1. Понятие программного обеспечения, классификация программного обеспечения.
2. Жизненный цикл ПО и его стандартизация, процессы ЖЦ ПО, группы процессов ЖЦ ПО.
3. Процесс разработки ПО: основные действия и их содержание.
4. Сертификация процессов разработки ПО, модель CMM.
5. Стратегии жизненного цикла ПО: понятие, виды и их сравнительная характеристика.
6. Каскадная модель жизненного цикла ПО: описание, преимущества и недостатки, критерии применения.
7. V-образная модель жизненного цикла ПО: описание, преимущества и недостатки, критерии применения.
8. Инкрементная модель жизненного цикла ПО: описание, преимущества и недостатки, критерии применения.
9. Спиральная модель жизненного цикла ПО: описание, преимущества и недостатки, критерии применения.
10. RAD модель жизненного цикла ПО: описание, преимущества и недостатки, критерии применения.
11. Структурный подход к разработке ПО: основные принципы и методы.
12. Методология IDEF0: назначение, ICOM модель, правила построения диаграммы.
13. Методология IDEF0: назначение, правила построения иерархии диаграмм, критерии завершения и стратегии декомпозиции.
14. Методология DFD: назначение, элементы диаграммы и их назначение, правила построения диаграммы.
15. Методология DFD: правила построения иерархии диаграмм, спецификации и их содержание.
16. Методология IDEF1X: назначение, сущности и связи: понятие и их обозначения.
17. Методология IDEF1X: назначение, виды и уровни моделей, порядок построения.
18. Методология IDEF3: назначение, единица работы, связи и их виды, соединения и их виды.
19. Основные этапы проектирования программных систем и их содержание.
20. Объектно-ориентированный подход к разработке ПО: основные понятия и принципы.
21. Язык UML: причины появления и история развития языка, структура языка.
22. Канонические диаграммы языка UML: их виды и типы, рекомендации построения.
23. Диаграмма вариантов использования: назначение, принципы построения.
24. Оценка трудоемкости разработки проекта на основе вариантов использования.
25. Диаграмма классов: назначение, классы, обозначение классов, их атрибутов и операций.
26. Диаграмма классов: назначение, отношения между классами и их применение.
27. Диаграмма композитной структуры: композитные классы и их части, принципы построения, кооперации и их использование.
28. Диаграмма пакетов: назначение, пакеты и отношения между ними.
29. Диаграмма последовательности: назначение, линии жизни.
30. Диаграмма деятельности: назначение, понятие, семантика и обозначение деятельности, действия и дуг.
31. Диаграмма деятельности: узлы управления, их виды и применение.
32. Дополнительные элементы диаграммы деятельности: действия приема и передачи сигналов, центральный буфер и хранилище данных.
33. Диаграмма конечного автомата: назначение, простое и композитное состояния.

34.Диаграмма конечного автомата: простые и составные переходы, правила срабатывания переходов.
35.Диаграмма конечного автомата: псевдосостояния, их виды и применение.
36.Диаграмма компонентов: назначение, компоненты, интерфейсы и порты, соединения и их виды.
37.Диаграмма развертывания: назначение, узлы, артефакты, соединения и их виды.
38.Проблемы классического похода к разработке ПО и причины появления гибких методологий.
39.Гибкие методолгии. Преимущества и область применения.
40.Экстремальное программирование: понятие, базис XP, структура XP цикла разработки.
41.SCRUM процесс: понятие, роли, мероприятия, уровни команд в SCRUM.
42.Разработка, управляемая тестированием (Test Driven Development).
43.Разработка, управляемая поведением (Behavior Driven Development).
44.CASE¬средства: понятие, история появления и развития, структура и состав, классификация.

1. Понятие программного обеспечения, классификация программного обеспечения.

Программное обеспечение - это совокупность программ, выполненных вычислительной системой. К программному обеспечению (ПО) относится также вся область деятельности по проектированию и разработке (ПО): технология проектирования программ (нисходящее проектирование, структурное программирование и др.), методы тестирования программ, методы доказательства правильности программ, анализ качества работы программ и др.

Программное обеспечение - программа или множество программ, используемых для управления компьютером.

Программное обеспечение - неотъемлемая часть ЭВМ. Оно является логическим продолжением технических средств ЭВМ, расширяющие их возможности и сферу использования.

Существует три категории:

1) Прикладные программы, непосредственно обеспечивающие выполнение необходимых пользователям работ.

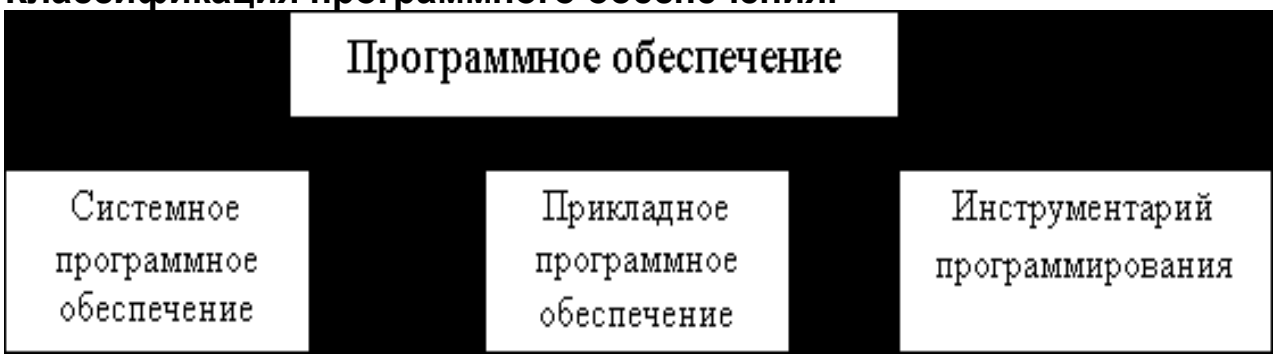
2) Системные программы:

- управление ресурсами ЭВМ.
- создание копий используемой информации.
- проверку работоспособности устройств компьютера.
- выдачу справочной информации о компьютере и др..

3) Инструментальные программные системы, облегчающие процесс создания новых программ для компьютера.

- Более или менее определенно сложились следующие группы программного обеспечения:
- операционные системы.
- системы программирования.
- инструментальные системы.
- интегрированные пакеты.
- динамические электронные таблицы.
- системы машинной графики.
- системы управления базами данных (СУБД).
- прикладное программное обеспечение.

Классификация программного обеспечения.



2. Жизненный цикл ПО и его стандартизация, процессы ЖЦ ПО, группы процессов ЖЦ ПО

Жизненный цикл программного обеспечения (ЖЦ ПО) – период времени, который начинается с момента принятия решения о необходимости создания ПО и заканчивается в момент его полного снятия с эксплуатации.

Основным нормативным документом, регламентирующим состав процессов ЖЦ ПО является международный стандарт ISO/IEC 12207:1995 «Information Technology – Software Life Cycle Processes».

С момента своего утверждения данный стандарт несколько раз пересматривался, и современная его версия носит обозначение ISO/IEC 12207:2008.

Процесс – совокупность взаимосвязанных действий (а каждое действие – набор задач), преобразующих некоторые входные данные в выходные.

Каждый процесс характеризуется задачами и методами их решения, исходными данными, полученными от других процессов, и результатами.

Каждый процесс, действие или задача инициируется и выполняется другим процессом по мере необходимости, причем не существует заранее определенных последовательностей выполнения (естественно, при сохранении связей по входным данным).

Согласно стандарту ISO/IEC 12207 все процессы ЖЦ ПО разделены на три группы:

- основные процессы;
- вспомогательные процессы;
- организационные процессы.

Основные процессы:

1. *приобретение* – регламентирует действия, выполняемые заказчиком в процессе приобретения программного продукта;

2. *поставка* – охватывает действия и задачи, выполняемые поставщиком, который снабжает заказчика программным продуктом или услугой;

3. *разработка* – предусматривает действия и задачи, выполняемые разработчиком, и охватывает работы по созданию ПО и его компонентов;

4. *эксплуатация* – охватывает действия и задачи оператора – организации, эксплуатирующей систему;

5. *сопровождение* – предусматривает действия и задачи, выполняемые сопровождающей организацией.

Вспомогательные процессы:

1. *документирование* – предусматривает формализованное описание информации, созданной в течении ЖЦ ПО;

2. *управление конфигурацией* – предполагает применение административных и технических процедур на всем протяжении ЖЦ ПО для определения состояния компонентов ПО в системе, управления модификациями ПО, описания и подготовки отчетов о состоянии компонентов ПО и т.д.

3. *обеспечение качества* – обеспечивает соответствующие гарантии того, что ПО и процессы его ЖЦ соответствуют заданным требованиям;

4. *верификация* – состоит в определении того, что программные продукты, являющиеся результатами некоторого действия, полностью удовлетворяют определенным требованиям;

5. *аттестация* – предусматривает определение полноты соответствия заданных требований и созданной системы или программного продукта их конкретному функциональному назначению;

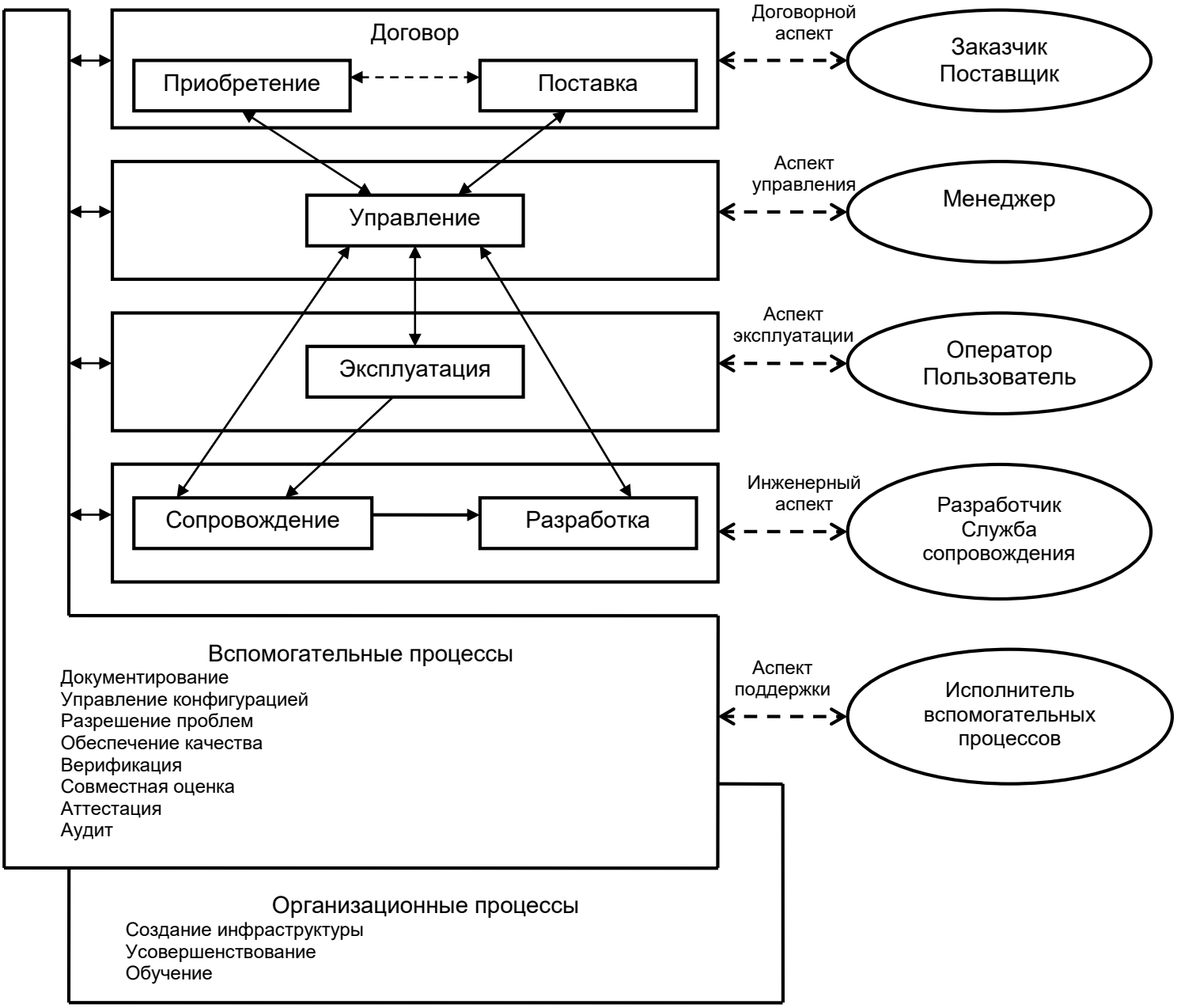
6. *совместная оценка* – предназначена для оценки состояния работ по проекту и ПО, создаваемому при выполнении данных работ (действий);

7. *аудит* – представляет собой определение соответствия требованиям, планам и условиям договора;

8. *разрешение проблем* – предусматривает анализ и решение проблем (включая обнаруженные несоответствия), независимо от их происхождения или источника, которые обнаружены в ходе разработки, эксплуатации, сопровождения или других процессов.

Организационные процессы ЖЦ ПО

1. ***управление*** – состоит из действий и задач, которые могут выполняться любой стороной, управляющей своими процессами;
2. ***инфраструктура*** – охватывает выбор и поддержку (сопровождение) технологии, стандартов и инструментальных средств, выбор и установку аппаратных и программных средств, используемых для разработки, эксплуатации или сопровождения ПО;
3. ***усовершенствование*** – предусматривает оценку, измерение, контроль и усовершенствование процессов ЖЦ ПО;
4. ***обучение*** – охватывает первоначальное обучение и последующее постоянное повышение квалификации персонала



3. Процесс разработки ПО: основные действия и их содержание

Процесс разработки предусматривает действия и задачи, выполняемые разработчиком, и охватывает работы по созданию ПО и его компонентов в соответствии с заданными требованиями, включая оформление проектной и эксплуатационной документации, подготовку материалов, необходимых для проверки работоспособности и соответствующего

Процесс разработки включает следующие действия:

1. подготовительную работу;
2. анализ требований к системе;
3. проектирование архитектуры системы;
4. анализ требований к ПО;
5. проектирование архитектуры ПО;
6. детальное проектирование ПО;
7. кодирование и тестирование ПО;
8. интеграцию ПО;
9. квалификационное тестирование ПО;
10. интеграцию системы;
11. квалификационное тестирование системы;
12. установку ПО;
13. приемку ПО.

Подготовительная работа:

- выбор модели ЖЦ ПО, соответствующей масштабу, значимости и сложности проекта;
- выбор и адаптация к условиям проекта стандартов, методов и средств разработки;
- составление плана выполнения работ.

Анализ требований к системе подразумевает определение:

- функциональных возможностей;
- пользовательских требований;
- требований к надежности и безопасности;
- требований к внешним интерфейсам;

и т.д.

Требования к системе оцениваются исходя из критериев реализуемости и возможности проверки при тестировании.

Проектирование архитектуры системы на высоком уровне заключается в определении:

- компонентов ее оборудования,
- программного обеспечения,
- операций, выполняемых эксплуатирующим систему персоналом.

Архитектура системы должна соответствовать требованиям, предъявляемым к системе, а также принятым проектным стандартам и методам.

Анализ требований к ПО предполагает определение следующих характеристик для каждого компонента ПО:

- функциональных возможностей, включая характеристики производительности и среды функционирования компонента;
- внешних интерфейсов;
- спецификаций надежности и безопасности;
- эргономических требований;
- требований к используемым данным;
- требований к установке и приемке;

- требований к пользовательской документации;
- требований к эксплуатации и сопровождению.

Проектирование архитектуры ПО включает следующие задачи (для каждого компонента ПО):

- трансформацию требований к ПО в архитектуру, определяющую на высоком уровне структуру ПО и состав его компонентов;
- разработку и документирование программных интерфейсов ПО и баз данных;
- разработку предварительной версии пользовательской документации;
- разработку и документирование предварительных требований к тестам и плана интеграции ПО.

Детальное проектирование ПО включает следующие задачи:

- описание компонентов ПО и интерфейсов между ними на более низком уровне, достаточном для их последующего самостоятельного кодирования и тестирования;
- разработку и документирование детального проекта базы данных;
- обновление (при необходимости) пользовательской документации;
- разработку и документирование требований к тестам и плана тестирования компонентов ПО;
- обновление плана интеграции ПО.

Кодирование и тестирование ПО охватывает следующие задачи:

- разработку (кодирование) и документирование каждого компонента ПО и базы данных, а также совокупности тестовых процедур и данных для их тестирования;
- тестирование каждого компонента ПО и базы данных на соответствие предъявляемым к ним требованиям;
- обновление (при необходимости) пользовательской документации;
- обновление плана интеграции ПО.

Интеграция ПО предусматривает сборку разработанных компонентов ПО в соответствии с планом интеграции и тестирование агрегированных компонентов.

Квалификационное требование – это набор критериев или условий, которые необходимо выполнить, чтобы квалифицировать программный продукт как соответствующий своим спецификациям и готовый к использованию в условиях эксплуатации.

Квалификационное тестирование ПО проводится разработчиком в присутствии заказчика (по возможности) для демонстрации того, что ПО удовлетворяет своим спецификациям и готово к использованию в условиях эксплуатации.

Интеграция системы заключается в сборке всех ее компонентов, включая ПО и оборудование. После интеграции система, в свою очередь, подвергается квалификационному тестированию на соответствие совокупности требований к ней.

Установка ПО осуществляется разработчиком в соответствии с планом в той среде и на том оборудовании, которые предусмотрены договором. В процессе установки проверяется работоспособность ПО и баз данных.

Приемка ПО предусматривает оценку результатов квалификационного тестирования ПО и системы и документирование результатов оценки, которые проводятся заказчиком с помощью разработчика.

4. Сертификация процессов разработки ПО, модель CMM

Гарантия качества процессов разработки программных продуктов является весьма значимой в современных условиях. Такую гарантию дают сертификаты качества процесса, подтверждающие его соответствие принятым международным стандартам. Каждый такой стандарт фиксирует свою модель обеспечения качества.

Основные модели стандартов:

- модели стандартов ISO 9001:2000,
- ISO/IEC 15504,
- модель зрелости процесса разработки ПО (Capability Maturity Model – CMM)

Зрелость процессов – это степень их управляемости, контролируемости и эффективности. Повышение технологической зрелости означает потенциальную возможность возрастания устойчивости процессов и указывает на степень эффективности и согласованности использования процессов создания и сопровождения ПО в рамках всей организации.

В CMM выделены **пять уровней** технологической зрелости:

- уровень 1: Начальный;
- уровень 2: Повторяемый;
- уровень 3: Определенный;
- уровень 4: Управляемый;
- уровень 5: Оптимизирующий.

Начальный уровень означает, что процесс в компании не формализован. Он не может строго планироваться и отслеживаться, его успех носит случайный характер. Результат работы целиком и полностью зависит от личностных качеств отдельных сотрудников, увольнение которых приводит к остановке проекта.

На ***повторяемом уровне*** внедряются формальные процедуры для выполнения основных элементов процесса конструирования. Результаты выполнения процесса соответствуют заданным требованиям и стандартам. Выполнение проекта на этом уровне планируется и контролируется, а применяемые для этих целей средства дают возможность повторения ранее достигнутых успехов.

Определенный уровень требует, чтобы все элементы процесса были определены, стандартизированы и задокументированы. На этом уровне все процессы планируются и управляются на основе единого стандарта компании. Качество разрабатываемого ПО уже не зависит от способностей отдельных личностей.

На ***управляемом уровне*** в компании принимаются количественные показатели качества как программных продуктов, так и технологических процессов. Это обеспечивает более точное планирование проекта и контроль качества его результатов. Основное отличие от предыдущего уровня состоит в более объективной, количественной оценке продукта и процесса.

На высшем, ***оптимизирующем***, уровне главной задачей компании становится постоянное улучшение и повышение эффективности существующих процессов, ввод новых технологий. Технология создания и сопровождения программных продуктов планомерно и последовательно совершенствуется.

Уровень модели CMM	Процессы
Начальный уровень	
Повторяемый уровень	Управление требованиями
	Управление конфигурацией
	Планирование проекта
	Мониторинг и контроль проекта
	Управление контрактами

	Измерения и анализ
	Обеспечение качества процесса и продукта
Определенный уровень	Спецификация требований
	Интеграция продукта
	Верификация
	Аттестация
	Стандартизация процессов организации
	Обучение
	Интегрированное управление проектом
	Управление рисками
	Анализ и принятие решений
Управляемый уровень	Управление производительностью и продуктивностью
	Количественное управление проектом
Оптимизирующий уровень	Внедрение технологических и организационных инноваций
	Причинно-следственный анализ и разрешение проблем

Оценка технологической зрелости компаний может использоваться:

- заказчиком при отборе лучших исполнителей;
- компаниями-производителями ПО для систематической оценки состояния своих технологических процессов и выбора направлений их совершенствования;
- компаниями, решившими пройти аттестацию, для оценки своего текущего состояния;
- аудиторами для определения стандартной процедуры аттестации и проведения необходимых оценок;
- консалтинговыми фирмами, занимающимися реструктуризацией компаний и служб поставщиков информационных технологий и связанных с ними услуг.

5. Стратегии ЖЦ ПО: понятие, виды и их сравнительная характеристика

Стратегия жизненного цикла программного обеспечения – порядок следования и содержания основных этапов процесса разработки.

Модель жизненного цикла программного обеспечения – структура, содержащая процессы действия и задачи, которые осуществляются в ходе разработки, использования и сопровождения программного продукта.

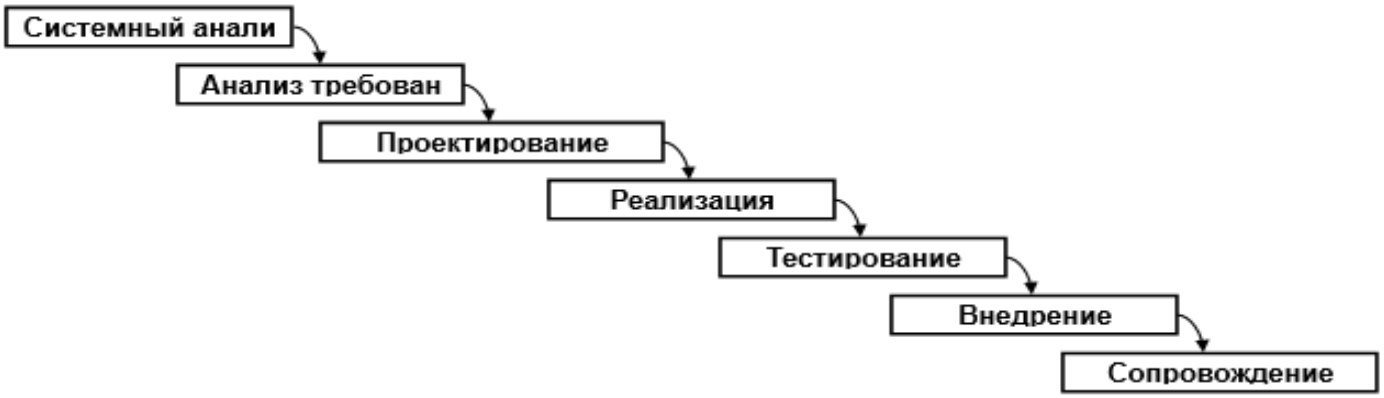
Виды стратегий жизненного цикла:

- однократный проход (водопадная стратегия) – линейная последовательность этапов конструирования;
- инкрементная стратегия – разработка ПО ведется с определения всех требований к ПО, а оставшаяся часть разработки выполняется в виде последовательности версий;
- эволюционная стратегия – ПО также разрабатывается в виде версий, но требования формируются и уточняются по ходу разработки от версии к версии

Стратегия конструирования	**Определение требований в начале разработки**	**Количество циклов разработки**	**Распространение промежуточного ПО**
Однократный проход	Все	Один	Нет
Инкрементная стратегия	Все	Несколько	Возможно
Эволюционная стратегия	Только часть требований	Несколько	Да

6. Каскадная модель ЖЦ ПО: описание, преимущества и недостатки, критерии применения

Каскадная модель ЖЦ ПО (или как ее еще называют, водопадная модель) реализует классический жизненный цикл ПО (автор Уинстон Ройс, 1970 г), являющийся старейшей парадигмой процесса разработки. Согласно этой модели, разработка ПО рассматривается как последовательность этапов, причем переход на



следующий этап осуществляется только по завершении всех работ на текущем этапе.

Этап системного анализа:

- задается роль каждого элемента в системе и их взаимодействие друг с другом;
- формируется интерфейс ПО с другими элементами (аппаратурой, базами данных, пользователями).
- начинается решение задачи планирования проекта.

Этап анализа требований:

- детальное определение функциональных и нефункциональных требований, предъявляемых к разрабатываемому ПО;
- завершается задача планирования проекта.

Этап проектирования:

- создание представлений:
- архитектуры ПО;
- модульной структуры ПО;
- алгоритмической структуры ПО;
- структуры данных;
- графического интерфейса пользователя.
- оценка качества будущего программного обеспечения.

Этап реализации:

- преобразование проектных спецификаций в текст на языке программирования (кодирование).

Этап тестирования:

- проверка корректности выполнения программы;
- обнаружение и исправление ошибок в функциях, логике и форме реализации программного продукта.

Этап внедрения:

- выполнение установки разработанного ПО у заказчика;
- обучение персонала;
- плавный переход от старого ПО (если оно есть) к использованию нового.

Этап сопровождения:

Внесение изменений в эксплуатируемое ПО с целями:

- исправления ошибок;
- адаптации к изменениям внешней для ПО среды;
- усовершенствования ПО по требованиям заказчика.

Преимущества:

- широкая известность модели;
- упорядоченность преодоления сложностей и хорошо срабатывает для тех проектов, которые достаточно понятны, но все же трудно разрешимы;
- она проста и удобна в применении;
- она отличается стабильностью требований;
- удобна, когда требования к качеству доминируют над требованиями к затратам и графику выполнения проекта;
- она способствует осуществлению строгого контроля менеджмента проекта;
- она облегчает работу менеджеру проекта по составлению плана и комплектации команды разработчиков;
- она позволяет участникам проекта, завершившим действия на выполняемой ими фазе, принять участие в реализации других проектов;
- она определяет процедуры по контролю за качеством;
- стадии модели довольно хорошо определены и понятны;

ход выполнения проекта легко проследить с помощью использования временной шкалы (или диаграммы Гантта).

Недостатки:

- в основе модели лежит последовательная линейная структура;
- невозможность предотвращения возникновение итераций между фазами;
- она не отображает основное свойство разработки ПО, направленное на разрешение задач;
- она может создать ошибочное впечатление о работе над проектом;
- интеграция всех полученных результатов происходит внезапно в завершающей стадии работы модели;
- у клиента практически нет возможности ознакомиться с системой заранее;
- пользователи не могут убедиться в качестве разработанного продукта до окончания всего процесса разработки;
- у пользователя нет возможности постепенно привыкнуть к системе;
- каждая фаза является предпосылкой для выполнения последующих действий;
- для каждой фазы создаются результативные данные, которые по его завершению считаются замороженными;
- все требования должны быть известны в начале жизненного цикла;
- необходимость в жестком управлении и контроле;
- модель основана на документации;
- весь программный продукт разрабатывается за один раз;
- отсутствует возможность учесть переделку и итерации за рамками проекта.

Критерии применения каскадной модели:

- требования к ПО и их реализация максимально четко определены и понятны;
- неизменяемое определение продукта и вполне понятные технические методики;
- если компания имеет опыт построения подобного рода систем.

Область применения: сложные системы с большим количеством задач вычислительного характера, системы управления производственными процессами повышенной опасности и др.

7. V-образная модель жизненного цикла ПО: описание, преимущества и недостатки, критерии применения.

В этой модели особое значение придается действиям, направленным на верификацию и аттестацию продукта. Является разновидностью каскадной модели, наследовала от нее такую же последовательную структуру. Модель демонстрирует комплексный подход к определению фаз процесса разработки ПО. В ней подчеркнуты взаимосвязи, существующие между аналитическими фазами и фазами проектирования, которые предшествуют кодированию, после которого следуют фазы тестирования. Пунктирные линии означают, что эти фазы необходимо рассматривать параллельно.

Планирование проекта и требований – определяются системные требования, а также то, каким образом будут распределены ресурсы организации.

Анализ требований к продукту и его спецификации– анализ существующей проблемы с ПО, завершается полной спецификацией создаваемой программной системы;

Архитектура или проектирование на высшем уровне– каким образом функции ПО должны применяться при реализации проекта;

Детализированная разработка проекта– обосновывает алгоритмы для каждого компонента, который был определен на фазе построения архитектуры.

Разработка программного кода– выполняется преобразование алгоритмов, определенных на этапе детализированного проектирования, в готовое ПО;

Модульное тестирование –проверка каждого закодированного модуля на наличие ошибок;

Интеграция и тестирование –установка взаимосвязей между группами модулей;

Системное и приемочное тестирование – проверка функционирования системы в целом;

Производство, эксплуатация и сопровождение – ПО запускается в производство;

Приемочные испытания (на рисунке нет) – позволяет пользователю протестировать функциональные возможности системы на соответствие исходным требованиям.

Преимущества:

- особое значение придается планированию, направленному на верификацию и аттестацию;
- предусмотрены аттестация и верификация всех внешних и внутренних полученных данных, а не только самого программного продукта;
- определение требований выполняется перед разработкой проекта системы, а проектирование ПО — перед разработкой компонентов;
- можно отслеживать ход процесса разработки - завершение фазы является контрольной точкой;
- модель проста в использовании.

Недостатки:

- непросто справиться с параллельными событиями;
- не предусмотрено внесение динамических изменений на разных этапах жизненного цикла;
- тестирование требований в жизненном цикле происходит слишком поздно;
- в модель не входят действия, направленные на анализ рисков.

Критерии применения:

- лучше всего срабатывает тогда, когда вся информация о требованиях доступна заранее.

- эффективна в том случае, когда доступными являются информация о методе реализации решения и технология, а персонал владеет умениями и опытом в работе с данной технологией.
- отличный выбор для систем, в которых требуется высокая надежность.

8. Инкрементная модель ЖЦ ПО: описание, преимущества и недостатки, критерии применения

Инкрементная разработка представляет собой процесс частичной реализации всей системы и медленного наращивания функциональных возможностей.

Инкрементная модель действует по принципу каскадной модели с перекрытиями.

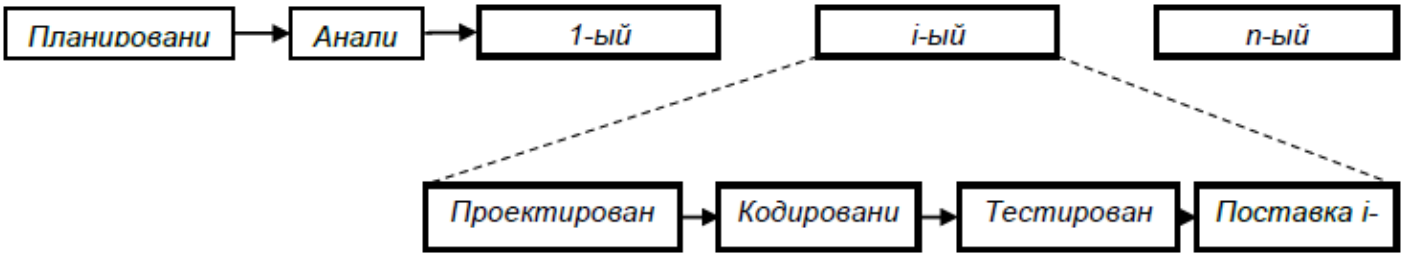
Два подхода к набору требований:

— полный заранее сформированный набор требований, которые выполняются в виде последовательных, небольших по размеру проектов,

— выполнение проекта может начаться с формулирования общих целей, которые затем уточняются и реализуются группами разработчиков.

Предполагается, что на ранних этапах ЖЦ (планирование, анализ и разработка проекта) выполняется конструирование системы в целом. На этих этапах определяются относящиеся к ним инкременты и функции. Каждый инкремент затем проходит через остальные фазы ЖЦ: кодирование, тестирование и поставку (рис).

Сначала выполняется конструирование, тестирование и реализация набора функций, формирующих основу продукта, или требований первостепенной важности, играющих основную роль для успешного выполнения проекта либо снижающих степень риска. Последующие итерации распространяются на ядро системы, постепенно улучшая ее функциональные возможности или рабочую характеристику. Добавление функций осуществляется с помощью выполнения существенных инкрементов с целью в комплексного удовлетворения потребностей пользователя. Каждая дополнительная функция аттестуется в соответствии с целым набором требований.



<u>Преимущества</u>:

— Не требуется заранее тратить средства, необходимые для разработки всего проекта

— В результате выполнения каждого инкремента получается функциональный продукт;

— Заказчик располагает возможностью высказаться по поводу каждой разработанной версии системы;

— Правило по принципу «разделяй и властвуй» позволяет разбить возникшую проблему на управляемые части

— Существует возможность поддерживать постоянный прогресс в ходе выполнения проекта;

— Снижаются затраты на первоначальную постановку программного продукта;

— Ускоряется начальный график поставки

— Снижается риск неудачи и изменения требований;

— Заказчики могут распознать самые важные и полезные функциональные возможности продукта на более ранних этапах разработки;

— Риск распределяется на несколько меньших по размеру инкрементов

— Требования стабилизируются на момент создания определенного инкремента

— Инкременты функциональных возможностей несут больше пользы и проще при тестировании

— Улучшается понимание требований для более поздних инкрементов

— В конце каждой инкрементной поставки существует возможность пересмотреть риски, связанные с затратами и соблюдением установленного графика;

— Использование последовательных инкрементов позволяет объединить полученный пользователями опыт в виде усовершенствованного продукта

— В процессе разработки можно ограничить количество персонала

— Возможность начать построение следующей версии проекта на переходном этапе предыдущей версии сглаживает изменения, вызванные сменой персонала;

— Потребности клиента лучше поддаются управлению

— Заказчик может привыкать к новой технологии постепенно;

— Ощутимые признаки прогресса при выполнении проекта помогают поддерживать вызванное соблюдением графика "давление" на управляемом уровне.

Недостатки:

— в модели не предусмотрены итерации в рамках каждого инкремента;

— определение полной функциональной системы должно осуществляться в начале жизненного цикла, чтобы обеспечить определение инкрементов;

— формальный критический анализ и проверку намного труднее выполнить для инкрементов, чем для системы в целом;

— заказчик должен осознавать, что общие затраты на выполнение проекта не будут снижены;

— поскольку создание некоторых модулей будет завершено значительно раньше других, возникает необходимость в четко определенных интерфейсах;

— использование на этапе анализа общих целей, вместо полностью сформулированных требований, может оказаться неудобным для руководства;

— для модели необходимы хорошее планирование и проектирование

— может возникнуть тенденция к оттягиванию решений трудных проблем на будущее с целью продемонстрировать руководству успех, достигнутый на ранних этапах разработки.

Критерии применения:

— если большинство требований можно сформулировать заранее, но их появление ожидается через определенный период времени;

— если рыночное окно слишком "узкое" и существует потребность быстро поставить на рынок продукт, имеющий функциональные базовые свойства;

— для проектов, на выполнение которых предусмотрен большой период времени разработки, как правило, один год;

— при равномерном распределении свойств различной степени важности;

— когда при рассмотрении риска, финансирования, графика выполнения проекта, размера программы, ее сложности или необходимости в реализации на ранних фазах оказывается, что самым оптимальным вариантом является применение принципа пофазовой разработки;

— при разработке программ, связанных с низкой или средней степенью риска;

— при выполнении проекта с применением новой технологии, что позволяет пользователю адаптироваться к системе путем выполнения более мелких инкрементных шагов, без резкого перехода к применению основного нового продукта;

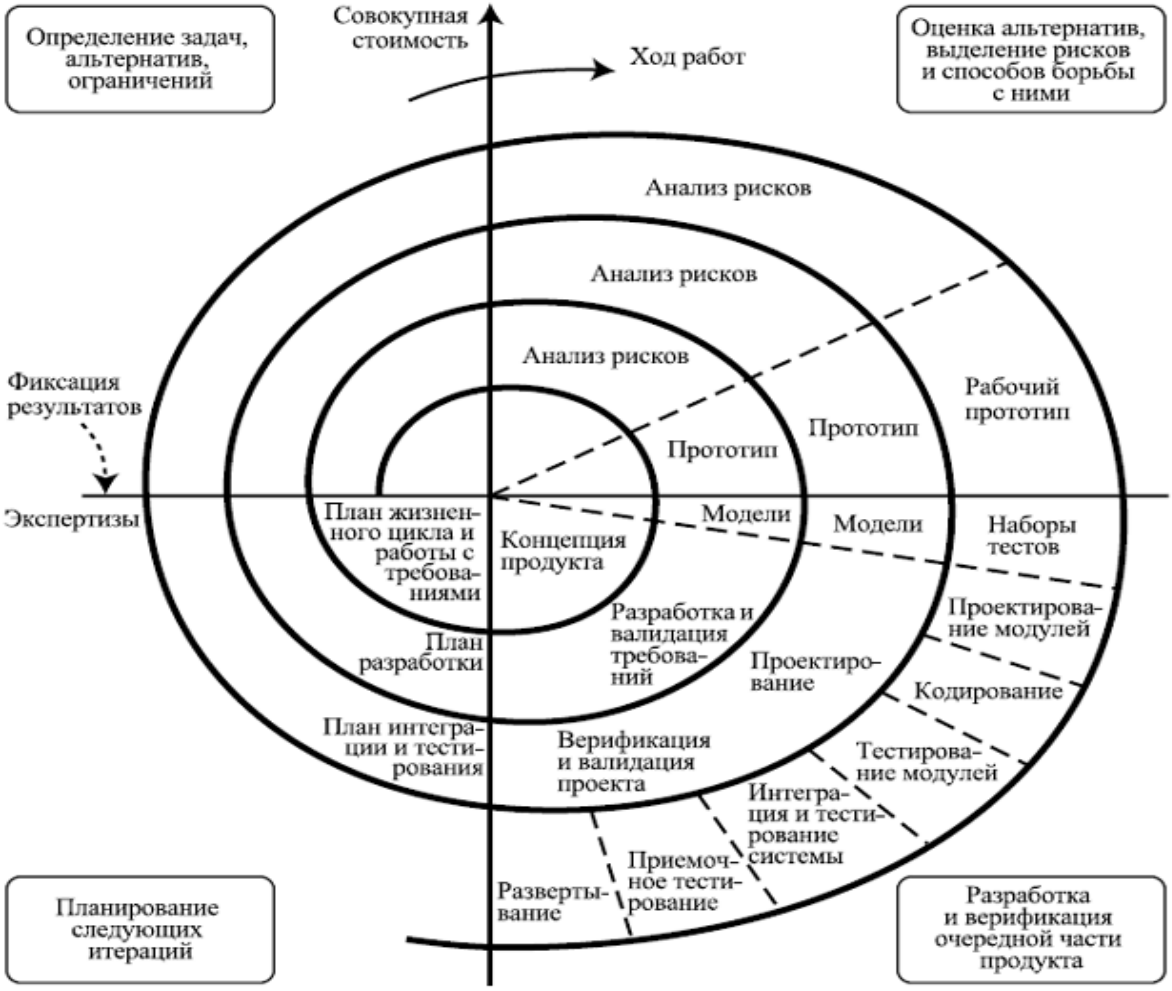
— когда однопроходная разработка системы связана с большой степенью риска;

— когда результативные данные получаются через регулярные интервалы времени.

9. Спиральная модель ЖЦ ПО: описание, преимущества и недостатки, критерии применения

Спиральная модель (автор: Барри Боэм, 1988) является реализацией эволюционной стратегии разработки программного обеспечения.

Модель отображает базовую концепцию, которая заключается в том, что каждый цикл представляет собой набор операций, которому соответствует такое же количество стадий, как и в модели каскадного процесса.



<u>Определение задач, альтернатив, ограничений</u>:

— Выполняется определение целей (рабочая характеристика, выполняемые функции, возможность внесения изменений, решающих факторов достижения успехам и аппаратного/программного интерфейса).

— Определяются альтернативные способы реализации этой части продукта (конструирование, повторное использование, покупка, субдоговор, и т.п.). Определяются ограничения, налагаемые на применение альтернативных вариантов (затраты, график выполнения, интерфейс, ограничения, относящиеся к среде и др.).

— Создается документация, подтверждающая риски, связанные с недостатком опыта в данной сфере, применением новой технологии, жесткими графиками, плохо организованными процессами и т.д.;

<u>Оценка альтернатив, выделение рисков и способов борьбы с ними</u>:

— Выполняется оценка альтернативных вариантов, относящихся к целям и ограничениям.

— Выполняется определение и разрешение рисков.

— Принятие решения о прекращении/продолжении работ над проектом.

<u>Разработка и верификация очередной версии продукта</u>:

— создание проекта;

— критический анализ проекта;

— разработку кода;
— проверку кода;
— тестирование и компоновку продукта.

Планирование следующих итераций:

— разработка плана проекта,
— разработка плана менеджмента конфигурацией,
— разработка плана тестирования;
— разработка плана установки программного продукта.

Преимущества:

– спиральная модель разрешает пользователям "увидеть" систему на ранних этапах
– обеспечивается определение непреодолимых рисков без особых дополнительных затрат;
– эта модель разрешает пользователям активно принимать участие при планировании, анализе рисков, разработке, а также при выполнении оценочных действий;
– она обеспечивает разбиение большого потенциального объема работы по разработке продукта на небольшие части
– в модели предусмотрена возможность гибкого проектирования
– реализованы преимущества инкрементной модели, а именно выпуск инкрементов, сокращение графика посредством перекрывания инкрементов
– здесь не ставится цель выполнить невозможное — довести конструкцию до совершенства;
– быстрая обратная связь по направлению от пользователей к разработчикам
– происходит усовершенствование административного управления
– повышается продуктивность благодаря использованию пригодных для повторного использования свойств;
– повышается вероятность предсказуемого поведения системы с помощью уточнения поставленных целей;
– при использовании спиральной модели не нужно распределять заранее все необходимые для выполнения проекта финансовые ресурсы;
– можно выполнять частую оценку совокупных затрат, а уменьшение рисков связано с затратами.

Недостатки:

– если проект имеет низкую степень риска или небольшие размеры, модель может оказаться дорогостоящей.
– модель имеет усложненную структуру, поэтому может быть затруднено ее применение разработчиками, менеджерами и заказчиками;
– серьезная нужда в высокопрофессиональных знаниях для оценки рисков;
– спираль может продолжаться до бесконечности, поскольку каждая ответная реакция заказчика на созданную версию может порождать новый цикл
– большое количество промежуточных стадий может привести к необходимости в обработке внутренней дополнительной и внешней документации;
– использование модели может оказаться дорогостоящим и даже недопустимым по средствам
– при выполнении действий на этапе вне процесса разработки возникает необходимость в переназначении разработчиков;
– могут возникнуть затруднения при определении целей и стадий, указывающих на готовность продолжать процесс разработки на следующей итерации;

– отсутствие хорошего средства или метода прототипирования может сделать использование модели неудобным;

– в производстве использование спиральной модели еще не получило такого широкого масштаба, как применение других моделей.

Критерии применения:

– когда создание прототипа представляет собой подходящий тип разработки продукта;

– когда важно сообщить, каким образом будет происходит увеличение затрат, и подсчитать затраты, связанные с выполнением действий из квадранта риска;

– когда организация обладает навыками, требуемыми для адаптации модели;

– для проектов, выполнение которых сопряжено со средней и высокой степенью риска;

– когда нет смыла браться за выполнение долгосрочного проекта из-за потенциальных изменений, которые могут произойти в экономических приоритетах, и когда такая неопределенность может вызвать ограничение во времени;

– когда речь идет о применении новой технологии и когда необходимо протестировать базовые концепции;

– когда пользователи не уверены в своих потребностях;

– когда требования слишком сложные;

– при разработке новой функции или новой серии продуктов;

– когда ожидаются существенные изменения, например, при изучении или исследовательской работе;

– когда важно сконцентрировать внимание на неизменяемых или известных частях

– в случае больших проектов;

– для организаций, которые не могут себе позволить выделить заранее все необходимые для выполнения проекта денежные средства, и когда в процессе разработки отсутствует финансовая поддержка;

– при выполнении затянувшихся проектов, которые могут вызывать раздражение у менеджеров и заказчиков;

– когда преимущества разработки невозможно точно определить, а достижение успеха не гарантировано;

– с целью демонстрации качества и достижения целей за короткий период времени;

– когда в процесс вовлекаются новые технологии, такие как впервые применяемые объектно-ориентированные принципы;

– при разработке систем, требующих большого объема вычислений, таких как систем, обеспечивающих принятие решений;

– при выполнении бизнес-проектов, а также проектов в области аэрокосмической промышленности, обороны и инжиниринга, где использование спиральной модели уже получило популярность.

10. RAD модель жизненного цикла ПО: описание, преимущества и недостатки, критерии применения.

RAD — модель быстрой разработки приложений, ключевыми для которой является скорость и удобство программирования. Первую версию RAD создал Барри Боэм в 1986 году, который назвал её «спиральная модель». Каждый виток спирали разбит на 4 сектора и соответствует разработке фрагмента или версии ПО. С каждым новым витком идёт углубление и уточнение целей, спецификаций проекта. В результате появляется возможность выбрать обоснованный вариант.

Принципы RAD сосредоточены на том, чтобы обеспечить основные преимущества методики быстрой разработки приложений: повышенная скорость разработки, низкая стоимость, высокое качество. С последним пунктом возникает больше всего проблем, потому что разработчик и заказчик видят предмет разработки по-разному. Для устранения этой и других проблем Джеймсом Мартином и его последователями были выделены следующие принципы RAD: минимизация временных затрат — инструментарий должен быть нацелен на уменьшение времени разработки, прототипирование — создание прототипов для конкретизации требований заказчика, цикличность разработки — каждая новая версия продукта основывается на оценке результата работы предыдущей версии заказчиком.

К однозначным преимуществам RAD относятся:

- высокое качество. Взаимодействие пользователей с прототипами повышает функциональность проектов, выполненных в рамках быстрой разработки приложений. Такое программное обеспечение может больше отвечать потребностям заказчика (конечного пользователя), чем при использовании методик Agile/Waterfall.

- контроль рисков — несмотря на то, что львиная часть материалов о RAD фокусируется на скорости и вовлечении пользователей как ключевым особенностям модели, нельзя исключать третье важное преимущество — снижение рисков. Интересно, но Боэм, создавший первую версию RAD, охарактеризовал спиральную модель как основанную на риске.

- Использование rapid application development позволяет уже на ранних стадиях сосредоточиться на главных факторах риска и приспособиться к ним.

- за единицу времени выполняется больше проектов в рамках бюджета — так как RAD подразумевает инкрементную модель разработки, шансы на критические ошибки, которые часто случаются в больших проектах по системе Waterfall, снижены.

- Если в проектах по водопадной системе реализация проекта была возможна после шести и более месяцев анализа и разработки, то в RAD вся необходима информация открывается раньше, во время самого процесса создания приложения.

К минусам rapid application development можно отнести:

- риск «новизны» — для большинства IT-компаний RAD было новинкой, требующей переосмысления привычных методик работы. Сопротивление новому, необходимость изучения с нуля инструментов и техник приводят к ошибкам во время первых внедрений rapid application development.

- если пользователи не могут постоянно брать участие в процессе разработки на протяжении всего жизненного цикла, это может негативно повлиять на конечный продукт — особенностью RAD является увеличенное взаимодействие пользователей и разработчиков в отличии от моделей Waterfall, в которых роль пользователей сводится к определению требований. Далее разработчики сами создают систему.

- уменьшенный контроль — гибкость, адаптивность процесса как одно из преимуществ RAD в идеале означает возможность быстро адаптироваться как к проблемам, так и возможностям.

- К сожалению, придётся отдать предпочтение чему-то одному — гибкости или контролю. В последнем случае методика быстрой разработки приложений не будет жизнеспособной.

- скудный дизайн — фокусирование на прототипах в некоторых случаях приводит к методике «взлом и тестирование», по которой разработчики постоянно вносят мелкие изменения в отдельные элементы и игнорируют проблемы системной архитектуры.

11. Структурный подход к разработке ПО: основные принципы и методы.

Сущность структурного подхода к разработке ИС заключается в ее декомпозиции (разбиении) на автоматизируемые функции: система разбивается на функциональные подсистемы, которые в свою очередь делятся на подфункции, подразделяемые на задачи и так далее. Процесс разбиения продолжается вплоть до конкретных процедур. При этом автоматизируемая система сохраняет целостное представление, в котором все составляющие компоненты взаимоувязаны.

При разработке системы "снизу-вверх" от отдельных задач ко всей системе целостность теряется, возникают проблемы при информационной стыковке отдельных компонентов.

Все наиболее распространенные методологии структурного подхода базируются на ряде общих принципов. В качестве двух базовых принципов используются следующие:

· принцип "разделяй и властвуй" – принцип решения сложных проблем путем их разбиения на множество меньших независимых задач, легких для понимания и решения;

· принцип иерархического упорядочивания – принцип организации составных частей проблемы в иерархические древовидные структуры с добавлением новых деталей на каждом уровне.

Выделение двух базовых принципов не означает, что остальные принципы являются второстепенными, поскольку игнорирование любого из них может привести к непредсказуемым последствиям (в том числе и к провалу всего проекта). Основными из этих принципов являются следующие:

· принцип абстрагирования – заключается в выделении существенных аспектов системы и отвлечения от несущественных;

· принцип формализации – заключается в необходимости строгого методического подхода к решению проблемы;

· принцип непротиворечивости – заключается в обоснованности и согласованности элементов;

· принцип структурирования данных –заключается в том, что данные должны быть структурированы и иерархически организованы.

В структурном анализе используются в основном две группы средств, иллюстрирующих функции, выполняемые системой и отношения между данными. Каждой группе средств соответствуют определенные виды моделей (диаграмм), наиболее распространенными среди которых являются следующие:

· SADT (Structured Analysis and Design Technique) модели и соответствующие функциональные диаграммы;

· DFD (Data Flow Diagrams) диаграммы потоков данных;

· ERD (Entity-Relationship Diagrams) диаграммы "сущность-связь".

На стадии проектирования ИС модели расширяются, уточняются и дополняются диаграммами, отражающими структуру программного обеспечения: архитектуру ПО, структурные схемы программ и диаграммы экранных форм.

Перечисленные модели в совокупности дают полное описание ИС независимо от того, является ли она существующей или вновь разрабатываемой. Состав диаграмм в каждом конкретном случае зависит от необходимой полноты описания системы.

12. Методология IDEF0: назначение, ICOM-модель, правила построения диаграммы

Методология функционального моделирования IDEF0 – это технология описания системы в целом как множества взаимозависимых действий, или функций.

Функции системы исследуются независимо от объектов, которые обеспечивают их выполнение.

Методология IDEF0 применяется на ранних этапах разработки проекта (анализ).

Построение модели IDEF0 заключается в выполнении следующих действий:

- сбор информации об объекте, определение его границ;
- определение цели и точки зрения модели;
- построение, обобщение и декомпозиция диаграмм;
- критическая оценка, рецензирование и комментирование.

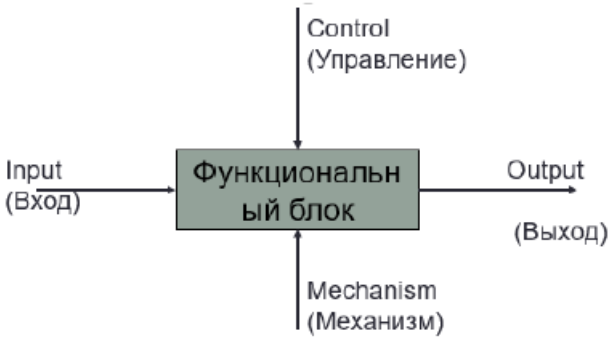
ICOM-модель

Действие, обычно в IDEF0 называемое функцией, обрабатывает или переводит входные параметры (сырье, информацию и т.п.) в выходные.

Функции изображаются на диаграммах как поименованные прямоугольники, или функциональные блоки.

Для отображения категорий информации существует аббревиатура ICOM, отображающая четыре возможных типа стрелок:

- I (Input) – вход – нечто, что потребляется в ходе выполнения процесса;
- C (Control) – управление – ограничения и инструкции, влияющие на ход выполнения процесса;
- O (Output) – выход – нечто, являющееся результатом выполнения процесса;
- M (Mechanism) – исполняющий механизм – нечто, что используется для выполнения процесса, но не потребляется само по себе.



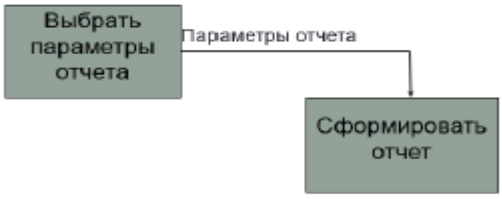
Соединения

В IDEF0 существует пять основных видов комбинированных стрелок:

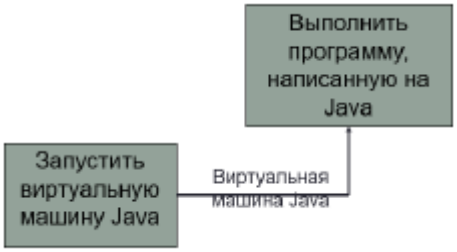
- выход – вход,
- выход – управление,
- выход – механизм исполнения,
- выход – обратная связь на управление,
- выход – обратная связь на вход.

Выход – вход

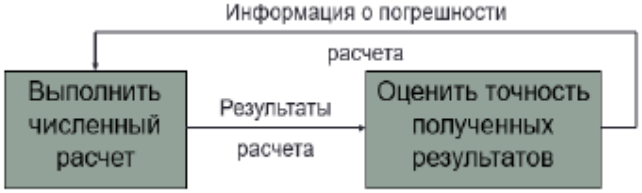
Выход – механизм



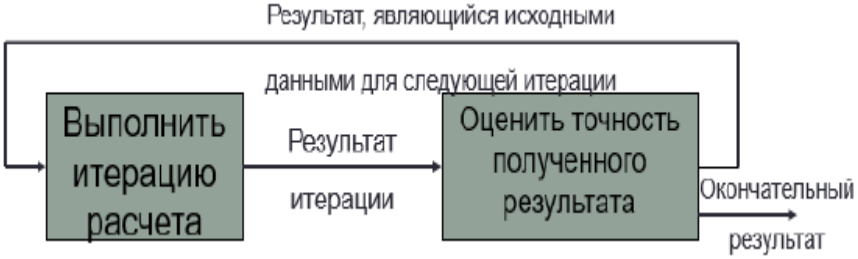
Выход – управление



Выход – обратная связь на управление



Выход – обратная связь на вход



Правила построения диаграмм:

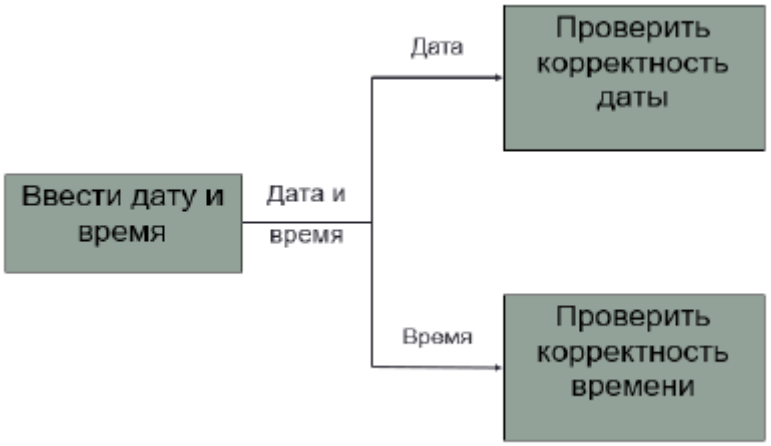
Синтаксис диаграмм определяется следующими правилами:

• диаграммы содержат блоки и дуги;

• блоки представляют функции;

• количество блоков на диаграмме ограничено: от 3 до 6-7

Разбиение и соединение стрелок:

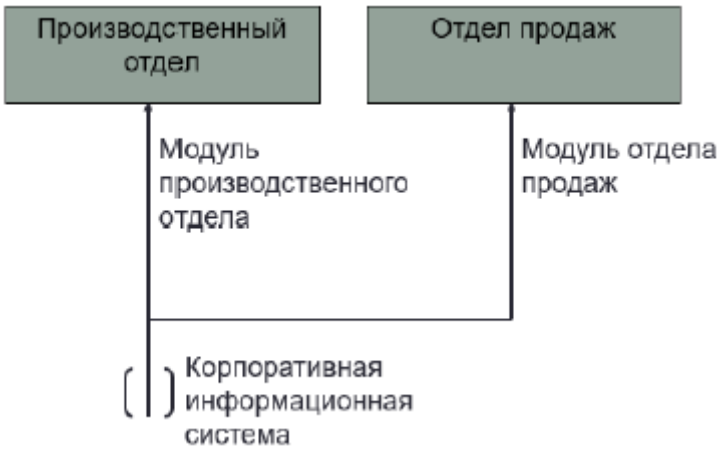


• блоки имеют доминирование (выражающееся в их ступенчатом расположении, причем доминирующий блок располагается в верхнем левом углу диаграммы);

• дуги изображают наборы объектов, передаваемых между блоками;

• дуги изображают взаимосвязи между блоками: выход-управление, выход-вход, обратная связь по управлению, обратная связь по входу, выход-механизм.

Туннели



Стратегии декомпозиции:

1. Функциональная декомпозиция.

1. Декомпозиция в соответствии с известными стабильными подсистемами.

2. Декомпозиция по физическому процессу.

Признаки завершения декомпозиции блока:

1. блок содержит достаточно деталей.

2. необходимо изменить уровень абстракции, чтобы достичь большей детализации блока.

3. необходимо изменить точку зрения, чтобы детализировать блок.

4. блок очень похож на другой блок той же модели или на блок другой модели.

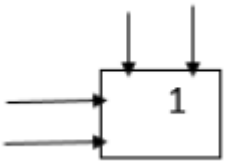
5. блок представляет тривиальную функцию.

В дополнение к контекстным диаграммам и диаграммам декомпозиции при разработке и представлении моделей могут применяться другие виды IDEF0-диаграмм:

• Дерево модели.

• Презентационные диаграммы.

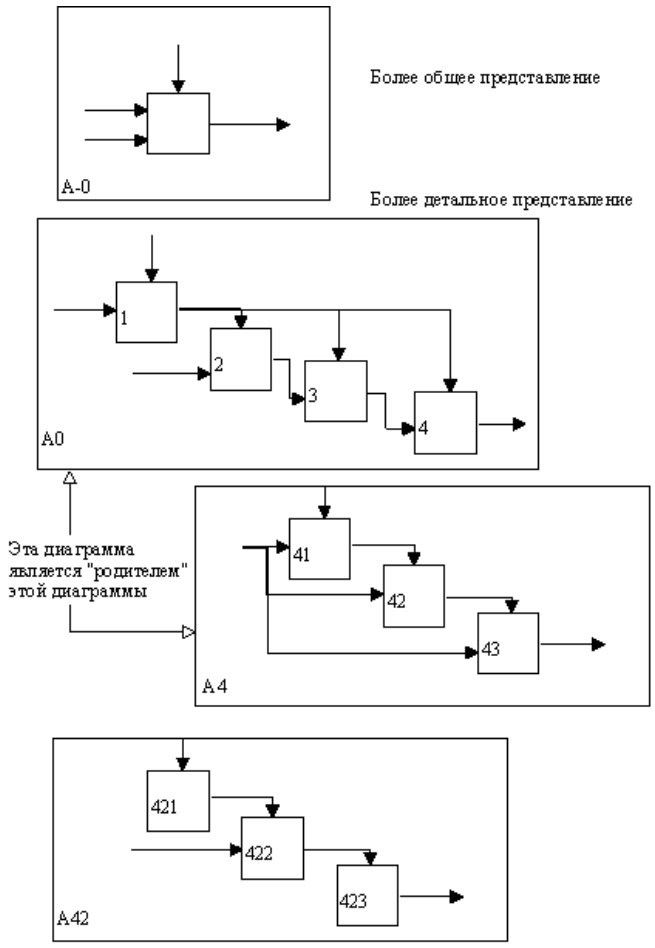
ICOM-коды


1

13. Методология IDEF0: назначение, правила построения иерархии диаграмм, критерии завершения и стратегии декомпозиции

IDEF0 сочетает в себе небольшую по объему графическую нотацию (она содержит только два обозначения: блоки и стрелки) со строгими и четко определенными рекомендациями, в совокупности предназначенными для построения качественной и понятной модели системы.

Методология IDEFO в некоторой степени напоминает рекомендации, существующие в книгоиздательском деле, часто набор напечатанных моделей IDEFO организуется в брошюру (называемую в терминах IDEFO комплект), имеющую содержание, глоссарий и другие элементы, характерные для законченной книги.



<u>Правила построения иерархии диаграмм</u>:

Построение модели IDEF0 начинается с представления всей системы в виде простейшей компоненты – одного блока и дуг, изображающих интерфейсы с функциями вне системы. Поскольку единственный блок представляет всю систему как единое целое, имя, указанное в блоке, является общим. Это верно и для интерфейсных дуг – они также представляют полный набор внешних интерфейсов системы в целом.

Затем блок, который представляет систему в качестве единого модуля, детализируется на другой диаграмме с помощью нескольких блоков, соединенных интерфейсными дугами. Эти блоки представляют основные подфункции исходной функции. Данная декомпозиция выявляет полный набор подфункций, каждая из которых представлена как блок, границы которого определены интерфейсными дугами. Каждая из этих подфункций может быть декомпозирована подобным образом для более детального представления.

Во всех случаях каждая подфункция может содержать только те элементы, которые входят в исходную функцию. Кроме того, модель не может опустить какие-либо элементы, т.е., как уже отмечалось, родительский блок и его интерфейсы обеспечивают контекст. К нему нельзя ничего добавить, и из него не может быть ничего удалено.

Модель SADT представляет собой серию диаграмм с сопроводительной документацией, разбивающих сложный объект на составные части, которые представлены в виде блоков. Детали каждого из основных блоков показаны в виде блоков на других диаграммах. Каждая детальная диаграмма является декомпозицией блока из более общей диаграммы. На каждом шаге декомпозиции более общая диаграмма называется родительской для более детальной диаграммы.

Дуги, входящие в блок и выходящие из него на диаграмме верхнего уровня, являются точно теми же самыми, что и дуги, входящие в диаграмму нижнего уровня

и выходящие из нее, потому что блок и диаграмма представляют одну и ту же часть системы.

Пример построения иерархии диаграмм приведен на рисунке.

Стратегии декомпозиции:

– ***Функциональная декомпозиция*** — декомпозиция в соответствии с функциями, которые выполняют люди или организация. Может оказаться полезной стратегией для создания системы описаний, фиксирующей взаимодействие между людьми в процессе их работы. Очень часто, однако, взаимосвязи между функциями весьма многочисленны и сложны, поэтому рекомендуется использовать эту стратегию только в начале работы над моделью системы.

– ***Декомпозиция в соответствии с известными стабильными подсистемами*** — приводит к созданию набора моделей, по одной модели на каждую подсистему или важный компонент. Затем для описания всей системы должна быть построена составная модель, объединяющая все отдельные модели. Рекомендуется использовать разложение на подсистемы, только когда разделение на основные части системы не меняется. Нестабильность границ подсистем быстро обесценит как отдельные модели, так и их объединение.

– ***Декомпозиция по физическому процессу*** — выделение функциональных стадий, этапов завершения или шагов выполнения. Хотя эта стратегия полезна при описании существующих процессов (таких, например, как работа промышленного предприятия), результатом ее часто может стать слишком последовательное описание системы, которое не будет в полной мере учитывать ограничения, диктуемые функциями друг другу При этом может оказаться скрытой последовательность управления. Эта стратегия рекомендуется, только если целью модели является описание физического процесса как такового или только в крайнем случае, когда неясно, как действовать.

Признаки завершения декомпозиции блока:

– ***блок содержит достаточно деталей.*** Одна из типичных ситуаций, встречающихся в конце моделирования, — это блок, который описывает систему с нужным уровнем подробности. Проверить достаточность деталей обычно совсем легко, достаточно просто спросить себя, отвечает ли блок на все или на часть вопросов, составляющих цель модели. Если блок помогает ответить на один или более вопросов, то дальнейшая декомпозиция может не понадобиться;

– ***необходимо изменить уровень абстракции, чтобы достичь*** *шей детализации блока.* Блоки подвергаются декомпозиции, если они недостаточно детализированы для удовлетворения цели модели. Но иногда при декомпозиции блока выясняется, что диаграмма начинает описывать, как функционирует блок, вместо описания того, что блок делает. В этом случае происходит изменение уровня абстракции — изменение сути того, что должна представлять модель (т.е. изменение способа описания системы)

– ***необходимо изменить точку зрения, чтобы детализировать*** *блок.* Изменение точки зрения происходит примерно так же, как изменение уровня абстракции. Это чаще всего характерно для ситуаций, когда точку зрения модели нельзя использовать для декомпозиции конкретного блока, т. е. этот блок можно декомпозировать, только если посмотреть на него с другой позиции. Об этом может свидетельствовать заметное изменение терминологии;

– ***блок очень похож на другой блок той же модели или на блок*** *другой модели. И*ногда встречается блок, чрезвычайно похожий на другой блок модели. Два блока похожи, если они выполняют примерно одну и ту же функцию и имеют почти

одинаковые по типу и количеству входы, управления и выходы. Если второй блок уже декомпозирован, то разумно отложить декомпозицию и тщательно сравнить два блока. Если нужны ничтожные изменения для совпадения первого блока со вторым, то внесение этих изменений сократит усилия на декомпозицию и улучшит модульность модели (т.е. сходные функции уточняются согласованным образом);

– ***блок представляет тривиальную функцию.*** Тривиальная функция — это такая функция, понимание которой не требует ни каких объяснений. В этом случае очевидна целесообразность отказа от декомпозиции, потому что роль SADT заключается **в** превращении сложного вопроса в понятный, а не в педантичной разработке очевидных деталей. В таких случаях декомпозиция определенных блоков может принести больше вреда, чем пользы.

В дополнение к контекстным диаграммам и диаграммам декомпозиции при разработке и представлении моделей могут применяться другие виды IDEF0-диаграмм:

• Дерево модели.

• Презентационные диаграммы.

Презентационные диаграммы (For Exposition Only diagrams – FEO diagrams) часто включают в модели, чтобы проиллюстрировать другие точки зрения или детали, выходящие за рамки традиционного синтаксиса IDEF0.

Виды презентационных диаграмм:

• копия диаграммы IDEF0, которая содержит все функциональные блоки, и стрелки, относящиеся только к одному из функциональных блоков;

• копия диаграммы IDEF0, которая содержит все функциональные блоки, и стрелки, непосредственно относящиеся только к входу и (или) к выходу родительского блока;

различные точки зрения, как правило, на глубину одного уровня декомпозиции.

14. Методология DFD: назначение, элементы диаграммы и их назначение, правила построения диаграммы

Диаграммы потоков данных (Data Flow Diagram – DFD) моделируют систему как набор действий, соединенных друг с другом стрелками. Диаграммы потоков данных также могут содержать два новых типа объектов: объекты, собирающие и хранящие информацию – хранилища данных и внешние сущности – объекты, которые моделируют взаимодействие с теми частями системы (или другими системами), которые выходят за границы моделирования.

Виды **элементов**: ***внешние сущности***, ***процессы***, ***накопители данных***, ***потоки данных***.

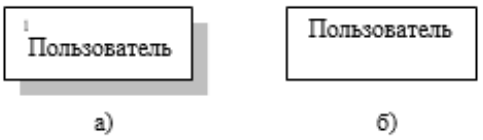


Рисунок 1 – Графическое изображение внешней сущности
а) – нотация Гейна – Сарсона, б) – нотация Йордана – Де Марко

Внешняя сущность представляет собой материальный объект или физическое лицо, источник или приемник информации (например, заказчики, персонал, поставщики, клиенты, склад).

Процесс представляет собой преобразование входных потоков данных в выходные в соответствии с определенным алгоритмом. Номер процесса служит для его идентификации.

Рисунок 2 – Графическое изображение процесса
а) – нотация Гейна – Сарсона, б) – нотация Йордана – Де Марко

В некоторых CASE-средствах используется модификация нотации Гейна – Сарсона, приведенная на рисунке. В этой нотации в нижней секции указывается механизм, выполняющий данный процесс.

Рисунок 3 – Модифицированная нотация Гейна – Сарсона для изображения процесса

Накопитель данных – это абстрактное устройство для хранения информации, которую можно в любой момент поместить в накопитель и через некоторое время извлечь, причем способы помещения и извлечения могут быть любыми. Накопитель данных может быть реализован физически в виде ящика в картотеке, таблицы в базе данных, файла на носителе информации и т.д.

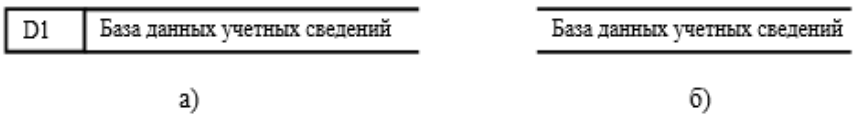


Рисунок 4 – Графическое изображение хранилища данных
а) – нотация Гейна – Сарсона, б) – нотация Йордана – Де Марко

Поток данных определяет информацию, передаваемую через некоторое соединение от источника к приемнику. Реальный поток данных может быть информацией, передаваемой по кабелю между двумя устройствами, пересылаемыми по почте письмами, магнитными лентами или дискетами, переносимыми с одного компьютера на другой, и т.д.

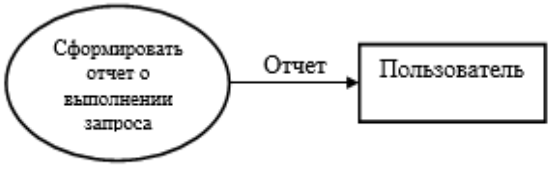


Рисунок 5 – Графическое изображение потока

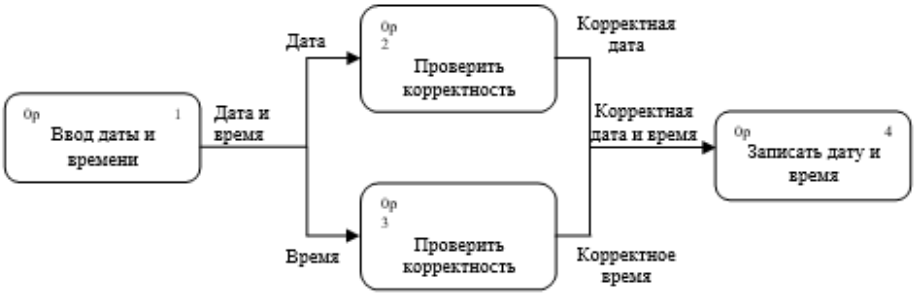


Рисунок 6 – Пример декомпозиции и объединения потока данных

Поскольку все стороны обозначающего функциональный блок DFD прямоугольника равнозначны (в отличие от IDEF0), стрелки могут начинаться и заканчиваться в любой части блока. В диаграммах потоков данных также используются двунаправленные стрелки, которые нужны для отображения взаимодействия между блоками.

Стрелки на DFD-диаграммах могут быть разбиты (разветвлены) на части, и при этом каждый получившийся сегмент может быть переименован таким образом, чтобы показать декомпозицию данных, переносимых данным потоком. Стрелки могут и соединяться между собой (объединяться) для формирования так называемых комплексных объектов. Пример разделения и объединения стрелок приведен на рисунке.

Главная цель построения иерархии DFD заключается в том, чтобы сделать описание системы ясным и понятным на каждом уровне детализации, а также разбить его на части с точно определенными отношениями между ними. Для достижения этого целесообразно пользоваться следующими **рекомендациями**:

- размещать на каждой диаграмме от 3 до 6—7 процессов (аналогично IDEF0);
- не загромождать диаграммы несущественными на данном уровне деталями;
- декомпозицию потоков данных осуществлять параллельно с декомпозицией процессов. Эти две работы должны выполняться одновременно, а не одна после завершения другой;
- выбирать ясные, отражающие суть дела, имена процессов и потоков, при этом стараться не использовать аббревиатуры.

Первым шагом при построении иерархии DFD является построение контекстных диаграмм. Обычно при проектировании относительно простых систем строится единственная контекстная диаграмма со звездообразной топологией, в центре которой находится так называемый главный процесс, соединенный с приемниками и источниками информации, посредством которых с системой взаимодействуют пользователи и другие внешние системы. Перед построением контекстной DFD необходимо проанализировать внешние события (внешние сущности), оказывающие влияние на функционирование системы. Количество потоков на контекстной диаграмме должно быть по возможности небольшим, поскольку каждый из них может быть в дальнейшем разбит на несколько потоков на следующих уровнях диаграммы.

Для проверки контекстной диаграммы можно составить список событий, который должен состоять из описаний действий внешних сущностей (событий) и соответствующих реакций системы на них. Каждое событие должно соответствовать одному или более потокам данных: входные потоки интерпретируются как воздействия, а выходные потоки – как реакции системы на входные потоки.

15. Методология DFD: правила построения иерархии диаграмм, спецификации и их содержание

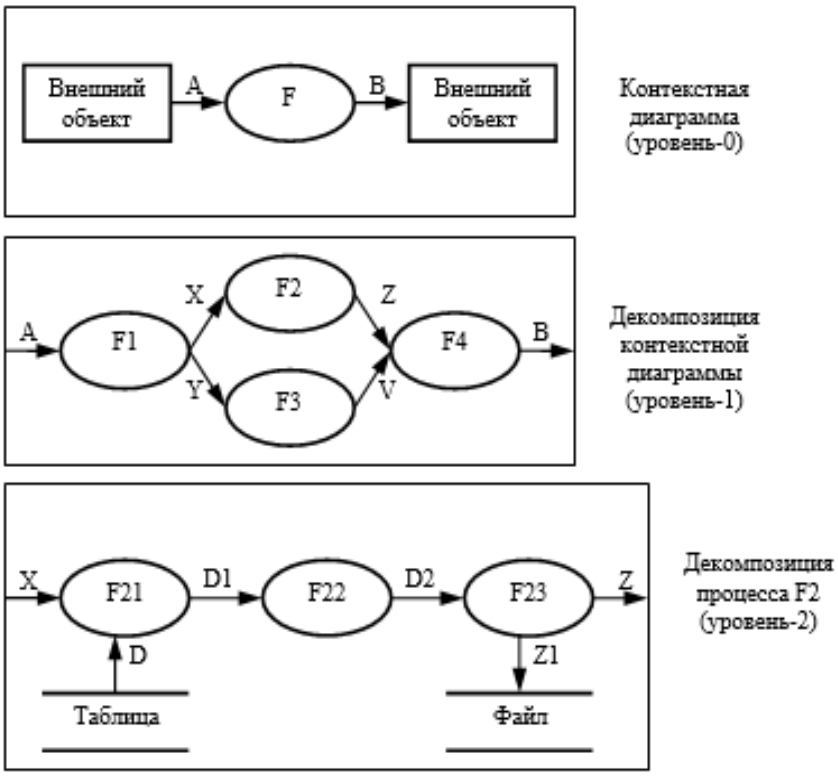


Рисунок 7 – Пример иерархии диаграмм потоков данных

Для **сложных систем** строится **иерархия контекстных диаграмм**. При этом контекстная диаграмма верхнего уровня содержит не единственный главный процесс, а набор подсистем, соединенных потоками данных. Контекстные диаграммы следующего уровня детализируют контекст и структуру подсистем.

Иерархия контекстных диаграмм определяет взаимодействие основных функциональных подсистем между собой, с внешними входными и выходными потоками данных и внешними объектами (источниками и приемниками информации), с которыми взаимодействует система.

Разработка контекстных диаграмм решает проблему строгого определения функциональной структуры системы на самой ранней стадии ее проектирования, что особенно важно для сложных многофункциональных систем, в создании которых участвуют разные организации и коллективы разработчиков.

После построения контекстных диаграмм полученную модель следует проверить на полноту исходных данных об объектах системы и изолированность объектов (отсутствие информационных связей с другими объектами).

Для каждой подсистемы, присутствующей на контекстных диаграммах, выполняется ее детализация при помощи DFD. Это можно сделать путем построения диаграммы для каждого события. Каждое событие представляется в виде процесса с соответствующими входными и выходными потоками, накопителями данных, внешними сущностями и ссылками на другие процессы для описания связей между этим процессом и его окружением. Затем все построенные диаграммы сводятся в одну диаграмму нулевого уровня.

Рекомендации построения диаграмм:

— размещать на каждой диаграмме от 3 до 6—7 процессов (аналогично IDEF0);

— не загромождать диаграммы несущественными на данном уровне деталями;

— декомпозицию потоков данных осуществлять параллельно с декомпозицией процессов;

— выбирать ясные, отражающие суть дела, имена процессов и потоков, при этом стараться не использовать аббревиатуры.

Правила детализации:

— ***правило балансировки*** – при детализации подсистемы или процесса детализирующая диаграмма в качестве внешних источников или приемников данных может иметь только те компоненты (подсистемы, процессы, внешние сущности, накопители данных), с которыми имеют информационную связь детализируемые подсистема или процесс на родительской диаграмме;

— ***правило нумерации*** – при детализации процессов должна поддерживаться их иерархическая нумерация. Например, процессы, детализирующие процесс с номером 12, получают номера 12.1, 12.2, 12.3 и т.д.

Спецификация процесса должна формулировать его основные функции таким образом, чтобы в дальнейшем специалист, выполняющий реализацию проекта, смог выполнить их или разработать соответствующую программу.

Критерии использования спецификации:

— наличия у процесса относительно небольшого количества входных и выходных потоков данных (2—3 потока);

— возможности описания преобразования данных процессом в виде последовательного алгоритма;

— выполнения процессом единственной логической функции преобразования входной информации в выходную;

— возможности описания логики процесса при помощи спецификации небольшого объема (не более 20-30 строк).

Требования к спецификации процессов:

— для каждого процесса нижнего уровня должна существовать одна и только одна спецификация;

— спецификация должна определять способ преобразования входных потоков в выходные;

— нет необходимости (по крайней мере, на стадии формирования требований) определять метод реализации этого преобразования;

— спецификация должна стремиться к ограничению избыточности – не следует переопределять то, что уже было определено на диаграмме;

— набор конструкций для построения спецификации должен быть простым и понятным.

Состав языка спецификации:

— глаголы, ориентированные на действие и применяемые к объектам;

— термины, определенные на любой стадии проекта ПО (например, задачи, процедуры, символы данных и т.п.);

— предлоги и союзы, используемые в логических отношениях;

— общеупотребительные математические, физические и технические термины;

— арифметические уравнения;

— таблицы, диаграммы, графы и т.п.;

— комментарии.

Соглашения использования структурированного естественного языка

— логика процесса выражается в виде комбинации последовательных конструкций, конструкций выбора и итераций;

— глаголы должны быть активными, недвусмысленными и ориентированными на целевое действие *(заполнить, вычислить, извлечь,* а не *модернизировать, обработать);*

— логика процесса должна быть выражена четко и недвусмысленно.

16. Методология IDEF1X: назначение, сущности и связи: понятие и их обозначение

Цель моделирования данных состоит в обеспечении разработчика системы концептуальной схемой базы данных в форме одной модели или нескольких локальных моделей, которые относительно легко могут быть отображены в любую систему баз данных. Наиболее распространенным средством моделирования данных (предметной области) является модель «сущность-связь» (ERM) (Автор Питер Чен, 1976 г.)

Базовыми понятиями ERM являются:

- сущность,
- связь,
- атрибут.

Сущность (Entity) – реальный, либо воображаемый объект, имеющий существенное значение для рассматриваемой предметной области.

Экземпляр сущности – это конкретный представитель данной сущности. Например, экземпляром сущности «Сотрудник» может быть «Сотрудник Иванов».

Каждая сущность должна обладать некоторыми **свойствами**:

- иметь уникальное имя;
- к одному и тому же имени должна всегда применяться одна и та же интерпретация;
- одна и та же интерпретация не может применяться к различным именам, если только они не являются псевдонимами;
- обладать одним или несколькими атрибутами, которые либо принадлежат сущности, либо наследуются через связь;
- обладать одним или несколькими атрибутами, которые однозначно идентифицируют каждый экземпляр сущности.

Атрибут (Attribute) – любая характеристика сущности, значимая для рассматриваемой предметной области и предназначенная для квалификации, идентификации, классификации, количественной характеристики или выражения состояния сущности.

Виды атрибутов

- простой – состоит из одного элемента данных;
- составной – состоит из нескольких элементов данных;
- однозначный – содержит одно значение для одной сущности;
- многозначный – содержит несколько значений для одной сущности;
- необязательный – может иметь пустое (неопределенное) значение;
- производный – представляет значение, производное от значения связанного с ним атрибута.

Уникальным идентификатором называется неизбыточный набор атрибутов, значения которых в совокупности являются *уникальными* для каждого экземпляра сущности.

Связь (Relationship) – поименованная ассоциация между двумя сущностями, значимая для рассматриваемой предметной области. Связь – это ассоциация между сущностями, при которой каждый экземпляр одной сущности ассоциирован с произвольным (в том числе нулевым) количеством экземпляров второй сущности, и наоборот.

Степенью связи называется количество сущностей, участвующих в связи. Связь степени 2 называется бинарной, степени N – N-арной. Связь, в которой одна и та же сущность участвует в разных ролях, называется рекурсивной, или унарной.

Мощность связи – максимальное число экземпляров сущности, которое может быть связано с одним экземпляром данной сущности. Мощность связи может быть равна 1, N (любое число) и может быть конкретным числом.

Класс принадлежности характеризует обязательность участия экземпляра сущности в связи: 0 (необязательное участие) или 1 (обязательное участие).

17. Методология IDEF1X: назначение, виды и уровни моделей, порядок построения

Цель моделирования данных состоит в обеспечении разработчика системы концептуальной схемой базы данных в форме одной модели или нескольких локальных моделей, которые относительно легко могут быть отображены в любую систему баз данных.

Наиболее распространенным средством моделирования данных (предметной области) является модель «сущность-связь» (ERM) (Автор Питер Чен, 1976 г.)

Методология IDEF1X

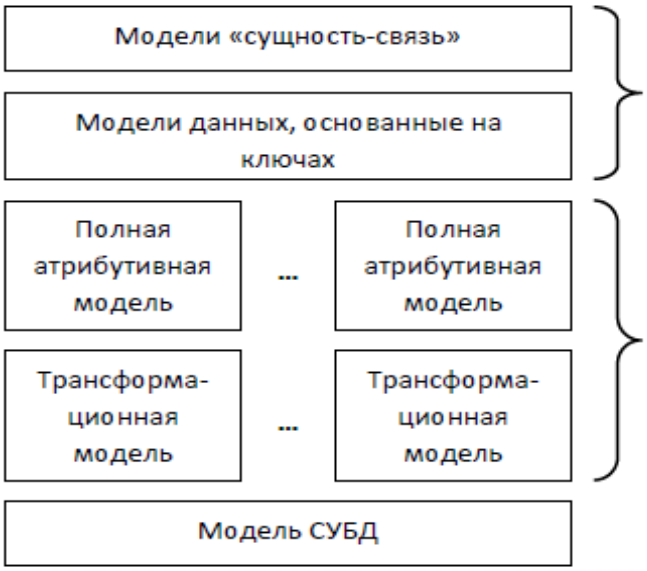
Сущность в методе IDEF1X является независимой от идентификаторов, или просто независимой, если каждый экземпляр сущности может быть однозначно идентифицирован без определения его отношений с другими сущностями.

Сущность называется зависимой от идентификаторов, или просто зависимой, если однозначная идентификация экземпляра сущности зависит от его отношения к другой сущности.

Связь

Мощности связей:

- каждый экземпляр сущности-родителя может иметь нуль, один или более одного связанного с ним экземпляра сущности-потомка;
- каждый экземпляр сущности-родителя должен иметь не менее одного связанного с ним экземпляра сущности-потомка;
- каждый экземпляр сущности-родителя должен иметь не более одного связанного с ним экземпляра сущности-потомка;
- каждый экземпляр сущности-родителя связан с некоторым фиксированным числом экземпляров сущности-потомка.
- Если экземпляр сущности-потомка однозначно определяется своей связью с сущностью-родителем, то связь называется идентифицирующей, в противном случае — неидентифицирующей.



Верхний уровень состоит из:

- Entity Relation Diagram (Диаграмма сущность-связь)
- Key-Based model (Модель данных, основанная на ключах).

Диаграмма сущность-связь определяет сущности и их отношения. Модель данных, основанная на ключах, дает более подробное представление данных. Она включает описание всех сущностей и первичных ключей, которые соответствуют предметной области.

Нижний уровень состоит из:

- Transformation Model (Трансформационная модель)
- Fully Attributed (Полная атрибутивная модель).

Трансформационная модель содержит всю информацию для реализации проекта, который может быть частью общей информационной системы и описывать предметную область. Трансформационная модель позволяет проектировщикам и администраторам БД представлять, какие объекты БД хранятся в словаре данных,

и проверить, насколько физическая модель данных удовлетворяет требованиям информационной системы.

18. Методология IDEF3: назначение. Единица работы, связи и их виды, соединения и их виды

Метод IDEF3 предназначен для таких моделей процессов, в которых важно понять последовательность выполнения действий и взаимозависимости между ними. Единица работы – действие, которое происходит в системе.

Модели IDEF3 могут использоваться для детализации функциональных блоков IDEF0, не имеющих диаграмм декомпозиции.

Как и в методе IDEFO, основной единицей модели IDEF3 является диаграмма. Другой важный компонент модели — *действие,* или в терминах IDEF3 ***«единица работы» (Unit of Work - UOW).*** Диаграммы IDEF3 отображают действие в виде прямоугольника. Действия именуются с использованием глаголов или отглагольных существительных, каждому из действий присваивается уникальный идентификационный номер. Этот номер не используется вновь даже в том случае, если в процессе построения модели действие удаляется. В диаграммах IDEF3 номер действия обычно предваряется номером его родителя (рисунок 6).

Рисунок 6 – Изображение и нумерация действия в диаграмме IDEF3

Таблица 1 – Типы связей IDEF3

Изображение	Название	Назначение
→	Временное предшествование (Temporal precedence)	Исходное действие должно завершиться, прежде чем конечное действие сможет начаться
→→	Объектный поток (Object flow)	Выход исходного действия является входом конечного действия (исходное действие должно завершиться, прежде чем конечное действие сможет начаться)
- - - - - →	Нечеткое отношение (Relationship)	Вид взаимодействия между исходным и конечным действиями задается аналитиком отдельно для каждого случая использования такого отношения

Типы соединений IDEF3

Указатели — это специальные символы, которые ссылаются на другие разделы описания процесса.

Они выносятся на диаграмму для привлечения внимания читателя к каким-либо важным аспектам модели.

Таблица 2 – Типы соединений

Графическое обозначение	Название	Вид	Правила инициализации
&	Соединение «И»	Разворачивающее	Каждое конечное действие обязательно инициируется
		Сворачивающее	Каждое исходное действие обязательно должно завершиться
X	Соединение «исключающее ИЛИ»	Разворачивающее	Одно и только одно конечное действие инициируется

		Сворачивающее	Одно и только одно исходное действие должно завершиться
О	Соединение «ИЛИ»	Разворачивающее	Одно или несколько конечных действий инициируются
		Сворачивающее	Одно или несколько исходных действий должны завершиться

19. Основные этапы проектирования программных систем и их содержание

Проектирование программного обеспечения - этап жизненного цикла программного обеспечения, во время которого исследуется структура и взаимосвязи элементов разрабатываемой системы. Результатом этого этапа является прежде всего документ (набор документов), содержащий(е) в себе достаточное количество информации для реализации системы.

Можно выделить 6 основных этапов проектирования ПС: *анализ требований, проектирование, кодирование, тестирование и отладка, внедрение, заключение.*

Анализ требований: в рамках этой стадии происходит максимально эффективное взаимодействие нуждающегося в программном решении клиента и сотрудников компании-разработчика, в ходе обсуждения деталей проекта помогающих более четко сформулировать предъявляемые к ПО требования. Результатом проведенного анализа становится формирование основного регламента, на который будет опираться исполнитель в своей работе — технического задания на разработку программного обеспечения. ТЗ должно полностью описывать поставленные перед разработчиком задачи и охарактеризовать конечную цель проекта в понимании заказчика.

Проектирование: Следующий ключевой этап в разработке программного обеспечения — стадия проектирования, то есть моделирования теоретической основы будущего продукта. В рамках данного этапа стороны должны осуществить: *оценку результатов проведенного первоначально анализа и выявленных ограничений; поиск критических участков проекта; формирование окончательной архитектуры создаваемой системы; анализ необходимости использования программных модулей или готовых решений сторонних разработчиков; проектирование основных элементов продукта — модели базы данных, процессов и кода; выбор среды программирование и инструментов разработки, утверждение интерфейса программы, включая элементы графического отображения данных; определение основных требований к безопасности разрабатываемого ПО.*

Кодирование: Следующим шагом становится непосредственная работа с кодом, опираясь на выбранный в процессе подготовки язык программирования. Кодирование может происходить параллельно со следующим этапом разработки — тестированием программного обеспечения, что помогает вносить изменения непосредственно по ходу написания кода. Уровень и эффективность взаимодействия всех элементов, задействованных для выполнения сформулированных задач компанией-разработчиком, на текущем этапе является самым важным — от слаженности действий программистов, тестировщиков и проектировщиков зависит качество реализации проекта.

Тестирование и отладка: после достижения задуманного программистами в написанном коде следуют не менее важные этапы разработки программного обеспечения, зачастую объединяемые в одну фазу — тестирование продукта и последующая отладка, позволяющая ликвидировать огрехи программирования и добиться конечной цели — полнофункциональной работы разработанной программы. Процесс тестирования позволяет смоделировать ситуации, при которых программный продукт перестает функционировать. Отдел отладки затем локализует и исправляет обнаруженные ошибки кода, «вылизывая» его до практически идеального состояния.

Внедрение: Процедура внедрения программного обеспечения в эксплуатацию является завершающей стадией разработки и нередко происходит совместно с отладкой системы. Как правило, ввод в эксплуатацию ПО осуществляется в три

этапа: *первоначальная загрузка данных; постепенное накопление информации; вывод созданного ПО на проектную мощность.* Ключевой целью поэтапного внедрения разработанной программы становится постепенное выявление не обнаруженных ранее ошибок и недочетов кода. В данном этапе разработки ПО можно выявить различные мелкие ошибки с несогласованностью данных в системе.

Заключение: Неотъемлемой частью завершающего этапа разработки программного обеспечения также является последующая техническая поддержка созданного продукта в процессе его эксплуатации на предприятии заказчика.

20. Объектно-ориентированный подход к разработке ПО: основные понятия и принципы

Это метод построения программ в виде множества взаимодействующих объектов. Базовыми понятиями объектно-ориентированного программирования являются объекты и классы. Объект определяется через его свойства: идентифицируемость (объект имеет имя), состояние, поведение.

Состояние объекта выражается всеми компонентами объекта (статическими) и текущими значениями этих компонентов (динамическими). Поведение объекта определяет, как объект изменяет свои состояния и взаимодействует с другими объектами, т.е. выражает динамику объекта и его реакцию на поступающие сообщения. Идентификация объекта–это свойство, которое позволяет отличить объект от других объектов того же или других класса.

Класс – это множество объектов, имеющих общую структуру и общее поведение. Базовыми правилами объектно-ориентированного программирования являются: определение классов, определение всех необходимых операций для каждого класса, обеспечение расширяемости (открытости) классов с использованием принципа наследования.

Объект - представляет собой конкретный опознаваемый предмет, единицу или сущность (реальную или абстрактную), имеющую четко определенное функциональное назначение в данной предметной области.

Свойства которыми может обладать объект могут быть трех видов.

- Атрибуты (attribute), такие как объем, положение или цвет, символизируют связи с другими объектами и состояние самого объекта.

- Процедуры или услуги, предоставляемые объектом, такие как перемещение или расширение. Их называют операциями (operation) или методами (method).

- Правила, которые устанавливают взаимосвязи свойств объекта или определяют условия его жизнеспособности. Их иногда называют инвариантами (invariant).

Строго говоря, методы являются функциями, которые реализуют операции, а операции — это абстрактные спецификации методов.

Объектно-ориентированное программирование (ООП) (Object-Oriented Programming) - это методология программирования, основанная на представлении программы в виде совокупности объектов, каждый из которых является экземпляром определенного класса, а классы образуют иерархию наследования.

К числу основополагающих понятий ООП обычно относят абстракцию данных, наследование, инкапсуляцию и полиморфизм. Абстракция- произвольное выражение языка программ-я,vявл отличным от идентификатора. Наследование- применимость всех или лишь некоторых св-в и/или методов базового (родительского) класса для всех классов, производных от него. Кроме того, сохранение св-в и/или методов базового класса д/обеспеч-ся и для всех конкретизаций(т.е. конкретных объектов) любого производного класса. Инкапсуляция - возможность доступа к объекту и манипулирования им исключ-но посредством предоставляемых именно этим объектом св-в и методов. Полиморфизм - возможность оперировать объектами, не обладая точным знанием их типов. Процесс создания объектно-ориентированного программного обеспечения включает три этапа:

объектно-ориентированный анализ(создание объектно-ориентированной модели предметной области приложения ПО; здесь объекты отражают реальные объекты-сущности, также определяются операции, выполняемые объектами);

объектно-ориентированное проектирование(разработка объектно-ориентированной модели ПО (системной архитектуры) с учетом системных

требований; в объектно-ориентированной модели определение всех объектов подчинено решению конкретной задачи);

объектно-ориентированное программирование(реализация архитектуры (модели) программы с помощью объектно-ориентированного языка программирования).

Этапы могут пересекаться, но на каждом этапе применяется одна и та же система нотации. Переход на следующий этап приводит к усовершенствованию результатов предыдущего этапа путем более детального описания определенных ранее классов объектов и определения новых классов. Так как данные скрыты внутри объектов, детальные решения о представлении данных обычно откладываются до этапа реализации ПО.

Объектно-ориентированное проектирование представляет собой стратегию, в рамках которой разработчики программной системы вместо операций и функций мыслят в понятиях объектов. Программная система состоит из взаимодействующих объектов, которые имеют собственное локальное состояние и могут выполнять определенный набор операций в зависимости от состояния объекта (рис. 2.12).

21. Язык UML: причины появления и история развития языка, структура языка

Для создания моделей анализа и проектирования объектно-ориентированных программных систем используют языки визуального моделирования. Неразбериха и разногласия, возникшие вследствие большого количества языков, привели к идее создания унифицированного (единого) языка визуального моделирования объектно-ориентированных программных систем. Таким языком и стал унифицированный язык моделирования (Unified Modeling Language – UML).

Работа над созданием нового языка началась в октябре 1994 г. В январе 1997 г. был издан документ с описанием языка UML версии 1.0, в ноябре 1997 г. выходит версия UML 1.1, а в июне 1999 г. – UML 1.3. После выхода очередной версии 1.4 языка UML в сентябре 2001 г. начинается официальная разработка стандарта языка UML 2.0. Хотя затем в марте 2003 г. выходит версия 1.5. Окончательный же стандарт языка UML 2.0 был издан только в августе 2005 г. В настоящее время используется версия 2.1 вышедшая в 2006 г.

С самой общей точки зрения описание языка UML состоит из двух взаимодействующих частей: семантики и нотации. Семантика– система правил и соглашений, определяющих толкование и придание смысла конструкциям некоторого языка. Графическая нотация– система условных обозначений, принятая в некотором языке для изображения и визуализации модели. В рамках языка UML семантика и нотация тесно взаимосвязаны: семантика языка описывается с использованием некоторого подмножества нотации, а, в свою очередь, нотация описывает соответствие или отображение графической нотации в базовые понятия семантики.

Назначение языка UML 2

• Предоставить в распоряжение всех пользователей легко воспринимаемую и выразительную нотацию для визуального моделирования, специально предназначенную для разработки и документирования моделей систем самого различного назначения.

• Снабдить исходные понятия языка UML 2 возможностью расширения и специализации для более точного представления моделей систем в конкретной предметной области.

• Описание языка UML 2 должно поддерживать такую спецификацию моделей, которая не зависит от конкретных языков программирования и инструментальных средств проектирования программных систем.

• Описание языка UML 2 должно включать в себя семантический базис для понимания общих особенностей ООАП.

• Поощрять развитие рынка программных инструментальных средств, поддерживающих методологию ООАП и MDA.

• Способствовать распространению объектных технологий и соответствующих понятий ООАП.

• Интегрировать в себя новейшие и наилучшие достижения практики ООАП.

Общая структура UML 2

Семантика языка определяется для двух категорий объектных моделей:

Структурные модели – модели, предназначенные для описания статической структуры сущностей или элементов некоторой системы, включая их классы, интерфейсы, атрибуты и отношения.

Модели поведения – модели, предназначенные для описания процесса функционирования элементов системы, включая их методы и взаимодействие

между ними, а также процесс изменения состояний отдельных элементов и системы в целом.

Язык UML использует более 200 элементов для описания моделей, число элементов возрастает с появлением новых версий языка. Все элементы языка UML иерархически сгруппированы в логические блоки – так называемые пакеты. Пакет, включающий другие пакеты, называется метапакетом или контейнером. Пакет, включенный в контейнер, называется подчиненным пакетом, или более коротко - подпакетом. Каждый пакет владеет всеми включенными в него элементами, при этом каждый элемент может принадлежать только одному пакету.

22. Канонические диаграммы языка UML: их виды и типы, рекомендации построения

Диаграмма (diagram) — графическое представление совокупности элементов *модели* в форме связного графа, вершинам и ребрам (дугам) которого приписывается определенная семантика. Нотация ***канонических диаграмм*** - основное средство разработки*моделей* на языке UML.

Диаграммы языка UML 2

Диаграммы структуры:

— ***диаграмма классов*** – диаграмма, которая служит для представления совокупности декларативных или статических элементов модели, таких как классы, типы, а также разнообразных отношений между ними;

— ***диаграмма композитной структуры*** – диаграмма, которая изображает внутреннюю структуру классификаторов, таких как класс, компонент или кооперация, включая точки взаимодействия классификатора с другими частями системы;

— ***диаграмма пакетов*** – диаграмма логического уровня, которая служит для представления организации элементов модели по пакетам, а также зависимостей между пакетами, включая импорт и слияние пакетов;

— ***диаграмма объектов*** – диаграмма, которая служит для представления объектов и отношений между ними в конкретный момент времени (может рассматриваться как специальный случай диаграммы коммуникации);

— ***диаграмма компонентов*** – диаграмма физического уровня, которая служит для представления программных компонентов и зависимостей между ними;

— ***диаграмма развертывания*** – диаграмма физического уровня, которая служит для представления архитектуры распределенных программных систем.

Диаграммы поведения:

— ***диаграмма вариантов использования*** – диаграмма концептуального уровня, которая служит для представления актеров и вариантов использования системы, а также отношений между ними;

— ***диаграмма деятельности*** – диаграмма, которая изображает поведение объекта или системы с использованием моделей потока данных и потока управления;

— ***диаграмма конечного автомата*** – диаграмма, которая служит для представления дискретного поведения, моделируемого посредством переходов в системах с конечным числом состояний.

Диаграммы взаимодействия:

— ***диаграмма последовательности*** – диаграмма, которая служит для представления взаимодействия элементов модели в форме последовательности сообщений и соответствующих событий на линиях жизни объектов;

— ***диаграмма коммуникации*** – диаграмма, которая служит для представления взаимодействия между линиями жизни объектов в форме элементов внутренней структуры системы и передаваемых между ними сообщений;

— ***диаграмма обзора взаимодействия*** – вариант диаграммы деятельности, которая служит для представления только взаимодействия потока управления в некоторой агрегированной форме;

— ***временная диаграмма*** – диаграмма взаимодействия, которая служит для представления изменения состояния или условий линии жизни отдельных экземпляров классификаторов во времени

Три типа визуальных графических изображения:

— Геометрические фигуры на плоскости, играющие роль вершин графов соответствующих диаграмм.

— Графические пути, которые представляются различными линиями на плоскости.

— Специальные графические символы, изображаемые вблизи от тех или иных визуальных элементов диаграмм и имеющие характер дополнительной спецификации.

Рекомендации при построении диаграмм языка UML 2

1. Каждая диаграмма должна служить законченным представлением соответствующего фрагмента моделируемой предметной области.
2. Все сущности на диаграмме модели должны быть одного концептуального уровня.
3. Вся информация о сущностях должна быть явно представлена диаграммах.
4. Диаграммы не должны содержать противоречивой информации.
5. Диаграммы не следует перегружать текстовой информацией.
6. Каждая диаграмма должна быть самодостаточной для правильной интерпретации всех ее элементов и понимания семантики всех используемых графических символов.
7. Количество диаграмм различных типов для модели конкретного приложения не является строго фиксированным.
8. Любая из моделей должна содержать только те элементы, которые определены в нотации языка UML 2.

Типы диаграмм UML 2, указываемые в заголовках фреймов:

- **activity (act)** – для фреймов диаграммы деятельности;
- **class (cs)** – для фреймов диаграммы классов;
- **component (cmp)** – для фреймов диаграммы компонентов;
- **interaction (sd)** – для фреймов диаграммы взаимодействия;
- **package (pkg)** – для фреймов диаграммы пакетов;
- **state machine (stm)** – для фреймов диаграммы конечных автоматов;
- **use case (uc)** – для фреймов диаграммы вариантов использования.

23. Диаграмма вариантов использования: назначение, принципы построения

Диаграмма вариантов использования – диаграмма, на которой изображаются варианты использования проектируемой системы, заключенные в границу субъекта, и внешние актеры, а также определенные отношения между актерами и вариантами использования.

Назначение диаграммы вариантов использования:

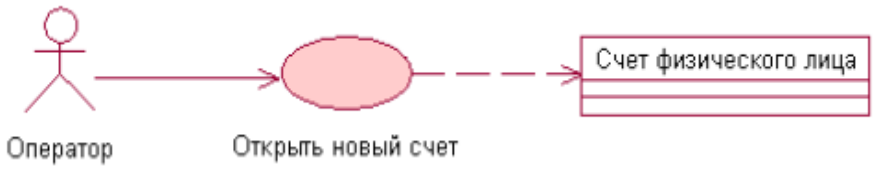
• определить общие границы функциональности проектируемой системы в контексте моделируемой предметной области;

• специфицировать требования к функциональному поведению проектируемой системы в форме вариантов использования;

• разработать исходную концептуальную модель системы для ее последующей детализации в форме логических и физических моделей;

• подготовить исходную документацию для взаимодействия разработчиков системы с ее заказчиками и пользователями.

Субъектом в контексте языка UML 2 называется любой элемент модели, который обладает функциональным поведением. Рис.1. Диаграмма вариантов использования



Основные графические элементы диаграммы вариантов использования:

• вариант использования;

• актер;

• примечание.

Вариант использования представляет собой общую спецификацию совокупности выполняемых системой действий с целью представления некоторого наблюдаемого результата, который имеет значение для одного или нескольких актеров.

Актер представляет собой любую внешнюю по отношению к проектируемой системе сущность, которая взаимодействует с системой и использует ее функциональные возможности для достижения определенных целей или решения частных задач.

Комментарий в языке UML 2 предназначен для включения в модель произвольной текстовой информации в форме примечания, которое может быть присоединено к одному или нескольким элементам разрабатываемой модели.

На диаграммах вариантов использования могут применяться следующие отношения:

• ассоциации; включение; расширение; обобщение.

Применительно к диаграммам вариантов использования отношение ассоциации может служить только для обозначения взаимодействия актера с вариантом использования. (прямая)

Отношение включения специфицирует тот факт, что некоторый вариант использования содержит поведение, определенное в другом варианте использования. (include)

Отношение расширения определяет взаимосвязь одного варианта использования с некоторым другим вариантом использования, функциональность или поведение которого задействуется первым не всегда, а только при выполнении некоторых дополнительных условий. (extend)

Отношение обобщения предназначено для спецификации того факта, что один элемент модели является специальным или частным случаем другого элемента модели (стрелка)

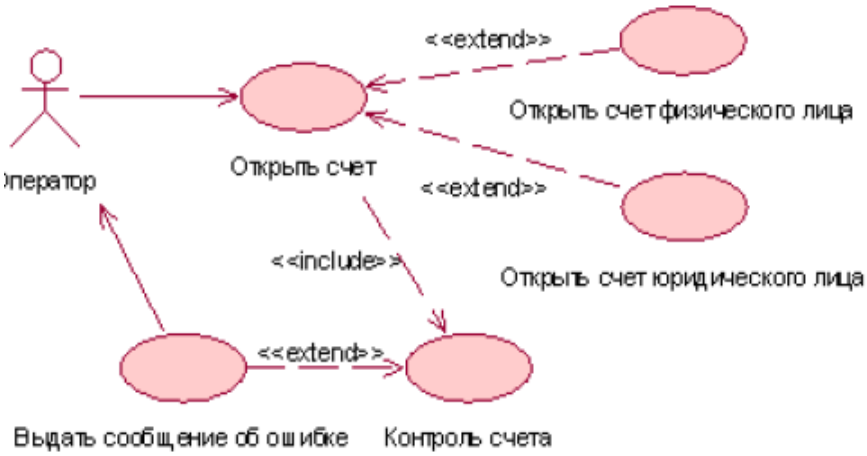


Рис. 2а. Пример диаграммы ВИ Актер «Оператор» активизирует выполнение ВИ «Открыть счет». В соответствии с заданным оператором типом счета выполняется либо ВИ «Открыть счет физического лица» либо «Открыть счет юридического лица», являющиеся расширениями первого. Открытие счета включает его контроль и при обнаружении ошибки – выдачу сообщения Оператору.

24. Оценка трудоемкости разработки проекта на основе вариантов использования.

Все действующие лица системы делятся на три типа: простые, средние и сложные. Простое действующее лицо представляет внешнюю систему с четко определенным программным интерфейсом (API). Весовой коэффициент простого действующего лица равен 1.

Среднее действующее лицо представляет либо внешнюю систему, взаимодействующую с данной системой посредством протокола наподобие TCP/IP, либо личность, пользующуюся текстовым интерфейсом (например, ASCII-терминалом). Весовой коэффициент среднего действующего лица равен 2.

Сложное действующее лицо представляет личность, пользующуюся графическим интерфейсом (GUI). Весовой коэффициент сложного действующего лица равен 3.

Подсчитанное количество действующих лиц каждого типа умножается на соответствующий весовой коэффициент, затем вычисляется общий весовой показатель A

Все варианты использования делятся на три типа: простые, средние и сложные, в зависимости от количества транзакций в потоках событий (основных и альтернативных).

В данном случае под транзакцией понимается атомарная последовательность действий, которая выполняется полностью или отменяется.

Подсчитанное количество вариантов использования каждого типа умножается на соответствующий весовой коэффициент, затем вычисляется общий весовой показатель UC.

Для простого типа варианта использования (3 или менее транзакций) весовой коэффициент принимается равным 5, для среднего (от 4 до 7 транзакций) – 10, для сложного (более 7 транзакций) – 15.

Общий весовой показатель равен:

$$UC = 21 * \ = 105$$

В результате получаем показатель UUCP:

$$UUCP = A + UC = 3 + 105 = 108.$$

Техническая сложность проекта (TCF — technical complexity factor) вычисляется с учетом показателей технической сложности. Каждому показателю присваивается значение Ti в диапазоне от 0 до 5 (0 означает отсутствие значимости показателя для данного проекта, 5 — высокую значимость). Значение TCF вычисляется по следующей формуле:

$$TCF = 0.6 + (0.01 * (\sum Ti\ Beci)).$$

Уровень квалификации разработчиков (EF — environmental factor) вычисляется с учетом показателей F1 – F8 по формуле:

$$EF = 1.4 + (-0.03 * (\sum Fi\ Beci)).$$

Вычисление трудозатрат проекта:

$$USE\ Case\ Points\ (UCP) = UUCP * TCF * ECF$$

Расчет трудозатрат проекта представлен ниже.

$$Estimated\ Work\ Effort(hours) = 10*UCP$$

$$Estimated\ Cost = EWE * Default\ hourly\ Rate$$

25. Диаграмма классов: назначение, классы, обозначение классов, их атрибутов и операций

Диаграмма классов – диаграмма, предназначенная для представления модели статической структуры программной системы в терминологии классов объектно-ориентированного программирования.

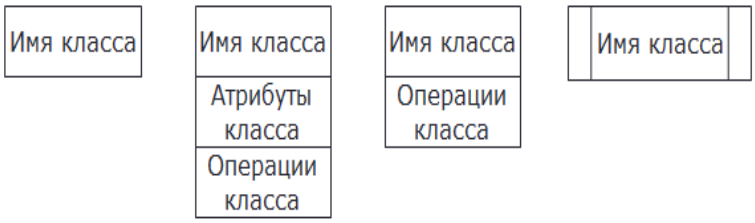
Разработка диаграммы классов преследует следующие **цели**:

- определить сущности предметной области и представить их в форме классов с соответствующими атрибутами и операциями;
- определить взаимосвязи между сущностями предметной области и представить их в форме типовых отношений между классами;
- разработать исходную логическую модель программной системы для ее последующей реализации в форме физических моделей;
- подготовить документацию для последующей разработки программного кода.

Класс – элемент модели, который описывает множество объектов, имеющих одинаковые спецификации характеристик, ограничений и семантики.

Активный класс – класс, каждый экземпляр которого имеет свою собственную нить управления.

Пассивный класс – класс, каждый экземпляр которого выполняется в контексте некоторого другого объекта.



Атрибут класса служит для представления отдельной структурной характеристики или свойства, которое является общим для всех объектов данного класса.

Виды видимости:

‘-’ – закрытый,
‘#’ – защищенный,
‘+’ – общедоступный,
‘~’ – пакетный.

Примеры:

+ имяСотрудника: String {readOnly}
# возрастСотрудника: Integer
+ номерТелефона: Integer [*] {unique}
- заработнаяПлата: Currency = $500

Операция класса служит для представления отдельной характеристики поведения, которая является общей для всех объектов данного класса.

Предусловие для операции определяет условие, которое должно быть истинным, когда эта операция вызывается.

Постусловие для операции определяет условие, которое должно быть истинным, когда вызов операции успешно завершился, в предположении, что все предусловия удовлетворены.

Примеры:

+ добавить(in номерТелефона: Integer [*] {unique})
- изменить (in заработнаяПлата: Currency)
+ создать () : Boolean

26. Диаграмма классов: назначение, отношения между классами и их применение

Диаграмма классов – диаграмма, предназначенная для представления модели статической структуры программной системы в терминологии классов объектно-ориентированного программирования.

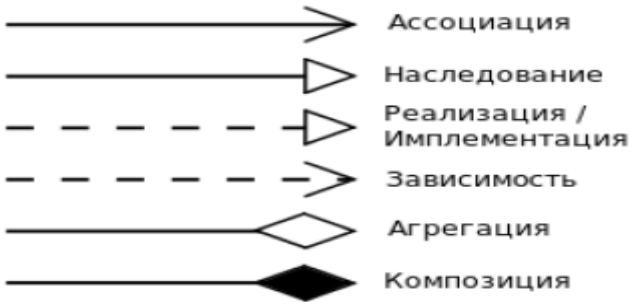
Разработка диаграммы классов преследует следующие **цели**:

• определить сущности предметной области и представить их в форме классов с соответствующими атрибутами и операциями;

• определить взаимосвязи между сущностями предметной области и представить их в форме типовых отношений между классами;

• разработать исходную логическую модель программной системы для ее последующей реализации в форме физических моделей;

• подготовить документацию для последующей разработки программного кода.



На диаграмме классов могут присутствовать следующие отношения между классами:

• ассоциация; зависимость; агрегация; композиция;

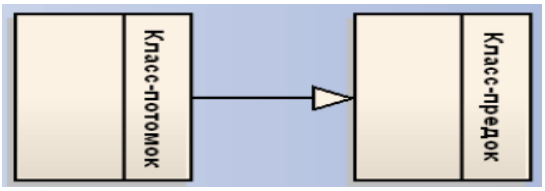
• обобщение; реализация.

Ассоциация – произвольное или взаимосвязь между классами.

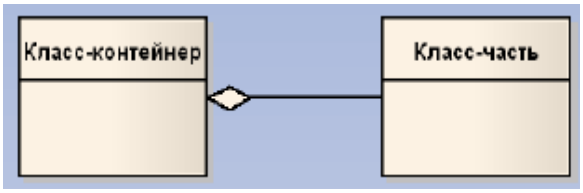
Имя конца ассоциации специфицирует роль, которую играет класс расположенный на соответствующем конце ассоциации.

Видимость конца ассоциации специфицирует возможность доступа к соответствующему концу ассоциации с других ее концов.

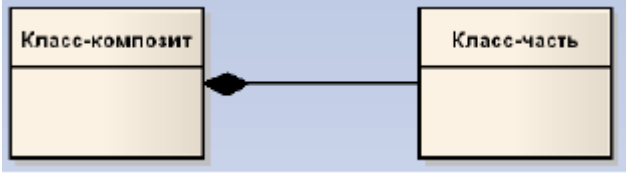
Кратность конца ассоциации специфицирует возможное количество экземпляров соответствующего класса, которое может соотносится с одним экземпляром класса на другом конце ассоциации.



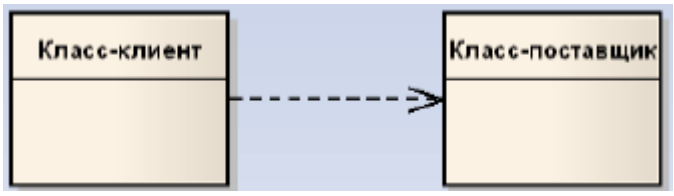
Обобщение – таксономическое отношение между более общим классификатором (родителем или предком) и более специальным классификатором (дочерним или потомком)



Агрегация – направленное отношение между двумя классами, предназначенное для представления ситуации, когда один из классов представляет собой некоторую сущность, которая включает в себя в качестве составных частей другие сущности.

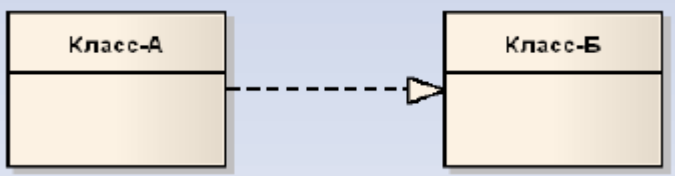


Композиция (или композитная агрегация) предназначена для спецификации более сильной формы отношения «часть-целое», при которой с уничтожением объекта класса-контейнера уничтожаются и все объекты, являющиеся его составными частями.



Зависимость – отношение, предназначенное для описания ситуации, когда отдельному элементу или множеству

элементов модели требуются другие элементы модели для своей спецификации или реализации.



Реализация – специализированное отношение зависимости между двумя элементами модели, один из которых представляет некоторую спецификацию (поставщик), а другой представляет его реализацию (клиент).

27. Диаграмма композитной структуры: композитные классы и их части, принципы построения

Композитная структура

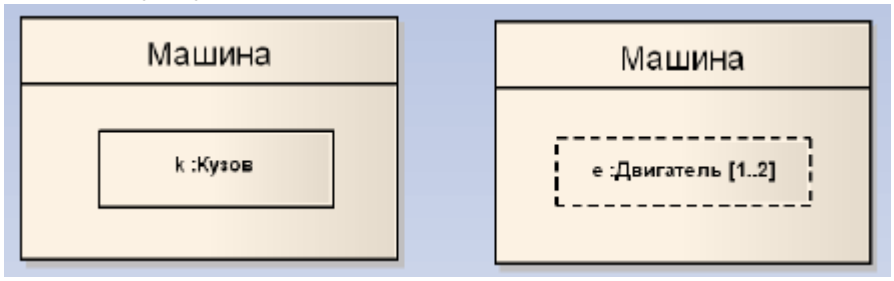
Внутренняя структура – это структура взаимодействующих элементов модели, которые создаются в экземпляре содержащего их классификатора.

Свойство – множество экземпляров, которые являются собственностью содержащего их экземпляра классификатора.

Для представления внутренней структуры классификаторов в общем случае предназначена диаграмма композитной структуры. При этом классификатор, имеющий некоторую внутреннюю структуру, называют также композитным, а его внутреннюю структуру – композитной структурой.

Композитный класс

Часть – свойство, которое является элементом внутренней структуры композитного классификатора, в частном случае – класса.



Соединитель – отношение, которое обеспечивает взаимосвязь или коммуникацию между двумя или более экземплярами классификаторов, в частном случае – экземплярами классов.

На диаграммах композитной структуры могут быть также изображены спецификации экземпляров класса. При этом для них должны быть указаны имена ролей, которые они играют в композитном классе.

Строка имени роли в спецификации экземпляра должна удовлетворять следующему синтаксису БНФ:

<имя-роли-в-спецификации-экземпляра> ::=

{<имя>[‘/’ <имя-роли>] | ‘/’ <имя-роли>}

[‘:’ <имя-классификатора> [‘,’ <имя-классификатора>]*]

Порт – свойство классификатора, которое специфицирует отдельную точку взаимодействия между этим классификатором и его окружением или между классификатором и его внутренними частями. С портами могут быть ассоциированы интерфейсы для спецификации характера взаимодействия, которое может осуществляться через этот порт. В этом случае все взаимодействия через этот порт специфицируются в форме и предоставляемых и требуемых интерфейсов.

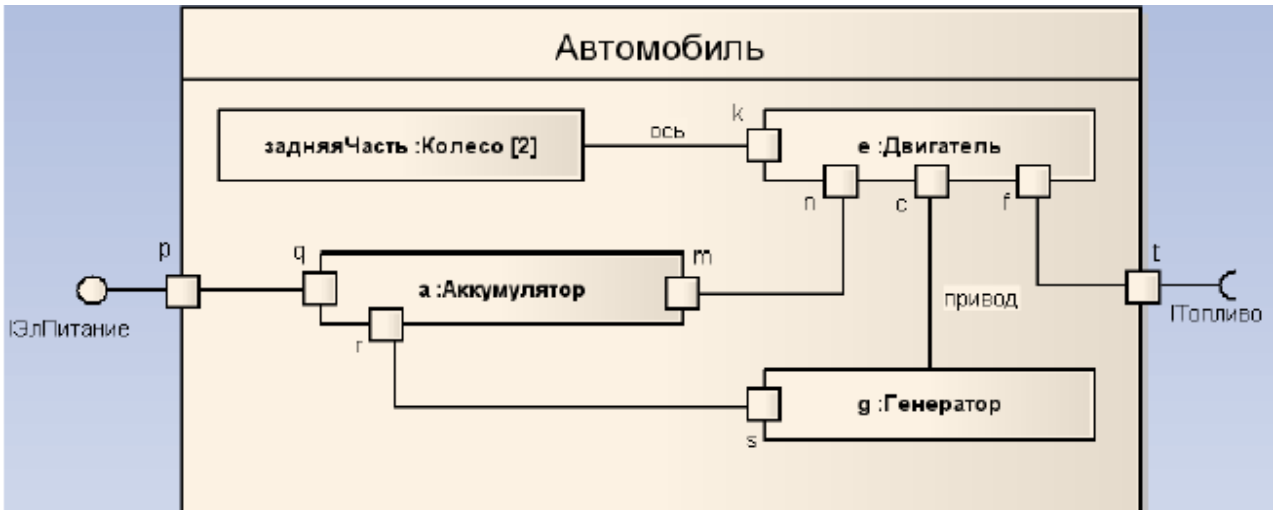
Предоставляемый интерфейс порта характеризует запросы, которые могут быть переданы через этот порт классу от его окружения.

Требуемый интерфейс порта характеризует запросы, которое могут быть переданы от класса к его окружению через этот порт.

Предоставляемый интерфейс порта характеризует запросы, которые могут быть переданы через этот порт классу от его окружения.

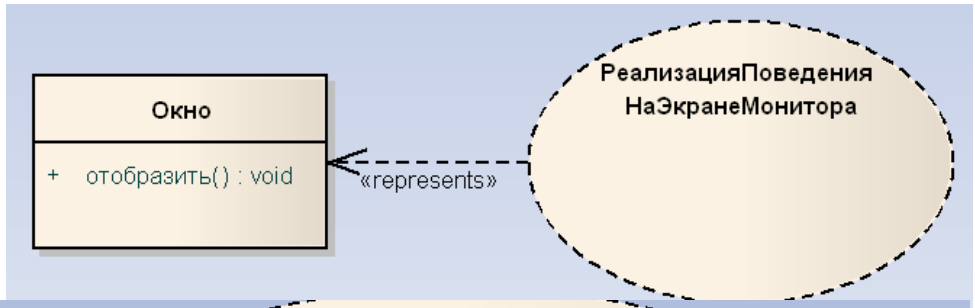
Требуемый интерфейс порта характеризует запросы, которое могут быть переданы от класса к его окружению через этот порт.

Пример композитной структуры класса Автомобиль



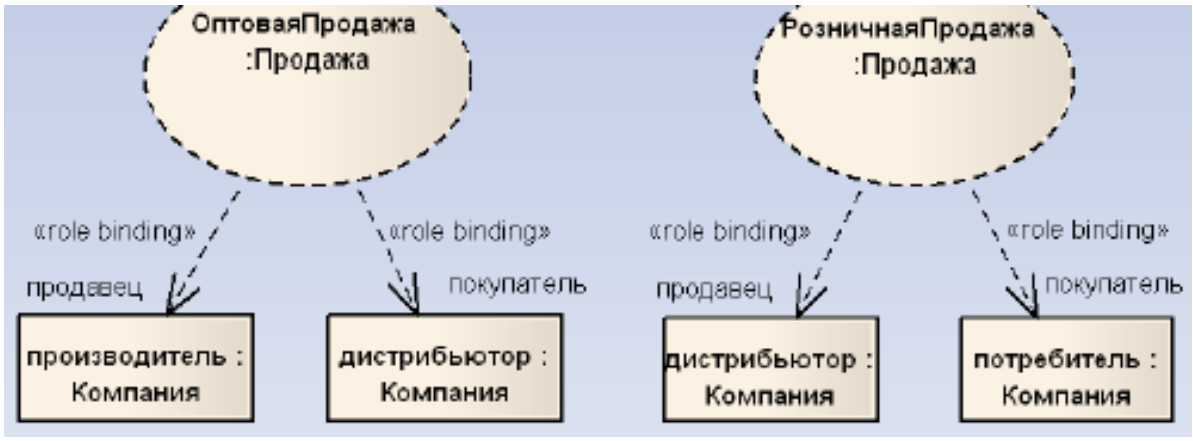
Кооперация – классификатор, предназначенный для описания некоторой структуры элементов или ролей, которые выполняют специализированные функции и совместно производят желаемую функциональность

Роль кооперации специфицирует требуемое множество характеристик, которые должен иметь соответствующий участник кооперации.



Применение кооперации представляет собой описание реализации кооперации в форме множества взаимодействующих элементов посредством связывания этих элементов с ролями данной кооперации.

Пример применения кооперации:



28. Диаграмма пакетов: назначение, пакеты и отношения между ними

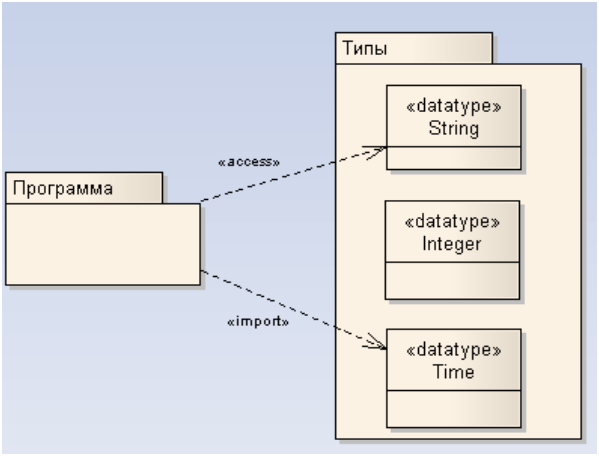
Диаграмма пакетов предназначена для представления размещения элементов модели в пакетах и спецификации зависимостей между пакетами и их элементами.

Пакет – элемент модели, используемый для группировки других элементов модели.

Элементы модели, которые входят в состав некоторого пакета, называются **членами этого пакета**.

Элементы модели, которые входят в пространство имен некоторого пакета, называются **элементами этого пакета**.

На диаграммах пакетов одни пакеты могут быть вложены в другие пакеты. В этом случае вложенный пакет называется подпакетом, а все элементы подпакета будут также принадлежать любому пакету, для которого рассматриваемый подпакет является вложенным.

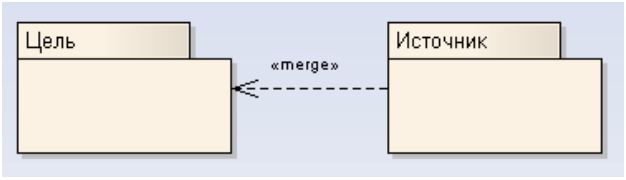


Импорт пакета – направленное отношение между пакетами, при котором члены одного пакета могут быть добавлены в пространство имен другого пакета.

Два вида импорта:

- <<import>> - для общедоступного импорта пакета;
- <<access>> - для закрытого импорта пакета

Импорт элемента – направленное отношение между импортирующим пространством имен и отдельным элементом пакета, которое позволяет ссылаться на этот элемент с использованием неквалифицированного имени.



Слияние пакетов – направленное отношение между двумя пакетами, один из которых расширяет свое содержание посредством добавления содержимого другого пакета

Сливаемый пакет – первый операнд слияния, т.е. пакет, который сливается в принимающий пакет и который является целью стрелки слияния на диаграммах.

Принимающий пакет – второй операнд слияния, т.е. пакет, который концептуально содержит результаты слияния и который является источником стрелки слияния на диаграммах.

Результирующий пакет – это пакет, который концептуально результаты слияния.

Сливаемый элемент – элемент модели, который находится в сливаемом пакете.

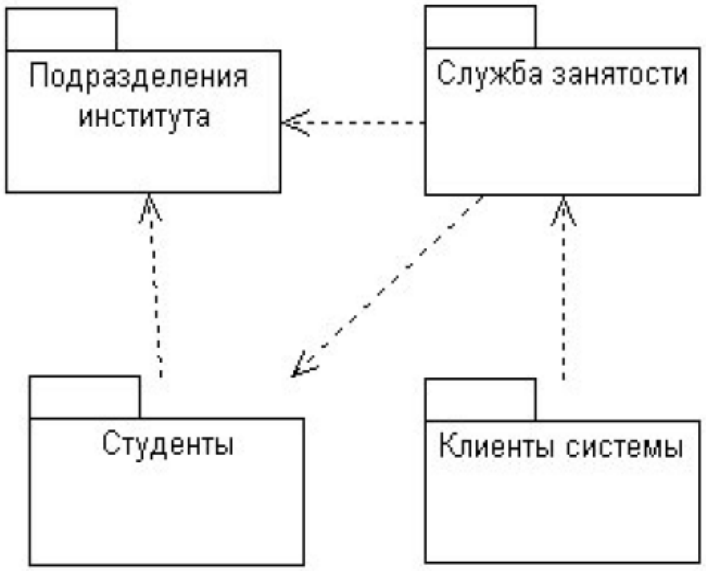
Принимающий элемент – элемент модели, который находится в принимающем пакете.

Результирующий элемент – элемент модели в результирующем пакете после выполнения слияния.

Тип элемента – любой допустимый тип элемента в языке UML 2, например, тип параметра или атрибута.

Метатип элемента – является типом MOF элемента модели, например, Классификатор, Ассоциация, Характеристика.

Диаграмма пакетов подсистемы «Служба занятости в рамках вуза» системы «Дистанционное обучение».


Подразделения института
Служба занятости
Студенты
Клиенты системы

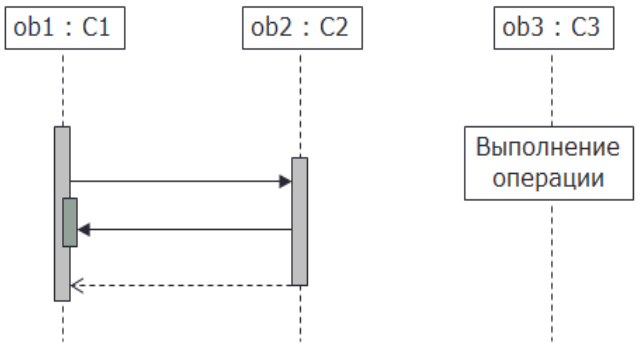
29. Диаграмма последовательности: назначение, линии жизни

Диаграмма последовательности – диаграмма, предназначенная для представления взаимодействия между элементами модели программной системы в терминологии линий жизни и сообщений между ними.

Графически диаграмма последовательности имеет **два измерения**:

- первое – слева направо в виде вертикальных линий каждая из которых соответствует линии жизни отдельного участника взаимодействия;
- второе – вертикальная временная ось, направленная сверху вниз.

Разработка диаграмм последовательностей осуществляется в ходе фазы детального проектирования, когда должна быть установлена точная коммуникация внутри процесса в соответствии с формальными протоколами.



В дальнейшем эта диаграмма может быть использована для тестирования приложения или его отдельных модулей.

Линия жизни представляет одного индивидуального участника взаимодействия или отдельную взаимодействующую сущность.

Сообщение – элемент модели, предназначенный для представления отдельной коммуникации между линиями жизни некоторого взаимодействия.

Тип сообщения представляет собой тип перечисления, который идентифицирует характер коммуникации, лежащей в основе генерации данного сообщения.

Тип сообщения является перечислением следующих значений:

'synchCall' – синхронное сообщение ——→

'asynchCall' – асинхронное сообщение ——→

'asynchSignal' – асинхронный сигнал ——→

ответное сообщение (reply) - - - - - - - →

сообщение создания объекта (object creation) - - - - - - - →

Виды параллельности вызова операции:

♦ 'sequential' – вызов последовательной операции, для которой не определен механизм управления параллельностью;

♦ 'guarded' – вызов охраняемой операции, для которой могут происходить одновременно несколько вызовов операций у одной линии жизни, но допускается к выполнению только один вызов;

♦ 'concurrent' – вызов параллельной операции, для которой могут происходить одновременно несколько вызовов операций у одной линии жизни, и все они могут быть обработаны параллельно.

Вид сообщения представляет собой тип перечисления, который идентифицирует тип сообщения.

Вид сообщения является перечислением следующих значений:

- 'complete' – полное сообщение, для которого существует событие передачи и приема.
- 'lost' – потерянное сообщение, для которого существует событие передачи и отсутствует событие приема.

- ‘found’ – найденное сообщение, для которого существует событие приема и отсутствует событие передачи.
- ‘unknown’ – неизвестное сообщение, для которого отсутствуют событие передачи и событие приема.

30. Диаграмма деятельности: назначение, понятие, семантика и обозначение деятельности, действия и дуг

Диаграммы деятельности используются для моделирования поведения, которое характеризуется некоторой деятельностью в форме последовательности действий, которые выполняются различными элементами, входящими в состав системы.

Деятельность – спецификация параметризированного поведения в форме координируемой последовательности подчиненных единиц, индивидуальными элементами которых являются действия.

Действие представляет собой элементарную единицу спецификации поведения, которая не может быть далее декомпозирована в форме деятельности.

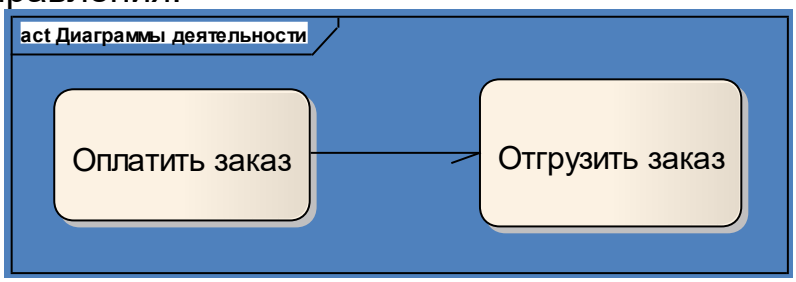
Узел деятельности является абстрактным классом для отдельных точек в потоке деятельности, соединенных дугами.

На диаграмме узлы деятельности могут быть связаны между собой при помощи дуг деятельности. Дуга деятельности является абстрактным классом для направленных соединений между двумя узлами деятельности.

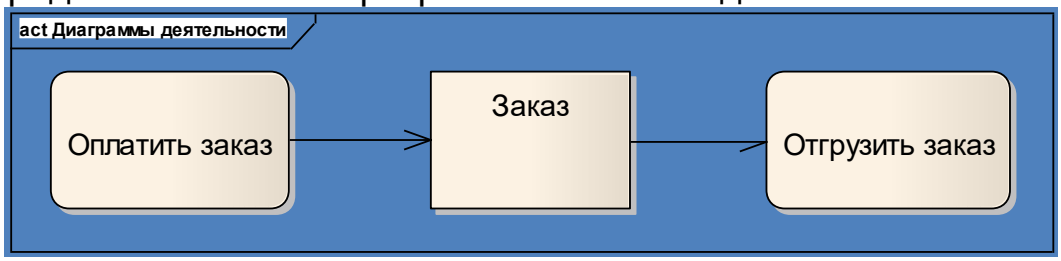
Две разновидности дуг деятельности:

- дуги потока управления,
- дуги потока данных.

Поток управления представляется в форме дуги деятельности, которая связывает между собой два узла деятельности и по которой передаются только маркеры управления.



Поток объектов представляется в форме дуги деятельности, по которой передаются только маркеры объектов или данных.



Семантика деятельности

Маркер – элемент модели, предназначенный для представления некоторого объекта, данных или управления и существующий на диаграмме деятельности в отдельном узле.

Особенности маркеров:

1. Каждый маркер отличается от любого другого, даже если он содержит то же значение, что и другой.
2. Маркеры могут протекать вдоль дуги, правила следования которых зависят от вида дуги и характеристик ее источника и цели.
3. Маркер перемещается по дуге, если он одновременно удовлетворяет правилам для узла цели, дуги и узла источника.
4. В общем случае вдоль дуги может одновременно следовать любое количество маркеров, индивидуально или в группах, в одно или разное время.

5. Если указано значение веса дуги, то оно определяет минимальное количество маркеров, которые могут последовать по дуге за один раз.
6. Если источник предлагает некоторое количество маркеров, то все эти маркеры предлагаются для цели одновременно.
7. Если количество маркеров, которое источник предлагает цели меньше веса дуги, то все маркеры совсем не предлагаются.
8. Неограниченный вес означает, что все маркеры в источнике могут быть предложены цели.
9. Несколько маркеров, предлагаемых некоторой дуге одновременно, являются такими же, как если бы они предлагались по одному в некоторый момент времени.

Свойства деятельности:

1. Деятельность может иметь несколько маркеров, которые втекают в нее или вытекают из нее в любой момент времени.
2. Во время выполнения деятельность имеет доступ к атрибутам и операциям объекта своего контекста и любых объектов транзитивно связанных с объектом контекста.
3. Деятельности могут быть анонимными, если они не назначены никакому классификатору.
4. Маркеры деятельности могут достигать узких мест и ожидать прохождения других маркеров перед ними, чтобы самим двигаться дальше.
5. Маркеры могут настигать друг друга во время выполнения вызываемого поведения и могут прерывать деятельность с помощью специальной конструкции – окончание деятельности.
6. Одна деятельность может вызывать другую деятельность.

Семантика действия

Правила выполнения действия:

1. Выполнение действия становится возможным, когда удовлетворены предварительное условие для его потоков управления и объектов.
2. Выполнение действия поглощает входные маркеры управления и маркеры объектов и удаляет их из источников дуг управления и из входных контактов.
3. Действие порождает выполнение до тех пор, пока оно не будет завершено.
4. После завершения действия оно предлагает маркеры объектов во все его выходные контакты, а маркеры управления во все выходящие из него дуги управления, и на этом формально оно заканчивается.
5. После окончания выполнения действия с помощью некоторой реализации должны быть восстановлены его ресурсы.

31. Диаграмма деятельности: узлы управления, их виды и применение

Диаграмма деятельности — UML-диаграмма, на которой показано разложение некоторой деятельности на её составные части. Под деятельностью понимается спецификация исполняемого поведения в виде координированного последовательного и параллельного выполнения подчинённых элементов — вложенных видов деятельности и отдельных действий, соединённых между собой потоками, которые идут от выходов одного узла ко входам другого.

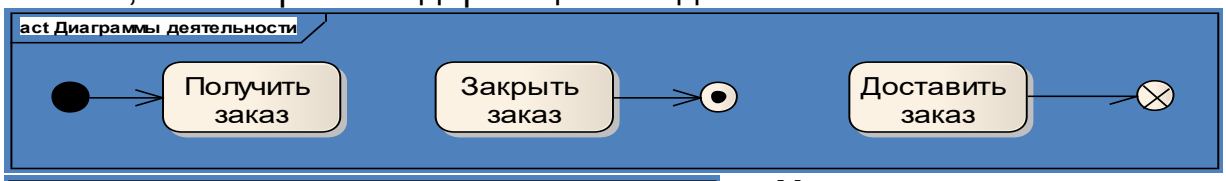
Узел управления является абстрактным узлом деятельности, который предназначен для координации потоков деятельности.

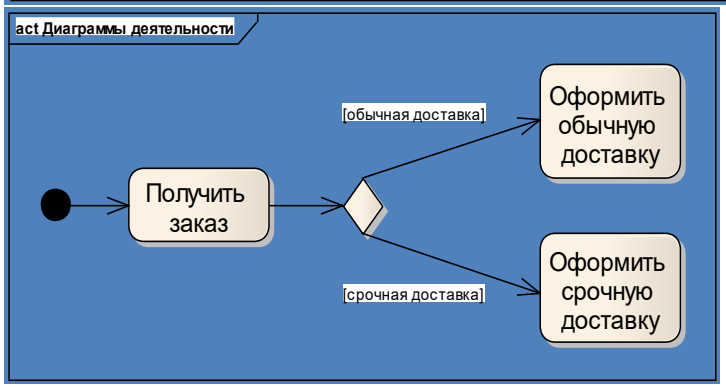
Существует несколько разновидностей конкретных узлов управления:

- начальный узел,
- финальный узел и его прямые потомки,
- узел разделения,
- узел соединения,
- узел решения,
- узел слияния.

Начальный узел является узлом управления, в котором начинается поток при вызове деятельности.

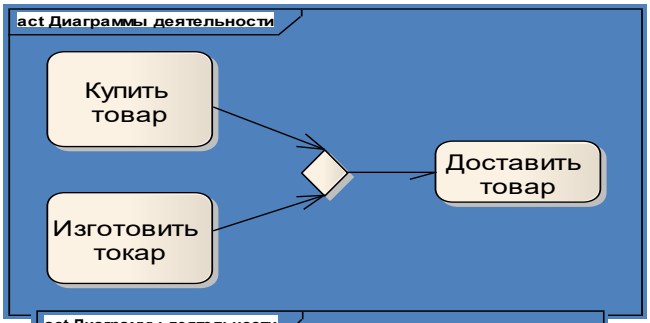
Узел финала деятельности является узлом управления, который прекращает или останавливает все потоки в деятельности. Узел финала потока является финальным узлом который завершает отдельный поток управления или поток объектов, не завершая содержащей его деятельности.





Узел решения является узлом управления, который выбирает между выходящими потоками.

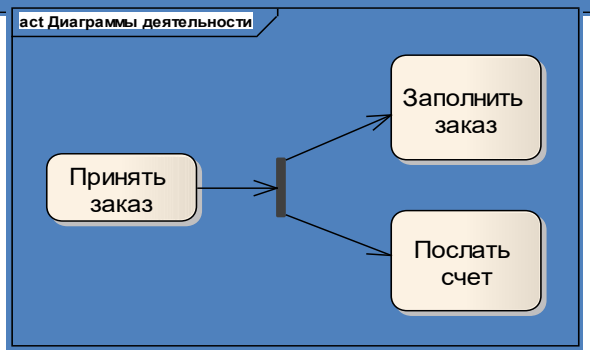
Узел слияния является узлом управления, который соединяет вместе несколько альтернативных потоков.

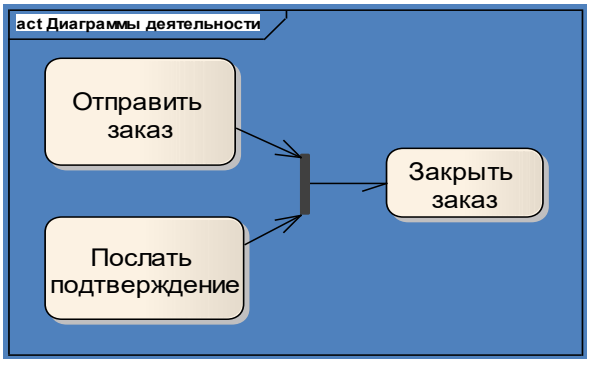


на

Узел разделения является узлом управления, который расщепляет поток несколько параллельных потоков.

Узел соединения является узлом управления, который синхронизирует несколько потоков.



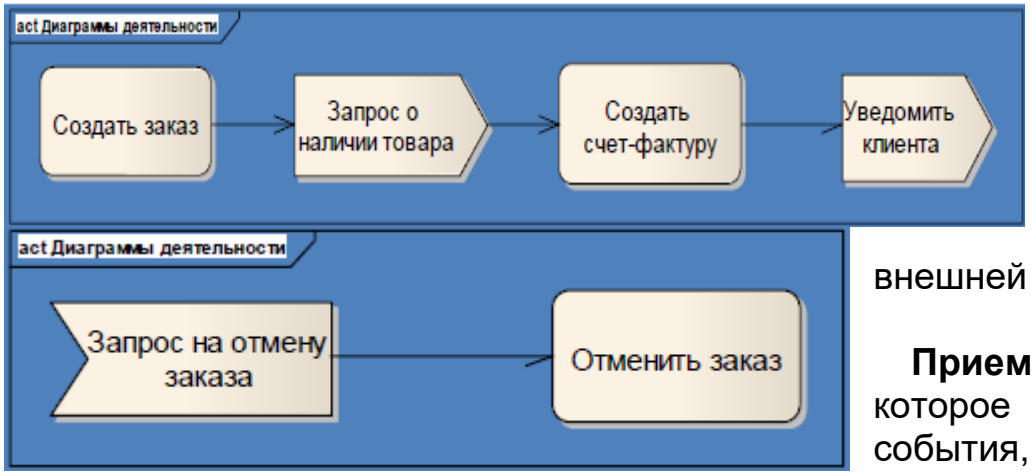


32. Дополнительные элементы диаграммы деятельности: действия приема и передачи сигналов, центральный буфер и хранилище данных

Диаграмма деятельности — UML-диаграмма, на которой показано разложение некоторой деятельности на её составные части. Под деятельностью понимается спецификация исполняемого поведения в виде координированного последовательного и параллельного выполнения подчинённых элементов — вложенных видов деятельности и отдельных действий, соединённых между собой потоками, которые идут от выходов одного узла ко входам другого.

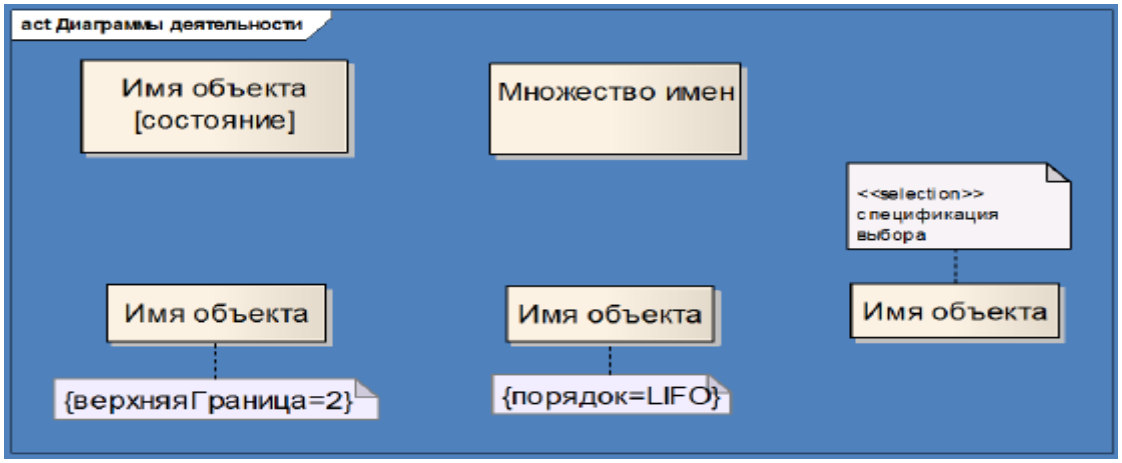
Также диаграмма действия может описывать поведение, на которое оказывают влияние внешние события, происходящие за пределами данной Системы.

На диаграмме это может быть показано при помощи изображения передачи сигнала. Передача сигнала может изображаться путем помещения между двумя действиями соответствующего элемента. Данная семантика была принята в UML 2.0.

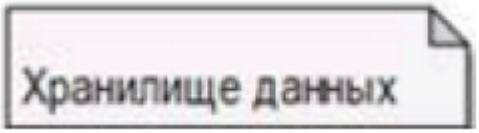


Передача сигнала - действие, которое на основе своих входов создает экземпляр сигнала и передает его внешней Системе.

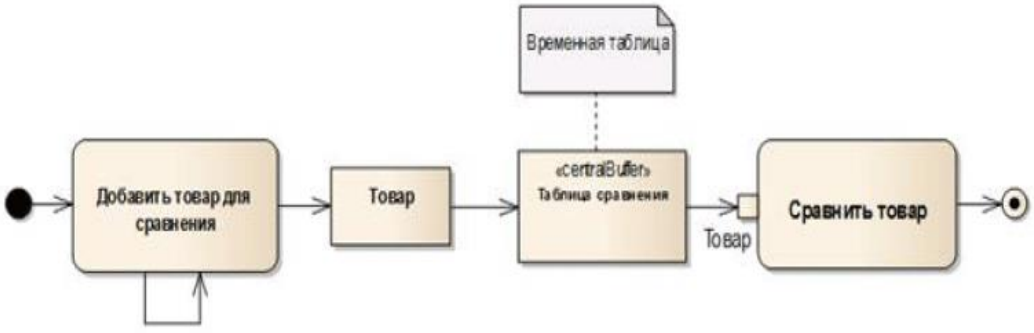
Прием события - действие, которое ожидает некоторого события, принимает и обрабатывает полученное сообщение.



Узел объекта является узлом абстрактной деятельности, которая служит частью определяющего потока объектов в деятельности.



Центральный буфер является узлом объекта для управления потоками из нескольких источников и мест назначения.



Хранилище данных является разновидностью центрального буфера для постоянного хранения объектов или другой информации.

33. Диаграмма конечного автомата: назначение, простое и композитное состояния

Диаграмма конечного автомата является графом, который представляет некоторый конечный автомат.

Вершинами диаграмм конечного автомата являются символы состояния (как обычные состояния, так и псевдосостояния). Граф является ориентированным: все состояния соединяются между собой дугами, называемыми переходами.

Конечный автомат представляет собой некоторый формализм для моделирования поведения отдельных элементов модели или системы в целом.

Поведение является спецификацией того, как экземпляр классификатора изменяет значения отдельных характеристик в течении своего времени жизни.

Состояние – элемент модели поведения, предназначенный для представления ситуации, в ходе которой поддерживается некоторое условие инварианта.

Переход представляет собой направленное отношение между двумя состояниями, одно из которых является вершиной источником, а другое – целевой вершиной.

Событие является спецификацией некоторых условий, которые оказывают влияния на поведение моделируемой сущности.

Триггер устанавливает отношение события с поведением которое может оказывать влияние на экземпляр классификатора.

Простым состоянием называется состояние, которое не имеет внутренних регионов и подсостояний.

Метки деятельности:

entry – специфицирует поведение, которое выполняется всякий раз когда происходит вход в данное состояние независимо от перехода, позволившего достичь это состояние.

exit – специфицирует поведение, которое выполняется всякий раз когда происходит выход из данного состояния независимо от перехода, который выводит из этого состояния.

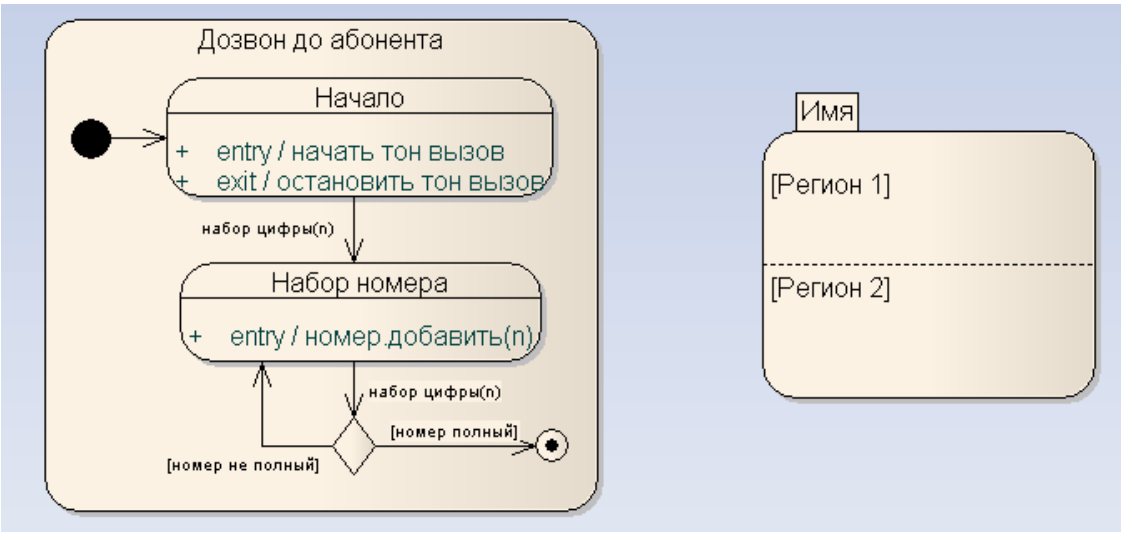
do – специфицирует поведение, которое выполняется до тех пор, пока моделируемый элемент находится в данном состоянии , или до тех пор пока не закончится деятельность, специфицированная соответствующим выражением.

Композитное состояние – состояние, содержащее в своем составе один регион или несколько ортогональных регионов.

Регион – специальный элемент модели, который содержит состояния и переходы и является частью композитного состояния или конечного автомата.

Ортогональное композитное состояние – композитное состояние, содержащее более одного региона, которые в этом случае называются ортогональными регионами.

Любое состояние, заключенное в регион композитного состояния называется подсостоянием этого композитного состояния. Оно называется *прямым* подсостоянием, если оно не содержится ни в каком другом состоянии; в противном случае оно называется *непрямым* подсостоянием.


Дозвон до абонента
Начало
+ entry / начать тон вызов
+ exit / остановить тон вызов
набор цифры(n)
Набор номера
+ entry / номер.добавить(n)
набор цифры(n)
[номер полный]
[номер не полный]
Имя
[Регион 1]
[Регион 2]

34. Диаграмма конечного автомата: просты и составные переходы, правила срабатывания переходов

Составной переход является **производным семантическим понятием**, которое представляет «семантически полный» путь, совершаемый одним или несколькими переходами.

Составной переход является **ациклической непрерывной цепочкой переходов**, которые могут быть соединены посредством псевдосостояний слияния, соединения, выбора или разделения.

Составной переход является **разрешенным**, если и только если:

• все его состояния-источники принадлежат активной конфигурации состояний;

• один из триггеров этого перехода удовлетворяет текущему наступлению события;

• существует, по крайней мере, один полный путь из конфигурации состояний источников либо к конфигурации целевых состояний, либо к некоторому узлу выбора, в котором хотя бы одно сторожевое условие для выходящих переходов принимает значение «истина».

Правило выполнения или срабатывания составного перехода представляет собой последовательность выполнения следующих шагов:

1. Происходит должным образом выход из главного состояния источника.

2. В последовательности переходов выполняются специфицированные действия в соответствии с их линейным порядком вдоль сегментов составного перехода.

3. Если встречается некоторый узел выбора, то динамически оцениваются сторожевые условия, следующие после этого узла выбора, и выбирается некоторый путь, сторожевые условия которого принимают значение «истина».

4. Происходит должным образом вход в главное целевое состояние.

Два и более разрешенных перехода называют **конфликтующими**, если все выходят из одного и того же состояния.

Правило приоритетов переходов:

В общем случае, если t1 является переходом, источником которого выступает состояние s1, и t2 имеет источником s2, то:

• если s1 является прямым или транзитивно вложенным подсостоянием s2, то t1 имеет более высокий приоритет, чем t2;

• если s1 и s2 не входят в одну и туже конфигурацию состояний, то не существует различия между приоритетами t1 и t2.

Простой переход представляет собой отношение между двумя последовательными состояниями, которое указывает на факт смены одного состояния объекта другим. Если пребывание моделируемого объекта в первом состоянии сопровождается выполнением некоторых действий, то переход во второе состояние будет возможен только после завершения этих действий и, возможно, после выполнения некоторых дополнительных условий, называемых сторожевыми условиями.

На диаграмме состояний переход изображается сплошной линией со стрелкой, которая направлена в целевое состояние. Каждый переход может быть помечен строкой текста, которая имеет следующий общий формат:

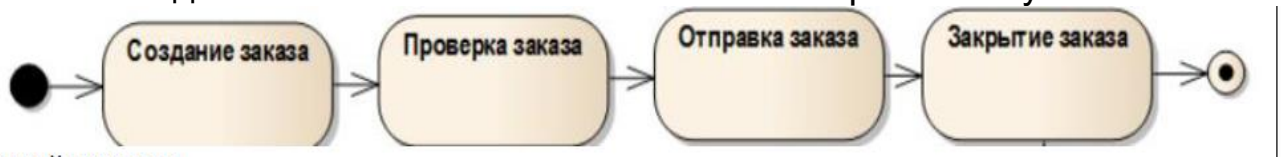
<сигнатура события>'['<сторожевое условие>']' <выражение действия>.

В отдельных случаях переход может иметь несколько состояний-источников и несколько целевых состояний. Такой переход получил название **параллельный переход**.

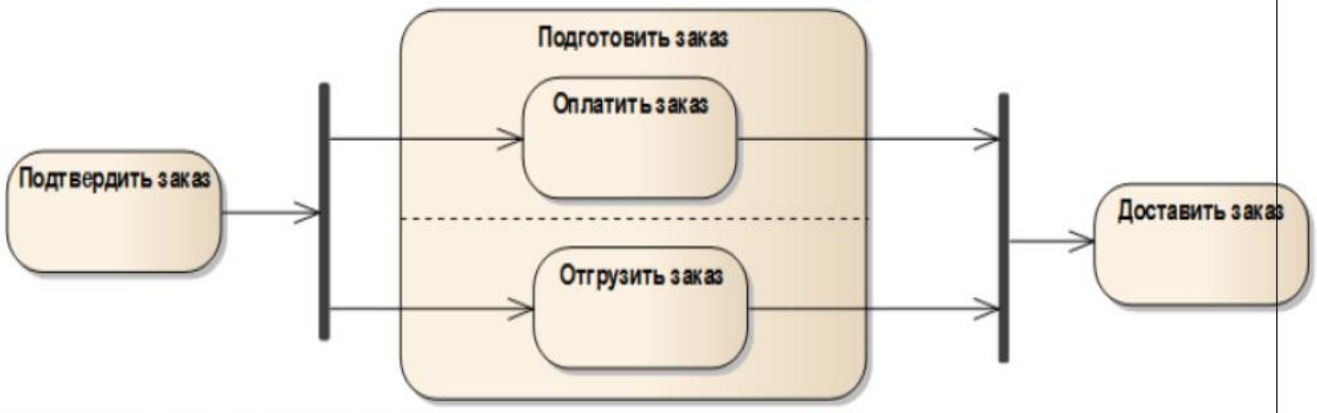
Графически такой переход изображается вертикальной черточкой. Если параллельный переход имеет две и более входящие дуги, его называют

соединением. Если же он имеет две или более исходящие дуги, то его называют ветвлением. Срабатывание параллельного перехода происходит следующим образом. Переход-соединение выполняется, если имеет место событие-триггер для всех исходных состояний этого перехода, и выполнено, если таковое есть, сторожевое условие. При срабатывании перехода-соединения одновременно покидаются все исходные состояния перехода и происходит переход в целевое состояние. При этом каждое из исходных состояний перехода должно принадлежать отдельному подавтомату, входящему в состав автомата.

Переход, стрелка которого соединена с границей некоторого составного состояния, обозначает **переход в составное состояние**. Такой переход эквивалентен переходу в начальное состояние каждого из подавтоматов, входящих в состав данного суперсостояния. Переход, выходящий из составного состояния, относится к каждому из вложенных подсостояний. Это означает, что объект может покинуть составное суперсостояние, находясь в любом из его подсостояний. Для этого вполне достаточно выполнения события и сторожевого условия.



Простой переход.



Параллельный переход в составное состояние.

35. Диаграмма конечного автомата: псевдосостояния, их виды и применение

Псевдосостояние – абстрактный элемент модели, который включает в себя различные типы вспомогательных вершин в графе конечного автомата.

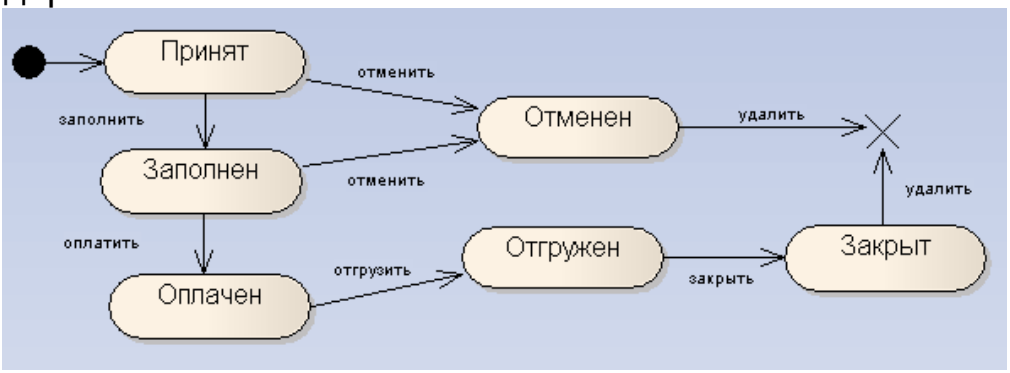
Виды псевдосостояний:

- начальное состояние;
- финальное состояние;
- завершение;
- выбор;
- соединение;
- разделение;
- слияние;
- точка входа;
- точка выхода;
- неглубокая история;
- глубокая история.

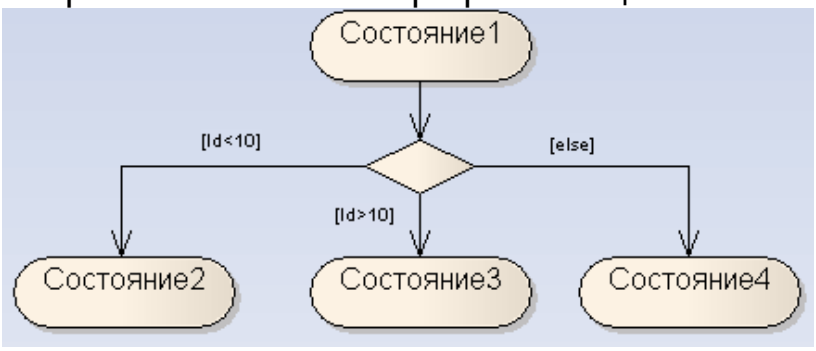
Начальное псевдосостояние представляет вершину графа конечного автомата, которая по умолчанию является состоянием-источником для начального перехода моделируемого поведения.

Узел завершения является псевдосостоянием, вход в который означает завершение выполнения поведения конечного автомата в контексте его объекта.

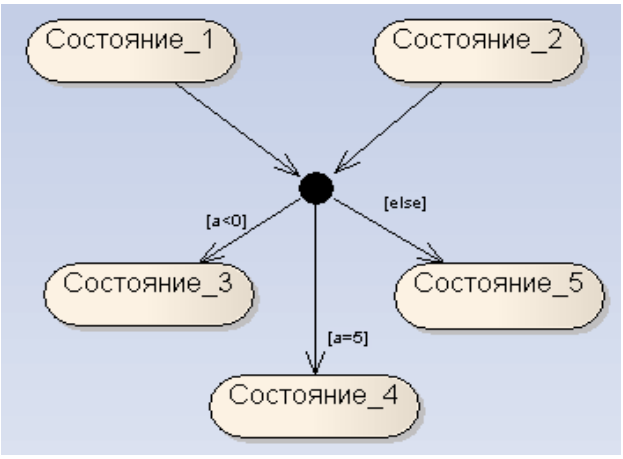
Финальное состояние – специальный вид состояние, предназначенного для моделирования завершения конечного автомата или региона, в котором оно содержится.



Псевдосостояние выбора предназначено для моделирования нескольких альтернативных ветвей при реализации конечного автомата.

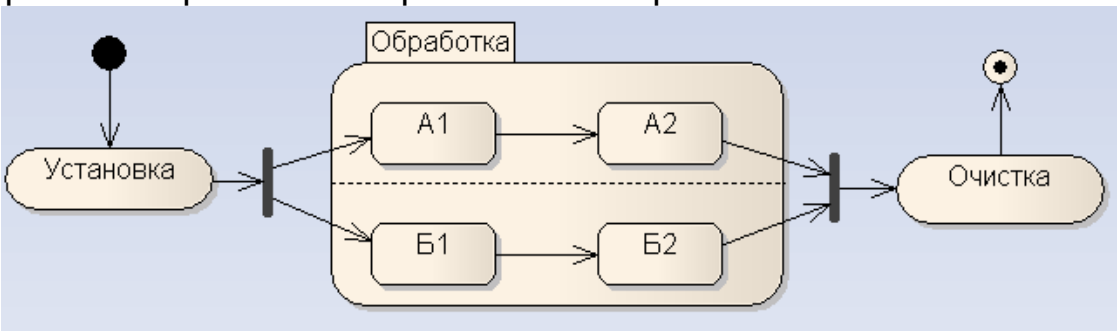


Псевдосостояние соединения является вершиной со свободной семантикой, которая используется для соединения вместе нескольких переходов.



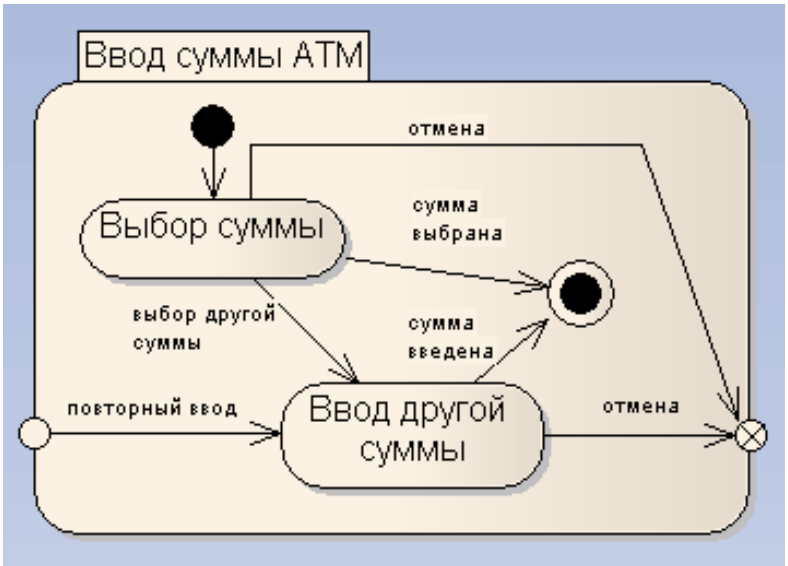
Вершина разделения является псевдосостоянием, предназначенным для разделения входящего перехода на два или более перехода, которые имеют в качестве своих целей вершины в ортогональных регионах композитного состояния.

Вершина слияния является псевдосостоянием, предназначенным для соединения нескольких переходов, которые имеют в качестве своих источников вершины из различных ортогональных регионов композитного состояния.

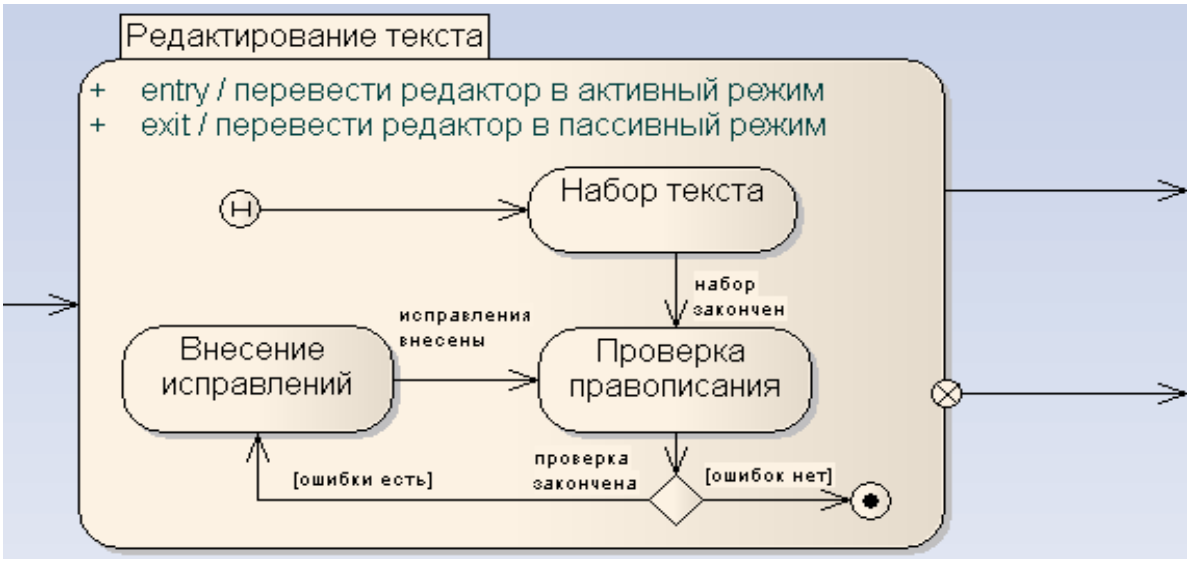


Точка входа является псевдосостоянием, предназначенным для моделирования входа в некоторый конечный автомат или композитное состояние.

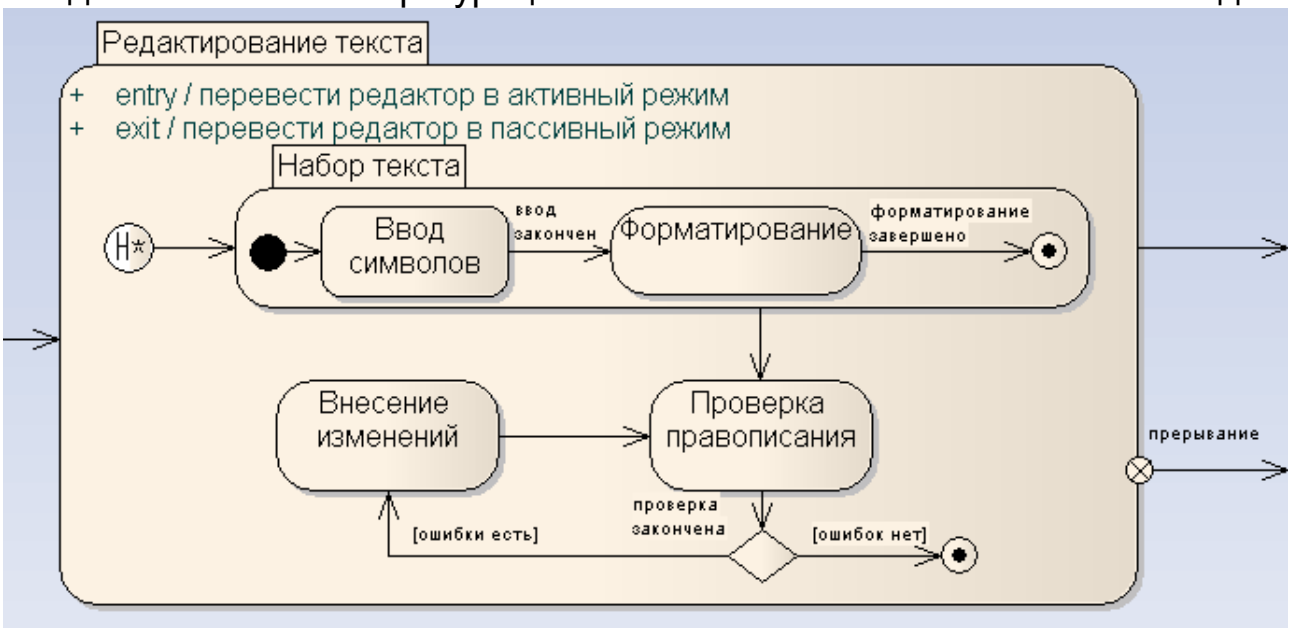
Точка выхода является псевдосостоняием, предназначенным для моделирования выхода из некоторого конечного автомата или композитного состояния.



Псевдосостояние неглубокой истории предназначено для представления самого последнего активного подсостояния композитного состояния после выхода из него.



Псевдосостояние глубокой истории предназначено для представления последней активной конфигурации композитного состояния после выхода из него.

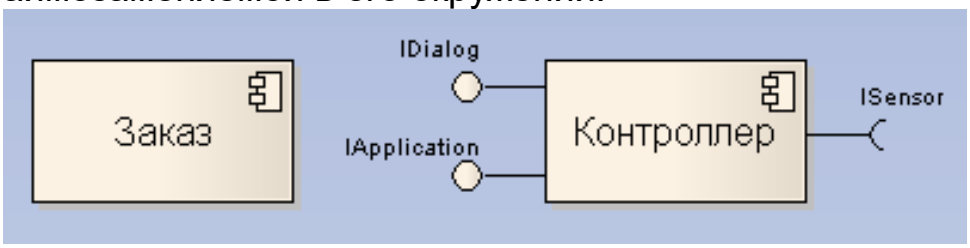


36. Диаграмма компонентов: назначение, компоненты, интерфейсы и порты, соединения и их виды

Диаграмма компонентов разрабатывается для следующих **целей**:

- визуализация общей структуры исходного кода программной системы;
- спецификация исполнимого варианта программной системы;
- обеспечение многократного использования отдельных фрагментов программного кода;
- представление концептуальной и физической схем баз данных.

Компонент – элемент модели, представляющий некоторую модульную часть системы с инкапсулированным содержимым, спецификация которого является взаимозаменяемой в его окружении.



Компонент отображается в двух **видах**:

- ***в виде черного ящика*** – в этом представлении внутренняя структура компонента полностью скрыта от его окружения, а открытыми и общедоступными извне являются только интерфейсы компонента, которые могут быть указаны в форме свойств компонента.
- ***в виде белого ящика*** – это представление показывает, как специфицированное и видимое извне поведение компонента реализуется внутри его.

Компонент может рассматриваться либо в качестве типа, либо в качестве экземпляра.

Компонент в качестве типа наделяется возможностью иметь атрибуты и операции, а также участвовать в ассоциациях и обобщениях.

Компонент в качестве экземпляра может быть инстанцирован от соответствующего компонента в качестве типа. На практике экземпляром компонента может быть некоторый объект времени выполнения или некоторый артефакт, который определяется только во время проектирования.

Интерфейс – вид класса, который представляет собой объявление множества общедоступных характеристик и обязанностей.

Предоставляемый интерфейс – интерфейс, который компонент предлагает для своего окружения.

Требуемый интерфейс – интерфейс, который необходим компоненту от своего окружения для выполнения заявленной функциональности, контракта или поведения.

Предоставляемые интерфейсы иногда называют реализуемыми, реже – обеспеченными интерфейсами

Порт – свойство классификатора, которое специфицирует отдельную точку взаимодействия между этим классификатором и его окружением или между классификаторами и его внутренними частями.

Связи.

Собирающий соединитель – соединитель, который связывает два компонента в контексте предоставляемых и требуемых сервисов.

Делегирующий соединитель – соединитель, который связывает внешний контракт компонента с реализацией этого поведения внутренними частями этого компонента.

В общем случае делегирующий соединитель выполняет одну из следующих **задач**:

- передача сообщений или сигналов, поступающих в порт компонента извне, для обработки в некоторую внутреннюю часть компонента или другой порт;
- передача сообщений или сигналов, поступающих из некоторой внутренней части компонента, для обработки во внешний порт компонента.

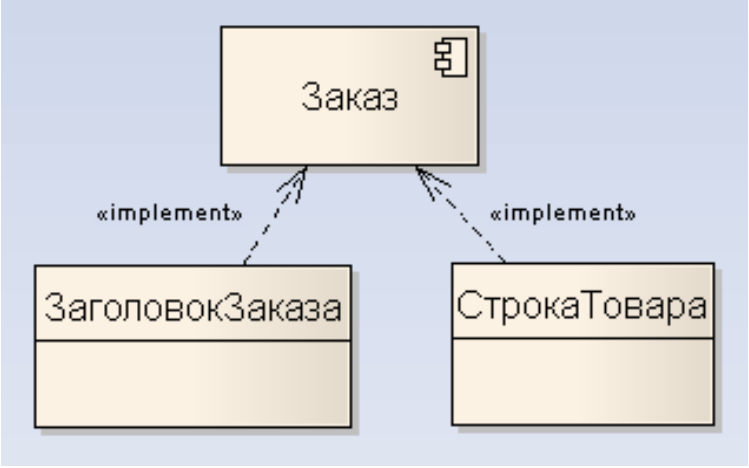
Делегирующий соединитель может быть определен только между интерфейсами или портами одного и того же вида.

Если делегирующий соединитель определяется между требуемым интерфейсом (портом) и некоторой внутренней частью, то эта внутренняя часть должна иметь отношение реализации с интерфейсом (портом) этого типа.

Если делегирующий соединитель определяется между интерфейсом (портом) в качестве источника и интерфейсом (портом) в качестве цели, то интерфейс в качестве цели должен поддерживать сигнатуру, совместимую с подмножеством операций интерфейса (порта) источника.

Зависимость

Реализация – специализация отношения зависимости для связи компонентов с классификаторами, которые реализуют функциональность этого компонента.



37. Диаграмма развертывания: назначение, узлы, артефакты, соединения и их виды

Диаграмма развертывания предназначена для представления общей конфигурации или топологии распределенной программной системы и содержит изображение размещения различных артефактов по отдельным узлам системы.

При разработке диаграмм развертывания преследуются следующие **цели**:

- специфицировать физические узлы, необходимые для размещения на них исполнимых компонентов программной системы;
- показать физические связи между узлами реализации системы на этапе ее исполнения;
- выявить узкие места системы и реконфигурировать ее топологию для достижения требуемой производительности.

Узел является элементом модели, который представляет некоторый вычислительный ресурс для развертывания на нем различных артефактов.

Среда выполнения представляет собой узел, который обладает функциональностью, необходимой для практического выполнения развернутых на нем исполнимых артефактов.

Устройство представляет собой узел, который обладает некоторым общим вычислительным ресурсом со способностью обрабатывать развернутые на нем артефакты.

Цель развертывания является абстрактным метаклассом для указания местоположения размещаемого артефакта.

Размещаемый артефакт является абстрактным метаклассом для представления артефакта или экземпляра артефакта, который должен быть размещен на цели развертывания.

Артефакт представляет собой элемент модели, который специфицирует некоторую физически существующую часть информации, используемую или производимую в ходе разработки программного обеспечения или в процессе развертывания и функционирования системы.

Спецификация экземпляра представляет собой элемент модели, предназначенный для указания **свойств конкретной цели развертывания или конкретного размещаемого артефакта.**

Спецификация развертывания описывает множество свойств, которые определяют параметры выполнения артефакта компонента, развертываемого на некотором узле.

Развертывание представляет собой размещение артефакта или экземпляра артефакта на некоторой цели развертывания.

Манифестация представляет собой отношение для спецификации конкретного физического воплощения одного или нескольких элементов модели посредством артефакта.

Путь коммуникации является ассоциацией между двумя целями развертывания, посредством которой они обладают способностью обмениваться сигналами и сообщениями.

38. Проблемы классического подхода к разработке ПО и причины появления гибких методологий

Суть проблемы состоит в том, что, начиная с самых первых программных проектов и по нынешний день, разработка программного обеспечения была и остаётся мало предсказуемым и далеко не всегда успешным делом. Значительный процент проектов по созданию программного обеспечения по-прежнему заканчивается с превышением бюджета, сроков, а созданные в результате программы часто не до конца отвечают требованиям пользователей либо приносят мало реальной пользы бизнесу. Перечисленные проблемы являются основными проявлениями так называемого кризиса программного обеспечения. Несмотря на значительные интеллектуальные усилия, потраченные на поиск способов преодоления кризиса, до сих пор так и не найдено сколько-нибудь универсальное решение (как часто говорят, "серебряной пули"). Гибкие методы - это современный ответ программной индустрии на вопрос как же всё-таки надо выполнять проекты, чтобы они с большей долей вероятности завершались успехом и приносили удовлетворение всем заинтересованным сторонам - и прежде всего, заказчику и команде проекта.

Классическая методология создания ПО, используемая в мире уже не один десяток лет, состоит из шести последовательных этапов - анализ требований, проектирование, кодирование, сборка, тестирование, внедрение/сопровождение. На ее основе разработаны конкретные популярные модели.

Модель “Водопад” предусматривает последовательный переход от фазы к фазе - поочередно выполняются определение системных требований, анализ требований к продукту, предварительное и детальное проектирование, кодирование, тестирование, объединение модулей, проверка работы всей системы, комплексное и системное тестирование и внедрение.

Последовательная модель также основана на возможности четкого определения требований к проекту, что позволяет создавать пилотные прототипы, постепенно увеличивая общие функциональные возможности системы и ведя разработку нескольких модулей параллельно.

Главная проблема подобных подходов - экспоненциальный рост времени на устранение выявляемых на последних этапах ошибок (особенно если они связаны с неверно спроектированной архитектурой системы или с плохо сформулированными требованиями).В течение 1990-х годов все больше разработчиков ПО начинали искать альтернативу традиционным, как правило, основанным на модели водопада, процессам разработки. К 2000 году существовало уже целое множество так называемых легковесных (lightweight) методологий. (Я использую термин «методология», а не «процесс», поскольку гибкие методологии включают в себя множество практик и технологий, выходящих за рамки описания процессов.)

В 2001 году группа создателей и экспертов по различным легковесным методологиям провела семинар, на котором были сформулированы основные принципы гибкой разработки ПО (так называемый Agile Manifesto). На том же семинаре было предложено новое название легковесных методологий – гибкая разработка (agile software development).

Общими особенностями гибких методологий являются:

Ориентированность на людей – как разработчиков, так и заказчиков. Считается, что умение собрать в проектной команде «правильных» людей определяет успех или неудачу проекта в значительно большей степени, чем любые процессы или технологии.

Использование устных обсуждений вместо формальных спецификаций везде, где это возможно. Обсуждения должны быть главным способом коммуникации как с заказчиком, так и внутри проектной команды.

Итеративная разработка с возможно более короткой (в разумных пределах) продолжительностью итерации, при этом в результате каждой итерации выпускается полноценная работающая версия продукта.

Ожидание изменений – в гибком процессе проектная команда не пытается зафиксировать требования в начале проекта и затем следовать жестко определенному плану. Изменения могут быть сделаны на сколь угодно позднем этапе проекта.

По всей видимости, из методологий гибкой разработки самое широкое

39. Гибкие методолгии. Преимущества и область применения.

Agile (Гибкая методология разработки)

Преимущества применения Agile

— частые релизы — требования не успевают устаревать, частью функционала уже можно пользоваться;

— фиксированная длина итераций — можно предсказывать скорость работы команды с учетом рисков;

— команда сама оценивает задачи — оценки реалистичны, команда мотивирована выполнить свои обязательства;

— команда самоуправляемая — 10 голов учтут больше чем одна очень умная;

— в конце каждой итерации процесс работы оценивается и вносятся улучшения;

— команда кроссфункциональная — границы отделов компании не являются препятствием при сотрудничестве, разнообразные навыки сочетаются и происходит синергия.

Область применения Agile

Спектр применения Agile довольно широк: от небольших студенческих стартапов, до крупных промышленных проектов размером в тысячи человеко-часов, как в локальной команде, так и в проекте с географически распределенными командами.

Не каждой команде может подойти применение гибкой методологии. Для некоторых проектов будет удачной и водопадная модель.

Не всегда применение гибких методологий может дать положительный эффект. Agile может негативно сказаться на эффективности:

— в проектах, которые являются инфраструктурными и имеют очень сложный процесс поддержки;

— в проектах, где подтверждение технического задания требует очень длительного формального цикла;

— если нет обратной связи с конечным пользователем системы или нет возможности у команды провести экспертизу в предметной области;

— если команда состоит из недостаточно квалифицированных специалистов, которые не готовы к изменениям и внедрению прогрессивных подходов.

Наиболее известные гибкие методологии

Agile Modeling — набор понятий, принципов и приёмов (практик), позволяющих быстро и просто выполнять моделирование и документирование в проектах разработки программного обеспечения.

— Agile Unified Process (AUP) описывает простое и понятное приближение (модель) для создания программного обеспечения для бизнес-приложений.

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

Getting Real — итеративный подход без функциональных спецификаций, использующийся для веб-приложений.

40. Экстремальное программирование: понятие, базис XP, структура XP цикла разработки

Экстремальное программирование (англ. *Extreme Programming*, *XP*) - одна из гибких методологий разработки программного обеспечения. (Автор: Кент Бек (1999 г.)) Экстремальное программирование - это упрощенная методика организации производства для небольших и средних по размеру команд специалистов, занимающихся разработкой программного продукта в условиях неясных или быстро меняющихся требований.

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.

Методика XP предназначена для работы над проектами, над которыми может работать от двух до десяти программистов, которые не зажаты в жесткие рамки существующего компьютерного окружения и в которых вся необходимая работа, связанная с тестированием, может быть выполнена в течение одного дня.

Базис XP

Короткий цикл обратной связи (Fine scale feedback)

• Разработка через тестирование (Test driven development) — непрерывное написание тестов для модулей, которые должны выполняться безупречно

• Игра в планирование (Planning game) быстрое определение области действия следующей реализации путем объединения деловых приоритетов и технических оценок. Заказчик формирует область действия, приоритетность и сроки с точки зрения бизнеса, а разработчики оценивают и прослеживают продвижение (прогресс).

• Заказчик всегда рядом (Whole team, Onsite customer) — в группе все время должен находиться представитель заказчика, действительно готовый отвечать на вопросы разработчиков.

• Парное программирование (Pair programming) — весь код пишется двумя программистами, работающими на одном компьютере.

Непрерывный, а не пакетный процесс

• Непрерывная интеграция (Continuous Integration) — система интегрируется и строится много раз в день, по мере завершения каждой задачи.

• Рефакторинг (Design Improvement, Refactor) — система реструктурируется, но ее поведение не изменяется; цель — устранить дублирование, улучшить взаимодействие, упростить систему или добавить в нее гибкость.

• Частые небольшие релизы (Small Releases) — быстрый запуск в производство простой системы. Новые версии реализуются в очень коротком (двухнедельном) цикле

Понимание, разделяемое всеми

• Простота (Simple design) — проектирование выполняется настолько просто, насколько это возможно в данный момент.

• Метафора системы (System metaphor)— вся разработка проводится на основе том, как работает вся система.

• Коллективное владение кодом (Collective code ownership) или выбранными шаблонами проектирования (Collective patterns ownership) — разработчик может улучшать в любое время любой код системы.

• Стандарт кодирования (Coding standard or Coding conventions) — должны выдерживаться правила, обеспечивающие одинаковое представление программного кода во всех частях программной системы.

Социальная защищенность программиста (Programmer welfare):

• 40-часовая рабочая неделя (Sustainable pace, Forty hour week)) — как правило, работают не более 40 часов в неделю. Нельзя удваивать рабочую неделю за счет сверхурочных работ.

Дополнительные принципы:

1. Открытое рабочее пространство (open workspace)
2. Изменение правил при необходимости (just rules)

Цикл разработки

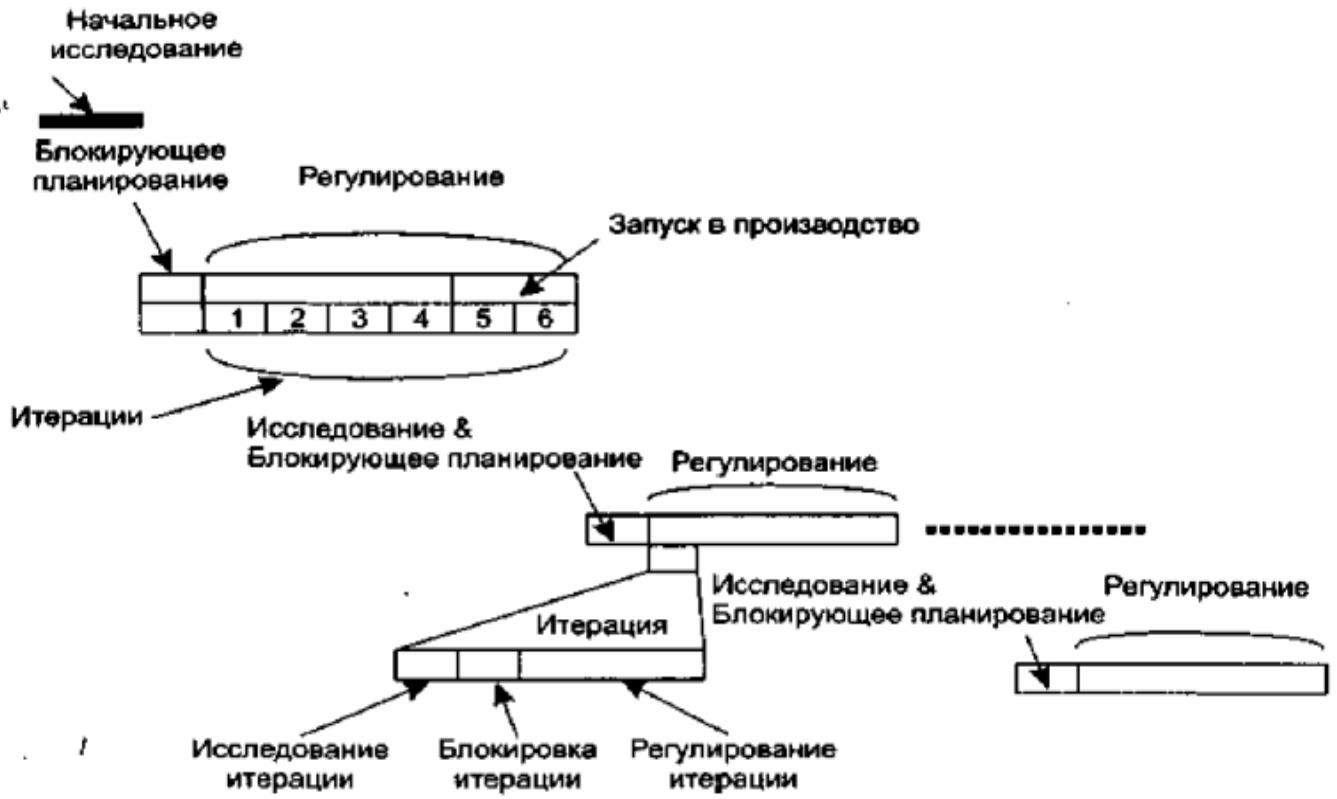
В масштабе месяцев и лет мы имеем дело с историями данной версии системы и затем с историями будущих версий. В масштабе недель и месяцев мы имеем дело с историями данной итерации и затем с историями, оставшимися в данной версии. В масштабе дней и недель мы имеем дело с задачей, над которой работаем в данный момент, а затем с задачами, оставшимися в итерации. Наконец, в масштабе минут и дней мы имеем дело с тестом, который прогоняем в данный момент, и затем с оставшимися тестовыми примерами, которые, возможно, придут нам в голову.

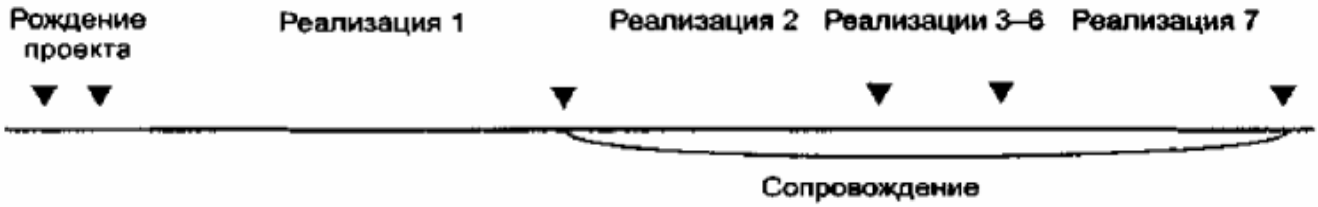
Заказчик определяет истории для следующей итерации, выбирая наиболее значимые истории из оставшихся в версии, вновь опираясь на оценку стоимости и скорости их разработки. Программисты разбивают истории на локальные задачи, и каждый берет на себя ответственность за одну из них. Затем программист преобразует свою задачу в набор тестовых примеров, успешное выполнение которых покажет, что задача решена полностью. Работая в паре с партнером, программист добивается нормальной работы тестов, одновременно развивая общий проект. Таким образом, удается реализовать максимально простую архитектуру системы в целом.

Основным структурным элементом процесса является XP-реализация, в которую многократно вкладывается базовый элемент — XP-итерация.

В состав XP-реализации и XP-итерации входят **три фазы**:

1. ***Исследование*** (exploration) — это поиск новых требований (историй, задач), которые должна выполнять система.
2. ***Блокировка*** (commitment) — выбор для реализации конкретного подмножества из всех возможных требований (иными словами, планирование).
3. ***Регулирование*** (steering) — проведение разработки, воплощение плана в жизнь.


Начальное исследование
Блокирующее планирование
Регулирование
Запуск в производство
1
2
3
4
5
6
Итерации
Исследование & Блокирующее планирование
Регулирование
Исследование & Блокирующее планирование
Регулирование
Итерация
Исследование итерации
Блокировка итерации
Регулирование итерации


Рождение проекта
Реализация 1
Реализация 2
Реализации 3–6
Реализация 7
Сопровождение

41. SCRUM процесс: понятие, роли , мероприятия, уровни команд в SCRUM.

Скрам (SCRUM – потасовка, драка за мяч в регби) – это подход, в рамках которого возможно решить сложные адаптивные проблемы, и в то же время продуктивно и с применением творческого подхода разработать продукт наивысшего качества. Скрам является:

— Легким

— Простым в понимании

— Чрезвычайно сложным в овладении

SCRUM основывается на теории управления эмпирическими процессами, или эмпиризме.

SCRUM использует итеративный, инкрементный подход для оптимизации прогнозируемости и управления рисками.

В основе управления эмпирическими процессами лежат три главных принципа:

• прозрачность (transparency),

• проверка (inspection),

• адаптация (adaptation).

Основные роли

— ***Скрам-мастер*** (Scrum Master) — 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.

— ***Владелец продукта*** (Product Owner) — представляет интересы конечных пользователей и других заинтересованных в продукте сторон.

— ***Скрам-команда*** (Scrum Team) — кросс-функциональная команда разработчиков проекта, состоящая из специалистов разных профилей: тестировщиков,архитекторов, аналитиков, программистов и т. д. Размер команды в идеале составляет от 3 до 9 человек. Команда является единственным полностью вовлечённым участником разработки и отвечает за результат как единое целое. Никто, кроме команды, не может вмешиваться в процесс разработки на протяжении спринта.

Дополнительные роли

— Пользователи (*Users*)

— Клиенты, Продавцы (*Stakeholders*) — лица, которые инициируют проект и для кого проект будет приносить выгоду. Они вовлечены в скрам только во время обзорного совещания по спринту (Sprint Review).

— Управляющие (*Managers*) — люди, которые управляют персоналом.

— Эксперты-консультанты (*Consulting Experts*)

мероприятия SCRUM

Четко установленные мероприятия используются в Скраме для того, чтобы придать процессу разработки регулярность и минимизировать потребность в совещаниях, не предписанных Скрамом. Скрам использует ограниченные по времени мероприятия, поэтому каждое мероприятие имеет свой верхний предел продолжительности. Это гарантирует, что планирование будет проводиться в предназначенное время, не позволяя потерь времени в процессе планирования.

Кроме главного мероприятия, собственно самого Спринта, который включает все остальные мероприятия, каждое мероприятие Скрама является возможностью что-то проверить и провести адаптацию чего-нибудь. Такие мероприятия являются специально разработанными для обеспечения необходимой прозрачности и контроля. Отказ от одного из таких мероприятий приводит к уменьшению

прозрачности и является потерянной возможностью провести инспекцию и адаптацию.

артефакты SCRUM

К **артефактам** SCRUM относят:

• ***Журнал продукта*** (Product backlog) – это упорядоченный список всего, что может быть нужным в продукте, он является единственным источником требований для любых изменений, которые может потребоваться внести в продукт.

• ***Журнал спринта*** (Sprint backlog) – это набор элементов из Журнала Продукта, выбранных для выполнения в текущем Спринте, а также план разработки Инкремента продукта и достижения Цели Спринта.

• ***Инкремент*** (Increment) – это сумма всех выполненных требований Журнала Продукта реализованных во время текущего Спринта и всех предыдущих Спринтов.

этапы командообразования в SCRUM

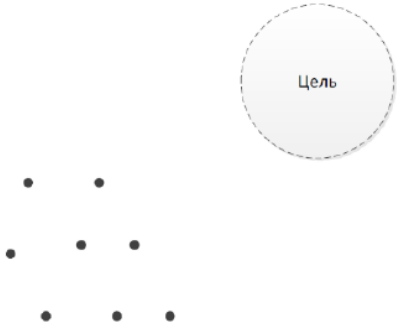
Команда — это небольшая группа людей, взаимодополняющих и взаимозаменяющих друг друга, которые собраны для совместного решения задач производительности и в соответствии с подходами, посредством которых они поддерживают взаимную ответственность.

Этапы командообразования

В своем развитии команды проходят несколько **этапов**, которые сменяют друг друга:

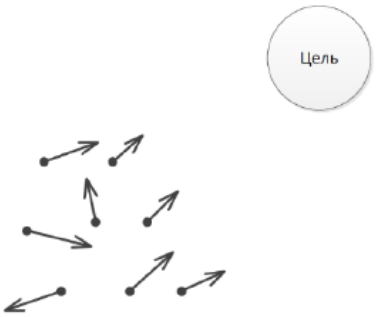
1. ***Формирование***

На этапе формирования происходит создание команды и постановка целей, распределение и закрепление ролей (в том числе и социальных). Отдельные члены команды еще не очень понимают цель и задачи, которые перед ними поставлены.



2. ***Бурление***

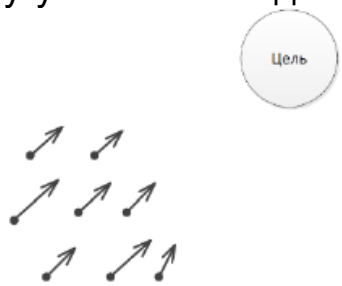
На этапе «бурления» участники осознают свои цели и определяют вектор движения. Обратите внимание, что эти векторы разнонаправленные между собой и с направлением, которое необходимо для достижения цели: которые перед ними поставлены (рисунок 3).



Цель также стала более четкой и понимаемой для команды. На данном этапе особенно часто возможны конфликты и противостояние между членами команды, поэтому особенно возрастает роль скрам-мастера как модератора.

3. ***Нормализация***

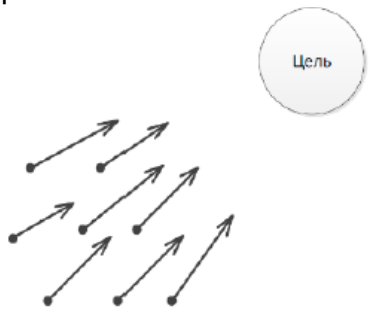
Следующим этапом идет нормализация, когда члены команды притираются к друг другу и начинают двигаться сонаправлено (рисунок 4).



Основной задачей скрам-мастера является фасилитация для перехода на следующий этап.

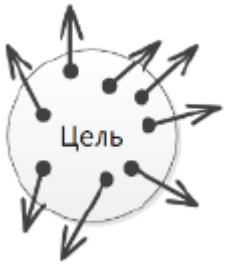
1. **Функционирование**

На этапе функционирования команда становится самоуправляемой и способной оптимизировать свою производительность, поэтому векторы, направленные к цели, удлиняются.



2. **Расформирование**

Когда цели, поставленные перед командой, достигнуты, наступает этап расформирования, и направление «движения» участников снова рассинхронизируется (рисунок 6).



Этап	**Быстрый переход**	**Средний переход**	**Долгий переход**
Формирование	0-ой спринт	2-ой спринт	2-ой спринт
Бурление	1-ый спринт	4-ый спринт	6-ой спринт

Нормализация	1-ый спринт	6-ой спринт	10-ый спринт
Функционирование	2-ой спринт	8-ой спринт	16-ый спринт
Расформирование	Завершение проекта		

42.Разработка, управляемая тестированием (Test Driven Development).

Тестирование кода раздражает, но приносит огромную пользу при работе над проектом. Сегодня мы будем использовать TDD, чтобы писать и тестировать код более эффективно.

Что такое TDD?

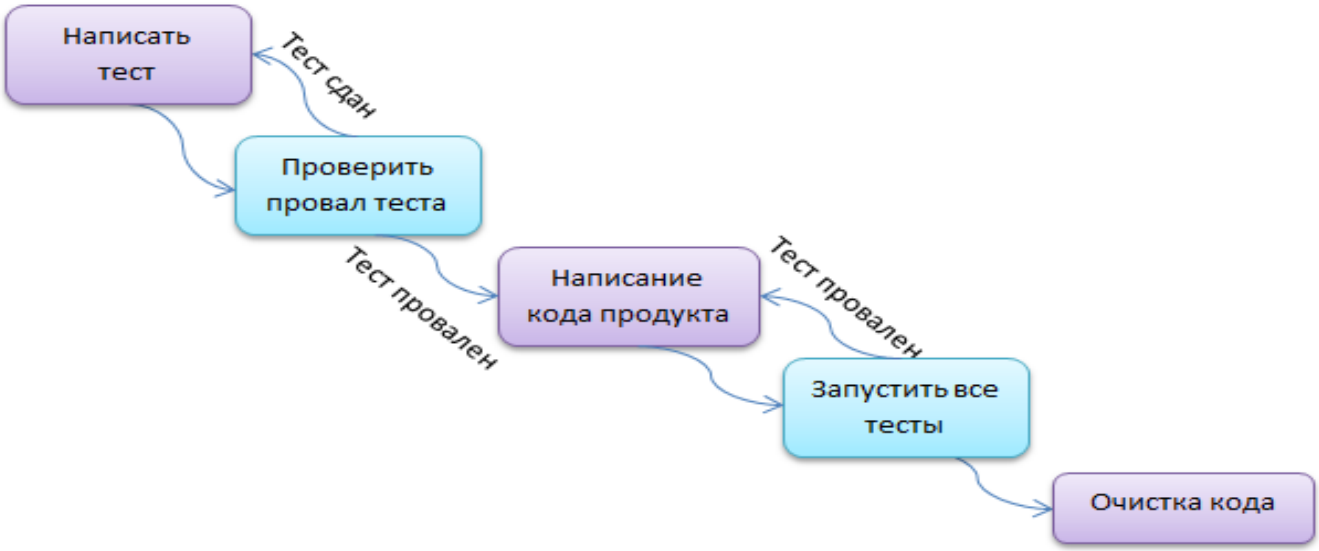
С самого начала компьютерной эры программисты и ошибки боролись за превосходство. Это неизбежное явление. Каждый программист становился жертвой ошибок. Код небезопасен. Вот почему мы проводим тесты.

Разработка через тестирование (test-driven development) — техника программирования, при которой модульные тесты для программы или её фрагмента пишутся до самой программы (test-first development) и, по существу, управляют её разработкой.

Как это работает?

Разработка через тестирование или TDD представляет собой короткий повторяющийся цикл развития:

1. Прежде чем писать любой код, необходимо сначала написать автоматический тест для вашего кода. Во время написания теста необходимо учитывать все возможные входные данные, ошибки и вывод в браузер
2. После этого можно начать написание кода. Необходимо вносить изменения до тех пор, пока код не будет проходить автоматические тесты
3. Как только код прошёл тест, можно начинать очистку кода (рефакторинг)



Отлично, но чем это лучше регулярного тестирования?

Вы не проводили тестирование программы по нескольким причинам:

- вы чувствовали, что это пустая трата времени, так как были незначительное поправки кода
- у вас не было достаточно времени для теста, так как менеджер проекта хотел получить продукт как можно быстрее
- вы откладывали это на завтра

Долгое время ничего не происходит, и вы перенесли ваш продукт на производство. Но иногда после того как продукт перенесён на производство, всё происходит не так. TDD ликвидирует наши ошибки. Когда программа была разработана с использованием разработки через тестирование, она позволяет вносить изменения и проводить тесты более эффективно и быстро. Всё что нам нужно сделать – это запустить тесты. Если код проходит все тесты, значит его можно применять, значит

мы внесли ошибки вместе с изменениями. Зная, какая часть испытаний провалилась, легче ликвидировать ошибки.

Как провести тесты?

Существует много фреймворков для тестирования PHP скриптов. Одним из наиболее широко используемых является PHPUnit.

PHPUnit это отличный фреймворк для тестирования, который можно легко интегрировать в свои проекты или другие проекты, созданные с помощью популярных PHP фреймворков.

43.Разработка, управляемая поведением (Behavior Driven Development).

BDD (сокр. от англ. Behavior-driven development, дословно «разработка через поведение») — это методология разработки программного обеспечения, являющаяся ответвлением от методологии разработки через тестирование (TDD).

Основной идеей данной методологии является совмещение в процессе разработки чисто технических интересов и интересов бизнеса, позволяя тем самым управляющему персоналу и программистам говорить на одном языке. Для общения между этими группами персонала используется предметно-ориентированный язык, основу которого представляют конструкции из естественного языка, понятные неспециалисту, обычно выражающие поведение программного продукта и ожидаемые результаты.

Считается, что данный подход эффективен, когда предметная область, в которой работает программный продукт, описывается очень комплексно.

BDD методология является расширением TDD в том смысле, что перед тем как написать какой-либо тест необходимо сначала описать желаемый результат от добавляемой функциональности на предметно-ориентированном языке. После того как это будет проделано, конструкции этого языка переводятся специалистами или специальным программным обеспечением в описание теста.

BDD фокусируется на следующих вопросах:

- С чего начинается процесс.
- Что нужно тестировать, а что нет.
- Сколько проверок должно быть совершено за один раз.
- Что можно назвать проверкой.
- Как понять, почему тест не прошёл.

Исходя из этих вопросов, BDD требует, чтобы имена тестов были целыми предложениями, которые начинаются с глагола в сослагательном наклонении и следовали бизнес целям. Описание приемочных тестов должно вестись на гибком языке пользовательской истории, например,

Как [роль того, чьи бизнес интересы удовлетворяются] я хочу, чтобы [описание функциональности так, как она должна работать], для того чтобы [описание выгоды].

Критерии приёмки должны быть описаны через сценарий, который реализует пользователь, чтобы достигнуть результата.

Принципы BDD

Как уже было отмечено, тесты для некоторой единицы программного обеспечения должны быть описаны с точки зрения желаемого поведения программируемого устройства. Под желаемым поведением здесь понимается такое, которое имеет ценность для бизнеса. Описание желаемого поведения даётся с помощью *спецификации поведения*.

Спецификация поведения строится в полуформальной форме. В настоящее время в практике BDD устоялась следующая структура:

1. *Заголовок*. В сослагательной форме должно быть дано описание бизнес-цели.
2. *Описание*. В краткой и свободной форме должны быть раскрыты следующие вопросы:
    1. Кто является заинтересованным лицом данной истории;
    2. Что входит в состав данной истории;
    3. Какую ценность данная история предоставляет для бизнеса.

3. *Сценарии*. В одной спецификации может быть один и более сценариев, каждый из которых раскрывает одну из ситуаций поведения пользователя, тем самым конкретизируя описание спецификации. Каждый сценарий обычно строится по одной и той же схеме:
    1. Начальные условия (одно или несколько);
    2. Событие, которое инициирует начало этого сценария;
    3. Ожидаемый результат или результаты.

BDD не предоставляет каких-либо формальных правил, но настаивает на том, чтобы использовался ограниченный стандартный набор фраз, который включал бы все элементы спецификации поведения. В 2007 году Дэном Нортом был предложен шаблон для спецификации, который получил популярность и впоследствии стал известен как язык *Gherkin*.

Способы реализации BDD концепции

Полуформальный формат спецификации поведения требует использования ограниченного набора предложений, о которых управляющий персонал, и разработчики должны предварительно договориться. Исходя из этого, фреймворки для поддержки BDD строятся по следующим принципам:

- Парсер может разбить спецификацию по её формальным частям, например, по ключевым словам, языка Gherkin. На выходе мы получаем набор предложений, каждое из которых начинается с ключевого слова.
- Каждое предложение может выражать один шаг теста.
- Некоторые части предложения могут являться входными параметрами, которые можно захватить, а остальные части могут никак не использоваться и служить только для понимания действия человеком. Обычно для такого захвата используется процессор регулярных выражений. Захваченные параметры могут быть переконвертированы и отправлены на вход конкретной исполняющей функции.

На этом принципе строятся такие фреймворки как JBehave и RBehave, которые основаны на языке Gherkin. Некоторые фреймворки строятся по аналогии, например, CBehave и Cucumber.

44.CASE¬средства: понятие, история появления и развития, структура и состав, классификация.

Обычно к CASE-средствам относят любое программное средство, автоматизирующее ту или иную совокупность процессов жизненного цикла ПО и обладающее следующими основными характерными особенностями:

- мощные графические средства для описания и документирования ИС, обеспечивающие удобный интерфейс с разработчиком и развивающие его творческие возможности;
- интеграция отдельных компонент CASE-средств, обеспечивающая управляемость процессом разработки ИС;
- использование специальным образом организованного хранилища проектных метаданных (репозитория).

Интегрированное CASE-средство (или комплекс средств, поддерживающих полный ЖЦ ПО) содержит следующие компоненты;

- репозиторий, являющийся основой CASE-средства. Он должен обеспечивать хранение версий проекта и его отдельных компонентов, синхронизацию поступления информации от различных разработчиков при групповой разработке, контроль метаданных на полноту и непротиворечивость;
- графические средства анализа и проектирования, обеспечивающие создание и редактирование иерархически связанных диаграмм (DFD, ERD и др.), образующих модели ИС;
- средства разработки приложений, включая языки 4GL и генераторы кодов;
- средства конфигурационного управления;
- средства документирования;
- средства тестирования;
- средства управления проектом;
- средства реинжиниринга.

История и развитие

С самого начала CASE-технологии развивались с целью преодоления ограничений ручных применений методологий структурного анализа и проектирования 60-70-х годов (сложности понимания, большой трудоемкости и стоимости использования, трудности внесения изменений в проектные спецификации и т.д.) за счет их автоматизации и интеграции поддерживающих средств. Таким образом, CASE-технологии не могут считаться самостоятельными методологиями, они только делают более эффективными пути их применения.

Традиционно выделяют шесть периодов, качественно отличающихся применяемой техникой и методами разработки ПО, которые характеризуются использованием в качестве инструментальных средств:

1. ассемблеров, дампов памяти, анализаторов;
2. компиляторов, интерпретаторов, трассировщиков;
3. символьных отладчиков, пакетов программ;
4. систем анализа и управления исходными текстами;
5. CASE-средств анализа требований, проектирования спецификаций и структуры, редактирования интерфейсов (первая генерация CASE-I);
6. CASE-средств генерации исходных текстов и реализации интегрированного окружения поддержки полного жизненного цикла разработки ПО (вторая генерация CASE-II)

CASE-I является первой технологией, адресованной непосредственно системным аналитикам и проектировщикам, и включающей средства для поддержки графических моделей, проектирования спецификаций, экранных редакторов и

словарей данных. Она не предназначена для поддержки полного жизненного цикла и концентрирует внимание на функциональных спецификациях и начальных шагах проекта – системном анализе, определении требований, системном проектировании, логическом проектировании БД [9].

CASE-II отличается значительно более развитыми возможностями, улучшенными характеристиками и исчерпывающим подходом к полному жизненному циклу. В ней, в первую очередь, используются средства поддержки автоматической кодогенерации, а также, обеспечивается полная функциональная поддержка для выполнения графических системных требований и спецификаций проектирования; контроля, анализа и связывания системной информации и информации по управлению проектированием; построение прототипов и моделей системы; тестирования, верификации и анализа сгенерированных программ; генерации документов по проекту; контроля на соответствие стандартам по всем этапам жизненного цикла. CASE-II может включать свыше 100 функциональных компонент, поддерживающих все этапы жизненного цикла, при этом пользователям предоставляется возможность выбора необходимых средств и их интеграции в нужном составе.

структура и состав

CASE-технологии предлагают новый, основанный на автоматизации подход к концепции ЖЦ ПО. При использовании CASE изменяются все фазы ЖЦ, при этом наибольшие изменения касаются фаз анализа и проектирования.

Простейшая модель ЖЦ:

Анализ
Проектирование
Кодирование
Тестирование
Сопровождение

CASE-модель ЖЦ:

Прототипирование
Проектирование спецификаций
Контроль проекта
Кодогенерация
Системное тестирование
Сопровождение

В CASE-модели фаза прототипирования заменяет традиционную фазу системного анализа. Необходимо отметить, что наиболее автоматизируемыми фазами являются фазы контроля проекта и кодогенерации (хотя все остальные фазы также поддерживаются CASE-средствами).

классификация

В функции CASE входят средства анализа, проектирования и программирования программных средств, проектирования интерфейсов, документирования и производства структурированного кода на каком-либо языке программирования.[4]

CASE-инструменты классифицируются по типам и категориям.

Классификация по типам отражает функциональную ориентацию средств на те или иные процессы жизненного цикла разработки программного обеспечения, и, в основном, совпадают с компонентным составом крупных интегрированных CASE-систем, и включает следующие типы:

- средства анализа — предназначены для построения и анализа модели предметной области;
- средства проектирования баз данных;

- средства разработки приложений;
- средства реинжиниринга процессов;
- средства планирования и управления проектом;
- средства тестирования;
- средства документирования.

Классификация по категориям определяет степень интегрированности по выполняемым функциям и включают — отдельные локальные средства, решающие небольшие автономные задачи, набор частично интегрированных средств, охватывающих большинство этапов жизненного цикла и полностью интегрированных средств, охватывающий весь жизненный цикл информационной системы и связанных общим репозиторием.

Типичными CASE-инструментами являются:

- инструменты управления конфигурацией;
- инструменты моделирования данных;
- инструменты анализа и проектирования;
- инструменты преобразования моделей;
- инструменты редактирования программного кода;
- инструменты рефакторинга кода;
- генераторы кода;
- инструменты для построения UML-диаграмм.