

**Министерство науки и высшего образования Российской Федерации
ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
«НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ ИТМО»
(Университет ИТМО)**

Факультет программной инженерии и компьютерной техники

ОТВЕТЫ НА ВОПРОСЫ К ЭКЗАМЕНУ

По дисциплине: Методологии программной инженерии

Руководитель дисциплины: Клименков С. В., старший преподаватель факультета
программной инженерии и компьютерной техники

Санкт-Петербург
2022

СОДЕРЖАНИЕ

ВОПРОС №1.....	7
Вопрос: Диаграммы прецедентов (use-case), основные понятия. (RUP&UML2 Главы 4.2, 4.3.1, 4.3.2, 4.3.3.) (Стр.:91-98).....	7
ВОПРОС №2.....	9
Диаграммы прецедентов: обобщение, включение, расширение.....	9
ВОПРОС №3.....	14
Диаграмма классов: основные понятия, содержимое класса. (RUP&UML2 Главы 7.4, 7.5.)	14
ВОПРОС №4.....	20
Диаграмма классов: ассоциации. Синтаксис и кратность.....	20
ВОПРОС №5.....	23
Диаграмма классов: ассоциации и атрибуты. Возможность навигации. (RUP&UML2 Главы 9.4.3, 9.4.4.)	23
ВОПРОС №6.....	26
Диаграмма классов: агрегация и композиция. (RUP&UML2 Главы 18.3, 18.4, 18.5.)	26
ВОПРОС №7.....	27
Диаграмма классов: наследование, абстрактные классы, множественное наследование. (RUP&UML2 Глава 10.3.).....	27
ВОПРОС №8.....	31
Диаграмма классов: квалифицированные ассоциации, классы ассоциаций. (RUP&XYIUML2 Главы 9.4.5, 9.4.6.)	31
ВОПРОС №9.....	35
Вопрос: Выражение зависимости в диаграммах. (RUP&UML2 Глава 9.5. (Стр. 219))	35
ВОПРОС №10.....	37
Диаграмма последовательности. Элементы диаграммы. (RUP&UML2 Глава 12.9.)	37
ВОПРОС №11.....	40
Диаграмма последовательности. Комбинированные фрагменты и операторы. (RUP&UML2 Глава 12.10.)	40
ВОПРОС №12.....	43
Коммуникационные (кооперативные) диаграммы.....	43
ВОПРОС №13.....	47
Диаграммы пакетов. (RUP&UML2 Главы 11.2, 11.3, 11.4, 11.5, 11.6.)	47

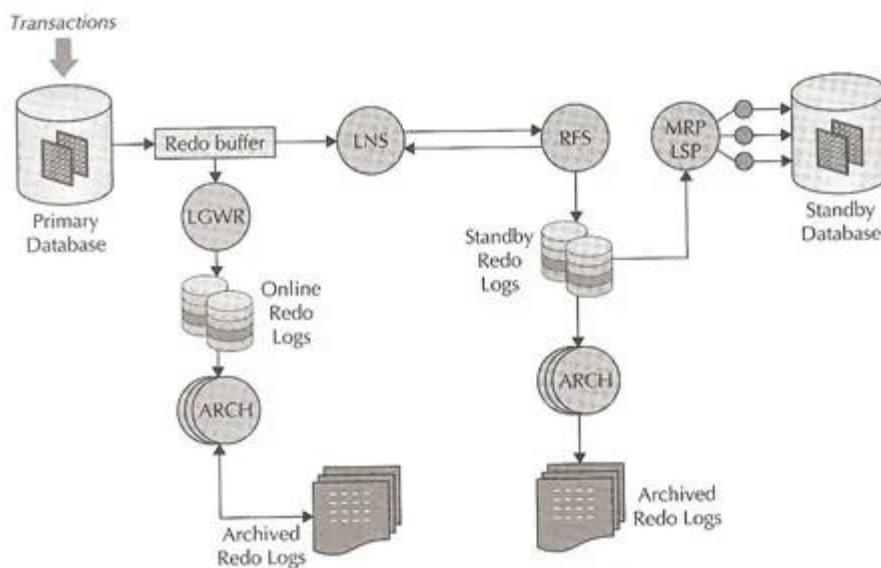
Вопрос №14	51
Диаграммы конечных автоматов (состояний).	51
ВОПРОС №15.....	55
Диаграммы деятельности.....	55
ВОПРОС №16.....	58
Диаграмма размещения. (RUP&UML2 Главы 24.3, 24.4, 24.5, 24.6.).....	58
ВОПРОС №17.....	60
Процессы разработки ПО - UP, RUP. Предназначение, связь и основные определения: исполнитель (роль), деятельность, артефакт, рабочий поток (дисциплина) и др.....	60
ВОПРОС №18.....	64
Итеративная разработка в UP. Рабочие потоки UP, Дисциплины RUP. Базовые версии. (RUP&UML2 Главы: 2.7, 2.7.1, 2.7.2 RUP_classic.2002.05.00/RationalUnifiedProcess/process/workflow/ovu_core.htm)	64
ВОПРОС №19.....	66
Аксиомы UP, риски. Структура UP. Фазы жизненного цикла, цели, контрольные точки (вехи).....	66
ВОПРОС №20.....	68
Рабочий поток определения требований.....	68
ВОПРОС №21.....	69
Понятие требования. Типы, организация и атрибуты требований. (RUP&UML2 Глава 3.6.).....	69
Атрибуты требований.	70
ВОПРОС №#22	72
Поиск и выявление требований. (RUP&UML2 Глава 3.7)	72
ВОПРОС № 23.....	74
Рабочий поток анализа.....	74
ВОПРОС №24.....	75
Практические правила построения аналитической модели (RUP&UML2 Глава 6.5).....	75
ВОПРОС №25.....	76
Анализ прецедента, классы анализа, практические правила создания классов анализа. (RUP&UML2 Главы 8.2,8.3.).....	76
ВОПРОС №26.....	79
Выявление классов анализа с помощью CRC модели.	79
ВОПРОС №27.....	80

Рабочий поток проектирования. Связь между аналитической и проектной моделями	80
ВОПРОС №28.....	82
Архитектура, проектирование архитектуры.....	82
ВОПРОС №29.....	83
Проектирование прецедента. (RXUP&UYMIL2 Главы 20.2, 20.3, [20.4]).....	83
ВОПРОС №30.....	85
Рабочий поток реализации. Модель реализации.....	85
ВОПРОС №31.....	87
Фаза Начало: цели, контрольная точка, состояние артефактов. (RUP&UML2 Главы 2.9.1, 2.9.2, 2.9.3)	87
ВОПРОС №32.....	89
Фаза Уточнение: цели, контрольная точка, состояние артефактов. (RUP&UML2 Главы 2.9.4, 2.9.5, 2.9.6)	89
ВОПРОС №33.....	92
Фаза Построение: цели, контрольная точка, состояние артефактов. (RUP&UML2 Главы 2.9.7, 2.9.8, 2.9.9)	92
ВОПРОС №34.....	93
Фаза Внедрение: цели, контрольная точка, состояние артефактов. (RUP&UML2 Главы 2.9.10, 2.9.11, 2.9.12)	93
ВОПРОС №35.....	94
Детализация прецедентов, артефакт Спецификация прецедента. (RUP&UML2 Глава 4.4, 4.5.)	94
ВОПРОС №36.....	97
Артефакт Vision (Концепция). Предназначение разделов. RUP_classic.2002.05.00/RationalUnifiedProcess/process/artifact/ar_vision.htm	97
ВОПРОС №37.....	102
Артефакт SDP (План разработки). Предназначение разделов.....	102
ВОПРОС №38.....	105
Артефакт SRS (Требования к продукту). Предназначение разделов.	105
ВОПРОС №39.....	107
Артефакт Risk List (Список Рисков). Предназначение разделов. Управление рисками.....	107
ВОПРОС №40.....	110
Артефакт Business Case (Бизнес обоснование). Предназначение разделов.....	110
ВОПРОС №41.....	112

Модели жизненного цикла разработки программного продукта.	112
ВОПРОС №42.....	114
Каскадная модель, особенности, область применения. Каскадная модель, особенности, область применения. (Рудаков 26, 27; Royce W. all)	114
ВОПРОС №43.....	116
V-образная модель, особенности, область применения.	116
ВОПРОС №44.....	118
Модель прототипирования, особенности, область применения.....	118
ВОПРОС №45.....	119
Вопрос: RAD-модель, особенности, область применения	119
ВОПРОС №46.....	121
Многопроходная модель, особенности, область применения.	121
ВОПРОС №47.....	123
Сpirальная модель, особенности, область применения	123
ВОПРОС №48.....	124
EUP. Фазы и дисциплины.	124
ВОПРОС №49.....	127
EUP. Дисциплина продуктивного использования и поддержки.....	127
ВОПРОС №50.....	129
EUP. Дисциплины архитектуры предприятия, управления людьми.....	129
ВОПРОС №51.....	133
EUP. Дисциплины управления портфолио, стратегического повторного использования.....	133
ВОПРОС №52.....	136
Гибкие методологии разработки. Agile Manifesto. Принципы Agile.....	136
ВОПРОС №53.....	137
Scrum. Основы процесса. Роли. Артефакты. (Вольфсон, стр. 8, 9, 16, 17)	137
ВОПРОС №54.....	139
Scrum. Процессы. Ретроспектива.....	139
ВОПРОС №55.....	143
Scrum. Этапы командообразования. Самоорганизация в командах, модель CDE	143
ВОПРОС № 56.....	145
Scrum. Покер-планирование.....	145

ВОПРОС №57.....	146
Scrum. Диаграмма сгорания. Улучшенная диаграмма сгорания (Вольфсон, 48-50, Подвижная мишень и дрожащие руки, Дорофеев Максим).....	146
ВОПРОС №58.....	151
Scrum. Доска задач. Теория X и Y (Вольфсон, 50-54)	151
ВОПРОС №59.....	152
Disciplined Agile 2.0 Введение в методологию. Процесс, фазы, дисциплины.....	152
ВОПРОС №60.....	154
XP: Подход к разработке ПО.....	154
ВОПРОС №61.....	156
XP: Проблемы и мотивации. Риски при разработке ПО. (Кент Бек, Глава 1.).....	156
ВОПРОС №62.....	158
Проблемы и мотивации. Модель четырех переменных.	158
ВОПРОС №63.....	160
Вопрос: XP: Проблемы и мотивации. Стоимость изменений. (Кент Бек, Глава 5.).....	160
ВОПРОС № 64	162
Scrum & XP. Инженерные практики(Вольфсон, 63-67, Кент Бек, Часть 2.)......	162
ВОПРОС №65.....	164
OpenUP: Жизненный цикл, принципы, дисциплины. (http://epf.eclipse.org/wikis/openup/).....	164
ВОПРОС №66.....	165
OpenUP: Артефакты, роли. Отличия от RUP.....	165
ВОПРОС №67.....	167
AUP: Жизненный цикл, философия, инкрементальные выпуски.	167
ВОПРОС №68.....	168
AUP: дисциплины, артефакты, роли. Отличия от RUP.	168
ВОПРОС №69.....	177
Архитектура монолитных приложений. Достоинства и недостатки.....	177
ВОПРОС №70.....	180
Двухуровневая архитектура клиент-сервер. Достоинства и недостатки.	180
ВОПРОС №71.....	182
Многоуровневая архитектура, особенности применения. (Software Architecture Pattern, гл. 1).....	182
ВОПРОС №72.....	184

Архитектура primary-standby (на примере Oracle DataGuard)



.....184

ВОПРОС №73.....	186
Кластеры высокой готовности (на примере SolarisCluster или VeritasCluster)	186
ВОПРОС №74.....	191
Высокопроизводительные кластеры (на примере Univa Grid Engine)	191
ВОПРОС №75.....	193
Обмен сообщениями. Очереди сообщений. (на примере ApacheMQ и MQTT)	193
ВОПРОС №76.....	196
Фильтры и конвейерная архитектура (на примере GStreamer).....	196
ВОПРОС №77.....	199
Архитектура, основанная на событиях (Software Architecture Pattern, гл. 2).....	199
ВОПРОС №78.....	202
Архитектура микроядра (Software Architecture Pattern, гл. 3).....	202
ВОПРОС №79.....	204
Архитектура микросервисов	204
ВОПРОС №80.....	206
Архитектура облака (Software Architecture Pattern, гл. 5).....	206

ВОПРОС №1

Вопрос: Диаграммы прецедентов (use-case), основные понятия. (RUP&UML2 Главы 4.2, 4.3.1, 4.3.2, 4.3.3.) (Стр.:91-98)

Ответ: Моделирование прецедентов – это форма выработки требований. Это дополнительный способ выявления и документирования требований.

Прецеденты – способ записи требований.

В модели прецедентов есть 4 компонента: Граница, Актёры, Прецеденты, Отношения.

Моделирование прецедентов включает выявление актеров и прецедентов.

1. Контекст системы:

Контекст системы отделяет систему от остального мира.

По факту – это границы системы. Контекст отделяет, что является частью системы, а что снаружи. Контекст системы определяется тем, кто или что использует систему (т.е. актерами), и тем, какие конкретные преимущества системы предлагает этим актерам (т.е. прецедентами). Отображается в виде прямоугольника с именем системы. Актеры - вне системы, а прецеденты внутри.

2. Актеры системы:

Актеры – это роли, исполняемые сущностями, непосредственно взаимодействующими с системой.

Актеры исполняют роли

Роль подобна шляпе, которую надевают в определенной ситуации.

Актер может играть несколько ролей. Вопрос “Какую роль играет эта сущность по отношению к системе?” помогает выявить роли. А для выявления актеров надо спросить себя: “Кто или что использует или взаимодействует с системой?”

Актеры являются внешними по отношению к системе.

Время тоже может быть актером! (Если надо смоделировать, что что-то происходит с системой в определенный момент времени, а не инициируется кем-либо)



«actor»
Customer

Рис. 4.3. Варианты изображения актера

3. Прецеденты:

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

Прецедент - это что-то, что всегда должна делать система по желанию актера. (прецеденты всегда инициируются актерами и описываются с точки зрения актеров).



Рис. 4.5. Пиктограмма прецедента

Чтобы найти прецедент, надо спросить: «Как каждый из актеров использует систему?» и «Что система делает для каждого актера?»

Каждому прецеденту должны быть присвоено конкретное описательно имя, представляющее собой глагольную группу.

4. Диаграмма прецедентов:

Отношения между актером и прецедентом обозначается сплошной линией. Это символ ассоциации в UML. Ассоциация между актером и прецедентом показывает, что актер и прецедент каким-то образом взаимодействуют.

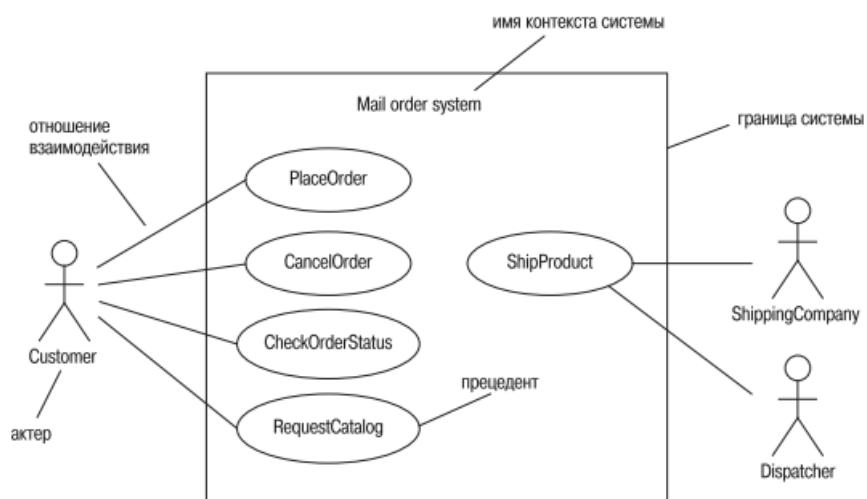


Рис. 4.6. Диаграмма прецедентов

ВОПРОС №2

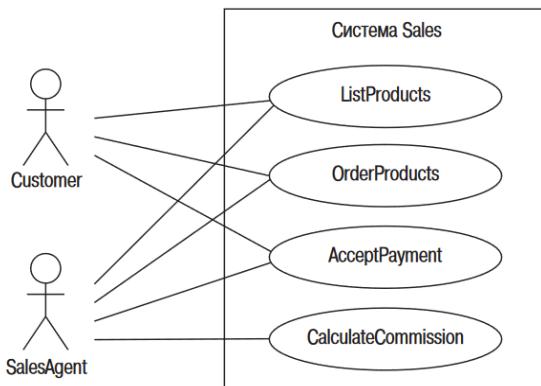
Диаграммы прецедентов: обобщение, включение, расширение.

- **обобщение актеров** – отношение обобщения между обобщенным актером и конкретным актером;
- **обобщение прецедентов** – отношение обобщения между обобщенным прецедентом и специализированным прецедентом;
- **«include» (включить)** – отношение между прецедентами, которое позволяет одному прецеденту включать в себя поведение другого;
- **«extend» (расширить)** – отношение между прецедентами, которое позволяет одному прецеденту расширять свое поведение одним или более фрагментами поведения другого.

Важно сохранять максимальную простоту модели и использовать **include** и **extend** только по необходимости.

1 Обобщение актеров

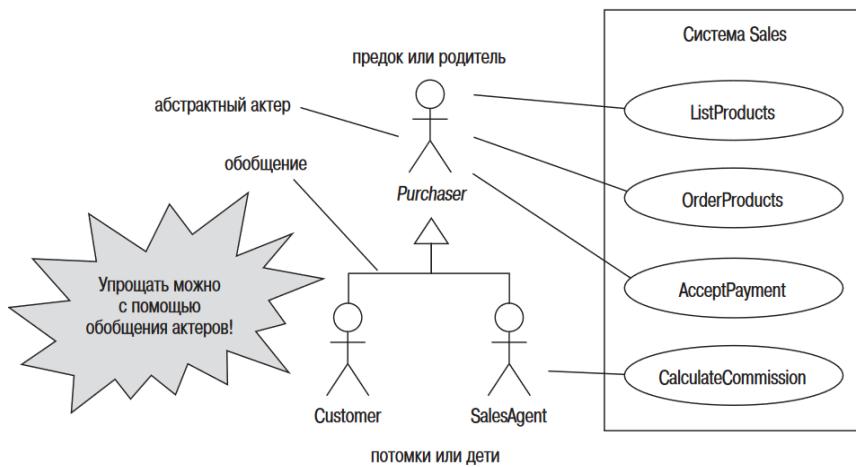
Рассмотрим пример с двумя актерами, имеющими общее поведение.



Отличие между ними в том, что SalesAgent может инициировать прецедент Calculate Commission

Кроме того, из-за сходства поведения этих актеров на диаграмме возникает масса пересекающихся линий. Это указывает на наличие некоего общего поведения, которое может быть вынесено и представлено в виде более обобщенного актера.

Обобщение выносит поведение, общее для двух или более актеров, в актера-родителя.



Общее поведение можно вынести путем обобщения актеров. Тем самым мы создаем абстрактного актера, называемого Purchaser (покупатель), который взаимодействует с прецедентами ListProducts, OrderProducts и AcceptPayment. Customer и SalesAgent – это конкретные актеры, потому что данные роли могут выполнять реальные люди (или другие системы). Purchaser – абстрактный актер, поскольку он является абстракцией, введенной просто для представления общего поведения (возможности делать покупки) двух конкретных актеров.

2 Обобщение прецедентов

Обобщение прецедентов используется, если есть один или более прецедентов, которые на самом деле являются специализациями более общего прецедента. Как и обобщение актеров, этот прием следует применять, только если он упрощает модель прецедентов.

Обобщение прецедентов выносит поведение, общее для одного или более прецедентов, в родительский прецедент.

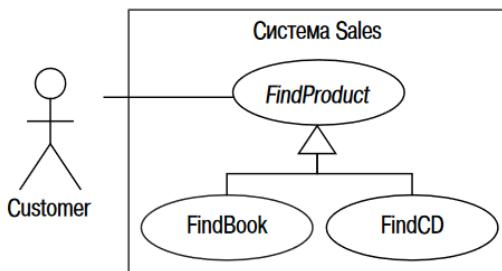
В обобщении прецедентов дочерние прецеденты представляют более специализированные формы их родителей. Потомки могут:

- наследовать возможности родительского прецедента;
- вводить новые возможности;
- переопределять (менять) унаследованные возможности.

Дочерний прецедент автоматически наследует все возможности своего родителя. Однако не все возможности прецедента могут быть переопределены. Ограничения приведены в таблице.

Возможность прецедента	Наследование	Добавление	Переопределение
Отношение	да	да	нет
Точка расширения	да	да	нет
Предусловие	да	да	да
Постусловие	да	да	да
Шаг основного потока	да	да	да
Альтернативный поток	да	да	да

Пример обобщения прецедентов:



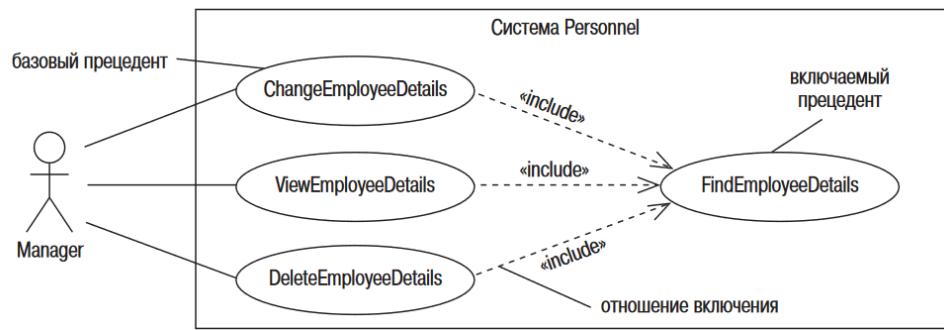
2.1 Документирование обобщения прецедентов.

- Каждый номер шага в потомке сопровождается номером эквивалентного шага родителя, если таковой имеется.
Например: 1. (2.). Некоторый шаг.
- Если шаг потомка переопределяет шаг родителя, его номер сопровождается буквой «о» (что значит overridden – переопределенный) и родительским номером шага.
Например: 6. (о6.) Другой шаг.

3 Отношение «include»

Например, рассмотрим систему Personnel (персонал). Практически любое действие системы начинается с получения данных о конкретном служащем. Если бы последовательность нужно было писать всегда, когда нужны данные служащего, прецеденты имели бы повторяющиеся части. Отношение «include», устанавливаемое между прецедентами, позволяет включить поведение одного в поток другого.

Отношение «include» выносит шаги, общие для нескольких прецедентов, в отдельный прецедент, который потом включается в остальные.

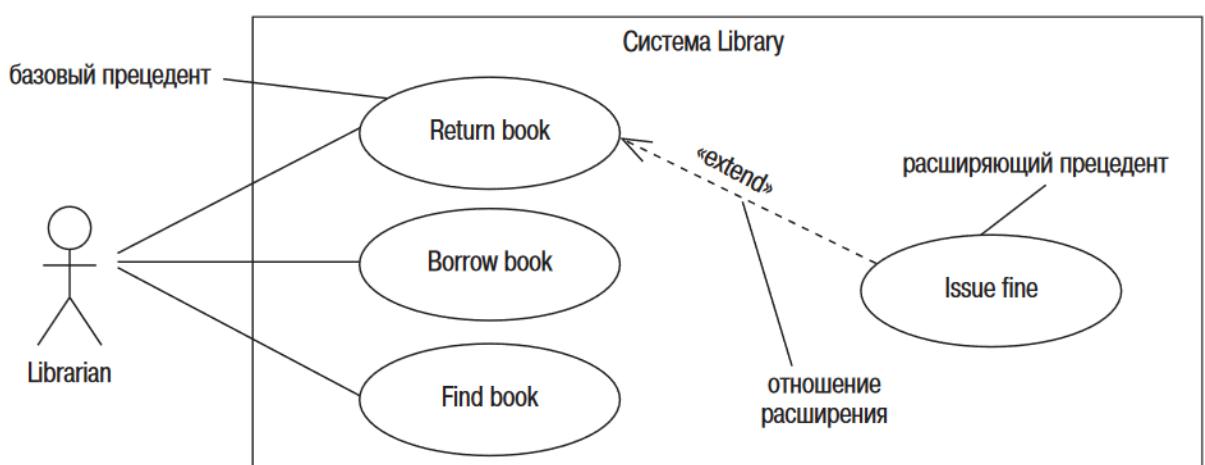


Включающий прецедент мы называем базовым, а тот прецедент, который включается, включаемым. Включаемый прецедент предоставляет поведение своему базовому прецеденту. В базовом прецеденте необходимо точно указать место, где должно быть включено поведение включаемого прецедента. Синтаксис и семантика отношения «include» немного напоминают вызов функции.

4 Отношение «extend»

Отношение «extend» предоставляет возможность ввести новое поведение в существующий прецедент. Базовый прецедент предоставляет набор точек расширения (extension points) – точек входа, в которые может быть добавлено новое поведение. А расширяющий прецедент предоставляет ряд сегментов вставки, которые можно ввести в базовый прецедент в места, указанные точками входа. Как вскоре будет показано, отношение «extend» может использоваться для того, чтобы точно указать, какие именно точки расширения базового прецедента подлежат расширению.

В отношении «extend» любопытно то, что базовый прецедент ничего не знает о расширяющих прецедентах, он просто предоставляет для них точки входа. Базовый прецедент абсолютно полон и без расширений. Это существенно отличает «extend» от отношения «include», где базовые



Источники

Джим Арлоу и Айла Нейштадт. UML2. Стр 119-129.

https://vk.com/doc161468248_590066525?hash=39d503ef225d8a0d41&dl=8e827044760c61f135

ВОПРОС №3

Диаграмма классов: основные понятия, содержимое класса. (RUP&UML2 Главы 7.4, 7.5.)

Объект - отдельная сущность с явно выраженнымными границами, которая инкапсулирует состояние и поведение; экземпляр класса. Объекты скрывают данные на уровне функций, которые называются операциями.

Идентификатор (identity) – это определение существования и единственности объекта во времени и пространстве. Это то, что отличает его от всех остальных объектов.

Состояние (state) – определяется значениями атрибутов объекта и его отношениями с другими объектами в конкретный момент времени.

Поведение объекта – это то, «что он может сделать для нас», т. е. его операции.

Операция – это описание части поведения. Реализация этого поведения называется **методом (method)**.

Пример класса - атрибута - состояния - значения:

Класс	Атрибут	Состояние	Значение
Smartphone	bluetooth	Отключен	off

Пример поведения - операции:

Таблица 7.2

Операция	Семантика
deposit()	Размещает некоторую сумму в объекте Account. Увеличивает значение атрибута balance.
withdraw()	Снимает некоторую сумму с Account. Уменьшает значение атрибута balance.
getOwner()	Возвращает владельца объекта Account – операция запроса.
setOwner()	Меняет владельца объекта Account.

Объекты в нотации UML:

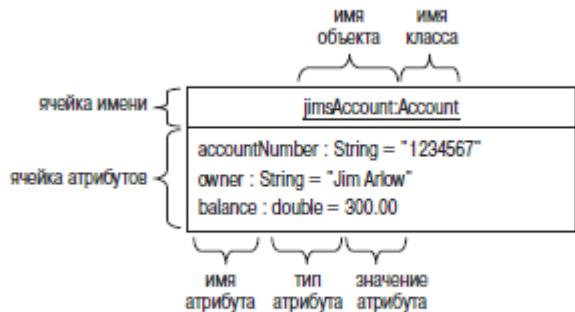


Рис. 7.4. Нотация объектов в UML

Классы

Класс описывает свойства ряда объектов. Каждый объект – это экземпляр только одного класса.

Класс – это спецификация или шаблон, которому должны следовать все объекты этого класса (экземпляры). Атрибуты, описанные классом, в каждом объекте имеют конкретные значения. Каждый объект будет отвечать на сообщения, инициируя операции, описанные классом.

Между классом и объектами этого класса устанавливается отношение «**«instantiate»**» (создать экземпляр).

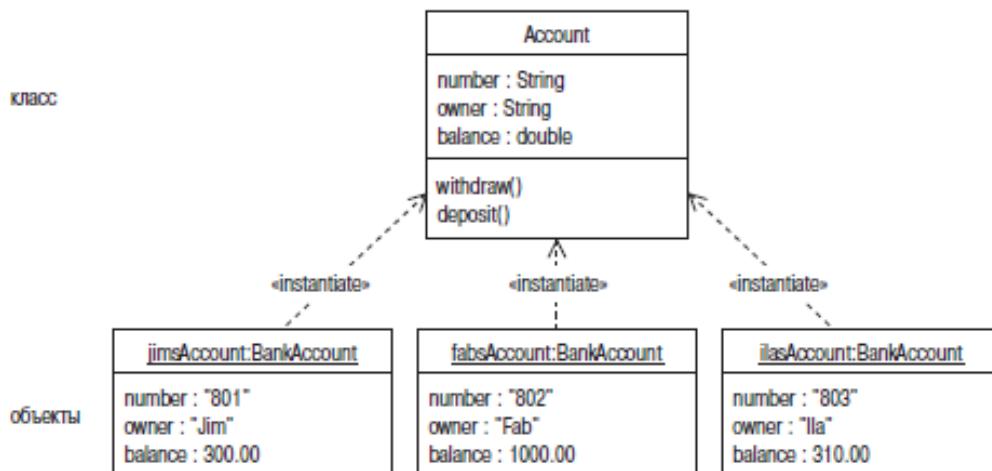


Рис. 7.6. Отношение «instantiate»

Всё, что находится во французских кавычках ("..."), называется **стереотипом**.

Стереотипы – один из трех механизмов расширения в UML. **Стереотип** – это способ настройки элементов модели, способ создания вариантов с новой семантикой. В данном случае стереотип «**«instantiate»**» превращает обычную зависимость в отношение конкретизации между классом и объектами этого класса.

Создание экземпляров (instantiation) – это создание новых экземпляров элементов модели. В данном случае создаются объекты классов. Создаются новые экземпляры классов. Класс используется как шаблон.

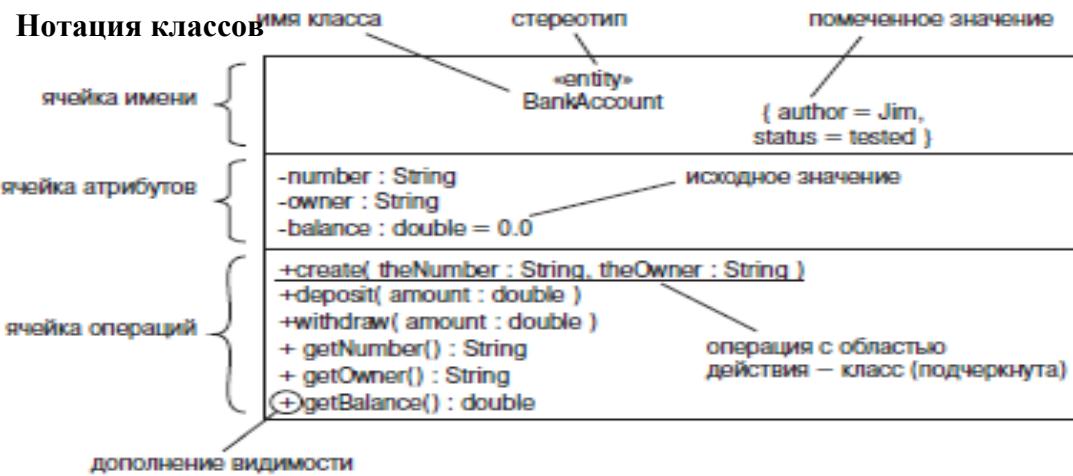


Рис. 7.7. Нотация классов в UML

Видимость

Видимость контролирует доступ к свойствам класса.

+ Public (Открытый)	Любой элемент, который имеет доступ к классу, имеет доступ к любой из его возможностей, видимость которой public .
- Private (Закрытый)	Только операции класса имеют доступ к возможностям, имеющим видимость private .
# Protected (Защищенный)	Только операции класса или потомка класса имеют доступ к возможностям, имеющим видимость protected .
~ Package (Пакетный)	Любой элемент, находящийся в одном пакете с классом или во вложенном подпакете, имеет доступ ко всем его возможностям, видимость которых package .

Тип

Типом атрибута может быть другой класс или примитивный тип.

Таблица 7.5

	Простой тип	Семантика
UML	Integer	Целое число.
	UnlimitedNatural	Целое число ≥ 0 .
	Boolean	Бесконечность обозначается как *.
	String	Может принимать значения true или false.
OCL	Real	Последовательность символов.
	Float	Строковые литералы заключаются в кавычки, например "Джим".

Кратность

Кратность позволяет моделировать коллекции сущностей или неопределенные значения.

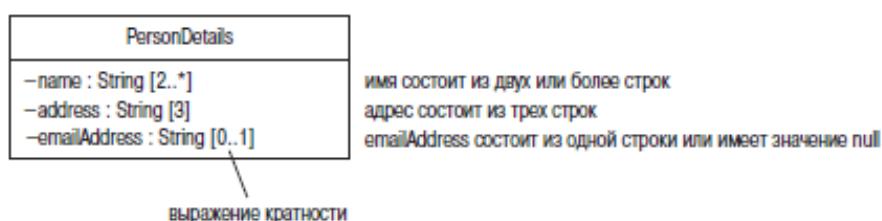


Рис. 7.9. Примеры синтаксиса кратности

Начальное значение

Начальное значение позволяет задавать значение атрибута в момент создания объекта.

Ячейка операции

Операции – это функции, закрепленные за определенным классом.

По сути, они обладают всеми характеристиками функций:

- имя
- список параметров
- возвращаемый тип

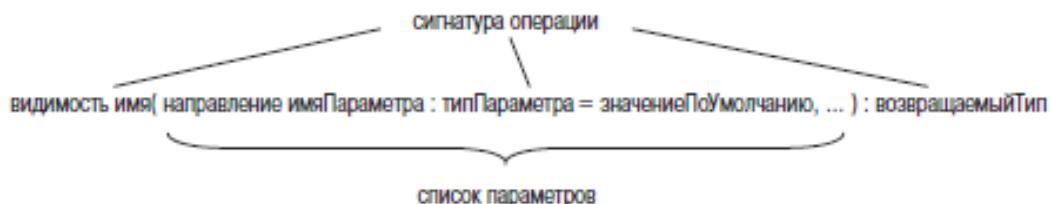


Рис. 7.10. Сигнатура операции

Имена операций записываются в стиле **lowerCamelCase**.

направление имяПараметра : типПараметра = значениеПоУмолчанию

Рис. 7.11. Синтаксис параметров операции

Направление параметра

Таблица 7.6

Параметр	Семантика
in p1:Integer	Применяется по умолчанию. Операция использует p1 как входной параметр. Значение p1 каким-то образом используется операцией. Операция <i>не</i> изменяет p1.
inout p2:Integer	Операция принимает p2 как параметр ввода/вывода. Значение p2 каким-то образом используется операцией. <i>и</i> принимает выходное значение операции. Операция может изменять p2.
out p3:Integer	Операция использует p3 как выходной параметр. Параметр служит хранилищем для выходного значения операции. Операция может изменять p3.
return p4:Integer	Операция использует p4 как возвращаемый параметр. Операция возвращает p4 как одно из своих возвращаемых значений.
return p5:Integer	Операция использует p5 как возвращаемый параметр. Операция возвращает p5 как одно из своих возвращаемых значений.

Операции запроса

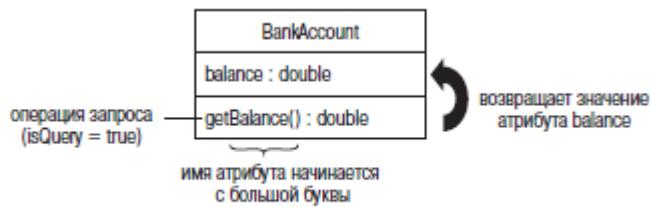


Рис. 7.13. Именование операций запроса

Синтаксис стереотипа класса

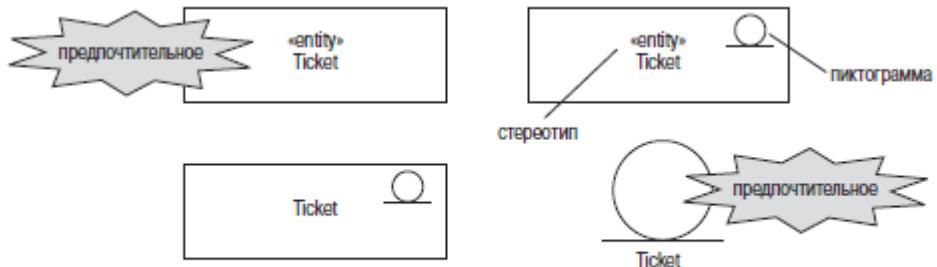


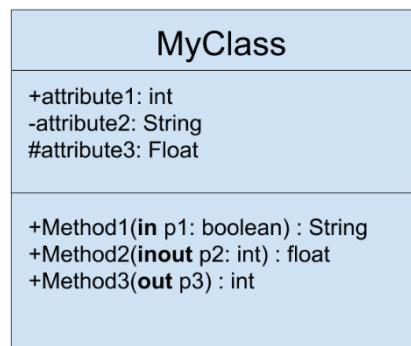
Рис. 7.14. Варианты отображения стереотипов

ВОПРОС №4

Диаграмма классов: ассоциации. Синтаксис и кратность.

Диаграмма классов (англ. class diagram) — структурная диаграмма языка моделирования UML, демонстрирующая общую структуру иерархии классов системы, их коопераций, атрибутов (полей), методов, интерфейсов и взаимосвязей между ними.

UML-нотация диаграммы классов



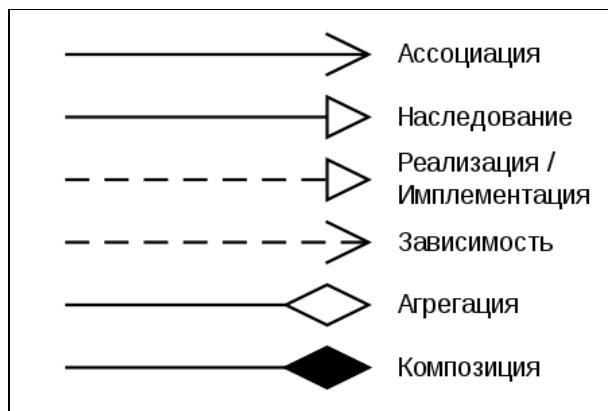
Графически класс изображается в виде прямоугольника, разделенного на 3 блока горизонтальными линиями:

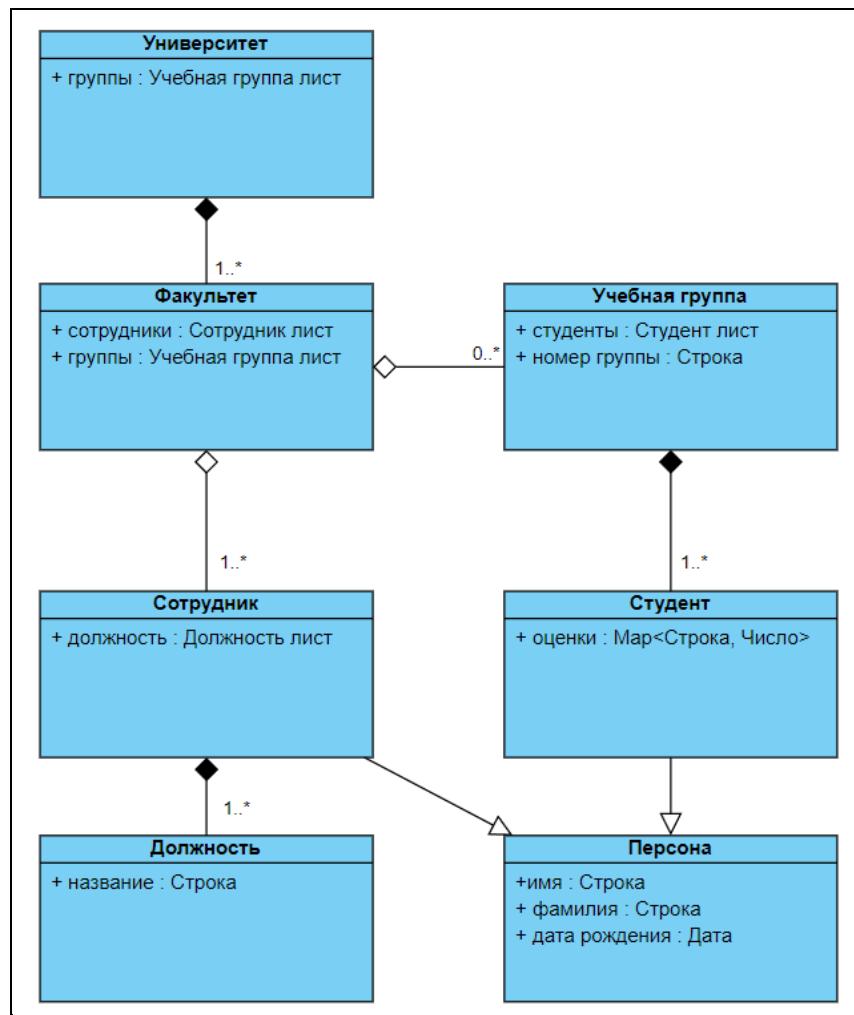
- имя класса;
- атрибуты (свойства) класса;
- операции (методы) класса.

Для атрибутов и операций может быть указан один из трех типов видимости:

- — private (частный)
- # — protected (зашитенный)
- + — public (общий)

Взаимосвязь классов





Агрегация — особая разновидность ассоциации, представляющая структурную связь целого с его частями. Как тип ассоциации, агрегация может быть именованной. Одно отношение агрегации не может включать более двух классов (контейнер и содержимое). Агрегация встречается, когда один класс является коллекцией или контейнером других. Причём, по умолчанию агрегацией называют агрегацию по ссылке, то есть когда время существования содержащихся классов не зависит от времени существования содержащего их класса. Если контейнер будет уничтожен, то его содержимое — нет.

Графически агрегация представляется пустым ромбом на блоке класса «целое», и линией, идущей от этого ромба к классу «часть».

Композиция — более строгий вариант агрегации. Известна также как агрегация по значению.

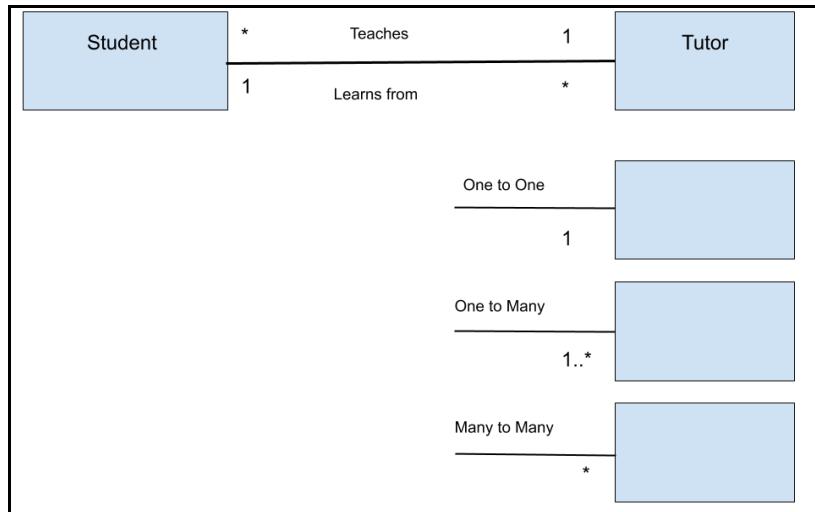
Композиция — это форма агрегации с четко выраженным отношениями владения и совпадением времени жизни частей и целого. Композиция имеет жёсткую зависимость времени существования экземпляров класса контейнера и экземпляров содержащихся классов. Если контейнер будет уничтожен, то всё его содержимое будет также уничтожено.

Графически представляется как и агрегация, но с закрашенным ромбиком.

Обобщение — выражает специализацию или наследование, в котором специализированный элемент (потомок) строится по спецификациям обобщенного элемента (родителя). Потомок разделяет структуру и поведение родителя. Графически обобщение представлено в виде сплошной линии с пустой стрелкой, указывающей на родителя.

Реализация – это семантическая связь между классами, когда один из них (поставщик) определяет соглашение, которого второй (клиент) обязан придерживаться. Это связи между интерфейсами и классами, которые реализуют эти интерфейсы. Это, своего рода, отношение «целое-часть». Поставщик, как правило, представлен абстрактным классом. В графическом исполнении связь реализации – это гибрид связей обобщения и зависимости: треугольник указывает на поставщика, а второй конец пунктирной линии – на клиента.

Кратность связей



Один к одному

Пример: у студента может быть только один студенческий билет, равно как и студенческий билет может принадлежать только одному студенту.

Один ко многим

Пример: в учебной группе может быть множество студентов, однако студент может быть только в одной группе.

Многие к одному

Тоже самое, только смотрим на отношение “Один ко многим” с другой стороны.

Многие ко многим

Пример: преподаватель может вести занятия у нескольких групп, при этом у каждой группы несколько преподавателей ведут занятия.

Источники

<https://www.visual-paradigm.com/guide/uml-unified-modeling-language/uml-class-diagram-tutorial/>

https://ru.wikipedia.org/wiki/Диаграмма_классов

<https://developer.ibm.com/articles/the-class-diagram/>

ВОПРОС №5

Диаграмма классов: ассоциации и атрибуты. Возможность навигации. (RUP&UML2 Главы 9.4.3, 9.4.4.)

Возможность навигации.

Возможность навигации указывает на возможность прохода от объекта исходного класса к одному или более объектам в зависимости от кратности целевого класса. Смысл навигации в том, что «сообщения могут посыпаться только в направлении, в котором указывает стрелка».

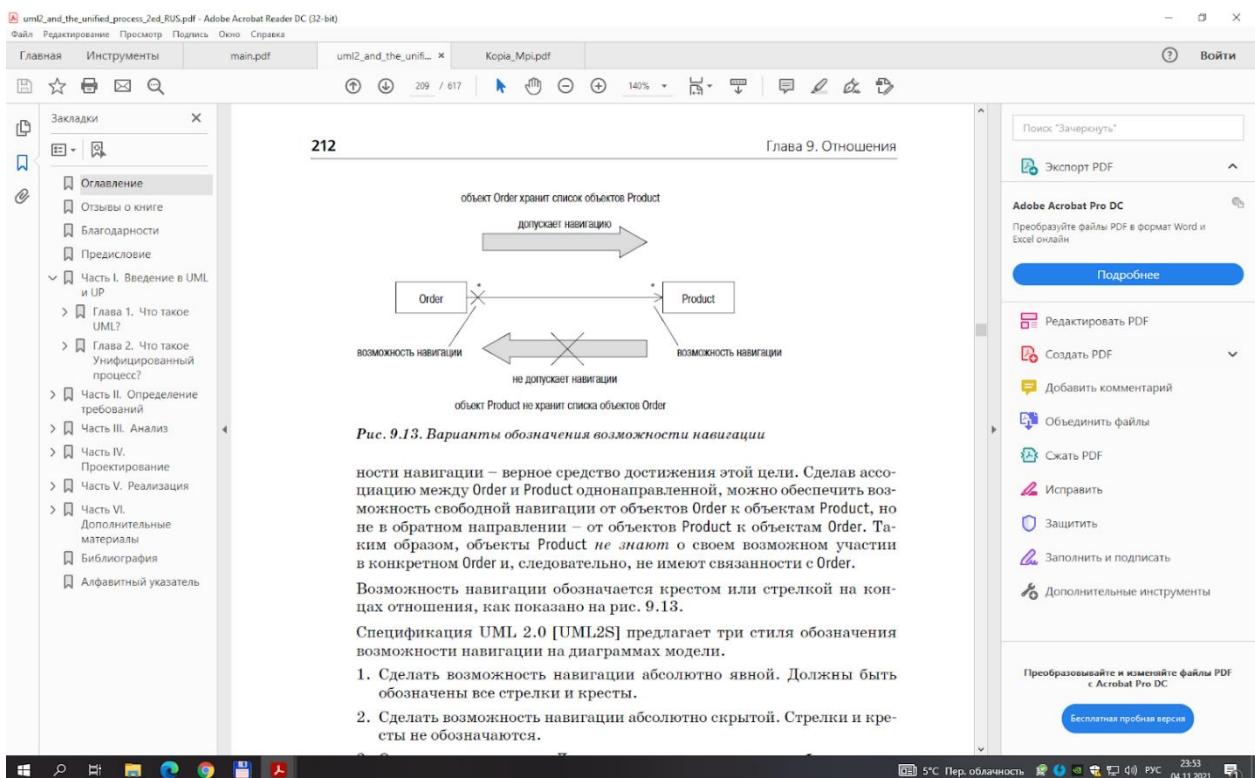


Рис. 9.13. Варианты обозначения возможности навигации

ности навигации – верное средство достижения этой цели. Сделав ассоциацию между Order и Product односторонней, можно обеспечить возможность свободной навигации от объектов Order к объектам Product, но не в обратном направлении – от объектов Product к объектам Order. Таким образом, объекты Product *не знают* о своем возможном участии в конкретном Order и, следовательно, не имеют связанных с Order.

Возможность навигации обозначается крестом или стрелкой на концах отношения, как показано на рис. 9.13.

Спецификация UML 2.0 [UML2S] предлагает три стиля обозначения возможности навигации на диаграммах модели.

1. Сделать возможность навигации абсолютно явной. Должны быть обозначены все стрелки и кrestы.
2. Сделать возможность навигации абсолютно скрытой. Стрелки и кресты не обозначаются.

Сделав ассоциацию между Order и Product односторонней, можно обеспечить возможность свободной навигации от объектов Order к объектам Product, но не в обратном направлении – от объектов Product к объектам Order. Возможность навигации обозначается крестом или стрелкой на концах отношения.

Спецификация UML 2.0 предлагает три стиля обозначения возможности навигации на диаграммах модели:

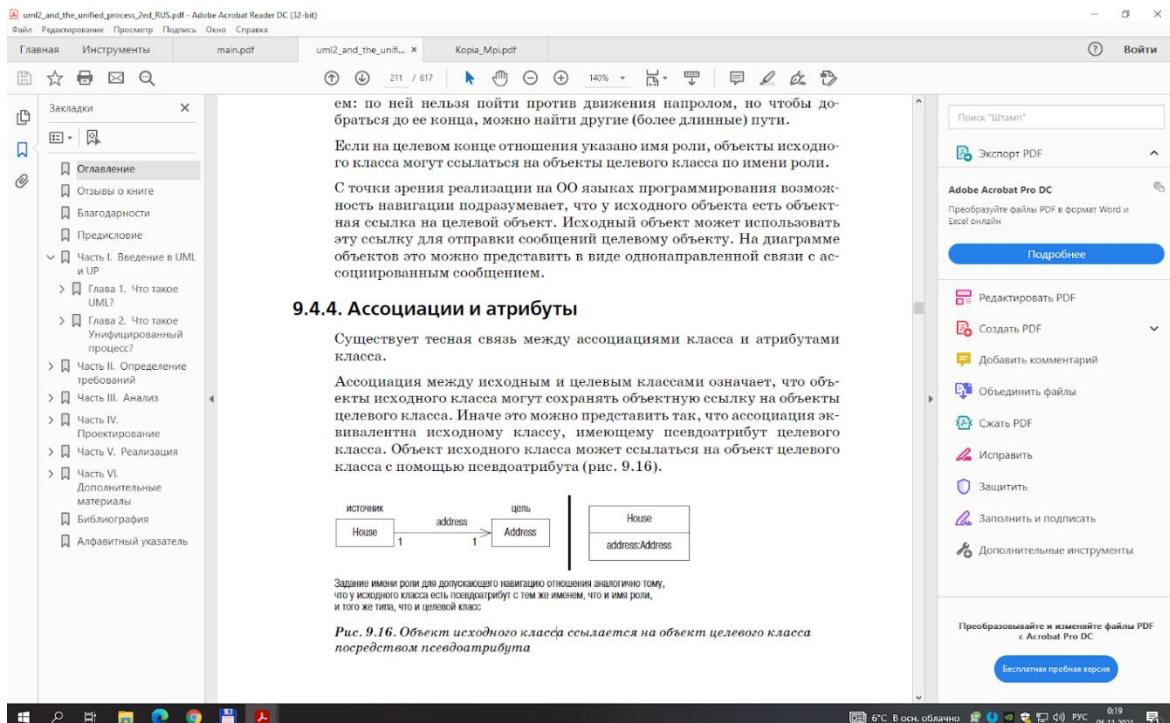
1. Сделать возможность навигации абсолютно явной. Должны быть обозначены все стрелки и кресты.
2. Сделать возможность навигации абсолютно скрытой. Стрелки и кресты не обозначаются.
3. Опускать все кресты. Двунаправленная ассоциация обозначается без стрелок. Односторонняя ассоциация обозначается с одной стрелкой.

На практике чаще всего используется стиль 3. Основные преимущества стиля 3: при его использовании диаграммы не загромождаются слишком большим количеством стрелок и крестов; он обратно совместим с предыдущими версиями UML.

С точки зрения реализации на ОО языках программирования возможность навигации подразумевает, что у исходного объекта есть объектная ссылка на целевой объект. Исходный объект может использовать эту ссылку для отправки сообщений целевому объекту.

Ассоциации и атрибуты.

Ассоциация между исходным и целевым классами означает, что объекты исходного класса могут сохранять объектную ссылку на объекты целевого класса. Иначе это можно представить так, что ассоциация эквивалентна исходному классу, имеющему псевдоатрибут целевого класса. Объект исходного класса может ссылаться на объект целевого класса с помощью псевдоатрибута.



```
public class House
{
    private Address address;
}
```

Нет широко используемого ОО языка программирования, имеющего специальную языковую конструкцию для поддержки ассоциаций. Поэтому при автоматическом генерировании кода из UML модели ассоциации один-к-одному превращаются в атрибуты исходного класса.

Если кратность целевого класса больше 1, она реализуется одним из следующих способов:

- как атрибут типа array (массив) (конструкция, поддерживаемая большинством языков программирования);
- как атрибут некоторого типа, являющегося коллекцией.

Ассоциации используются только тогда, когда целевой класс является важной частью модели, в противном случае отношения моделируются с помощью атрибутов. Важными являются бизнесклассы, описывающие часть прикладной области. Не имеющими большого значения классами являются компоненты библиотеки, такие как классы String, Date и Time.

Если кратность целевого класса больше 1, это ясно показывает, что целевой класс важен для модели. Таким образом, для моделирования такого отношения используются ассоциации.

Если кратность целевого класса равна 1, целевой объект может быть только частью исходного, и поэтому не стоит показывать отношение с ним как ассоциацию. Лучше смоделировать его как атрибут.

ВОПРОС №6

Диаграмма классов: агрегация и композиция. (RUP&UML2 Главы 18.3, 18.4, 18.5.)

Агрегация – это тип отношения целое-часть, в котором агрегат образуется многими частями. В отношении целое-часть один объект (целое) использует сервисы другого объекта (части).



Как итог:

- Агрегат может существовать как независимо от частей, так и вместе с ними.
- Части могут существовать независимо от агрегата.
- Агрегат является в некотором смысле неполным в случае отсутствия некоторых частей.
- Части могут принадлежать одновременно нескольким агрегатам.

Агрегация:

- транзитивна – если С является частью В, и В является частью А, тогда С также является частью А
- асимметрична – объект никогда не может быть частью самого себя

Композиция – более строгая форма агрегации. Она имеет сходную (но более ограниченную) семантику. Как и агрегация, это отношение целое-часть, являющееся как транзитивным, так и асимметричным.



Ключевое различие между агрегацией и композицией в том, что в композиции у частей нет независимой жизни вне целого. Композит имеет исключительное право владения и ответственности за свои части. Часть композита эквивалентна атрибуту объекта.

Как итог:

- Каждая часть принадлежит максимум одному и только одному целому.
- Композит обладает исключительной ответственностью за все свои части, это означает, что он отвечает за их создание и уничтожение.
- Композит может высвобождать части, передавая ответственность за них другому объекту.
- В случае уничтожения композита он должен или уничтожить все свои части, или передать ответственность за них другому объекту.

ВОПРОС №7

Диаграмма классов: наследование, абстрактные классы, множественное наследование. (RUP&UML2 Глава 10.3.)

I. НАСЛЕДОВАНИЕ

Наследование -- концепция объектно-ориентированного программирования, согласно которой абстрактный тип данных может наследовать данные и функциональность некоторого существующего типа, способствуя повторному использованию компонентов программного обеспечения.

Посредством наследования подклассы наследуют все возможности своих суперклассов. Чтобы быть более специальными, подклассы наследуют:

- атрибуты;
- операции;
- отношения;
- ограничения.

В примере на рис. 10.3 подклассы Square и Circle класса Shape наследуют все его атрибуты, операции и ограничения. Это означает, что хотя мы и не видим этих элементов в подклассах, они присутствуют в них неявно. Говорят, что Square и Circle типа Shape.

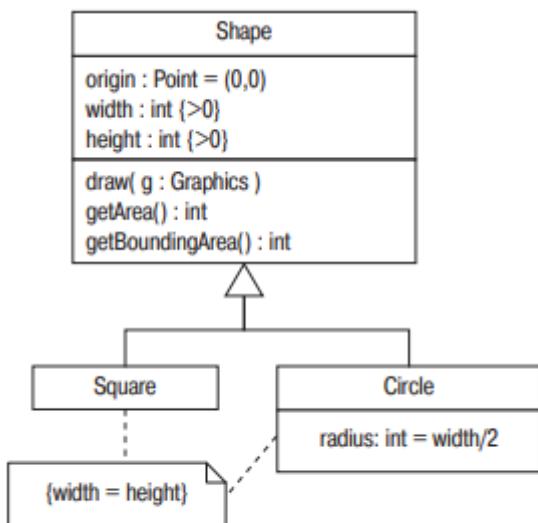


Рис. 10.3. Наследование характеристик суперкласса

Подклассы также могут вводить новые возможности и переопределять операции суперкласса. Подклассы всегда должны представлять «особую разновидность чего-либо», а не «выполняемую роль».

Чтобы переопределить операцию надкласса, подкласс должен предоставить операцию с точно такой же сигнатурой, что и у переопределяемой операции надкласса. UML определяет сигнатуру операции как имя операции, ее возвращаемый тип и типы всех параметров в порядке перечисления. Имена параметров не учитываются, поскольку они являются просто удобным способом обращения к определенному параметру в теле операции и поэтому не считаются частью сигнатуры.

Важно знать, что разные языки программирования могут по-разному определять «сигнатуру операции». Например, в C++ и Java возвращаемый тип операции не является частью сигнатуры операции. Таким образом, если операции подкласса и надкласса будут отличаться только возвращаемым типом, в этих языках будет сформирована ошибка компилятора или интерпретатора.

На рис. 10.4 все это показано в действии: подклассы Square и Circle предоставили собственные операции draw() и getArea(), имеющие соответствующее поведение.

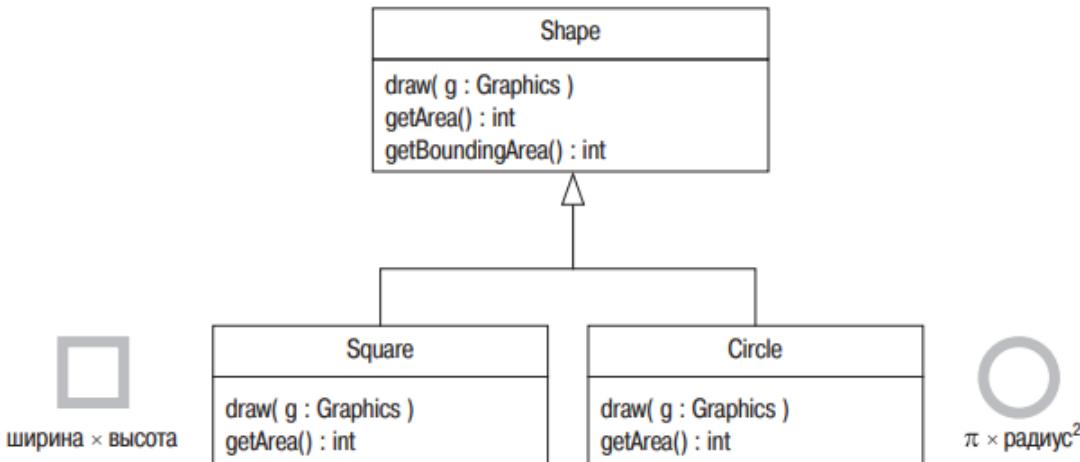


Рис. 10.4. Переопределение унаследованных операций

Наследование имеет определенные нежелательные характеристики:

- Это самая строгая из возможных форм связности двух или более классов.
- В иерархии классов инкапсуляция низкая. Изменения базового класса передаются вниз по иерархии и приводят к изменениям подклассов. Это явление называют проблемой «хрупкости базового класса», когда изменения базового класса имеют огромное влияние на другие классы системы.
- Это очень жесткий тип отношения. Во всех широко используемых ОО языках программирования отношения наследования постоянны во время выполнения. Создавая и уничтожая отношения во время выполнения, можно изменять иерархии агрегации и композиции, но иерархии наследования остаются неизменными.

II. АБСТРАКТНЫЕ КЛАССЫ

Отсутствие реализации операции можно обозначить, сделав ее абстрактной операцией. В UML для этого имя операции просто записывается курсивом.

Абстрактные классы имеют одну или более абстрактных операций. Создать экземпляр абстрактного класса невозможно. Чтобы показать, что класс является абстрактным, его имя записывается курсивом.

В примере на рис. 10.5 абстрактный класс Shape имеет две абстрактные операции: Shape::draw(g : Graphics) и Shape::getArea() : int. Эти операции реализуются подклассами Square и Circle. Хотя Shape является неполным, и его экземпляр не может быть создан, оба его подкласса предоставляют недостающие реализации, являются полными и могут иметь экземпляры. Любой класс, экземпляр которого может быть создан, называется конкретным

классом. Операция `getBoundingArea()` является конкретной операцией класса `Shape`, потому что контактная площадь (bounding area) любой фигуры вычисляется одинаково: ширина фигуры умножается на высоту.

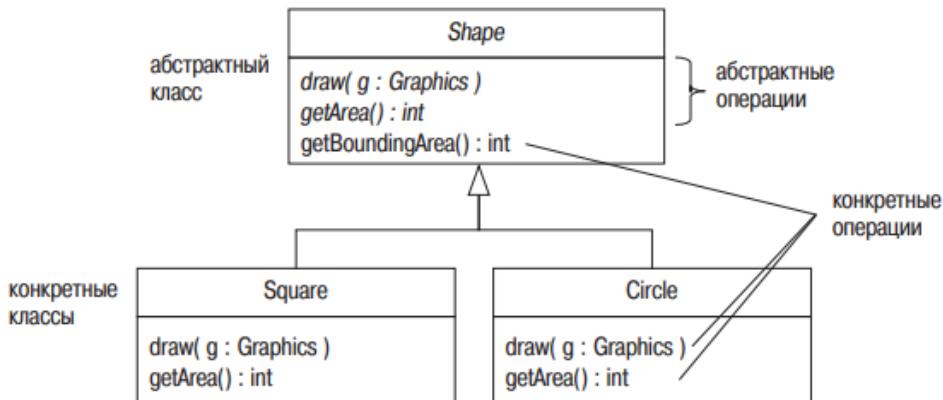


Рис. 10.5. Абстрактный класс Shape и конкретные подклассы Square и Circle

Использование абстрактных классов и операций обеспечивает два серьезных преимущества:

- В абстрактном суперклассе можно определять ряд абстрактных операций, которые должны быть реализованы всеми подклассами `Shape`. Это можно рассматривать как определение «контракта», который должны реализовать все конкретные подклассы `Shape`.
- Можно написать код управления фигурами и затем подставить `Circle`, `Square` и другие подклассы `Shape` соответственно. Согласно принципу замещаемости код, написанный для управления `Shape`, должен работать для всех подклассов `Shape`.

Важно помнить, что сущности, располагающиеся на одном уровне иерархии обобщения, должны находиться на одном уровне абстракции.

III. МНОЖЕСТВЕННОЕ НАСЛЕДОВАНИЕ

UML позволяет классу иметь несколько непосредственных надклассов. Это называется множественным наследованием (multiple inheritance). Подкласс наследуется от всех его непосредственных надклассов.

Основные положения множественного наследования:

- Все участвующие родительские классы должны быть семантически не связанными. В случае наличия какого-либо совмещения в семантике базовых классов между ними возможны непредвиденные взаимодействия. Это может привести к необычному поведению подкласса.
- Между подклассом и всеми его суперклассами должны действовать принцип замещаемости и принцип «является разновидностью».
- У суперклассов не должно быть общих родителей. В противном случае в иерархии наследования образуется цикл и возникает множество путей наследования одних и тех же возможностей от более абстрактных классов.

Один из общепринятых способов эффективного использования множественного наследования – «смешанный» (mixin) класс. Эти классы не являются по-настоящему автономными классами. Они разрабатываются специально для того, чтобы быть «смешанными» с другими классами с помощью наследования. На рис. 17.8 класс Dialer (набиратель номера) – простой смешанный класс. Его единственная функция – набор телефонного номера.



Рис. 17.8. Смешанный класс Dialer

ВОПРОС №8

Диаграмма классов: квалифицированные ассоциации, классы ассоциаций. (RUP&XYIUML2 Главы 9.4.5, 9.4.6.)

9.4.5. Классы ассоциации

В ОО моделировании распространена следующая проблема: когда между классами установлено отношение многие-ко-многим, встречаются такие атрибуты, которые не удается поместить ни в один из классов.

Пример 1:

Безобидная модель:

- *каждый человек (объект Person) может работать во многих компаниях (объект Company);*
- *каждая компания (Company) может нанимать много людей (объект Person).*

Однако, что происходит, если добавить бизнес-правило, заключающееся в том, что каждый Person получает зарплату в каждой нанявшей его Company? Где должна быть записана эта зарплата: в классе Person или в классе Company? Действительно, нельзя сделать зарплату Person атрибутом класса Person, потому что каждый экземпляр Person может работать на многие Company и в каждой получать разную зарплату. Аналогично нельзя сделать зарплату атрибутом Company, поскольку каждый экземпляр Company нанимает множество Person, зарплата которых может быть разной. Решение кроется в том, что зарплата на самом деле является собственностью самой ассоциации. У каждой ассоциации найдя устанавливаемой между объектом Person и объектом Company, своя индивидуальная зарплата.



Рис. 9.18. Отношение многие-ко-многим

UML позволяет моделировать эту ситуацию с помощью класса-ассоциации. Класс-ассоциация – это линия ассоциации (включая все имена ролей и кратности), пунктирная нисходящая линия и прямоугольник класса на конце пунктирной линии.

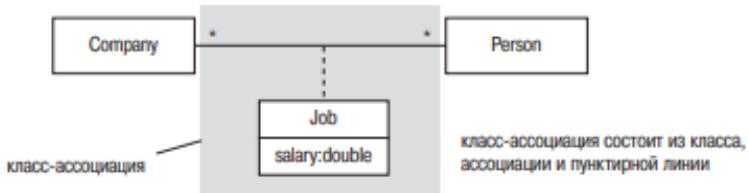


Рис. 9.19. Класс-ассоциация

Как показано на рисунке 9.19 все что входит в затемненную область, является классом ассоциацией.

Класс-ассоциация – это ассоциация, являющаяся еще и классом. Она соединяет два класса, как обычная ассоциация, также определяет набор характеристик, принадлежащих самой ассоциации. У классов-ассоциаций могут быть атрибуты, операции и другие ассоциации.

Класс-ассоциация означает, что в любой момент времени между любыми двумя объектами может существовать только одна связь.

Экземпляры класса-ассоциации – это связи, у которых есть атрибуты и операции. Уникальная идентификация этих связей определяется индивидуальностью объектов, находящихся на каждом конце. Этот фактор ограничивает семантику класса-ассоциации: его можно использовать только тогда, когда между двумя объектами в любой момент времени установлена единственная уникальная связь. Это обусловлено тем, что каждая связь, которая является экземпляром класса-ассоциации, должна быть уникальной.

Пример 2:

На рис. 9.19 применение класса-ассоциации означает, что на модель накладывается следующее ограничение: для данного объекта Person и данного объекта Company может существовать только один объект Job (должность). Иначе говоря, каждый Person может занимать только одну Job в данной Company. Однако если ситуация такова, что данный объект Person может занимать несколько Job в данном объекте Company, класс-ассоциацию использовать нельзя – семантика не соответствует. Но нам по-прежнему надо куда-то сохранять зарплату для каждой группы Company/Job/Person. Поэтому мы материализуем (делаем реальным) отношение, представляя его в виде обычного класса. На рис. 9.20 Job является обычным классом. Как видите, у Person может быть несколько Job, каждая Job в каждой конкретной Company.

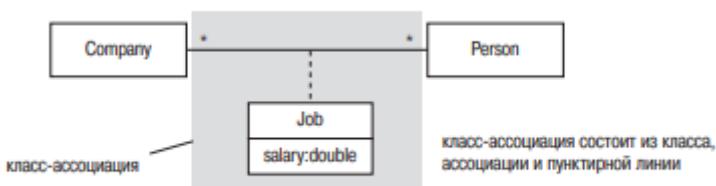


Рис. 9.19. Класс-ассоциация

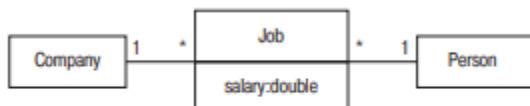


Рис. 9.20. Материализованное отношение в виде обычного класса

Материализованное отношение делает возможным существование более одной связи между любыми двумя объектами в конкретный момент времени.

Разница между классами-ассоциациями и материализованными отношениями: классы-ассоциации могут использоваться только в том случае, когда каждая связь имеет уникальную индивидуальность. Индивидуальность связи определяется индивидуальностью объектов, располагающихся на ее концах.

9.4.6. Квалифицированные ассоциации

Квалифицированные ассоциации могут использоваться для превращения ассоциации п-ко-многим в ассоциацию п-к-одному путем задания одного объекта (или группы объектов) из целевого набора.

Пример:

Рассмотрим модель, изображенную на рис. 9.21. Объект *Club* (клуб) связан с набором объектов *Member* (член), а объект *Member* подобным же образом связан с только одним объектом *Club*.

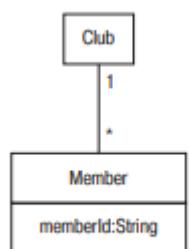


Рис. 9.21. Отношение один-ко-многим

Квалифицированная ассоциация выбирает один объект из целевого набора.

Возникает следующий вопрос: как от объекта *Club*, связанного с набором объектов *Member*, перейти к одному конкретному объекту *Member*? Очевидно, что необходим некоторый уникальный ключ, который можно использовать для поиска определенного объекта *Member* из набора. Такой ключ называют *квалификатором* (*qualifier*). Квалификаторы могут быть разными (имя, номер кредитной карточки, номер социальной страховки). В приведенном выше примере у каждого объекта *Member* есть значение атрибута *memberId*, уникальное для данного объекта. Это и есть ключ поиска в данной модели. В модели такой поиск можно обозначить путем добавления квалификатора на конце ассоциации со стороны *Club*. Важно

понимать, что этот квалификатор принадлежит концу ассоциации, а не классу *Club*. Этот квалификатор задает уникальный ключ и таким образом превращает отношение один-ко-многим в отношение один-к-одному, как показано на рис. 9.22.

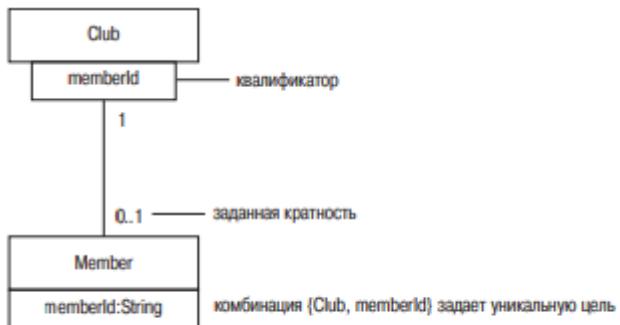


Рис. 9.22. Квалификатор превращает отношение один-ко-многим в отношение один-к-одному

Квалифицированные ассоциации – это способ показать, как с помощью уникального ключа происходит выбор определенного объекта из набора. Квалификаторы обычно ссылаются на атрибут целевого класса, но возможны и другие выражения, с помощью которых выбирается один объект из набора.

ВОПРОС №9

Вопрос: Выражение зависимости в диаграммах. (RUP&UML2 Глава 9.5. (Стр. 219))

Ответ: Зависимость отношение между двумя или более элементами модели, при котором изменение одного элемента может повлиять или предоставить информацию другому элементу.

В отношении зависимости клиент некоторым образом зависит от поставщика.

Три типа зависимостей (больше всего используется use из типа usage)

Тип	Семантика
Usage (Использование)	Клиент использует некоторые из доступных сервисов поставщика для реализации собственного поведения
Abstraction (Абстракция)	Отношения между клиентом и поставщиком, где поставщик “абстрактнее” (н-р.: поставщик на другой стадии разработки)
Permission (Доступ)	Поставщик предоставляет клиенту разрешение на доступ к своему содержимому - позволяет поставщику контролировать и ограничивать доступ к своему содержимому

Зависимости могут существовать между:

- классами и классами
 - пакетами и пакетами
 - объектами и классами

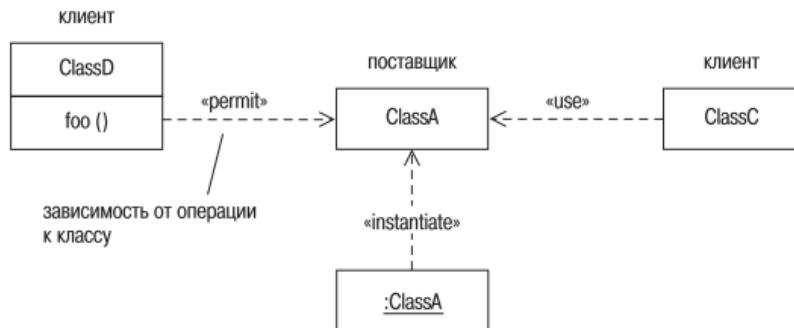


Рис. 9.23. Типы зависимостей

Зависимости usage:

“Use”: обозначает, что клиент каким-то образом использует поставщика

“Call”: это когда операция-клиент вызывает операцию-поставщика.

“Parameter”: поставщик является параметром операции клиента.

“Send”: операция посылающая поставщика в некоторую неопределенную цель.

“Instantiate” Клиент - это экземпляр поставщика

Зависимости abstraction:

“Trace”: отношения, где поставщик и клиент предоставляют одно понятие, но в разных моделях. Также для функциональных требований (Банкомат разрешает снять деньги до тех пор, пока они у вас на карте).

“Substitute”: клиент может заменять поставщика.

“Refine”: trace - это для разных моделей, а refine - это для элементов одной модели.

“Derive”: показывает возможность получения одной сущности как производной от другой.

Зависимости permission:

“Access”: разрешает одному пакету доступ ко всему открытому содержимому другого пакета.

“Import”: аналог access, но пространство имен поставщика объединяется с пространством имен клиента

“Permit”: обеспечивает возможность управляемого нарушения инкапсуляции.

ВОПРОС №10

Диаграмма последовательности. Элементы диаграммы. (RUP&UML2 Глава 12.9.)

Диаграммы последовательностей представляют взаимодействия между линиями жизни как упорядоченную последовательность событий.

Согласно спецификации UML 2, имена диаграмм взаимодействий могут начинаться с приставки sd, для обозначения того, что данная диаграмма является **диаграммой взаимодействий**.

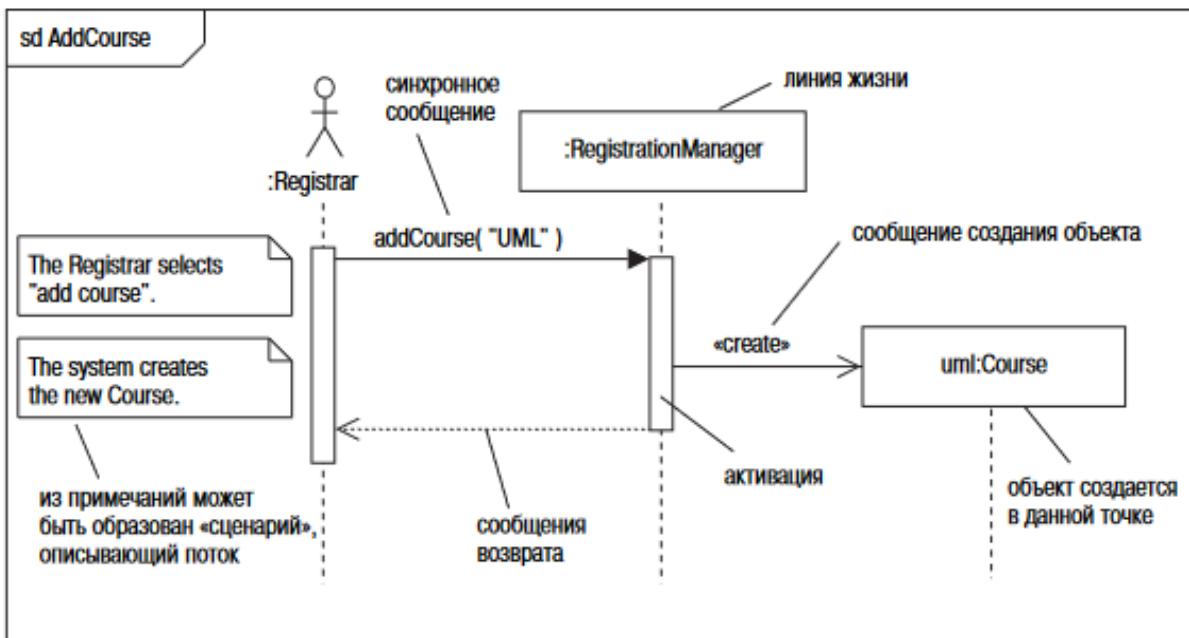


Рис. 1 Диаграмма последовательности AddCourse

Рассмотрим диаграмму последовательностей, приведенную на рис.1. Время на диаграммах последовательностей развивается сверху вниз, линии жизни выполняются слева направо. Линии жизни размещаются горизонтально, чтобы минимизировать число пересекающихся линий на диаграмме. Их местоположение относительно вертикальной оси отражает момент их создания. Пунктирная линия, находящаяся внизу, показывает существование линии жизни в течение времени.

Уничтожение объекта обозначается большим крестом, завершающим линию жизни, как показано на рис. 2. Если момент уничтожения объекта неизвестен или неважен, линия жизни завершается без креста.

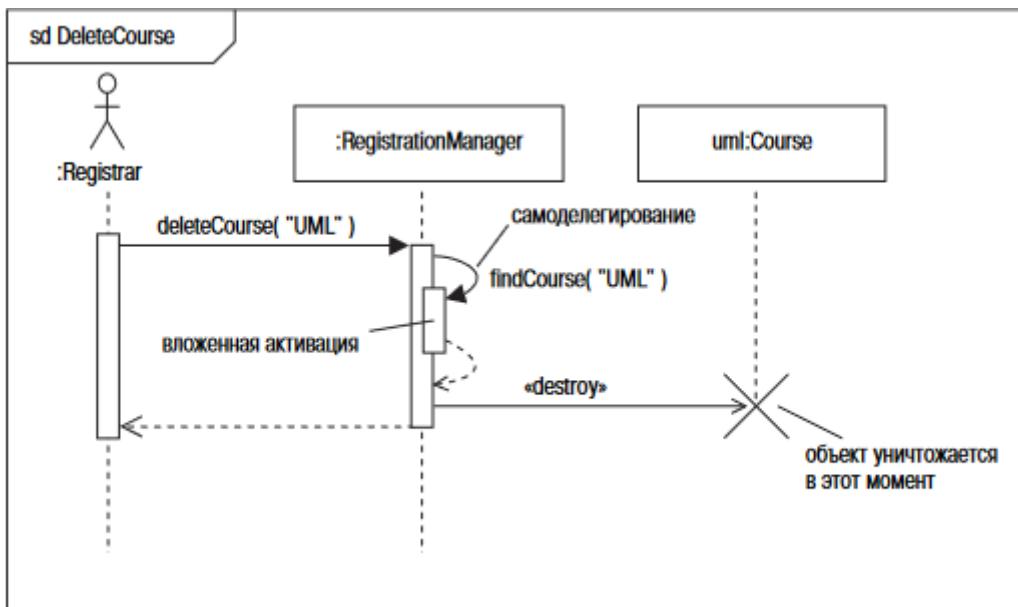


Рис. 2 Диаграмма последовательности DeleteCourse

Активации

Вытянутые прямоугольники, расположенные на пунктирной линии под линией жизни, показывают, когда на данной линии жизни находится фокус управления. Эти прямоугольники называются активациями, или фокусом управления.

Активации показывают, когда фокус управления располагается в данной линии жизни.

Сценарий может состоять из фактических шагов прецедента или краткого обзора того, что происходит на диаграмме, представленного в текстовой форме.

Инварианты состояния и ограничения

Сообщение, получаемое экземпляром, может обусловить изменение его состояния. Состояние определяется как «условие или ситуация в ходе жизни объекта, в течение которой он удовлетворяет некоторому условию, осуществляет некоторую деятельность или ожидает некоторое событие».

Рисунок 3 иллюстрирует применение ограничений. Они записываются в фигурных скобках и размещаются на (или рядом) линиях жизни.

Ограничение, обозначенное на линии жизни, указывает на то, что должно быть истинным для экземпляров, начиная с этого момента и далее. Ограничения часто формулируются на естественном языке. На линию жизни может быть помещен любой тип ограничения. Широко распространены ограничения значений атрибутов экземпляров.

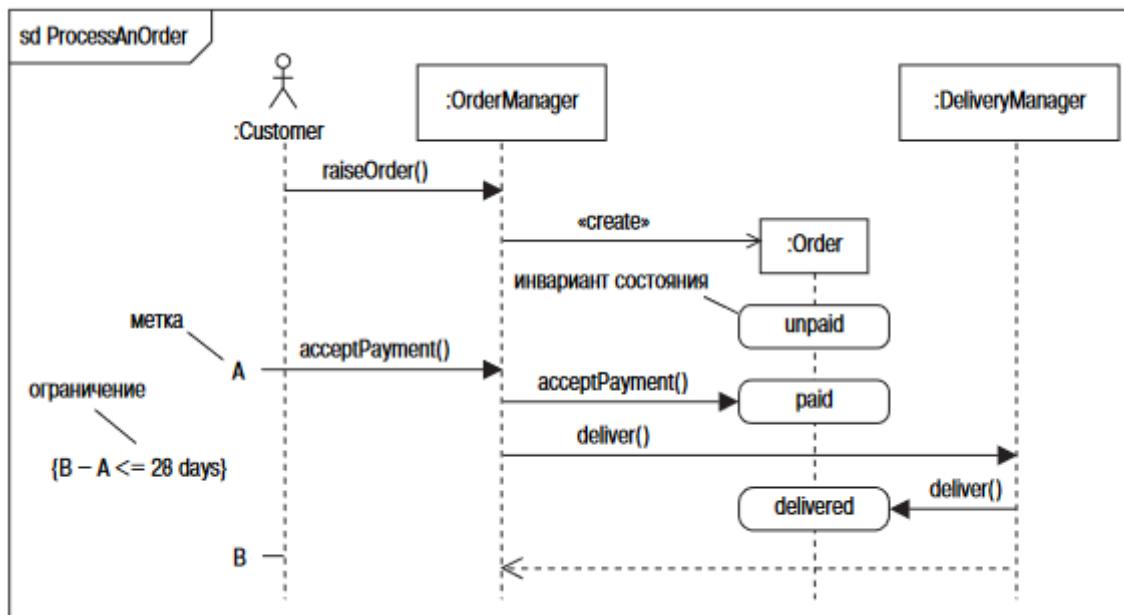


Рис. 3 Диаграмма последовательности ProcessAnOrder

ВОПРОС №11

Диаграмма последовательности. Комбинированные фрагменты и операторы. (RUP&UML2 Глава 12.10.)

Диаграммы последовательностей можно разделить на области, называемые комбинированными фрагментами. Рис. 1 иллюстрирует довольно богатый синтаксис комбинированного фрагмента. Комбинированные фрагменты разделяют диаграмму последовательностей на области с различным поведением.

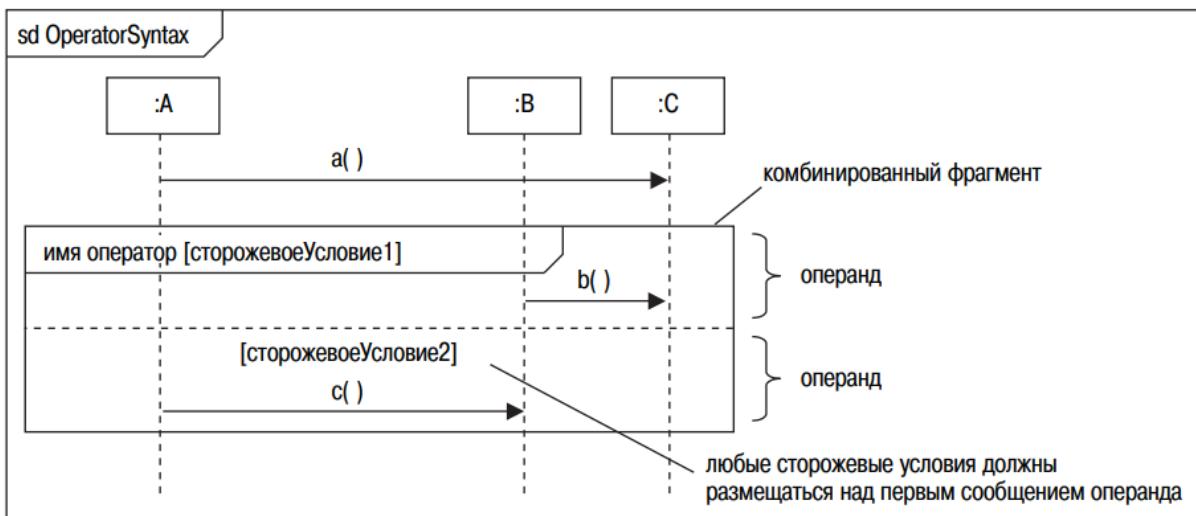


Рис.1 - Синтаксис комбинированного фрагмента

У каждого комбинированного фрагмента есть один оператор, один или более operandов и нуль или более сторожевых условий. Оператор определяет, как исполняются его operandы. Сторожевые условия определяют, будут ли исполняться эти operandы. Сторожевое условие – это логическое выражение, и operand исполняется тогда и только тогда, когда это выражение истинно. Одно сторожевое условие может применяться ко всем operandам или каждый operand может иметь собственное уникальное сторожевое условие.

Полный список операторов приведен в табл. 1. Самые важные из них: opt, alt, loop, break, ref, par и critical.

Таблица 1

Оператор (Полное имя)	Семантика
opt (option)	Единственный operand выполняется, если условие истинно (как if ... then).
alt (alternatives)	Выполняется тот operand, условие которого истинно. Вместо логического выражения может использоваться ключевое слово else (как select ... case).
loop	Имеет специальный синтаксис: loop min, max [condition] повторять минимальное количество раз, затем, пока условие истинно, повторять еще (max – min) число раз.

break	Если сторожевое условие истинно, выполняется операнд, а не все остальное взаимодействие, в которое он входит.
ref (reference)	Комбинированный фрагмент ссылается на другое взаимодействие.
par (parallel)	Все операнды выполняются параллельно.
critical	Операнд выполняется автоматически без прерывания.
seq (weak sequencing)	Все операнды выполняются параллельно при условии выполнения следующего ограничения: последовательность поступления событий на одну линию жизни от разных operandов такая же, как и последовательность operandов. В результате наблюдается слабая форма упорядочения; отсюда название (weak sequencing – слабое упорядочение).
strict (strict sequencing)	Операнды выполняются в строгой последовательности.
neg (negative)	Операнд демонстрирует неверные взаимодействия. Применяется, когда необходимо показать, что не должно произойти.
ignore	Перечисляет сообщения, которые намеренно исключены из взаимодействия. Разделенный запятыми список имен проигнорированных сообщений в фигурных скобках помещается после имени оператора, например {m1, m2, m3}. Например, взаимодействие может представлять тестовый пример, в котором принято решение игнорировать некоторые сообщения.
consider	Перечисляет сообщения, намеренно включенные во взаимодействие. Разделенный запятыми список имен сообщений в фигурных скобках помещается после имени оператора. Например, взаимодействие может представлять тестовый пример, в который решено включить подмножество возможных сообщений.
assert (assertion)	Операнд является единственным допустимым поведением в данный момент взаимодействия, любое другое поведение было бы ошибочным. Используется как средство обозначения того, что некоторое поведение должно иметь место в определенной точке взаимодействия.

Рисунок 2 иллюстрирует синтаксис операторов opt и alt.

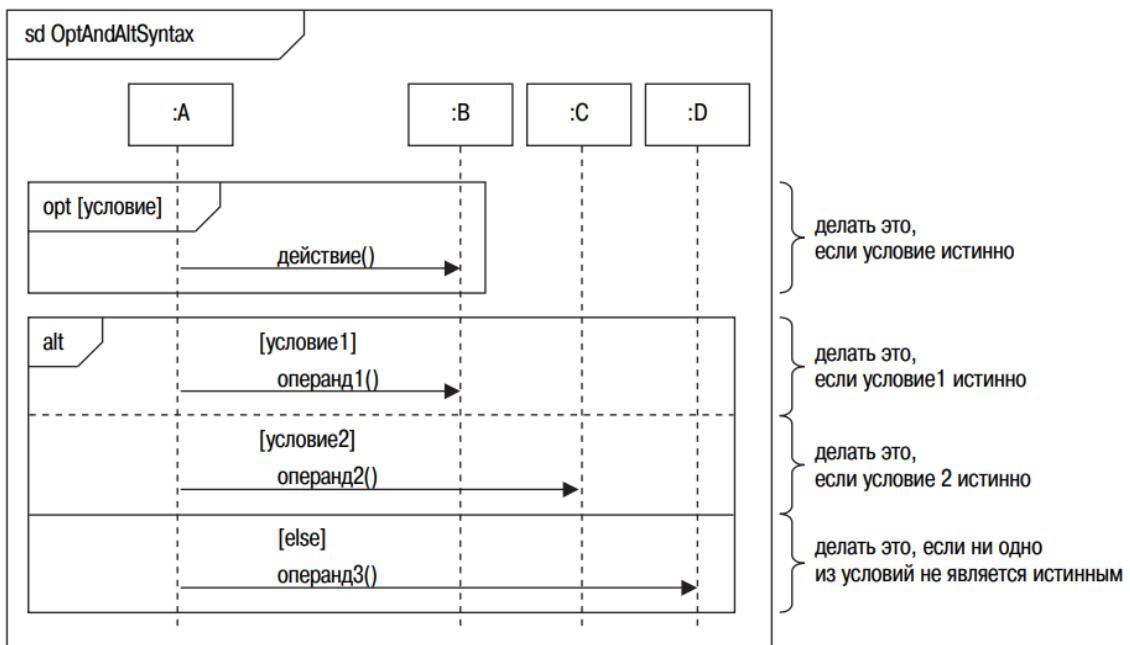


Рис. 2 - Синтаксис операторов opt и alt

Очень часто на диаграммах последовательностей необходимо показать циклы. Сделать это можно с помощью операторов loop и break, как показано на рис. 3.

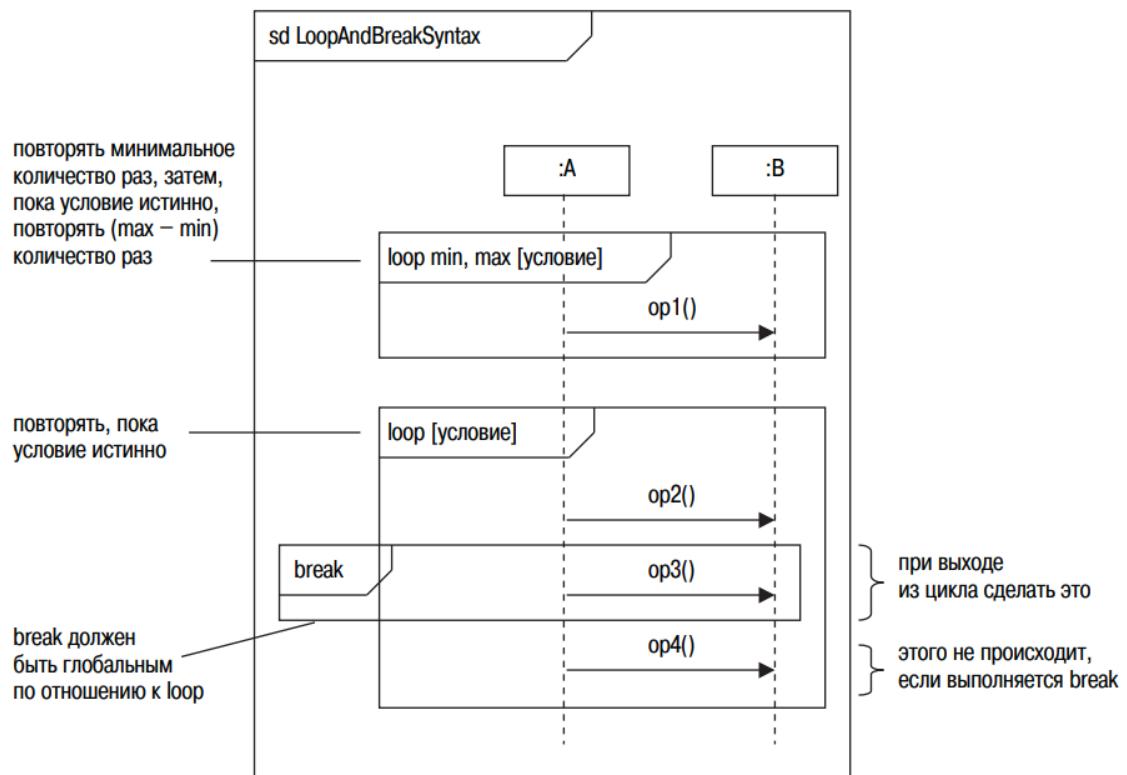


Рис.3 - Отображение циклов на диаграмме последовательностей с помощью операторов loop и break

ВОПРОС № 12.

Коммуникационные (кооперативные) диаграммы.

Диаграмма коммуникации (англ. *communication diagram*, в UML 1.x — **диаграмма кооперации**, collaboration diagram) — диаграмма, на которой изображаются взаимодействия между частями композитной структуры или ролями кооперации. В отличие от диаграммы последовательности, на диаграмме коммуникации явно указываются отношения между объектами, а время как отдельное измерение не используется (применяются порядковые номера вызовов).

В UML есть четыре типа диаграмм взаимодействия (неточно):

- Диаграмма последовательности
- Диаграмма коммуникации
- Диаграмма обзора взаимодействия
- Диаграмма синхронизации.

Диаграмма коммуникации моделирует взаимодействия между объектами или частями в терминах упорядоченных сообщений. Коммуникационные диаграммы представляют комбинацию информации, взятой из диаграмм классов, последовательности и вариантов использования, описывая сразу и статическую структуру, и динамическое поведение системы.

Коммуникационные диаграммы имеют свободный формат упорядочивания объектов и связей как в диаграмме объектов. Чтобы поддерживать порядок сообщений при таком свободном формате, их хронологически нумеруют. Чтение диаграммы коммуникации начинается с сообщения 1.0 и продолжается по направлению пересылки сообщений от объекта к объекту.

Подобно диаграммам последовательностей, диаграммы кооперации предназначены для описания динамических аспектов моделируемой системы. Обычно они применяются для того, чтобы:

- показать набор взаимодействующих объектов в реальном окружении "с высоты птичьего полета";
- распределить функциональность между классами, основываясь на результатах изучения динамических аспектов системы;
- описать логику выполнения сложных операций, особенно в тех случаях, когда один объект взаимодействует еще с несколькими объектами;
- изучить роли, выполняемые объектами внутри системы, а также отношения между объектами, в которые они вовлекаются, выполняя эти роли.

На диаграммах взаимодействия видим объекты, классы, сообщения, связи и кооперации. Это статическая конструкция для моделирования набора сущностей, взаимодействующих друг с другом. *Кооперация* определяет набор взаимодействующих ролей, используемых вместе, чтобы показать некую функциональность. *Кооперация* часто реализует некоторый *паттерн* (*шаблон проектирования*).

Кооперация изображается в виде эллипса с *пунктирной границей*, причем символ этот может использоваться двумя способами. Вот первый способ (рис.1):

*рис.1*

Эта диаграмма буквально иллюстрирует наши слова о кооперации как наборе ролей, используемых вместе, чтобы показать некую функциональность, в данном случае - выполнение ежемесячного резервного копирования. Второй способ показывает прикрепленные к объектам (классам) роли в рамках данной кооперации. Назначение роли изображается пунктирной линией со стрелкой на конце, направленной в сторону объекта. *Имя роли* указывается на конце линии, рядом с объектом. Посмотрите, например, на эту диаграмму (*рис.2*):

*рис.2*

Следующие элементы, которые можно увидеть на диаграмме взаимодействия - это *объекты и классы*.

Поскольку *диаграмма кооперации* - всего лишь альтернативная форма представления той же информации, которая содержится в диаграмме последовательностей, то и обозначения объектов (классов) в ней, по сути, такие же, как и на диаграмме последовательностей (и на других диаграммах). Чтобы проиллюстрировать это утверждение, приведем пример *диаграммы взаимодействия*, позаимствованный нами с сайта <http://www.agilemodeling.com/> (а точнее, <http://www.agilemodeling.com/style/collaborationDiagram.htm>) (*рис.3*):

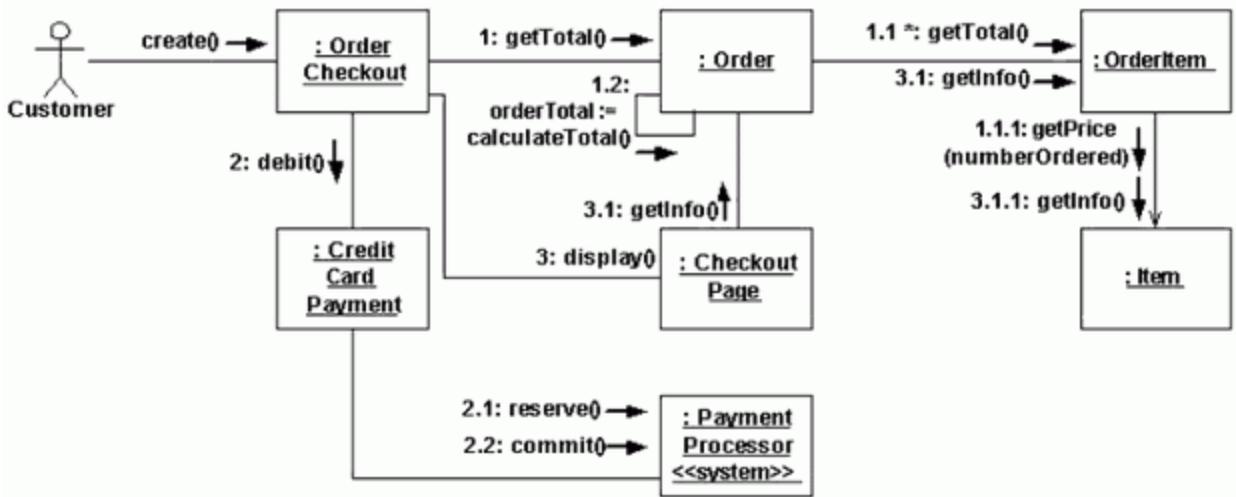


рис.3

Диаграмма иллюстрирует покупку некоторого товара (вероятно, в онлайне) и оплату с помощью кредитной карты. Еще одна интересная вещь, которую можно увидеть на этой диаграмме - это *сообщения*, вернее, то, как они изображаются. Сообщения показаны в виде текста (название метода) со стрелкой. Но есть один нюанс: на диаграмме последовательностей было легко показать последовательность отправки сообщений, так как линии жизни служили одновременно "осами времени", направленными вниз, и, естественно, было видно, что нижние сообщения отправлены позже верхних. В диаграммах взаимодействия проблему отображения очередности сообщений решили просто - перед названием каждого сообщения просто пишут его номер. Выглядит эта конструкция так: *номер:название_сообщения*. Причем часто используют и *составные номера*. Например, *объект* отправил другому объекту сообщение с номером 1. Когда *объект-получатель* в свою очередь отправляет сообщения другим объектам, они получают номера 1.1, 1.2 и т. д. Иногда нужно показать одновременную отправку сообщений. Чтобы отметить параллельные потоки сообщений, их номера предваряют буквами А, В, С, Д и т. д. Вот пример таких обозначений, позаимствованный опять-таки с <http://www.agilemodeling.com/> (рис.4):

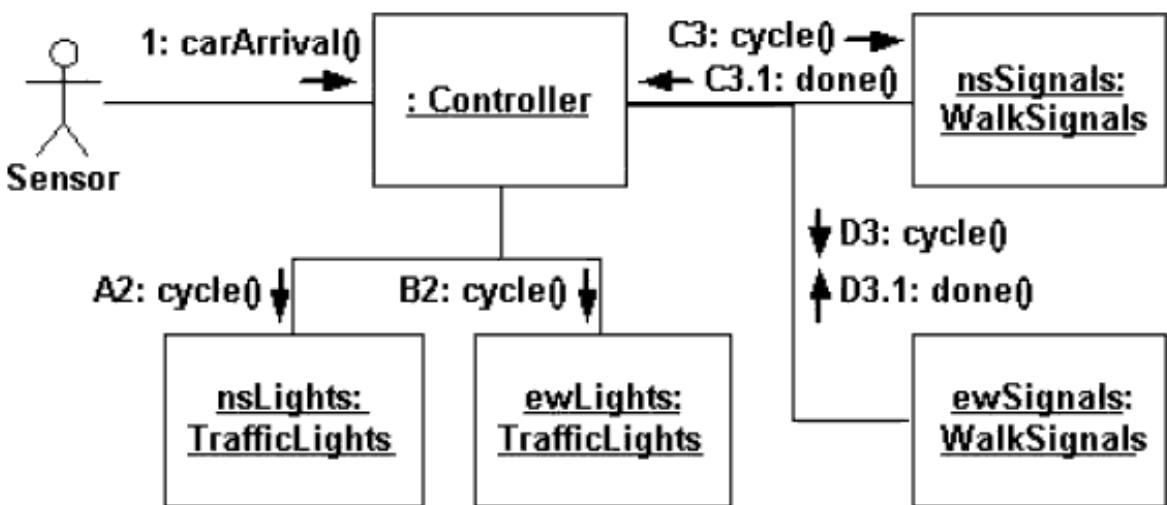


рис.4

Мультиобъект показывает, что на "далнем" конце ассоциации находится не один, а целый набор объектов. Такая конструкция используется, чтобы показать операцию, которая нацелена на целый набор объектов. *Мультиобъект* изображается как два

прямоугольника, смещенных *по* отношению друг к другу, что создает впечатление "колоды карт". Сообщение, отправленное мультиобъекту, означает сообщение к набору объектов, например, операция выбора - поиска определенного объекта. Пример подобной диаграммы показан на рисунке (рис.5):



рис.5

Смысл диаграммы вполне понятен: для печати документа из некоторого приложения необходимо выбрать из всех доступных некий конкретный принтер. Символ композиции применен для того, чтобы показать, что принтер входит в состав набора объектов.

Выводы:

- Диаграмма последовательностей - диаграмма взаимодействия, в которой основной акцент сделан на упорядочении сообщений во времени.
- Диаграмма кооперации - альтернативная форма представления информации, содержащейся в диаграмме последовательностей.
- Диаграмма кооперации - диаграмма взаимодействий, в которой основной акцент сделан на структурной организации объектов, посылающих и получающих сообщения.
- Существуют различные типы сообщений: синхронные, асинхронные и ответные, потерянные и найденные.
- Диаграммы кооперации бывают двух "уровней" - уровня экземпляров и уровня спецификации.
- Кооперация - это статическая конструкция для моделирования набора сущностей, взаимодействующих друг с другом.

С диаграммами кооперации связаны такие понятия, как *мультиобъекты*, композитные объекты и активные объекты.

ВОПРОС №13

Диаграммы пакетов. (RUP&UML2 Главы 11.2, 11.3, 11.4, 11.5, 11.6.)

Пакет – это UML-механизм логической группировки сущностей.

Используется для:

- предоставления инкапсулированного пространства имен, в рамках которого все имена должны быть уникальными;
- группировки семантически взаимосвязанных элементов;
- определения «семантической границы» модели;
- предоставления элементов для параллельной работы и управления конфигурацией.

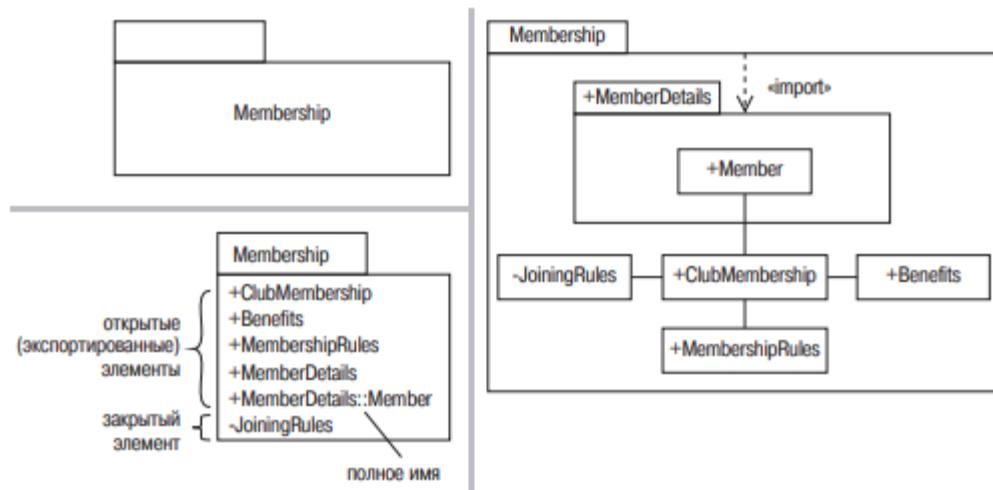


Рис. 11.2. Синтаксис пакета: три способа представления пакета на разных уровнях детализации

Каждый элемент модели принадлежит только одному пакету. Иерархия принадлежности образует дерево, где корень – пакет высшего уровня. Для обозначения этого пакета используется специальный стереотип UML «*topLevel*» (высший уровень). Если элемент моделирования явно не помещен в какой-либо пакет, он по умолчанию отправляется в пакет высшего уровня.

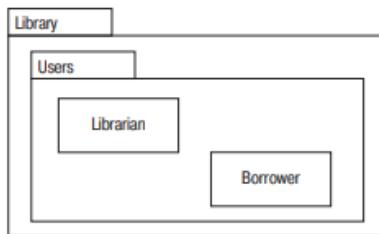
Для элементов, находящихся в пакете, может быть задана видимость, показывающая, видят ли их клиенты пакета.

Символ	Видимость	Семантика
+	открытая (public)	Элементы с открытой видимостью видимы вне пакета – они <i>экспортируются</i> пакетом.
-	закрытая (private)	Элементы с закрытой видимостью полностью скрыты внутри пакета.

Пространство имен

Пакет создает границу, в рамках которой имена всех элементов должны быть уникальными. Если элементу из одного пространства имен необходимо обратиться к элементу из другого пространства имен, он должен указать и имя необходимого элемента, и путь к этому элементу, чтобы его можно было найти в пространствах имен. Этот путь навигации называют полным или составным именем элемента.

Пример:



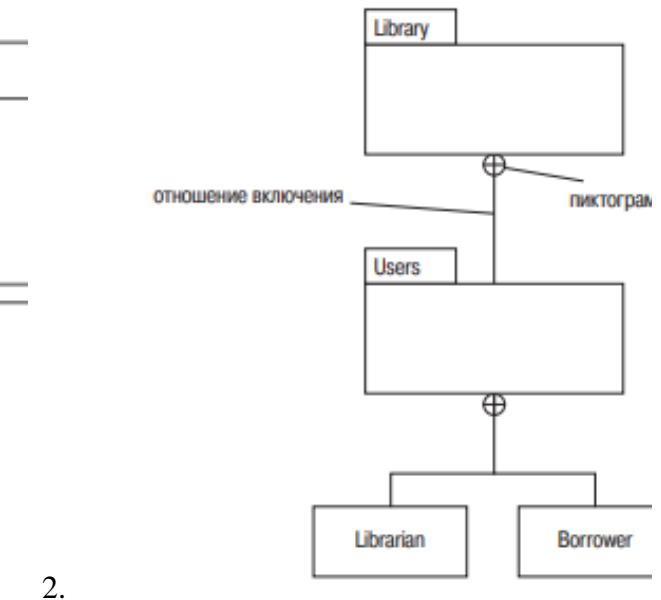
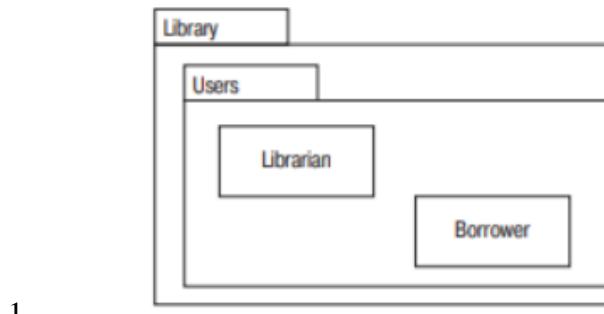
Полное имя класса Librarian (библиотекарь) будет таким: Library::Users::Librarian

Вложенность

Пакеты могут быть вложены в другие пакеты с любой глубиной вложенности. Обычно достаточно всего двух или трех уровней. Иначе модель может стать трудной для понимания и в ней будет сложно ориентироваться.

Вложенные пакеты имеют доступ к пространству имен своего пакета-владельца. Таким образом, элементы пакета Users (пользователи) могут организовывать доступ ко всем элементам пакета Library (библиотека) через неполные имена. Однако обратное невозможно. Пакет-владелец для доступа к содержимому пакетов, которыми он владеет, должен использовать полные имена

Два способа представления вложенности:



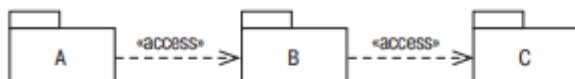
Зависимости пакетов

Между пакетами может быть установлено отношение зависимости. Оно показывает, что один пакет зависит от другого.

Отношение зависимости пакетов	Семантика
	Элемент клиентского пакета некоторым образом использует открытый элемент пакета-поставщика – клиент зависит от поставщика. Если зависимость пакета показана без стереотипа, необходимо предполагать зависимость «use».
	Открытые элементы пространства имен поставщика добавляются как открытые элементы в пространство имен клиента. Элементы клиента могут организовывать доступ ко всем открытым элементам поставщика через неполные имена.
	Открытые элементы пространства имен поставщика добавляются как закрытые элементы в пространство имен клиента. Элементы клиента могут организовывать доступ ко всем открытым элементам поставщика с помощью неполных имен.
	«trace», как правило, представляет историческое развитие одного элемента в другую более развитую версию; обычно это отношение между моделями, а не элементами (межмодельное отношение).
	Открытые элементы пакета-поставщика объединяются с элементами клиентского пакета. Эта зависимость используется только при создании метамодели; в обычном ОО анализе и проектировании она не должна встречаться.

«import» – транзитивное отношение, «access» – нет*

*Транзитивность (transitivity) – термин, применяемый к отношениям. Он означает, что если существует отношение между сущностями А и В и отношение между сущностями В и С, то существует неявное отношение между сущностями А и С.



Rис. 11.6. «access» – нетранзитивная зависимость

Отсутствие транзитивности в «access» означает следующее:

- открытые элементы пакета С становятся закрытыми элементами пакета В;
- открытые элементы пакета В становятся закрытыми элементами пакета А;
- следовательно, элементы пакета А *не имеют* доступа к элементам пакета С.

Обобщение пакетов

Дочерние пакеты наследуют элементы от своих родителей. Они могут переопределять родительские элементы. Они могут вводить новые элементы

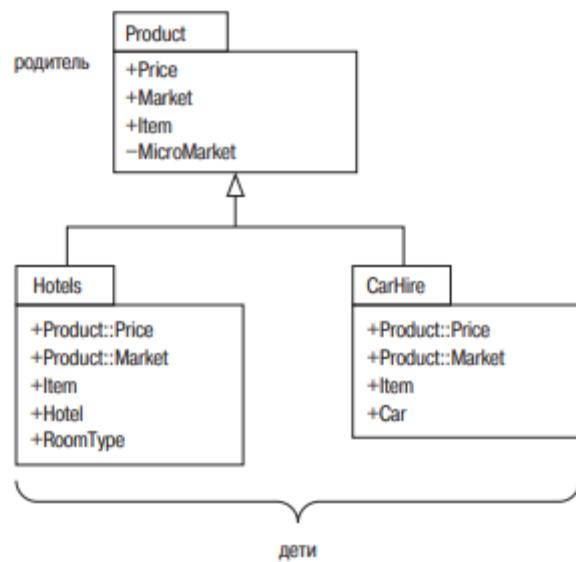


Рис. 11.7. Дочерние элементы наследуют элементы родителя

Ссылки:

1.RUP&UML2 Главы 11.2, 11.3, 11.4, 11.5, 11.6.

Вопрос №14

Диаграммы конечных автоматов (состояний).

Спецификация UML 2 определяет два типа конечных автоматов, имеющих общий синтаксис: поведенческие автоматы и протокольные автоматы

Диаграмма состояний содержит только один конечный автомат для единственного реактивного объекта.

- Состояния обозначаются прямоугольниками со скругленными углами, за исключением начального состояния (закрашенный кружок) и конечного состояния (бычий глаз).
- Переходы указывают на возможные пути между состояниями и моделируются с помощью стрелок.
- События записываются над инициируемыми ими переходами.

У каждого конечного автомата должно быть начальное состояние, обозначающее первое состояние последовательности. Если смена состояний не бесконечна, должно присутствовать и конечное состояние, которое завершает последовательность переходов.

События обуславливают переходы состояний.

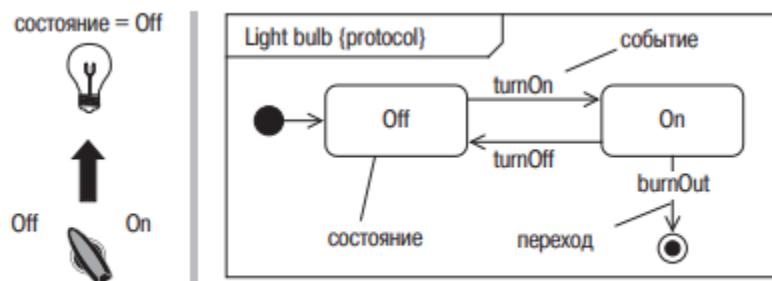


Рис. 21.2. Диаграмма состояний электрической лампочки

Состояние объекта меняется со временем, но в любой отдельный момент оно определяется:

- значениями атрибутов объекта;
- отношениями с другими объектами;
- осуществляемыми действиями.

Состояние – это семантически значимое состояние объекта. Должны быть выявлены значимые состояния системы.

Действия считаются мгновенными и непрерываемыми, тогда как **деятельности** занимают конечное время и могут быть прерваны. Каждое действие в состоянии ассоциируется с внутренним переходом, инициируемым событием. В состоянии может быть любое число действий и внутренних переходов.

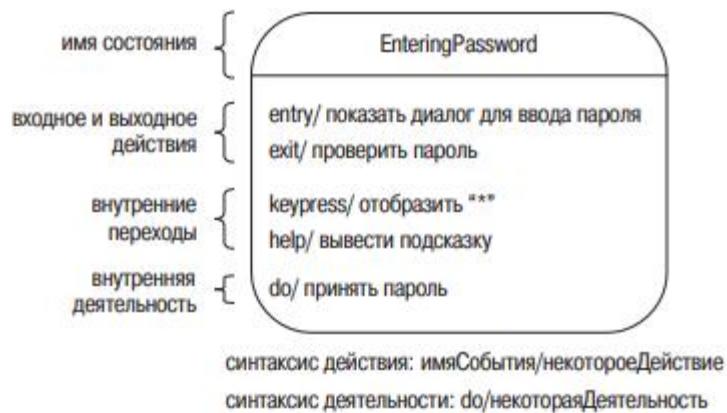


Рис. 21.4. Синтаксис состояния

Переходы показывают движение между состояниями.

Каждый переход имеет три необязательных элемента.

1. Нуль или более событий – определяют внешнее или внутреннее происшествие, которое может инициировать переход.
2. Нуль или одно сторожевое условие – логическое выражение, которое должно быть выполнено (иметь значение true), прежде чем может произойти переход. Условие указывают после событий.
3. Нуль или более действий – часть работы, ассоциированная с переходом и выполняемая при срабатывании перехода.

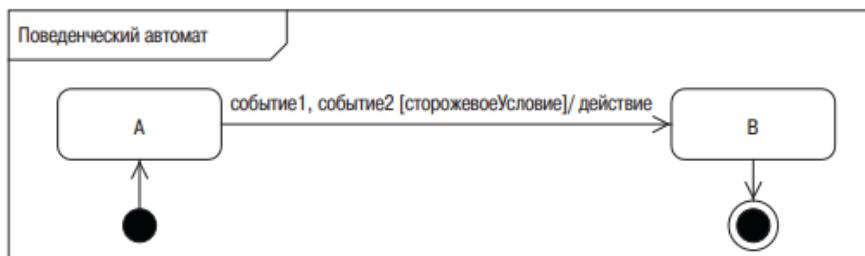
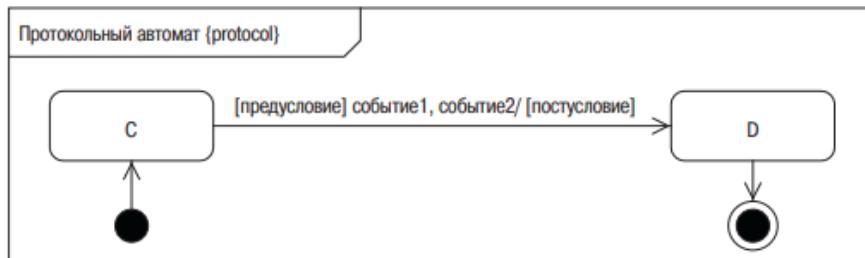
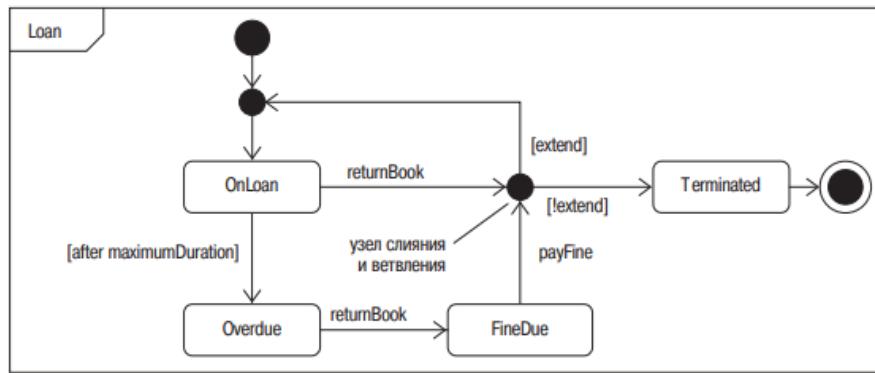


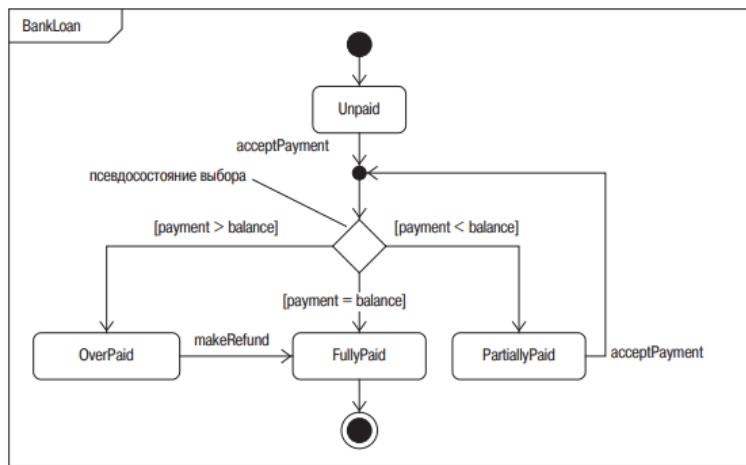
Рис. 21.5. Синтаксис переходов для поведенческих автоматов



Соединение переходов – переходные псевдосостояния объединяют или разветвляют переходы. Они изображаются в виде закрашенных кружков с одним или более входными переходами и одним или более исходящими переходами.

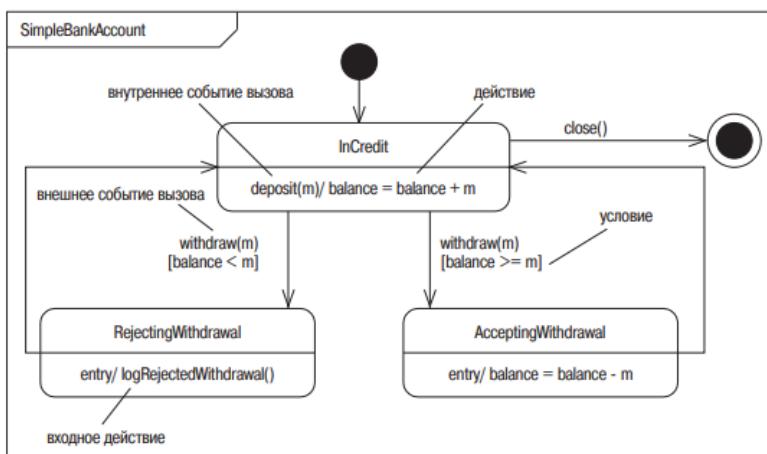


Ветвление переходов – псевдосостояние выбора - направляет поток конечного автомата согласно заданным условиям.



События инициируют переходы. Существует четыре семантически различных типа событий:

- **событие вызова** – это запрос на вызов конкретной операции.



- **сигнал** – это односторонняя асинхронная связь между объектами.

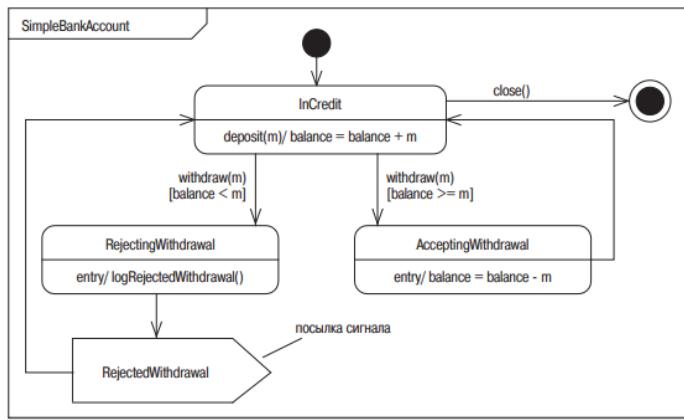


Рис. 21.13. Посылка сигнала

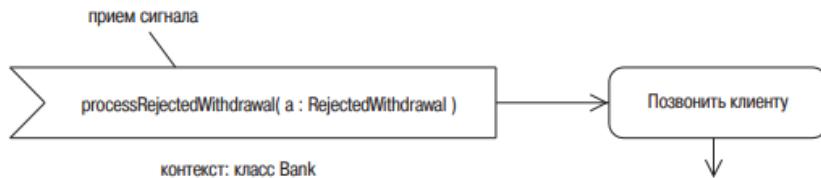
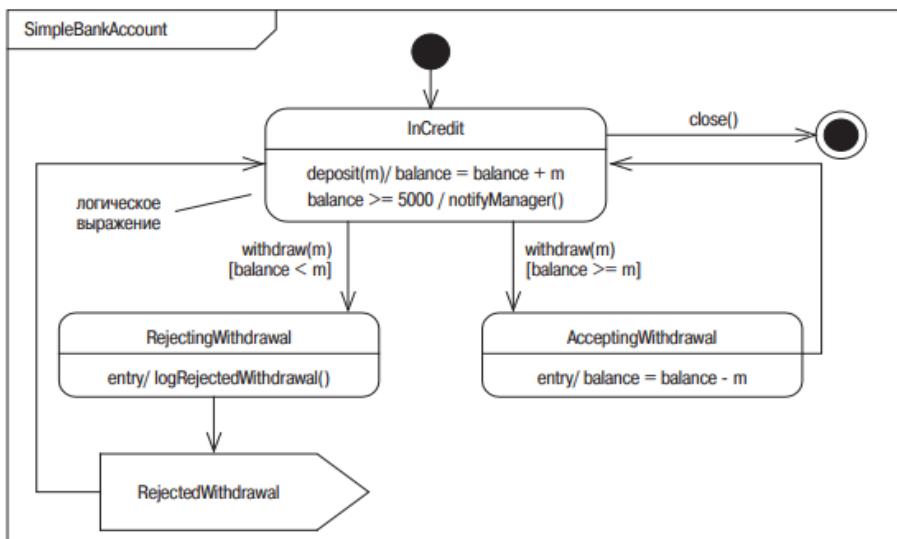
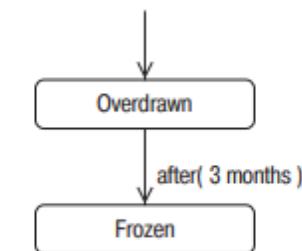


Рис. 21.14. Прием сигнала

- **события изменения** имеют место, когда значение логического выражения меняется с false на true. События изменения являются срабатывающими по положительному фронту (positive edge triggered). Это означает, что они инициируются при каждом изменении значения логического выражения с false на true.



- **событие времени** происходят в ответ на время.



контекст: класс CreditAccount

ВОПРОС №15

Диаграммы деятельности.

(RUP&UML2 Глава 14.)

Диаграммы деятельности – это «ОО блок схемы». Они позволяют моделировать процесс как деятельность, состоящую из коллекции соединенных ребрами узлов.

- Они используются для моделирования всех типов процессов;
- Диаграммы деятельности можно создать для любого элемента модели для описания его поведения;
- Хорошая диаграмма деятельности описывает один конкретный аспект поведения системы;
- В UML 2 диаграммы деятельности имеют семантику технологии сетей Петри.

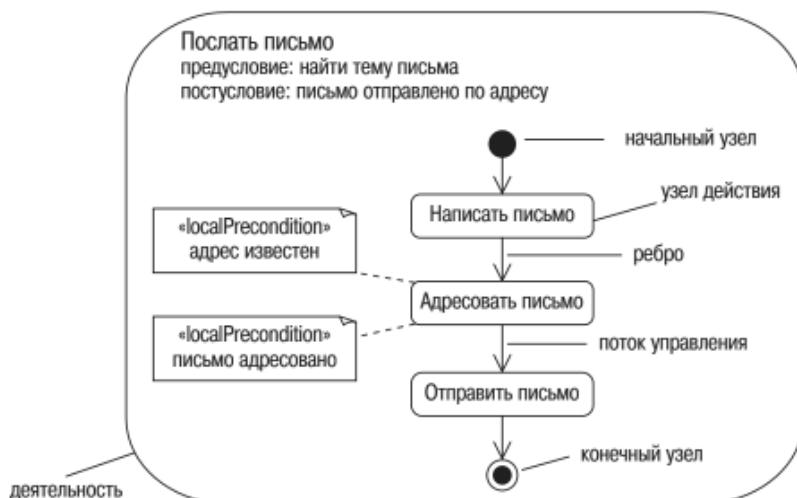


Рис. 14.2. Диаграмма деятельности бизнес-процесса Послать письмо

Деятельности – это схемы, состоящие из узлов, соединенных ребрами.

- Категории узлов:
 - узлы **действия** – элементарные единицы работы в рамках деятельности;
 - узлы **управления** – управляют потоком деятельности;
 - объектные узлы** – представляют объекты, используемые в деятельности.
- Категории ребер:
 - потоки управления** – представляют поток управления деятельности;
 - потоки объектов** – представляют поток объектов деятельности.
- Маркеры перемещаются по сети (network) и могут представлять:
 - поток управления; объект; некоторые данные.*
- Движение маркеров по ребрам от начального узла к целевому узлу зависит от:
 - постусловий начального узла; сторожевых условий ребра; предусловий целевого узла.*
- Деятельности могут иметь предусловия и постусловия.

Узлы действия – выполняются при одновременном поступлении маркеров по всем входным ребрам и удовлетворении *всех* предусловий. После выполнения узлы действия предлагают маркеры одновременно на всех выходных ребрах, постусловия которых удовлетворены *неявное ветвление*.

- Узел вызова действия - инициирует деятельность – используется символ «грабли», поведение или операцию.
- Узел действия, посылающий *сигнал*.
- Узел действия, принимающий *событие*.
- Узел действия, принимающий *событие времени* – выполняется, когда временное выражение становится истинным: некоторое событие (например, конец финансового года); конкретный момент времени (например, 11/03/1960); временной интервал (например, ожидать 10 секунд).

Узлы управления: начальный узел показывает, где начинается поток при вызове деятельности; конечный узел деятельности заканчивает деятельность; конечный узел потока заканчивает конкретный поток деятельности;

- Узел решения – поток направляется по исходящему ребру, сторожевое условие которого истинно: может иметь стереотип «decisionInput»;
- Узел слияния копирует входные маркеры в единственное исходящее ребро;
- Узел ветвления разделяет поток на несколько параллельных потоков;
- Узел объединения синхронизирует несколько параллельных потоков: может иметь {объединение}.

Таблица 14.2

Синтаксис	Имя	Семантика	Раздел
● →	Начальный узел	Указывает, где начинается поток при вызове деятельности.	14.8.1
→ ●	Конечный узел деятельности	Завершает деятельность.	14.8.1
→ ⊗	Конечный узел потока	Завершает определенный поток деятельности – другие потоки не затрагиваются.	14.8.1
«decisionInput» условие принятия решения 	Узел решения	Поток проходит по исходящему ребру, сторожевое условие которого истинно. Может иметь входные данные (не обязательно).	14.8.2
→ ⊕	Узел слияния	Копирует входные маркеры в единственное выходное ребро.	14.8.2
→ □	Узел ветвления	Разделяет поток на несколько параллельных потоков.	14.8.3
{описание объединения}	Узел объединения	Синхронизирует несколько параллельных потоков. Может иметь описание объединения (не обязательно) для изменения его семантики.	14.8.3

Разделы деятельности – высокоуровневая группировка взаимосвязанных действий. Разделы формируют иерархию, корнем которой является измерение.

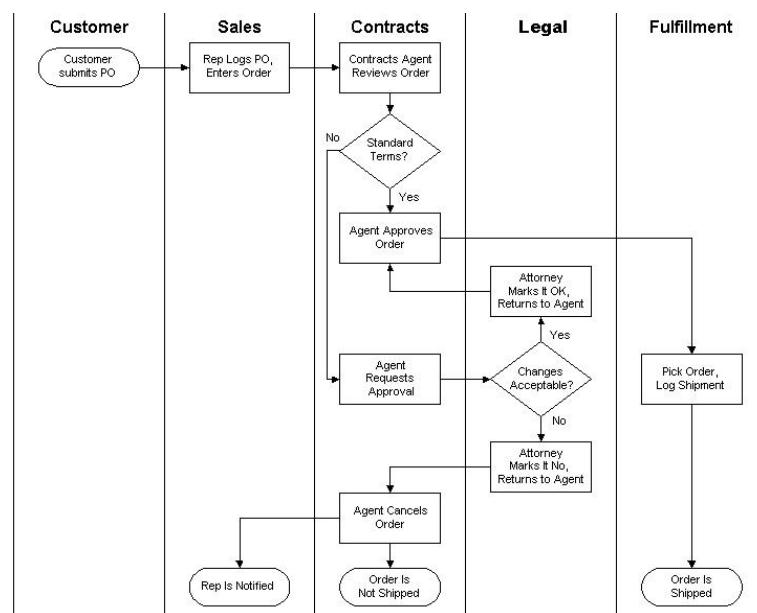
Объектные узлы представляют экземпляры классификатора.

- Входящие и исходящие ребра – потоки объектов – представляют движение объектов.
- Исходящие ребра объектного узла конкурируют за каждый исходящий маркер.
- Объектные узлы работают как буферы:
 - { upperBound = n }
 - { ordering = FIFO } XOR { ordering = LIFO };
 - по умолчанию применяется { ordering = FIFO };
 - могут иметь стереотип «selection».
- Объектные узлы могут представлять объекты, находящиеся в определенном состоянии: *должны соответствовать конечным автоматам*.

- Параметры деятельности – это объектные узлы, входящие в или исходящие из деятельности. На диаграмме перекрывают рамку деятельности; входящие параметры имеют один или более исходящих ребер, поступающих в деятельность; исходящие параметры имеют один или более входящих ребер, поступающих из деятельности.

Контакт – это объектный узел, представляющий один вход в или выход из действия или деятельности.

Swimlane – визуальный элемент, используемый в процессе технологических схем, которые описывают — что или кто работает на определенной части процесса. «Плавательные дорожки» расположены либо по горизонтали, либо по вертикали и используются для группировки процессов или задач в соответствии с обязанностями этих ресурсов, ролей или отделов. В сопроводительном примере, swimlanes называются Customer (Клиенты), Sales (Продажи), Contracts (Контракты), Legal (Юристы) и Fulfillment (Исполнение).



ВОПРОС №16

Диаграмма размещения. (RUP&UML2 Главы 24.3, 24.4, 24.5, 24.6.)

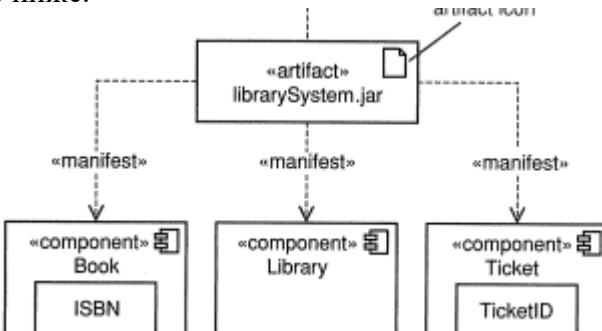
(Источник: UML and The Unified Process, главы 24.3, 24.4, 24.5, 25.6)

Диаграммы развертывания - диаграммы, описывающие размещение разработанных программных компонентов на аппаратном обеспечении. Диаграммы компонентов разрабатываются в ходе реализации архитектуры.

Реализация архитектуры - активность по выявлению архитектурно значимых компонентов и отображению их на аппаратное обеспечение - моделирование физической структуры и размещения системы.

Артефакт - физическое представление чего-либо, созданное человеком. Артефакт подразумевает взаимодействие (четкого определения нет, увы).

Артефакт может задавать отношение “манифестирует” т.е. предоставляет какие-либо компоненты. На диаграммах развертывания артефакт помечается иконкой “Файл” как на рисунке ниже.



Нода (узел) - тип вычислительного ресурса, на котором может быть развернут артефакт

Выделяют два стереотипа для нод:

- девайс - аппаратное обеспечение
- среда выполнения - среда, в которой будет исполняться артефакт (например, если артефакт - war файл, то среда исполнения - веб-сервер)

Ноды могут быть вложены в другие ноды, а также быть связаны друг-с-другом. Связь "ассоциация" представляет канал связи между двумя нодами

Существует два вида диаграмм развертывания:

- **дескрипторные (descriptor form)** - описывают размещение артефактов (исполняемое ПО) на нодах (аппаратное обеспечение). Подходят для моделирования физической архитектуры.
- **экземплярные (instance form)** - описывают размещение конкретных экземпляров артефактов на конкретных экземплярах нод (например, конкретный ПК Иван Иваныча), а также отношения между этими экземплярами. Подходят для моделирования процесса деплоя на конкретных ресурсах.

Создание диаграммы развертывания проходит в два этапа:

- в ходе дизайна продумываются ноды или экземпляры нод (фактически, какие ресурсы требуются)
- в ходе реализации фокус смещается на отображение артефактов на ноды.

Пример диаграммы развертывания с нодами “девайс” и “среда выполнения”

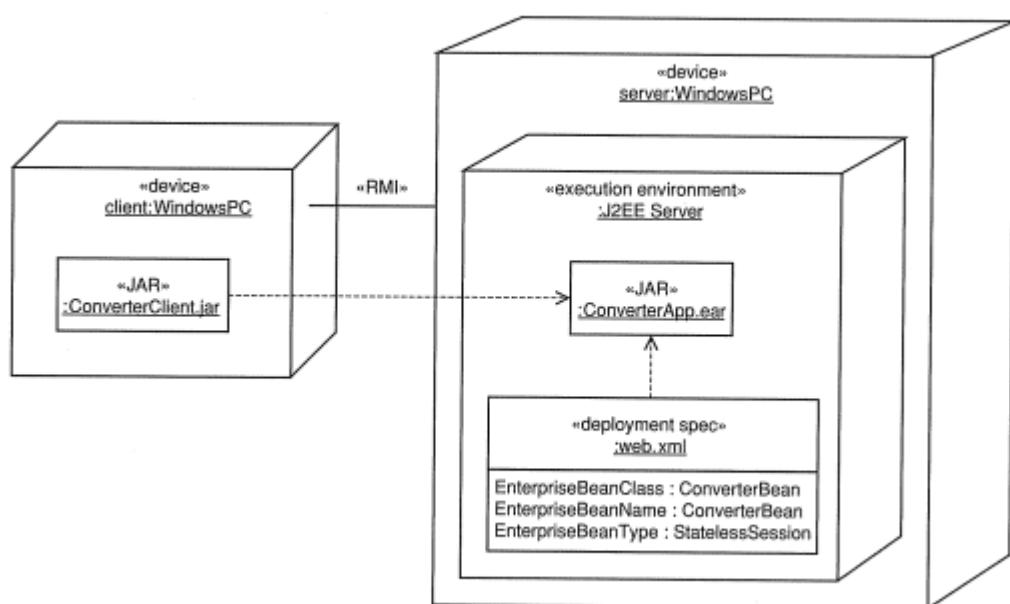


Figure 24.9

ВОПРОС №17

Процессы разработки ПО - UP, RUP. Предназначение, связь и основные определения: исполнитель (роль), деятельность, артефакт, рабочий поток (дисциплина) и др.

UP: методология для построения процессов разработки программного обеспечения, позволяющий команде разработки преобразовывать требования заказчика в работоспособный продукт. В зависимости от требований и доступных ресурсов, процесс разработки может быть адаптирован путем включения или исключения определенных проектных активностей. Для описания всех аспектов процесса создания продукта, а также для описания самого продукта UP использует язык UML.

Принципы UP

Unified Process основан на **сценариях использования**, описывающих одно или несколько действующих лиц, взаимодействующих с системой и получающих результаты, представляющие ценность для участников процесса.

Согласно Unified Process, в центре процесса разработки лежит **архитектура** — фундаментальная организация всей системы. Она может включать в себя статические и динамические элементы, их взаимодействие, и позволяет решать вопросы эффективности работы продукта, расширяемости, возможности переиспользования элементов, помогать преодолеть экономические и технические ограничения.

Третим фундаментальным принципом Unified Process является использование **итеративного и инкрементного подхода**. Итерациями называются миниатюрные проекты, которые позволяют запустить более новую версию системы. Результат итерации, изменения, внесенные в систему, называются **инкрементом**.

Определения

В рамках Unified Process **артефактом** называется любой фрагмент информации, играющий важную роль в процессе разработки. В число артефактов, используемых инженерами, входят модели, прототипы, результаты тестирования и пр. Артефактами менеджера являются план проекта, бизнес-кейсы и др. Важной составляющей Unified Process является то, что система не считается готовой к использованию до тех пор, пока все соответствующие артефакты не завершены.

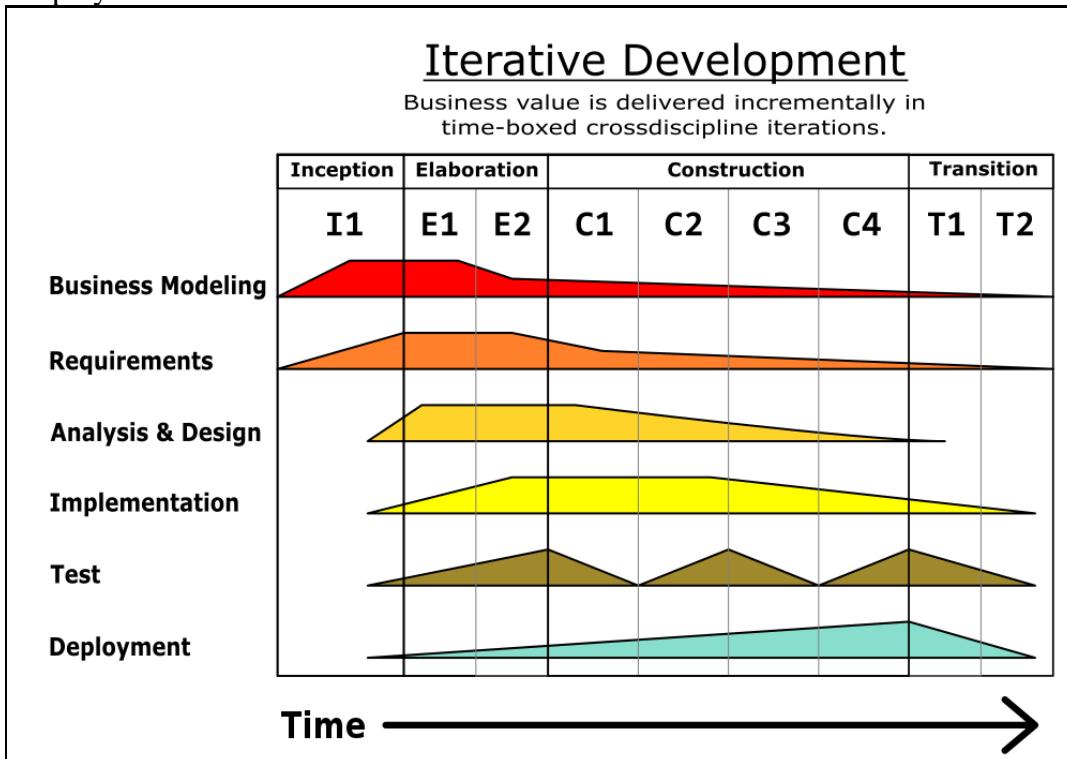
Под **исполнителем** понимается роль, которую отдельный работник может выполнять на проекте. Разница между исполнителем и действующим лицом (из сценариев использования) заключается в том, что последний смотрит на систему «снаружи», в то время как исполнитель — «изнутри». Исполнители создают артефакты, самостоятельно или в группах или командах.

Каждая из разновидностей рабочего процесса включает в себя несколько **активностей (деятельностей)** — задач, над которыми работают исполнители с целью получения артефактов.

Дисциплина — это проблемная область или тема, в которой рабочие процессы описывают ход работы, и в которой детали рабочего процесса описывают набор действий (с их связанными ролями и артефактами), часто выполняемых вместе.

В UP есть шесть дисциплин:

1. Business Modeling — формализация бизнес процесса и построение набора диаграмм, описывающих его, а также определение роли разрабатываемой системы в автоматизации рассматриваемого процесса. На выходе данной дисциплины получается артефакт “Business model”;
2. Requirements — формализация функциональных требований к системе на основе бизнес-требований. На выходе данной дисциплины получается артефакт “Use-Case Diagram”;
3. Analysis & Design — определение нефункциональных требований и разработка архитектуры системы. На выходе — “Analysis/Design model” (SRS, SAD);
4. Implementation — собственно разработка системы в соответствии с разработанной моделью;
5. Test — тестирование системы на предмет соответствия требованиям. Для тестирования необходима Модель тестирования (Test plan);
6. Deployment — непосредственное развертывание системы в соответствии с Deployment Model.



RUP: это универсальная методология распределения задач и сфер ответственности. Вобравшая в себя известные характеристики других методологий. Процесс основан на инкрементно-эволюционной методологии. RUP основан на UP.

Цель – создание высококачественного программного обеспечения. RUP оптимизирует командную работу – обеспечивает команде разработчиков свободный доступ к базе знаний с инструкциями для использования программных средств. RUP ориентирован на создание и поддержание моделей. UML позволяет команде легко донести свои требования к проекту, его архитектуру и план реализации. В RUP есть *Роли* и *Дисциплины*.

В основе RUP лежит шесть главных принципов

1. Итеративная модель разработки – устранение рисков на каждой стадии проекта позволяет лучше понять проблему и вносить необходимые изменения, пока не будет найдено приемлемое решение;
2. Управление требованиями – RUP описывает процесс организации и отслеживания функциональных требований, документации и выбора оптимальных решений (как в процессе разработки, так и при ведении бизнеса);
3. Компонентная архитектура – архитектура системы разбивается на компоненты, которые можно использовать как в текущем, так и в будущих проектах;
4. Визуальное моделирование ПО – RUP методология разработки показывает, как создать визуальную модель программного обеспечения, чтобы понять структуру и поведение архитектуры и его компонентов;
5. Проверка качества ПО – в процессе разработки программного обеспечения контролируется качество всех действий команды;
6. Контроль внесённых изменений – отслеживание изменений позволяет выстроить непрерывный процесс разработки. Создается благоприятная обстановка, в рамках которой команда будет защищена от изменений в рабочем процессе.

Жизненный Цикл RUP

Любая фаза заканчивается вехами – принятие решения о будущем.

1. *Начала*
Команда определяет структуру и основную идею проекта. Предлагают технические решения. Команда решает, стоит ли вообще заниматься этим проектом, исходя из его предполагаемой стоимости, необходимых ресурсов и цели, которую нужно достичь.
2. *Проектирования*
Цель этой фазы — анализ требований к системе и ее архитектуре, разработка плана проекта и устранение элементов наивысшего риска. Создаются прототипы. Это самая важная фаза из всех, она знаменует переход от низкого уровня риска к высокому. Уточняются сроки и стоимость системы
3. *Построение*
В этой фазе RUP методологии команда начинает разработку всех компонентов и функций программного обеспечения, интегрирует их в конечный продукт. Это производственный процесс, в рамках которого команда сосредоточена на управлении ресурсами, чтобы оптимизировать расходы, время и качество продукта. Проводятся плановые демонстрации. В конце происходит подготовка к передаче заказчику.
4. *Внедрение*
Фаза, когда продукт готов и доставлен покупателям. Но после того как пользователи получают продукт, могут возникнуть новые трудности. Команде нужно будет исправить ошибки, отловить баги и доделать функции, которые не были реализованы в первично установленный срок. Происходит оценка удовлетворенности пользователя.

В конце каждой фазы существует отметка завершения этапа (Project Milestone) — момент, когда ваша команда оценивает, достигнуты ли поставленные цели. При этом команда принимает важные решения, влияющие на ход следующей фазы.

Источники

<https://www.methodsandtools.com/archive/archive.php?id=32>

https://ru.wikipedia.org/wiki/Unified_Process

ВОПРОС №18

Итеративная разработка в UP. Рабочие потоки UP, Дисциплины RUP. Базовые версии. (RUP&UML2 Главы: 2.7, 2.7.1, 2.7.2
RUP_classic.2002.05.00/RationalUnifiedProcess/process/workflow/ovu_core.htm)

Итеративная разработка в UP

В UP проект по разработке ПО разбивается на несколько меньших «мини-проектов», которыми легче управлять и успешно выполнить. Каждый из этих «мини-проектов» и есть *итерация*. Основной момент: каждая итерация включает все элементы обычного проекта по разработке ПО.

Разбиение проекта на серию итераций обеспечивает возможность гибкого подхода к его планированию. Самый простой подход – упорядоченная во времени последовательность итераций, в которой каждая последующая итерация является результатом предыдущей. Однако часто итерации можно расположить параллельно. Преимущество параллельных итераций – меньшее время вывода нового изделия на рынок и, возможно, более рациональное использование команды, но при этом первостепенным является тщательное планирование.

Рабочие потоки UP

В каждой итерации *пять основных рабочих потоков* определяют, что должно быть сделано и какие навыки для этого необходимы. Наряду с пятью основными рабочими потоками будут присутствовать и другие, такие как планирование, оценка и все, что характерно для этой конкретной итерации. Однако эти этапы не выделены в UP. К пяти основным рабочим потокам относятся:

- *определение требований* – сбор данных о том, что должна делать система;
- *анализ* – уточнение и структурирование требований;
- *проектирование* – реализация требований в архитектуре системы;
- *реализация* – построение программного обеспечения;
- *тестирование* – проверяется, отвечает ли реализация предъявляемым требованиям.

Дисциплины RUP

Дисциплины описывают все действия, которые вы можете проделать, чтобы произвести определенный набор артефактов. Дисциплины RUP:

- Бизнес моделирование(Business Modeling)
- Анализ и проектирование
- Развертывание
- Реализация
- Среда(Environment)
- Тестирование
- Требования
- Управление конфигурацией и изменениями
- Управление проектом

Базовые версии

Каждая итерация UP формирует базовую версию. Это версия для внутреннего (или внешнего) использования набора рассмотренных и утвержденных артефактов, сгенерированных в данной итерации. Каждая базовая версия:

- предоставляет базу для дальнейшего рассмотрения и разработки;
- может изменяться только через формальные процедуры управления конфигурацией и изменениями.

Инкременты – это просто разница между двумя последовательными базовыми версиями. Это шаги по направлению к окончательной выпускаемой системе.

Источники:

(RUP&UML2 Главы: 2.7, 2.7.1, 2.7.2
RUP_classic.2002.05.00/RationalUnifiedProcess/process/workflow/ovu_core.htm)

ВОПРОС №19

Аксиомы UP, риски. Структура UP. Фазы жизненного цикла, цели, контрольные точки (вехи).

Аксиомы UP

- управляется требованиями и риском
- архитектурно-центричный
- итеративный и инкрементный

UP призывает разработчиков “атаковать” риски, закладывая их анализ в основу процесса разработки ПО. Чем раньше стадия, на которой обнаруживается неучтённый риск, тем экспоненциально больше связанные с ним расходы. По этой причине анализ рисков является управляющим механизмом в UP.

Структура UP

Фазы UP:

- Начало (Inception)

Цели:

- обоснование выполнимости
- разработка экономического обоснования
- определение основных требований
- выявление наиболее опасных рисков

Контрольная точка – цели жизненного цикла. Её условия:

- Заинтересованные стороны согласовали цели проекта.
- Заинтересованные стороны определили и одобрили предметную область системы.
- Заинтересованные стороны определили и одобрили ключевые требования.
- Заинтересованные стороны одобрили затраты и план работы.
- Руководитель проекта сформировал экономическое обоснование проекта.
- Руководитель проекта провел оценку рисков.
- Посредством технических исследований и/или создания прототипа была подтверждена выполнимость.
- Архитектура намечена в общих чертах.

- Уточнение (Elaboration)

Цели:

- создание исполняемой базовой версии архитектуры
- детализация оценки рисков
- определение атрибутов качества (скорости выявления дефектов, приемлемые плотности дефектов и т. д)
- выявление прецедентов, составляющих до 80% от функциональных требований (что именно сюда входит, вы увидите в главах 3 и 4)
- создание подробного плана фазы Построение
- формулировка предложения, включающего ресурсы, время, оборудование, штат и стоимость.

Контрольная точка – архитектура жизненного цикла. Её условия:

- Создана гибкая надежная исполняемая базовая версия архитектуры.
- Исполняемая базовая версия архитектуры демонстрирует, что важные риски были выявлены и учтены.
- Представление продукта стабилизировалось.

- Оценка рисков пересмотрена.
- Экономическое обоснование проекта пересмотрено и одобрено всеми заинтересованными сторонами.
- Создан достаточно детальный план проекта, что обеспечило возможность сформулировать реалистичную заявку на затраты времени, денег и ресурсов в следующих фазах. Заинтересованные стороны одобрили план проекта.
- Проведена проверка экономического обоснования проекта согласно плану проекта.
- Заинтересованные стороны достигли соглашения о продолжении проекта.

• Построение (Construction)

Цели:

- завершение определения требований, анализа и проектирования
- развить базовую версию архитектуры в завершённую систему

Контрольная точка – базовая функциональность. Её условия:

- Программный продукт достаточно стабилен и качественен для распространения среди пользователей.
- Заинтересованные стороны одобрили и готовы к введению программного продукта в свое окружение.
- Описание данной версии. Расхождения реальных расходов с предполагаемыми приемлемы.

• Внедрение (Transition)

Цели:

- исправление дефектов
- подготовка пользовательских сайтов под новое программное обеспечение
- настройка работоспособности программного обеспечения на пользовательских сайтах
- изменение программного обеспечения в случае возникновения непредвиденных проблем
- создание руководств для пользователей и другой документации
- предоставление пользователям консультаций
- проведение послепроектного анализа.

Контрольная точка – выпуск продукта. Её условия:

- Бета-тестирование завершено, необходимые изменения сделаны и пользователи согласны с тем, что система успешно развернута.
- Сообщество пользователей активно использует продукт.
- Стратегии поддержки продукта согласованы с пользователями и реализованы.

Источники: RUP&UML2 Главы: 2.6, 2.8

ВОПРОС №20

Рабочий поток определения требований

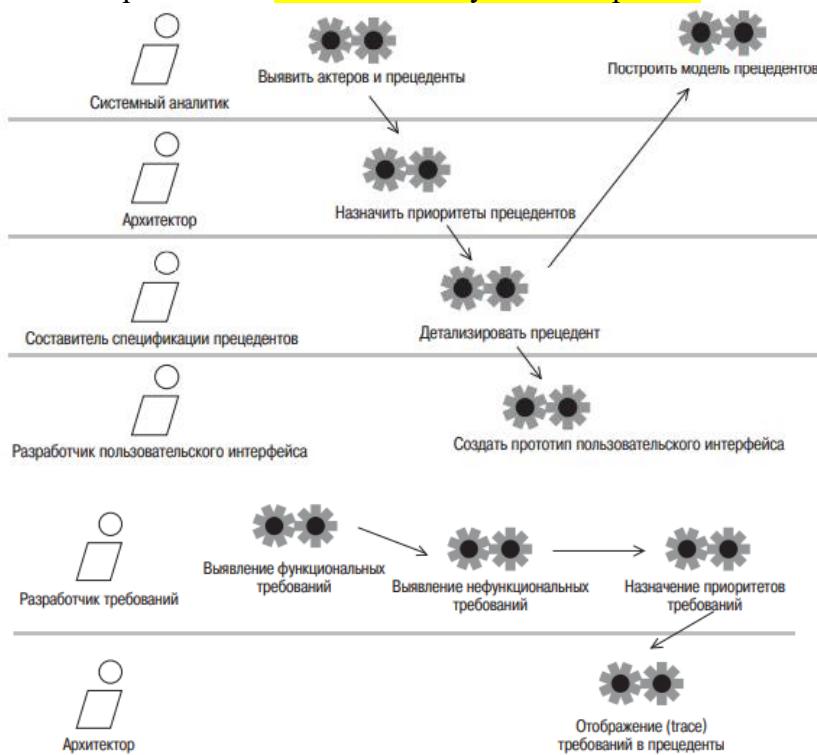
Определение требований производится в самом начале процесса разработки, на фазах начала и уточнения.

Цель рабочего потока определения требований - поиск и достижение соглашения о функциях системы, написанного на языке пользователей системы.

Выработка требований состоит в выявлении и классификации требований, предъявляемых к системе заинтересованными сторонами, представляет собой переговорный процесс.

Спецификация требований к ПО включает в себя модель требований (функциональных и нефункциональных) и модель прецедентов (пакеты прецедентов с акторами, прецедентами и отношениями).

Диаграмма UP для задач рабочего потока определения требований относительно прецедентов и требований выглядит следующим образом:



Немного доп. информации о самих требованиях и их выработке:

Выработка требований – термин, используемый для описания деятельности по выявлению, документированию и обслуживанию ряда требований, предъявляемых к программной системе. Эффективная выработка требований – решающий фактор успеха в проектах по разработке программного обеспечения.

Требование можно определить как «подробное описание того, что должно быть реализовано». Функциональное требование – это формулировка того, что должна делать система. Нефункциональное требование – это ограничение, накладываемое на систему.

ВОПРОС №21

Понятие требования. Типы, организация и атрибуты требований. (RUP&UML2 Глава 3.6.)

Понятие требования.

Требование - это подробное описание того, что должно быть реализовано, что должна делать система. Требования указывают, что должно быть построено, но не говорят, как это сделать.

Спецификация требований к программному обеспечению (Software requirements specification, SRS) включает Модель требований (Requirements model) и Модель прецедентов (Use case model).

Каждое требование имеет уникальный идентификатор (обычно это число), ключевое слово (shall (должен)) и описание действия.



Рис. 3.6. Формат формулировки требований

Типы требований.

Функциональное требование – это формулировка того, что должна делать система, это описание назначения системы. Например, для банкомата можно было бы определить следующие функциональные требования:

1. Система shall (должна) проверять действительность вставленной в банкомат карточки.

Нефункциональное требование – это ограничение, накладываемое на систему. Система ATM может иметь следующие нефункциональные требования?

1. Система ATM shall (должна) быть написана на C++.

Организация требований.

При использовании инструментального средства управления требованиями можно организовать требования в иерархию типов требований, которая может использоваться при классификации требований. Типы требований применяются главным образом для создания небольших групп из громадного числа неструктурированных требований. Такими группами легче управлять, что делает работу с требованиями более эффективной.

Базовое разделение на функциональные и нефункциональные требования, описанное выше, является очень простой таксономией. Но можно пойти дальше и ввести категории требований, расширяя таксономию, как показано на рисунке.

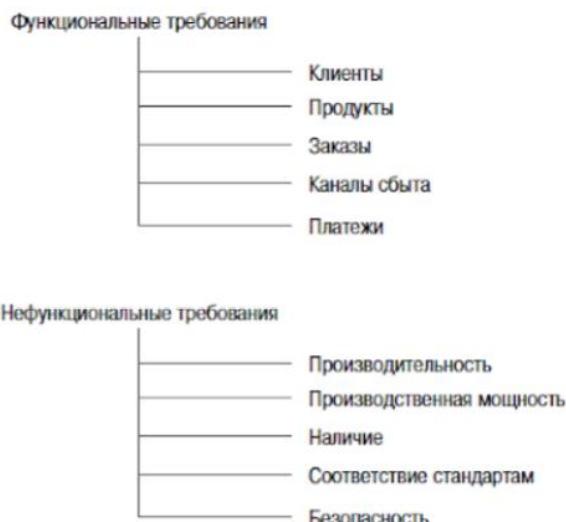


Рис. 3.7. Иерархия типов требований

Атрибуты требований.

У каждого требования может быть ряд атрибутов, фиксирующих дополнительную информацию (метаданные) о требовании.

Каждый атрибут требования имеет описательное имя и значение. Например, у требования может быть атрибут dueDate, значением которого является дата выполнения данного требования.

Самым распространенным атрибутом требований является priority (приоритет). Обычно для назначения приоритетов применяется набор критериев MoSCoW.

Значения атрибута Priority	Семантика
Must have (обязан иметь)	Обязательные требования, являющиеся фундаментальными для системы.
Should have (должен иметь)	Важные требования, которые могут быть опущены.
Could have (мог бы иметь)	По-настоящему необязательные требования (реализуются, если есть на это время).
Want to have (хотел бы иметь)	Требования, которые могут подождать до следующих версий системы.

Атрибуты RUP:

Атрибут	Семантика
Status (статус)	Может иметь одно из следующих значений: Proposed (предложенные) – требования, которые находятся в состоянии обсуждения и еще не утверждены. Approved (одобренные) – требования, утвержденные для реализации. Rejected (отклоненные) – требования, которые решено не реализовывать. Incorporated (включенные) – требования, которые были реализованы в определенной версии.
Benefit (полезность)	Может иметь одно из следующих значений: Critical (критичное) – требование <i>должно</i> быть реализовано, в противном случае система не будет принята заинтересованными сторонами. Important (важное) – требование может быть опущено, но это неблагоприятно отразится на удобстве использования системы и удовлетворении заинтересованных сторон. Useful (полезное) – требование может быть опущено без существенного влияния на приемлемость системы.
Effort (трудоемкость)	Оценка времени и ресурсов, необходимых для реализации возможности, выраженная в человеко-часах или другими методами, например методом функциональных точек (www.ifpug.org)
Атрибут	Семантика
Risk (риск)	Риск, связанный с добавлением этой возможности: High (высокий), Medium (средний) или Low (низкий).
Stability (стабильность)	Оценка вероятности того, что по каким-то причинам требование будет изменено: High (высокая), Medium (средняя) или Low (низкая).
TargetRelease (целевая версия)	Версия продукта, в которой требование должно быть реализовано.

ВОПРОС №#22

Поиск и выявление требований. (RUP&UML2 Глава 3.7)

Требования следуют из контекста моделируемой системы. В этот контекст входят:

1. Непосредственные пользователи системы;
2. Другие заинтересованные стороны (например, руководители, специалисты обслуживания, установщики);
3. Другие системы, с которыми взаимодействует данная система;
4. Аппаратные устройства, с которыми взаимодействует данная система;
5. Правовые и регулирующие ограничения;
6. Технические ограничения;
7. Коммерческие цели.

Как правило, выработка требований начинается с документа, описывающего в общих чертах (vision) то, что собирается делать система и какие услуги она будет предоставлять ряду заинтересованных сторон. Назначение этого документа – обозначить наиболее важные цели системы с точки зрения заинтересованных сторон.

Выяснение требований

Три фильтра – пропуск, искажение и обобщение.

1. пропуск – информация отфильтровывается;
2. искажение – информация изменяется взаимосвязанными механизмами вымысла и представления;
3. обобщение – информация обобщается в правила, убеждения и понятия об истинности и ложности.

Интервью

Проведение интервью с заинтересованными сторонами является самым прямым способом сбора требований. Обычно более полную информацию можно получить при интервьюировании один на один. Основные моменты указаны ниже.

1. Не заблуждайтесь по поводу решения
2. Задавайте контекстно свободные вопросы - Вопросы, которые не предполагают какого либо конкретного ответа и заставляют интервьюируемого говорить о проблеме.
3. Слушать - Единственный способ выяснить, чего хотят заинтересованные стороны, поэтому дайте им возможность поговорить.
4. Не занимайтесь телепатией. Телепатия – это заблуждение по поводу того, что вам известны чьи то чувства, желания или мысли, базирующееся на том, что вы чувствовали бы, желали бы или думали бы в подобной ситуации.
5. Запаситесь терпением! - Обстановка, в которой проводится интервью, может оказывать большое влияние на качество получаемой информации.
6. Лучший способ записи информации во время интервью – бумага и ручка!

Анкеты

Анкеты не заменяют интервью. Анкеты могут быть полезным дополнением к интервью. Они хорошо подходят для получения ответов на конкретные закрытые вопросы. Можно выделить из интервью ключевые вопросы, объединить их в анкету, а затем

распространить ее на более широкую аудиторию. Это поможет проверить, правильно ли вы понимаете требования.

Семинар по определению требований

Семинар – это один из самых эффективных способов сбора информации, относящейся к требованиям. Для выявления ключевых требований организуется семинар и основные заинтересованные стороны приглашаются принять в нем участие. Во встрече должны принимать участие руководитель проекта, разработчик требований, основные заинтересованные стороны и специалисты в определенной области. Процедура проведения подобного семинара следующая:

1. Объясните, что сбор требований проводится методом мозговой атаки.
 - 1.1 Все идеи принимаются как хорошие.
 - 1.2 Идеи записываются, но не обсуждаются – никогда ни о чем не спорьте, просто записывайте и двигайтесь дальше. Все будет проанализировано позже
2. Попросите членов команды назвать ключевые требования к системе.
 - 2.1 Напишите каждое требование на стикере.
 - 2.2 Приклейте стикер на стену или доску.
3. Затем можно еще раз просмотреть все выявленные требования и напротив каждого записать дополнительные атрибуты.

После встречи результаты анализируются и превращаются в требования. Выработка требований – это итеративный процесс, в котором требования выявляются по мере уточнения понимания нужд заинтересованных сторон.

ВОПРОС № 23

Рабочий поток анализа

Аналитическая работа начинается в конце фазы Начало и является главной задачей фазы Уточнение. Сильно пересекается с определением требований.

Цель - создание аналити КАК она это делает).

Артефакты:

- классы анализа - ключевые бизнес-понятия;
- реализаций прецедентов - иллюстрация взаимодействия классов анализа для реализации поведения системы, описанного прецедентами.

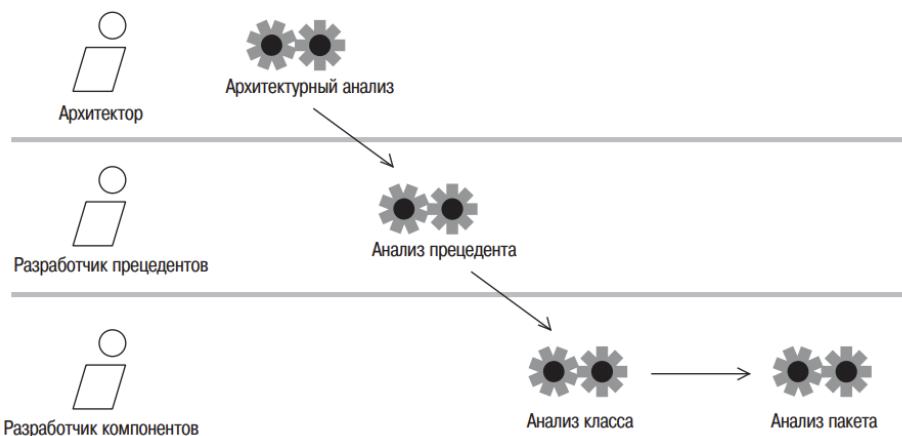


Рис. 6.4. Рабочий поток анализа в UP. Воспроизведено с рис. 8.18 [Jacobson 1] с разрешения издательства Addison-Wesley

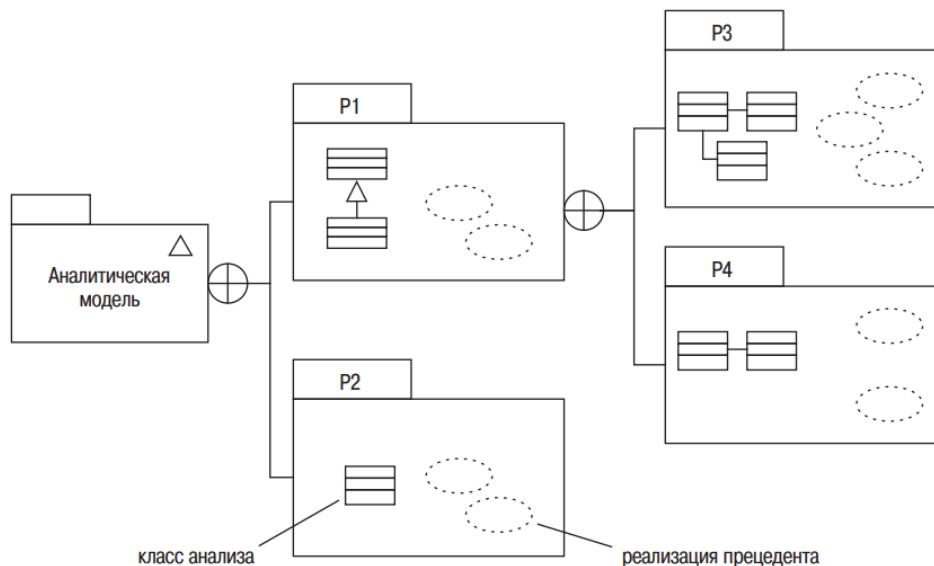


Рис. 6.3. Метамодель аналитической модели

ВОПРОС №24

Практические правила построения аналитической модели (RUP&UML2 Глава 6.5)

Источник: RUP&UML2 Глава 6.5

Цель рабочего потока анализа (с точки зрения ОО анализа) – создание аналитической модели. Данная модель фокусируется на том, **что** должна делать система(вопрос “**как**” решается в рабочем потоке *проектирования*).

Артефакты в рабочем потоке анализа:

1. классы анализа
2. реализации прецедентов

Аналитическая модель системы среднего размера и сложности включает от 50 до 100 классов анализа.

- **Важно ограничиться** лишь теми **классами**, которые являются частью словаря предметной области (не помещать проектные классы, например, классы для установки соединения. это -- в рабочий поток проектирования, так как это относится к реализации)
- Аналитическая модель создается **на языке сферы деятельности** и абстракции формируют часть словаря бизнес-сферы.
- Каждая диаграмма должна раскрывать **важные части поведения системы**.
- **Не надо углубляться** в детали того, как будет работать система (это в проектировании).
- **Минимизировать связанность** (меньше ассоциаций между классами)
- Возникновение **наследования при естественной и очевидной иерархии** абстракций, не применять для повторного использования кода.
- Модель **полезна для всех заинтересованных сторон**.
- Стремление к **простоте** модели

ВОПРОС №25

Анализ прецедента, классы анализа, практические правила создания классов анализа. (RUP&UML2 Главы 8.2,8.3.)

Анализ прецедента

Результатом рабочего потока UP Анализ прецедента являются классы анализа и реализации прецедентов.

Входные данные анализа прецедента:

- Бизнесмодель – в распоряжении разработчиков модели может быть (а может и не быть) бизнесмодель моделируемой системы. Если она уже есть, это превосходный источник требований.
- Модель требований – процесс создания этой модели описан в главе 3. Требования (этот артефакт затушеван, чтобы показать изменение по сравнению с исходным рисунком) обеспечивают полезные входные данные для процесса моделирования прецедентов. В частности, функциональные требования предложат прецеденты и актеров, а не функциональные требования – то, что, возможно, надо иметь в виду при создании модели прецедентов.
- Модель прецедентов – результат деятельности моделирования прецедентов, т. е. формы выработки требований. В этой модели четыре компонента:
 - Граница системы – прямоугольник, очерчивающий прецеденты для обозначения края, или границы, моделируемой системы. В UML 2 эту границу называют контекстом системы (subject).
 - Актеры – роли, выполняемые людьми или сущностями, использующими систему.
 - Прецеденты – то, что актеры могут делать с системой.
 - Отношения – значимые отношения между актерами и прецедентами.

Модель прецедентов является основным источником объектов и классов. Это основные исходные данные для моделирования классов.

- Описание архитектуры – представление важных с архитектурной точки зрения аспектов системы. Может включать фрагменты UML моделей, вставленные в пояснительный текст. Создается архитекторами на основании данных, полученных от аналитиков или проектировщиков.

Классы анализа

Классы анализа – это классы, которые:

- представляют четкую абстракцию предметной области (problem domain);
- должны проецироваться на реальные бизнеспонятия (и быть аккуратно поименованы соответственно этим понятиям).

Самое важное свойство класса анализа – он должен четко и однозначно проецироваться в некоторое реальное прикладное понятие, например покупатель, продукт или счет.

В классах анализа содержатся только ключевые атрибуты и обязанности, определенные на очень высоком уровне абстракции.

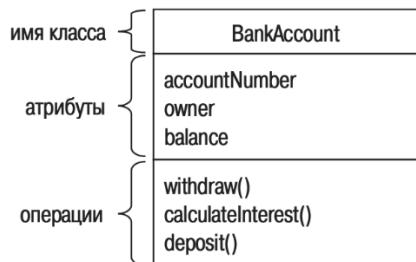
Базовый синтаксис класса анализа всегда избегает деталей реализации.

Минимальная форма класса анализа включает следующее:

- Имя – обязательно.
- Атрибуты – имена атрибутов являются обязательными, хотя на данном этапе могут моделироваться только важные предполагаемые атрибуты. Типы атрибутов считаются необязательными.

- Операции – в анализе операции могут быть всего лишь очень приблизительными формулировками обязанностей класса. Параметры и возвращаемые типы операций приводятся только в том случае, если они важны для понимания модели.
- Видимость – обычно не указывается.
- Стереотипы – могут указываться, если это приводит к улучшению модели.
- Помеченные значения – могут указываться, если это улучшает модель.

Пример класса анализа:



Основное назначение класса анализа состоит в том, что в нем делается попытка уловить суть абстракции, а детали реализации оставляют на этап проектирования.

Признаки хорошего класса анализа можно кратко охарактеризовать следующим образом:

- его имя отражает его назначение;
- он является четкой абстракцией, моделирующей один конкретный элемент предметной области;
- он проецируется на четко определяемую возможность предметной области;
- у него небольшой четко определенный набор обязанностей;
- у него высокая внутренняя связность (cohesion);
- у него низкая связанность с другими классами (coupling).

Очень важно, чтобы классы анализа имели связный набор обязанностей, абсолютно соответствующих назначению класса (которое выражено в его имени) и реальной «сущности», моделируемой классом. Обязанность – это контракт или обязательство класса по отношению к его клиентам. По существу, обязанность – это сервис, который класс предлагает другим классам.

Хорошие классы обладают минимальной связанностью (coupling) с другими классами. Связанность измеряется количеством классов, с которыми данный класс имеет отношения. Равномерное распределение обязанностей между классами обеспечит в результате низкую связность. Размещение управления или множества обязанностей в одном классе приводит к повышению связанности этого класса.

Практические правила создания класса анализа

- В каждом классе должно быть три-пять обязанностей – необходимо стремиться сохранить максимальную простоту классов. Обычно число обязанностей, которые они могут поддерживать, ограничиваются тремя-пятью.
- Ни один класс не является изолированным – суть хорошего ОО анализа и проектирования во взаимодействии классов друг с другом с целью предоставить пользователям значимый результат. По существу, каждый класс должен быть ассоциирован с небольшим количеством других классов, взаимодействуя с которыми он обеспечивает требуемый результат. Классы могут делегировать некоторые из своих обязанностей другим «вспомогательным» классам, предна значенным для выполнения именно этой конкретной функции.

- Остерегайтесь создания множества очень мелких классов – иногда это может нарушить баланс распределения обязанностей. Если в модели масса мелких классов, каждый из которых имеет одну-две обязанности, необходимо тщательно проанализировать ситуацию с целью объединения нескольких мелких классов в большие.
- Следует опасаться и противоположной ситуации, когда в модели несколько очень больших классов, многие из которых обладают большим числом (> 5) обязанностей. Стратегия в этом случае такова: по очереди рассмотреть эти классы и проанализировать возможность их разбиения на несколько меньших классов с допустимым количеством обязанностей.
- Остерегайтесь «функций»; функция – это на самом деле обычная процедурная функция, выдаваемая за класс.
- Опасайтесь всемогущих классов – классов, которые, кажется, делают все. Для решения этой проблемы необходимо посмотреть, можно ли разбить обязанности всемогущего класса на связные подмножества. Если да, то, вероятно, каждый из этих связных наборов обязанностей можно выделить в отдельный класс. Тогда поведение, предлагаемое исходным всемогущим классом, можно было бы получить за счет взаимодействия этих меньших классов.
- Необходимо избегать «глубоких» деревьев наследования (3 и более уровней) – суть проектирования хорошей иерархии наследования в том, что каждый уровень абстракции должен иметь четко определенное назначение. Широко распространенной ошибкой является использование иерархии для реализации некоторого рода функциональной декомпозиции, где каждый уровень абстракции имеет только одну обязанность. В анализе наследование используется только там, где есть явная и четкая иерархия наследования, происходящая непосредственно из предметной области.

ВОПРОС №26

Выявление классов анализа с помощью CRC модели.

CRC – class-responsibilities-collaborators, класс-обязанности-участники. Использует стикер (клеящуюся записку).

CRC – это техника мозгового штурма, при которой важные моменты предметной области записываются на стикерах.

Имя класса: BankAccount	
Обязанности: поддерживать остаток	Участники: Bank



Рис. 8.4. Разметка стикера

Процедура CRC-анализа проста. Ее основная идея – рассортировать данные, поступающие в результате анализа информации. По этой причине CRC-анализ лучше проводить в два этапа.

Этап 1: мозговой штурм – сбор информации

Привлечение заинтересованных сторон – важнейшая составляющая успеха CRC.

1. Объясните участникам, что это настоящий мозговой штурм.
 - 1.1. Все идеи принимаются как хорошие.
 - 1.2. Идеи записываются, но не обсуждаются – никаких споров, просто записывайте идеи и двигайтесь дальше. Все будет анализироваться позже.
2. Попросите членов команды назвать «сущности» их области деятельности, например покупатель, продукт.
 - 2.1. Каждую сущность запишите на kleящуюся записку в качестве предполагаемого класса или атрибута класса.
 - 2.2. Приклейте записку на стену или доску.
3. Попросите команду сформулировать обязанности этих сущностей, запишите их в ячейке обязанностей записи.
4. Работая с командой, попытайтесь установить классы, которые могут работать совместно. Перегруппируйте записи на доске соответственно такой организации классов и нарисуйте линии между ними. В качестве альтернативы впишите имена участников в соответствующую ячейку записи.

Этап 2: анализ информации

Важные бизнес-понятия обычно становятся классами.

Участники – ОО аналитики и эксперты

Классы анализа должны представлять четкую абстракцию предметной области. Определенные kleящиеся записи будут представлять ключевые бизнес-понятия и, непременно, должны стать классами. Другие записи могут стать классами или атрибутами. Если по логике кажется, что записка является частью другой записи, то это верный признак того, что она представляет атрибут. Кроме того, если записка не кажется особенно важной или не отличается интересным поведением, стоит рассмотреть возможность сделать ее атрибутом другого класса. Если по поводу записи есть какие-то сомнения, просто сделайте ее классом. Важно сделать лучшее приближение, а затем довести этот процесс до логического конца. Всегда можно уточнить модель позже.

ВОПРОС №27

Рабочий поток проектирования. Связь между аналитической и проектной моделями.

Аналитическая модель - это логическая модель системы, отражающая функциональность, которую должна предоставлять система для удовлетворения требований пользователя. Проектная модель в полном объеме определяет как будет реализовываться эта функциональность. Она объединяет в себе технические решения (библиотеки классов, механизмы сохранения объектов и т. д.).

Проектная модель содержит ряд проектных подсистем. Подсистемы – это компоненты, которые могут включать различные типы элементов модели.

Проектные модели образуются:

- проектными подсистемами;
- проектными классами;
- интерфейсами;
- реализациями прецедентов – проектными;
- диаграммой развертывания.

Одними из ключевых артефактов проектирования являются интерфейсы. Они позволяют разложить систему на подсистемы, которые могут разрабатываться параллельно.

Диаграмма развертывания создается только в первом приближении, она показывает распределение программной системы на физических вычислительных узлах.

Рисунок 16.4 иллюстрирует простой пример отношения «trace» между аналитической и проектной моделями: проектная модель базируется на аналитической и может считаться просто ее улучшенной и уточненной версией.

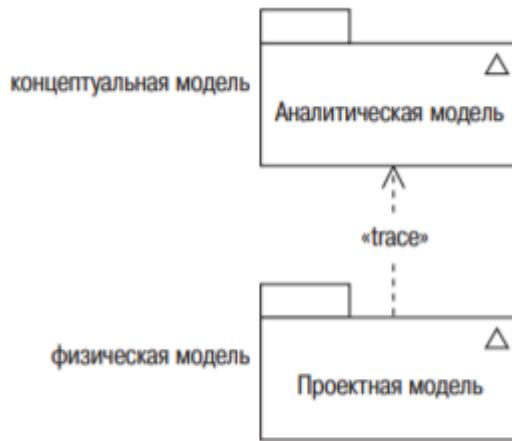


Рис. 16.4. Проектная модель – уточненная версия аналитической модели

Проектную модель можно рассматривать как уточнение аналитической модели с добавлением деталей и конкретных технических решений. Проектная модель содержит все то же самое, что и аналитическая, но все артефакты в ней проработаны более основательно и должны включать детали реализации.

На рис. 16.5 показаны отношения между ключевыми артефактами анализа и проектирования.

В силу архитектурных и технических причин один пакет анализа может быть разбит на несколько подсистем.

Классы анализа являются высокоуровневым концептуальным представлением классов системы с парой атрибутов и только ключевыми операциями. Проектный класс должен быть совершенно точно определен - все атрибуты и операции (включая возвращаемые типы и списки параметров) должны быть полностью описаны. Для реализации концептуального класса может понадобиться один или более проектных классов и/или интерфейсов.

Реализация прецедента при проектирования просто становится более детализированной.

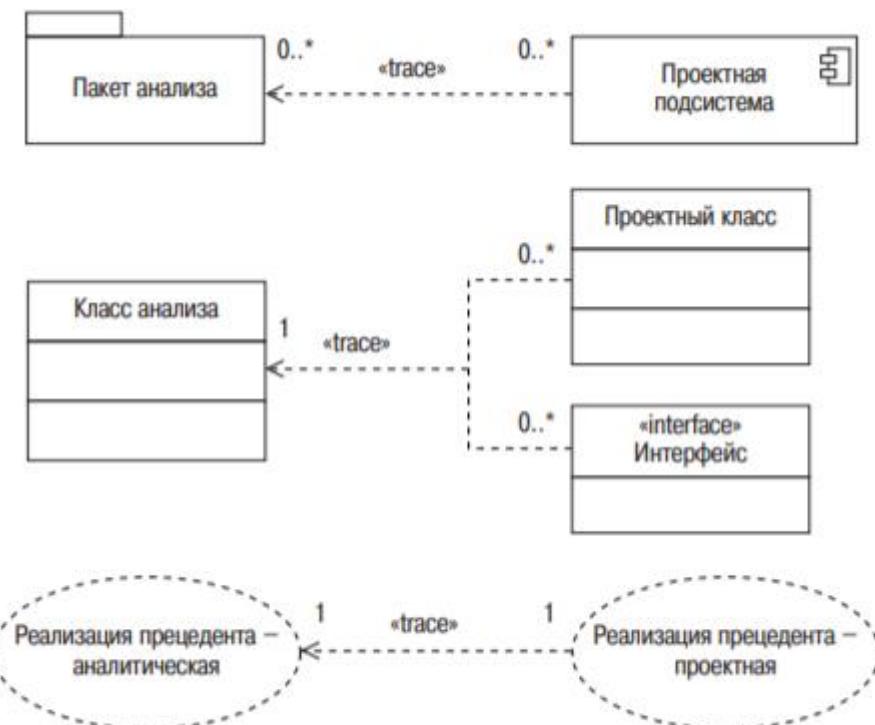


Рис. 16.5. Отношения между ключевыми артефактами анализа и проектирования

Та

к как проектная модель является уточнением аналитической, не всегда имеет смысл поддерживать обе модели. Аналитическая модель более понятная, так как содержит всего 1-10% классов подробного проектного представления. Следует поддерживать две отдельные модели, аналитическую и проектную, если система:

- большая;
- сложная;
- стратегически важная;
- подвержена частым изменениям;
- предположительно с большим сроком службы;
- для ее разработки привлекаются внешние ресурсы.

Для небольших систем может быть допустимо восстановление аналитической модели по проектной с использованием инструмента моделирования.

Источники:

- RUP&UML2, Главы 16.2, 16.3, 16.4

ВОПРОС №28

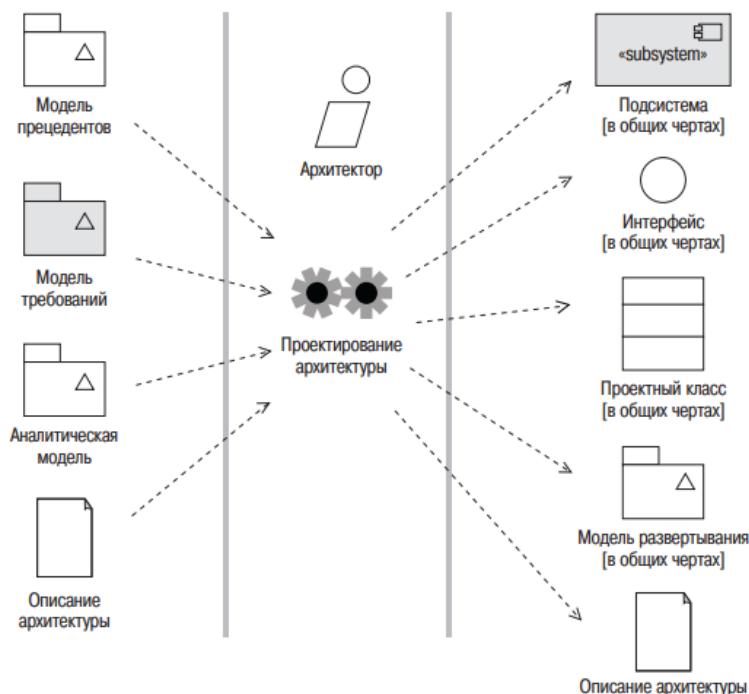
Архитектура, проектирование архитектуры.

UML основывается на 4+1 представлениях архитектуры системы:

- логическое представление – функциональность системы и словарь
- представление процессов – производительность, масштабируемость и пропускная способность системы
- представление реализации – сборка системы и управление конфигурацией
- представление развертывания – топология, распространение, поставка и установка системы
- все представления объединены представлением прецедентов, которое описывает требования заинтересованных сторон.

Проектирование архитектуры - деятельность, осуществляемая одним или более архитекторами, результатом которой является предварительное описание артефактов, важных с архитектурной точки зрения, с целью создания общего плана архитектуры системы. Описанные в общих чертах артефакты затем проходят через более детальное проектирование, в процессе которого происходит их конкретизация. Проработка деталей архитектуры системы ведется на всем протяжении завершающих этапов Уточнения и в начале Построения.

Общие и конкретизированные артефакты архитектуры представлены на картинке ниже (“Проектирование архитектуры” на ней ИМХО было бы неплохо заменить на “Конкретизация”, но в учебнике так):



Источники: RUP&UML2 Главы 1.10, 16.5

ВОПРОС №29

Проектирование прецедента. (RXUP&UYMIL2 Главы 20.2, 20.3, [20.4])

20.2. Деятельность UP: Проектирование прецедента

Деятельность UP Проектирование прецедента, заключается в выявлении проектных классов, интерфейсов и компонентов, взаимодействие которых обеспечивает поведение, описанное прецедентом.

448

Глава 20. Реализация прецедента на этапе проектирования

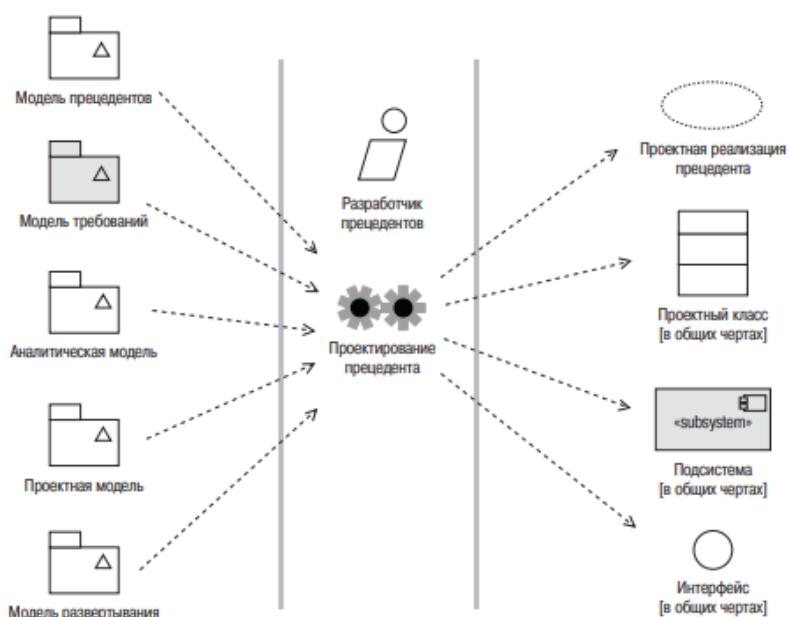


Рис. 20.2. Деятельность UP Проектирование прецедента. Адаптировано с рис. 9.34 [Jacobson 1] с разрешения издательства Addison-Wesley

Особенности процесса проектирования:

- При проектировании в реализациях прецедентов участвуют проектные классы, интерфейсы и компоненты, а не классы анализа.
- Процесс создания реализаций прецедентов на этапе проектирования часто выявляет новые нефункциональные требования и новые проектные классы.
- Проектирование реализаций прецедента помогает найти центральный механизм. Это стандартные пути решения конкретных проблем проектирования (например, организация доступа к базе данных), которые остаются неизменными в течение всего процесса разработки.

Входными артефактами Проектирования прецедента являются:

- **Модель прецедентов**
- **Модель требований**
- **Аналитическая модель**
- **Проектная модель**

(UP представляет проектную модель как входной артефакт Проектирования прецедента, чтобы обозначить итеративную природу этого процесса. По мере выявления в процессе проектирования все большего числа деталей системы происходит уточнение каждого из артефактов)

- **Модель развертывания**

(Модель развертывания также представлена как входной артефакт этой деятельности проектирования, чтобы проиллюстрировать, как все артефакты совместно эволюционируют во времени. Так же необходимо помнить, что информация, выявленная в отношении одного артефакта, может повлиять на остальные артефакты. Синхронизация всех артефактов является составной частью проектирования.)

20.3. Проектная реализация precedента

«Проектная реализация precedента» – это взаимодействие проектных объектов и проектных классов, реализующих precedент. Между аналитической и проектной реализациами precedента установлено отношение «trace». Проектирование реализации precedента определяет решения уровня реализации и реализует нефункциональные требования. Проектная реализация precedента состоит из:

- Проектных диаграмм взаимодействий;
- Диаграмм классов, включающих участвующие в ней проектные классы.

При анализе основное внимание в реализации precedентов было сосредоточено на том, что должна делать система, в проектировании, как система собирается это делать. Для этого необходимо определить детали реализации, которые игнорировались на этапе анализа. Поэтому проектные реализации precedентов более сложные и детализированные, чем исходные аналитические реализации precedентов.

20.4. Диаграммы взаимодействий при проектировании

При проектировании можно уточнять основные диаграммы взаимодействий или создавать новые для иллюстрации центральных механизмов, таких как сохранение объектов.

Диаграммы взаимодействий – основная часть проектной реализации precedента.

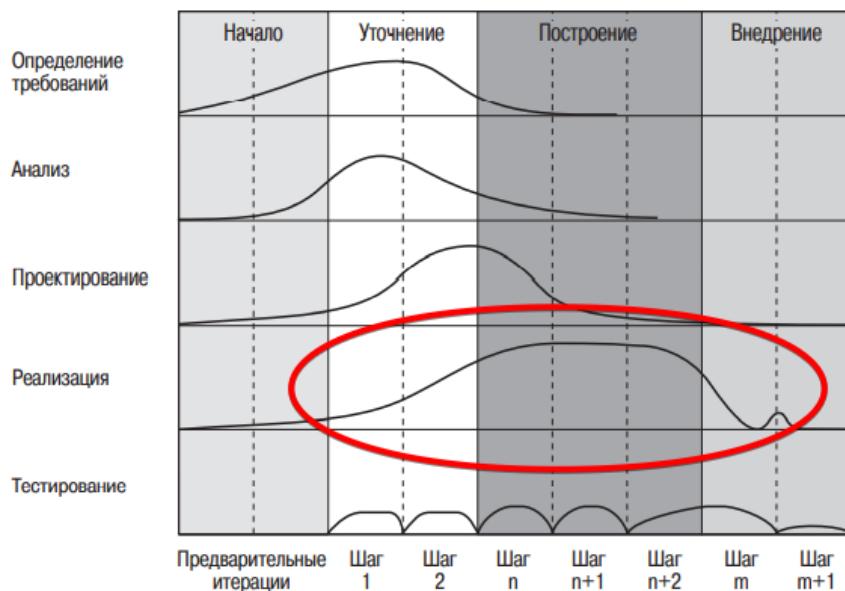
Диаграммы взаимодействий в проектировании могут быть:

- уточнением основных аналитических диаграмм взаимодействий с дополнением деталей реализации;
- абсолютно новыми диаграммами, созданными для иллюстрации технических вопросов, возникающих при проектировании.

При проектировании вводится ограниченное число центральных механизмов, таких как сохранение объектов, распределение объектов, транзакции и т. д. Часто диаграммы взаимодействий создаются именно для представления этих механизмов. Диаграммы взаимодействий, иллюстрирующие центральные механизмы, нередко охватывают несколько precedентов.

ВОПРОС №30

Рабочий поток реализации. Модель реализации.



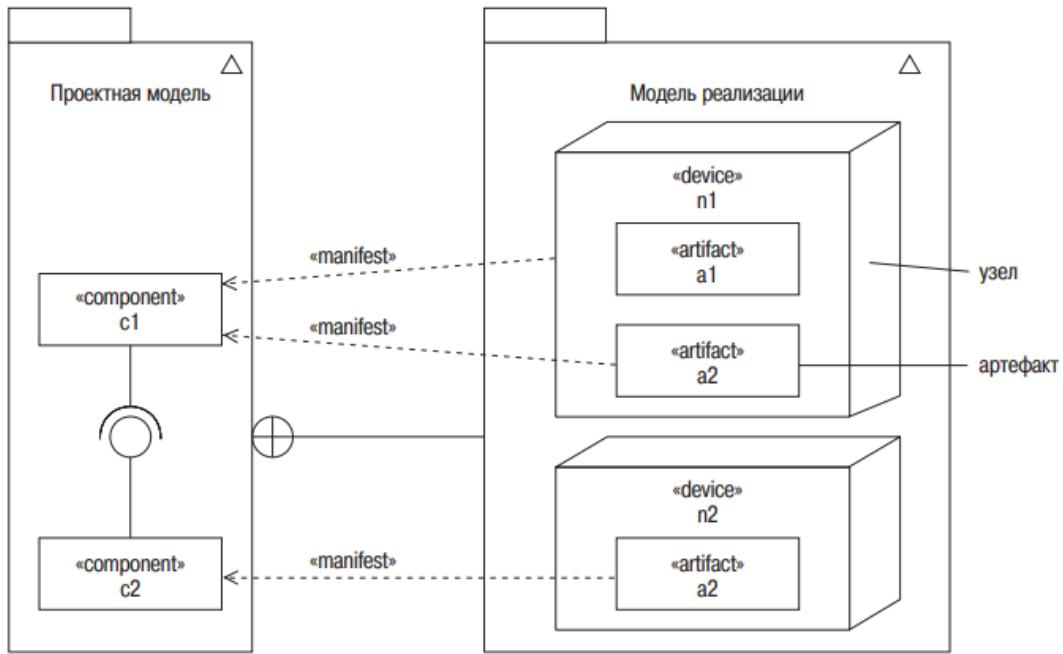
Поток реализации – основной поток фазы Построение. Реализация состоит в преобразовании проектной модели в исполняемый код.

Основное внимание в процессе реализации направлено на производство исполняемого кода. Создание модели реализации может быть побочным продуктом этого процесса, но не явной деятельностью моделирования. На самом деле многие инструментальные средства моделирования позволяют создавать модель реализации из исходного кода путем обратного проектирования. Это предоставляет программистам возможность эффективно выполнять моделирование реализации.

Однако есть два случая, когда очень важно, чтобы опытные аналитики или проектировщики провели явное моделирование реализации.

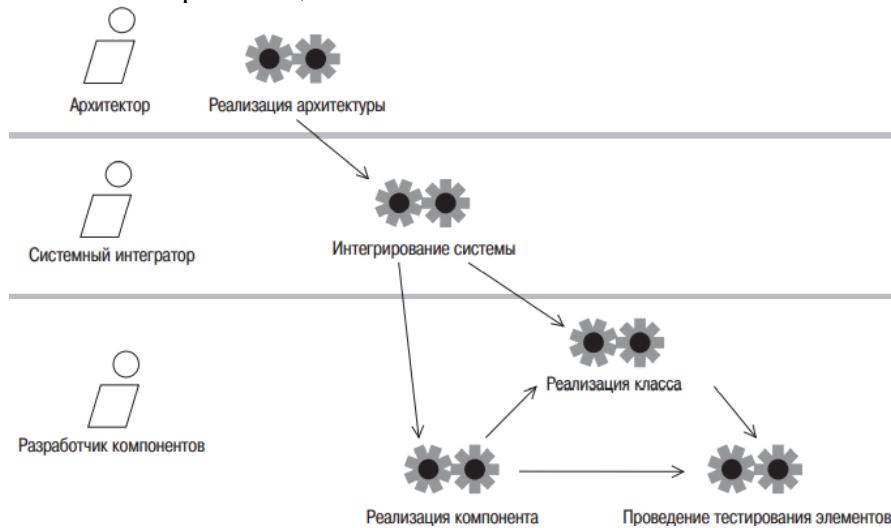
- Если предполагается генерировать код прямо из модели, понадобится определить такие детали, как исходные файлы и компоненты (если не используются применяемые по умолчанию значения у средства моделирования).
- Если осуществляется компонентно-ориентированная разработка (CBD) с целью повторного использования компонентов, распределение проектных классов и интерфейсов по компонентам становится стратегически важным вопросом. Вероятно, вы захотите сначала смоделировать эту часть проекта, а не перекладывать все на плечи одного программиста.

Отношения между моделью реализации и проектной моделью очень просты. Фактически модель реализации – это представление проектной модели с точки зрения реализации. Иными словами, модель реализации – это часть проектной модели. Она определяет, как проектные элементы представляются артефактами и как эти артефакты развертываются на узлах. Артефакты представляют описания реальных сущностей, таких как исходные файлы, а узлы представляют описания оборудования или сред выполнения, в которых эти сущности развертываются.



Отношение `«manifest»` между артефактами и компонентами указывает на то, что артефакты являются физическими представлениями компонентов. Например, компонент может состоять из класса и интерфейса, которые реализованы единственным артефактом: файлом, содержащим исходный код.

Проектные компоненты – это логические сущности, которые группируют проектные элементы. А артефакты реализации проецируются на реальные, физические механизмы группировки целевого языка реализации.



В рабочем потоке реализации участвуют архитектор, системный интегратор и разработчик компонентов. В любой из этих трех ролей могут выступать отдельные аналитики или проектировщики или небольшие команды аналитиков или проектировщиков. Их основное внимание будет направлено на производство моделей развертывания и реализации (часть реализации архитектуры).

ВОПРОС №31

Фаза Начало: цели, контрольная точка, состояние артефактов. (RUP&UML2 Главы 2.9.1, 2.9.2, 2.9.3)

Цель фазы Начало – «сдвинуть проект с мертвой точки». Начало включает:

- Обоснование выполнимости – может включать разработку технического прототипа с целью проверки правильности технологических решений или концептуального прототипа для проверки бизнес-требований.
- Разработка экономического обоснования для демонстрации того, что проект обеспечит выраженную в количественном отношении коммерческую выгоду.
- Определение основных требований для создания предметной области системы.
- Выявление наиболее опасных рисков.

Основными исполнителями в данной фазе являются руководитель проекта и архитектор системы.

В фазе Начало основное внимание обращено на определение требований и анализ. Однако если принято решение о создании технического или подтверждающего концепцию прототипа, может быть проведено некоторое проектирование и реализация. Тестирование обычно не применяется в данной фазе, поскольку единственными программными артефактами здесь являются прототипы, которые не будут больше нигде использоваться.

Тогда как многие SEP фокусируются на создании ключевых артефактов, UP применяет иной подход, ориентированный на цель. Каждая контрольная точка устанавливает определенные цели, которые должны быть выполнены для того, чтобы контрольная точка считалась пройденной. В частности, некоторые цели могут состоять в производстве определенных артефактов.

Контрольной точкой фазы Начало являются Цели жизненного цикла. Условия, которые должны быть выполнены, чтобы эта контрольная точка была пройдена, приведены в табл. 1. Мы также предлагаем набор поставляемых артефактов, которые, возможно, потребуется создать для реализации этих условий. Однако следует запомнить, что поставляемый артефакт создается, если он действительно необходим в проекте.

Таблица 1

Условия принятия	Поставляемые артефакты
Заинтересованные стороны согласовали цели проекта.	Общее описание, определяющее основные требования, характеристики и ограничения проекта.
Заинтересованные стороны определили и одобрили предметную область системы.	Исходная модель прецедентов (выполненная только на 10–20%).
Заинтересованные стороны определили и одобрили ключевые требования.	Глоссарий проекта.
Заинтересованные стороны одобрили затраты и план работы.	Исходный план проекта.
Руководитель проекта сформировал экономическое обоснование проекта.	Экономическое обоснование.

Руководитель проекта провел оценку рисков.	Документ или база данных оценки рисков.
Посредством технических исследований и/или создания прототипа была подтверждена выполнимость.	Один или более одноразовых прототипов.
Архитектура намечена в общих чертах.	Документ с исходной архитектурой.

ВОПРОС №32

Фаза Уточнение: цели, контрольная точка, состояние артефактов. (RUP&UML2 Главы 2.9.4, 2.9.5, 2.9.6)

Фазы UP и объем работ изображены на рис. 2.7. У каждой фазы есть цель, основная деятельность с акцентом на одном или более рабочих потоках, и контрольная точка.

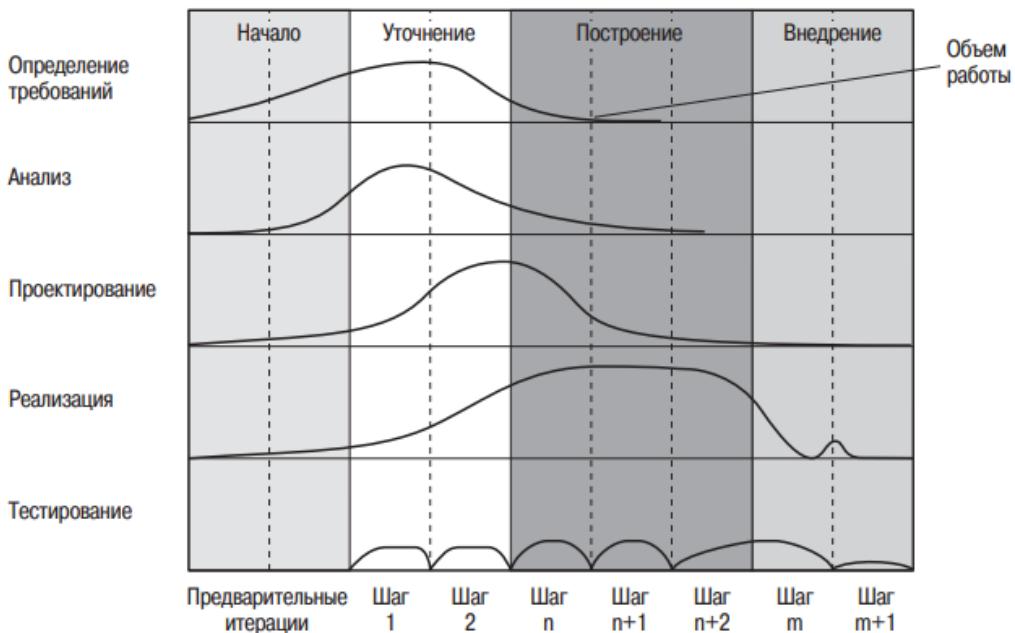


Рис. 2.7. Относительный объем работы, выполняемый в каждом из пяти рабочих потоков по мере прохождения проекта по фазам. Адаптировано с рис. 1.5 [Jacobson 1] с разрешения издательства Addison-Wesley

Основная цель фазы Уточнение – создание исполняемой базовой версии архитектуры. Это реальная исполняемая система, построенная соответственно заданной архитектуре. Это не прототип (который уйдет в корзину), а скорее всего, «первый срез» требуемой системы. Эта исполняемая базовая версия архитектуры будет дополняться по мере развития проекта и разовьется в окончательную поставляемую систему в фазах Построение и Внедрение. Поскольку следующие фазы основываются на результатах Уточнения, можно сказать, что Уточнение – решающая фаза.

Задачи Уточнения можно описать следующим образом:

- создание исполняемой базовой версии архитектуры;
- детализация оценки рисков;
- определение атрибутов качества (скорости выявления дефектов, приемлемые плотности дефектов и т. д.);
- выявление прецедентов, составляющих до 80% от функциональных требований;
- создание подробного плана фазы Построение (Цель фазы Построение – завершить определение требований, анализ и проектирование и развить исполняемую базовую версию архитектуры, созданную в фазе Уточнение, в завершенную систему.);

- формулировка предложения, включающего ресурсы, время, оборудование, штат и стоимость.

Основное внимание в фазе Уточнение направлено на рабочие потоки определения требований, анализа и проектирования. Реализация приобретает значение в конце фазы при создании исполняемой базовой версии архитектуры.

На что обращается внимание в фазе Уточнение в каждом из основных рабочих потоков:

- определение требований* – детализация предметной области системы и требований;
- анализ* – выяснение, что необходимо построить;
- проектирование* – создание стабильной архитектуры;
- реализация* – построение базовой версии архитектуры;
- тестирование* – тестирование базовой версии архитектуры.

Контрольная точка – Архитектура жизненного цикла. Условия принятия контрольной точки перечислены в таблице.

Условия принятия	Поставляемые артефакты
Создана гибкая надежная исполняемая базовая версия архитектуры.	Исполняемая базовая версия архитектуры.
Исполняемая базовая версия архитектуры демонстрирует, что важные риски были выявлены и учтены.	Статическая UML модель (объекты, типы). Динамическая UML модель (поведение объектов, взаимодействие, жц). UML модель прецедентов.
Представление продукта стабилизировалось.	Общее описание проекта.
Оценка рисков пересмотрена.	Обновленная оценка рисков.
Экономическое обоснование проекта пересмотрено и одобрено всеми заинтересованными сторонами.	Обновленное экономическое обоснование проекта.
Создан достаточно детальный план проекта, что обеспечило возможность сформулировать реалистичную заявку на затраты времени, денег и ресурсов в следующих фазах. Заинтересованные стороны одобрили план проекта.	Обновленный план проекта.

Проведена проверка экономического обоснования проекта согласно плану проекта.	Экономическое обоснование проекта.
Заинтересованные стороны достигли соглашения о продолжении проекта.	Подписанный документ.

ВОПРОС №33

**Фаза Построение: цели, контрольная точка, состояние артефактов.
(RUP&UML2 Главы 2.9.7, 2.9.8, 2.9.9)**

Фаза Построение: цели

Построение превращает исполняемую базовую версию архитектуры в законченную рабочую систему.

Цель фазы Построение – завершить определение требований, анализ и проектирование и развить исполняемую базовую версию архитектуры, созданную в фазе Уточнение, в завершенную систему.

Работа, выполняемая в рамках каждого рабочего потока:

- определение требований – выявление всех неучтенных требований;
- анализ – завершение аналитической модели;
- проектирование – завершение модели проектируемой системы;
- реализация – создание базовой функциональности;
- тестирование – тестирование базовой функциональности.

Фаза Построение: контрольная точка

Контрольная точка: базовая функциональность (программная система готова к бета-тестированию пользователем).

Условия принятия данной контрольной точки:

Условия принятия:	Поставляемые артефакты:
Программный продукт достаточно стабилен и качественен для распространения среди пользователей.	Программный продукт. UML-модель. Тестовый комплект.
Заинтересованные стороны одобрили и готовы к введению программного продукта в свое окружение.	Руководство для пользователя. Описание данной версии.
Расхождения реальных расходов с предполагаемыми приемлемы.	План проекта.

Источники: (RUP&UML2 Главы 2.9.7, 2.9.8, 2.9.9)

ВОПРОС №34

Фаза Внедрение: цели, контрольная точка, состояние артефактов. (RUP&UML2 Главы 2.9.10, 2.9.11, 2.9.12)

Внедрение - это завершающая фаза разработки ПО. Она начинается, когда завершено бета-тестирование и система окончательно развернута. При внедрении происходит устранение дефектов, найденных при бета-тестировании и подготовка к массовому выпуску программного обеспечения на все пользовательские сайты.

Цели внедрения:

- исправление дефектов;
- подготовка пользовательских сайтов под новое программное обеспечение;
- настройка работоспособности программного обеспечения на пользовательских сайтах;
- изменение программного обеспечения в случае возникновения непредвиденных проблем;
- создание руководств для пользователей и другой документации;
- предоставление пользователям консультаций;
- проведение послепроектного анализа.

В данной фазе основное внимание концентрируется на рабочих потоках реализации и тестирования. Также выполняется существенный объем проектирования. Надо стремиться к тому, чтобы в фазе Внедрение рабочие потоки определения требований и анализа оставались практически незадействованными. В противном случае с проектом не все в порядке. Ниже представлена краткая характеристика активностей для каждого рабочего потока фазы Внедрение.

- Определение требований – не проводится.
- Анализ – не проводится.
- Проектирование – изменение конструкции в случае выявления проблем при бета-тестировании.
- Реализация – настройка ПО под пользовательский сайт и исправление проблем, не выявленных при бета-тестировании.
- Тестирование – бета-тестирование и приемочные испытания на пользовательском сайте.

Это последняя контрольная точка: бета-тестирование, приемочные испытания и исправление дефектов завершены, продукт выпущен и принят в сообществе пользователей. Условия принятия этой контрольной точки приведены ниже.

- Бета-тестирование завершено, необходимые изменения сделаны и пользователи согласны с тем, что система успешно развернута.
- Сообщество пользователей активно использует продукт.
- Стратегии поддержки продукта согласованы с пользователями и реализованы.

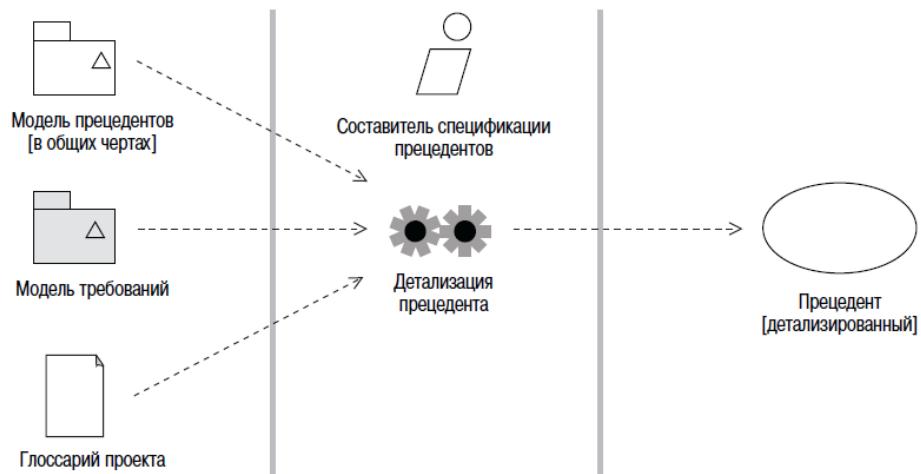
Артефактами фазы внедрения являются:

- Программный продукт.
- План поддержки пользователя.
- Обновленные руководства для пользователей.

ВОПРОС №35

Детализация прецедентов, артефакт Спецификация прецедента. (RUP&UML2 Глава 4.4, 4.5.)

После создания диаграммы прецедентов и выявления актеров и ключевых прецедентов приступают к точному определению каждого прецедента по очереди. Эту деятельность UP известна как Детализация прецедента (рисунок ниже).



Целью данной деятельности является подробное описание потока работ прецедента, так чтобы его смогли понять заказчики и пользователи.

Действие детализации прецедента включает в себя следующие шаги:

- Сбор информации о прецеденте
- Детализация потока работ прецедента
- Структурирование потока работ прецедента
- Иллюстрирование связей между субъектами и другими прецедентами
- Описание специальных требований делового прецедента
- Оценка результатов

Спецификация прецедента

Чаще всего (В разных предприятиях может использоваться свой) используется следующий шаблон для спецификации. В примере описан прецедент “выплата налога с оборота”.

имя прецедента	Прецедент: PaySalesTax
идентификатор прецедента	ID: 1
краткое описание	Краткое описание: Выплата налога с оборота в Налоговое управление по окончанию налогового периода.
актеры, вовлеченные в прецедент	Главные актеры: Time (Время)
состояние системы до начала прецедента	Второстепенные актеры: TaxAuthority (налоговое управление)
фактические этапы прецедента	Предусловия: 1. Конец налогового периода. Основной поток: неявный актер Time 1. Прецедент начинается в конце налогового периода. 2. Система определяет сумму Налога с оборота, которую необходимо выплатить Налоговому управлению. 3. Система посыпает электронный платеж в Налоговое управление.
состояние системы после окончания прецедента	Постусловия: 1. Налоговое управление получает соответствующую сумму Налога с оборота.
альтернативные потоки	Альтернативные потоки: Нет.

Rис. 4.8. Шаблон спецификации прецедента

Подробнее про пункты спецификации ([источник с простыми примерами](#) либо методичка про UML2 и UP Глава 4.5, стр. 101)

- **Имя варианта использования (ВИ или прецедент)**
Нет стандарта.
Рекомендуем начинать с глагола или глагольной группы (отглагольного существительного).
Является уникальным идентификатором в рамках модели ВИ
- **ID варианта использования**
Суррогат идентификатора. Используется для облегчения и краткости при описании ссылок
- **Краткое описание**
Изложение цели или сути варианта использования. Один абзац.
- **Действующие лица (актеры)**
Каждый ВИ всегда инициируется одним ДЛ.
Но в разные моменты времени один и тот же вариант использования может быть инициирован разными ДЛ.
Любое действующее лицо, которое может инициировать вариант использования, является основным действующим лицом. Все остальные ДЛ — второстепенные.
- **Предусловия и постусловия**
Ограничения
Предусловия определяют в каком состоянии должна находиться система, чтобы запуск ВИ был возможным. Постусловия определяют, какие условия будут истинными после завершения ВИ
- **Основной поток**
«Идеальный» ход развития событий. Все идет, как ожидается и хочется
Нет ошибок, отклонений, прерываний или ответвлений
Шаблон записи шага — <номер> <кто-либо> <совершает некоторое действие>
- **Альтернативные потоки (ветвления и повторения)**
Ветвления потока или повторения в потоке можно сократить, уменьшая число вариантов использования, но пользоваться этим надо умеренно!
Примеры:
 - Условный выбор — Если
 - 2. Если Покупатель выбирает «удалить позицию»

2.1. Система удаляет позицию из корзины

- Повторение в потоке

5.1. Для каждого найденного продукта

5.1.1. система выводит на экран миниатюрное представление продукта

- **Моделирование альтернативных потоков (как отдельный ВИ)**

У каждого ВИ есть один основной поток и может быть множество альтернативных потоков.

Альтернативные потоки часто не возвращаются в основной поток ВИ. Это связано с тем, что они обычно обрабатывают ошибки и исключения основного потока и имеют другие постуслугия. Альтернативные потоки можно задокументировать в конце ВИ или отдельно. Спецификация альтернативного потока подобна спецификации всего ВИ.

Альтернативные потоки могут быть инициированы тремя разными способами:

1. вместо основного потока. Такая ситуация часто возникает в вариантах использования типа CRUD (Create, Read, Update, Delete)
2. после определенного этапа основного потока
3. в любой момент в ходе выполнения основного потока

- **Выявление альтернативных потоков**

1. Изучите каждый шаг основного потока

2. Выделите возможные альтернативы основному потоку

1. Ошибки, которые могут возникнуть в основном потоке
2. Прерывания, которые могут случиться в конкретной точке основного потока
3. Прерывания, которые могут произойти в любой точке основного потока

3. Задокументируйте только самые важные альтернативные потоки

ВОПРОС №36

Артефакт Vision (Концепция). Предназначение разделов.
RUP_classic.2002.05.00/RationalUnifiedProcess/process/artifact/ar_vision.htm

“Видение” определяет взгляд заинтересованных сторон на продукт, который будет разработан, с учетом основных потребностей и характеристик заинтересованных сторон. Содержащий схему предполагаемых основных требований, он обеспечивает договорную основу для более подробных технических требований.

Цель:

Документ «Видение» обеспечивает высокоуровневую - иногда договорную - основу для более подробных технических требований. Также может быть формальная спецификация требований. Vision фиксирует очень высокоуровневые требования и конструктивные ограничения, чтобы дать читателю представление о системе, которую необходимо разработать. Он обеспечивает входные данные для процесса утверждения проекта и, следовательно, тесно связан с экономическим обоснованием. Он сообщает основные «почему и что», относящиеся к проекту, и является мерой, по которой должны проверяться все будущие решения.

Документ «Видение» будут читать менеджеры, финансирующие органы, занимающиеся моделированием вариантов использования, и разработчики в целом.

Сроки

Документ «Видение» создается на ранней стадии начального этапа и используется в качестве основы для бизнес-обоснования (см. Артефакт: бизнес-модель) и первого черновика списка рисков (см. «Артефакт: список рисков»).

Документ Vision служит входными данными для моделирования вариантов использования, а также обновляется и поддерживается как отдельный артефакт на протяжении всего проекта

Обязанности

Системный аналитик несет ответственность за целостность документа Vision, гарантируя, что:

- Документ Vision обновляется и распространяется.
- Принимается во внимание вклад всех заинтересованных сторон.

Структура документа Vision

РАЗДЕЛ	ПОДРАЗДЕЛ	ОПИСАНИЕ							
Introduction (Введение) - данный раздел дает обзор на весь документ, здесь описывается цель разрабатываемой системы, сфера применения данной системы, различные определения и аббревиатуры необходимые для разъяснения.	Purpose (Назначение)	Здесь описывается цель документа							
	Scope (Область применения)	Краткое описание области применения данного документа, к какому проекту он относится, кто им будет пользоваться и т.д							
	Definitions, Acronyms, and Abbreviations (Определения и аббревиатуры)	Значение терминов и аббревиатур, которые употребляются в данном документе. Возможно указание ссылки на Глоссарий проекта							
	References (Ссылки)	Список названия документов, на которые ссылаешься в данном, укажите их источники							
	Overview (Обзор документа)	Краткое описание остальных разделов документа.							
РАЗДЕЛ	ПОДРАЗДЕЛ	ОПИСАНИЕ							
Positioning (Позиционированье)	Business Opportunity (Возможности для бизнеса)	Здесь описывается бизнес-задача, которую решает или будет решать проект, система, актуальность создания проекта, аргументация актуальности.							
	Problem Statement (Постановка задачи)	Данный подраздел описывает как проект решает проблемы конкретных групп пользователей. Описываются <ul style="list-style-type: none"> • проблемы • группы пользователей, которым существенна данные проблемы • результат проблем (каким образом проблема влияет на пользователей) • решение (что может решить проблему в рамках проекта) <table border="1" style="width: 100%; border-collapse: collapse;"> <tr> <td style="padding: 2px;">The problem of</td> <td style="padding: 2px;"><i>[describe the problem]</i></td> </tr> <tr> <td style="padding: 2px;">affects</td> <td style="padding: 2px;"><i>[the stakeholders affected by the problem]</i></td> </tr> <tr> <td style="padding: 2px;">the impact of which is</td> <td style="padding: 2px;"><i>[what is the impact of the problem?]</i></td> </tr> <tr> <td style="padding: 2px;">a successful solution would be</td> <td style="padding: 2px;"><i>[list some key benefits of a successful solution]</i></td> </tr> </table>	The problem of	<i>[describe the problem]</i>	affects	<i>[the stakeholders affected by the problem]</i>	the impact of which is	<i>[what is the impact of the problem?]</i>	a successful solution would be
The problem of	<i>[describe the problem]</i>								
affects	<i>[the stakeholders affected by the problem]</i>								
the impact of which is	<i>[what is the impact of the problem?]</i>								
a successful solution would be	<i>[list some key benefits of a successful solution]</i>								
Product Position Statement (Позиция продукта на рынке)	Здесь описываются уникальные особенности проекта, которые отличают его от аналогов на рынке Описывается <ul style="list-style-type: none"> • для кого проект (потенциальный клиент) • потребность, которая необходима клиенту • наименование продукта и его категория • ключевой функционал, причина приобретения продукта клиентом • название и характеристики продуктов-конкурентов • отличительная особенность разрабатываемого продукта (список новых возможностей и характеристик, которые новый продукт предлагает этой группе потребителей) 								

		<table border="1"> <tr><td>For</td><td>[target customer]</td></tr> <tr><td>Who</td><td>[statement of the need or opportunity]</td></tr> <tr><td>The (product name)</td><td>is a [product category]</td></tr> <tr><td>That</td><td>[statement of key benefit; that is, the compelling reason to buy]</td></tr> <tr><td>Unlike</td><td>[primary competitive alternative]</td></tr> <tr><td>Our product</td><td>[statement of primary differentiation]</td></tr> </table>	For	[target customer]	Who	[statement of the need or opportunity]	The (product name)	is a [product category]	That	[statement of key benefit; that is, the compelling reason to buy]	Unlike	[primary competitive alternative]	Our product	[statement of primary differentiation]
For	[target customer]													
Who	[statement of the need or opportunity]													
The (product name)	is a [product category]													
That	[statement of key benefit; that is, the compelling reason to buy]													
Unlike	[primary competitive alternative]													
Our product	[statement of primary differentiation]													
РАЗДЕЛ	ПОДРАЗДЕЛ	ОПИСАНИЕ												
Stakeholder and User Descriptions (Описание заинтересованных лиц) - в данном разделе дается описание заинтересованных лиц и пользователей проекта, а также описание проблем, которые они хотят решить с помощью разрабатываемого продукта. Здесь не описываются конкретные требования этих людей, но скорее причины, почему данные требования возникли	Market Demographics (Демография рынка)	Описывается ситуация на рынке сбыта продукта, оценивается его размер и рост числа потенциальных пользователей, основные тенденции в использовании технологий.												
	Stakeholder Summary (Описание заинтересованных лиц)	Описываютя группы лиц, заинтересованные в разработке вашего продукта (не все из них являются его пользователями). <ul style="list-style-type: none"> • наименование группы лиц • описание • обязанности (что делают эти лица) 												
	User Summary (Описание пользователей)	Здесь приводится описание всех групп потенциальных пользователей <ul style="list-style-type: none"> • наименование группы лиц • описание (кем они являются в рамках разрабатываемого продукта) • обязанности (что делают эти лица) • заинтересованное лицо 												
	User Environment (Описание рабочего окружения пользователей)	Здесь описывается рабочее окружение пользователей в виде ответов на вопросы: <ul style="list-style-type: none"> • Какое количество людей требуется для выполнения 1 задачи? Меняется ли это число? • Сколько длится выполнение 1 задачи? Сколько длится выполнение 1 подзадачи? Меняется ли это число? • Есть ли особые ограничения на рабочее окружение? (мобильность и т.п.) • Какой платформой пользуются пользователи? В будущем будут изменения? • Какими другими приложениями пользуются пользователи? Нужна ли разрабатываемому продукту интеграция с ними? 												
	Stakeholder Profiles (Профили заинтересованных лиц)	Отличается от Stakeholder Summary тем, что добавляется следующая информация: <ul style="list-style-type: none"> • оценка уровня компетенций • критерий успеха (в чем выгода при использовании продукта) • вовлеченность (роль в контексте RUP, каким образом лицо вовлечено в разработку) • ожидаемый результат (что, какие результаты ожидает стейххолдер кроме разрабатываемой системы) • проблем, которые мешают успешно использовать систему / любые уместные комментарии 												
	User Profiles (Профили пользователей)	Отличается от User Summary тем, что добавляется следующая информация: <ul style="list-style-type: none"> • оценка уровня компетенций 												

		<ul style="list-style-type: none"> • критерий успеха (в чем выгода при использовании продукта) • вовлеченность (роль в контексте RUP, каким образом лицо вовлено в разработку) • ожидаемый результат (что, какие результаты ожидает стейхолдер кроме разрабатываемой системы) • проблем, которые мешают успешно использовать систему / любые уместные комментарии 				
	Key Stakeholder or User Needs (Ключевые потребности заинтересованных лиц или пользователей)	<p>Для каждой проблемы ответьте на следующие вопросы:</p> <ul style="list-style-type: none"> • Какова причина возникновения данной проблемы? • Как она решается сейчас? • Какое решение требуется заинтересованному лицу/пользователям? <p>Важно осознавать, что у каждой проблемы есть свой приоритет, это позволяет выявить те из них, которые обязательно необходимо решить.</p>				
	Alternatives and Competition (Конкурентные решения и альтернативы)	Перечислите альтернативы разрабатываемому продукту - это могут быть как продукты-конкуренты от других компаний, так и такие условия, при которых разработка не потребуется. Опишите преимущества и недостатки каждой из альтернатив				
РАЗДЕЛ	ПОДРАЗДЕЛ	ОПИСАНИЕ				
Product Overview (Обзор продукта) - данный раздел описывает разрабатываемый продукт - его возможности, взаимодействие с другими приложениями и системами.	Product Perspective (Перспектива продукта) Summary of Capabilities (Обзор возможностей)	<p>Данный раздел описывает продукт в контексте других связанных с ним, а также окружением пользователя. Продукт может быть полностью независим от других систем или быть частью другой системы (тогда данный раздел стоит посвятить взаимодействию подсистем между собой, желательно описать данное взаимодействие с помощью диаграмм)</p> <p>Опишите главные преимущества и ключевые особенности продукта. Здесь заполняется таблица, где описывается выгода ключевые возможности, выгоды пользователей, и каким функционалом это реализуется.</p> <table border="1"> <thead> <tr> <th>Customer Benefit (Выгода пользователей)</th><th>Supporting Features (Описание функционала)</th></tr> </thead> <tbody> <tr> <td>Повышается скорость реагирования на происшествия.</td><td>Датчики обнаружения, установленные в вентиляц скрытых местах, посыпают сигналы о приближении нарушителя</td></tr> </tbody> </table>	Customer Benefit (Выгода пользователей)	Supporting Features (Описание функционала)	Повышается скорость реагирования на происшествия.	Датчики обнаружения, установленные в вентиляц скрытых местах, посыпают сигналы о приближении нарушителя
Customer Benefit (Выгода пользователей)	Supporting Features (Описание функционала)					
Повышается скорость реагирования на происшествия.	Датчики обнаружения, установленные в вентиляц скрытых местах, посыпают сигналы о приближении нарушителя					
	Assumptions and Dependencies (Влияющие факторы и зависимости)	Перечислите факторы, которые могут повлиять на преимущества продукта, а также на изменение данного документа				
	Cost and Pricing (Цены)	В данном разделе зафиксируйте все ценовые факторы, которые могут повлиять на сбыт продукта - например, цена носителей для распространения, цена печати руководства пользователя, упаковки и т.п				
	Licensing and Installation (Лицензирование и установка)	Дополнительные требования к лицензированию и установке продукта могут также повлиять на разработку системы и должны быть перечислены в данном разделе.				

Product Features (Особенности продукта) - перечислите и кратко опишите особенности продукта - возможности системы, которые необходимо реализовать для того, чтобы система приносила пользу пользователям. При описании не стоит вдаваться в технические детали, оно должно быть всем понятно, лучше фокусироваться на обслуживаемом функционале и на причинах необходимости его разработки. Каждая особенность должна быть подробно описана в Описании Прецедента
Constraints (Ограничения) - Перечислите требования к архитектуре клиента и сервера, установленным программам и версиям библиотек
Quality Ranges (Оценка качества) - В данном разделе определите границы параметров производительности, надежности, отказоустойчивости, удобства использования и т.п
Precedence and Priority (Приоритетные особенности) - В данном разделе определите приоритет разрабатываемых особенностей продукта, поясните свою оценку. Лучше воспользоваться таблицей, приведенной ниже
Other Product Requirements (Прочие требования) - В данном разделе перечислите все применяемые стандарты, а также требования к производительности и окружению пользователя.

Дополнительная информация

Видение проекта должно изменяться по мере развития понимания требований, архитектуры, планов и технологий. Однако он должен меняться медленно и нормально на протяжении более ранней части жизненного цикла.

Важно выразить видение в терминах его вариантов использования и основных сценариев по мере их разработки, чтобы вы могли видеть, как видение реализуется с помощью вариантов использования. Сценарии использования также обеспечивают эффективную основу для развития набора тестовых примеров.

Первоначальным автором может быть кто угодно, но после создания проекта за видение отвечает системный аналитик.

Другое название этого документа - Документ с требованиями к продукту.

Портняжное дело

При необходимости адаптируйте под нужды вашего проекта. Как правило, рекомендуется держать Видение кратким, чтобы иметь возможность как можно скорее передать его заинтересованным сторонам, а также облегчить заинтересованным сторонам обзор и усвоение. Это достигается путем включения только самых важных запросов и функций заинтересованных сторон и избегания подробных требований. Подробности могут быть зафиксированы в других артефактах требований или в приложениях.

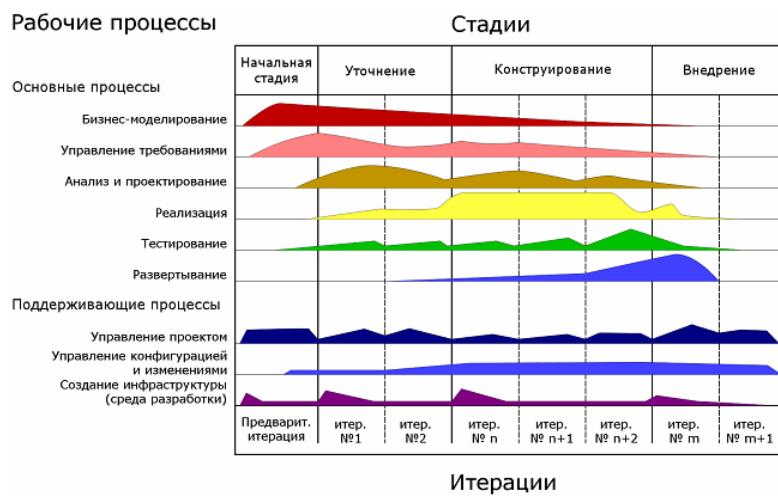
ВОПРОС №37

Артефакт SDP (План разработки). Предназначение разделов.

RUP_classic.2002.05.00/RationalUnifiedProcess/process/artifact/ar_sdp.html

Rational Unified Process (RUP) — методология разработки программного обеспечения, созданная компанией Rational Software. В основе RUP лежат следующие принципы:

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



Администратор проекта составляет план разработки программного обеспечения (SDP), координируя разработку всех его компонентов и публикуя в главном SDP.

SDP или план разработки программного обеспечения – это составной документ, содержащий всю информацию, необходимую для управления проектом. Информация или непосредственно включается в содержимое SDP, или туда помещаются ссылки на другие спецификации и планы.

Одним из важных аспектов SDP является более точное определение процесса, который будет использовать проект: в этом и состоит роль описания процесса разработки проекта.

Разделы документа SDP (план разработки программного обеспечения) –

- **Introduction (Введение)**

Введение представляет собой обзор на весь документ в целом и включает в себя следующие разделы - назначение, область применения, определения и аббревиатуры,

ссылки и обзор. Может содержать в себе следующие разделы: *Purpose, Scope, Definitions, Acronyms and Abbreviations, References, Overview*

- **Project Overview (Обзор продукта)**

Project Purpose, Scope, and Objectives (Назначение, цели и контекст продукта) – в данном разделе кратко описываются назначение проекта, цели его разработки, какие артефакты будут получены в результате работы над ним.

Assumptions and Constraints (Влияющие факторы и ограничения) – перечисляются ограничения, накладываемые на данный план разработки проекта - бюджет, персонал, сроки и т.д.

Project Deliverables (Ожидаемые результаты проекта) – описываются артефакты, которые будут созданы в результате проекта.

Evolution of the Software Development Plan (План развития данного документа) – план версий данного документа, на каких этапах они должны создаваться, при каких условия нужно изменить данный документ.

- **Project Organization (Организация проекта)**

Organizational Structure (Структура команды разработки) – структура команды проекта - список сотрудников с указанием ролей, которые они занимают.

External Interfaces (Внешние интерфейсы) – как команда связана с внешним миром - контактных лиц и каналы связи.

Roles and Responsibilities (Роли и обязанности) – описание ролей в проекте и их обязанности.

- **Management Process (Процесс управление)**

Project Estimates (Оценка сроков разработки проекта) Укажите примерные сроки и стоимость разработки проекта.]

Project Plan (План проекта) – Включает в себя

- *Phase Plan (План фаз)* – какие фазы будет включать в себя разработка проекта с указанием их длительности и основных вех.
- *Iteration Objectives (Цели итераций)* – цели каждой итерации.
- *Releases (Релизы)* – релизы проекта, их назначение и даты выпуска
- *Project Schedule (Штатное расписание)* – расписание разработки проекта, с указанием трудозатрат на реализацию каждой из фаз и и итераций.
- *Project Resourcing (Ресурсы проекта)* – Staffing Plan (План по сотрудникам), Resource Acquisition Plan (План поиска сотрудников)
- *Budget (Бюджет)* – бюджет проекта

Project Monitoring and Control (Мониторинг и контроль проекта)

- *Requirements Management Plan (План управления требованиями)* – Определите, каким образом будет контролироваться процесс работы с требованиями проекта.
- *Schedule Control Plan (План управления расписанием)* – каким образом будут контролироваться сроки выполнения работ проекта.
- *Budget Control Plan (План управления бюджетом)* – каким образом будет контролироваться бюджет проекта.
- *Quality Control Plan (План управления качеством)* – каким образом будет контролироваться качество проекта.
- *Reporting Plan (План отчетности)* – как часто будут создаваться отчеты по прогрессу разработки проекта, а также каково будет их содержание.

Risk Management plan (План управления рисками) – каким образом будут контролироваться риски проекта.

Close-out Plan (План завершения проекта) – каким образом будет определяться завершение разработки проекта, какие цели должны быть достигнуты для этого, что должно быть сделано после завершения разработки.

ВОПРОС №38

Артефакт SRS (Требования к продукту). Предназначение разделов.

Спецификация требований к программному обеспечению (SRS) фиксирует полные требования к программному обеспечению для системы или ее части. При использовании моделирования вариантов использования (Use-cases) этот артефакт состоит из пакета, содержащего варианты использования модели вариантов использования и применимые дополнительные спецификации.

Спецификации требований к программному обеспечению идут рука об руку с вариантами использования (Use-cases) и дополнительными спецификациями, подразумевая, что:

- первоначально они рассматриваются на начальном этапе как дополнение к определению объема системы.
- они постепенно улучшаются на этапах разработки и написания кода.

Раздел «Введение»

Введение представляет собой обзор на весь документ в целом и включает в себя следующие разделы - назначение, область применения, определения и аббревиатуры, ссылки и обзор.

Раздел «Общее описание»

Данный раздел содержит описание факторов, влияющих на требования к продукту, сами требования отписываются в следующем разделе.

- Функционал продукта (описание основного функционала разрабатываемой системы, что она должна уметь делать).
- Описание пользователей (описание группы пользователей разрабатываемой системы, как именно они будут взаимодействовать с ней).
- Влияющие факторы и зависимости (дополнительные зависимости, которые могут повлиять на требования к системе).
- Ограничения (ограничения, накладываемые на функционал системы).

Раздел «Спецификация требований»

Данный раздел содержит описание всех требований к разрабатываемой системе. Данное описание будет использоваться как разработчиками при разработке системы, так и тестировщиками в процессе проверки её функционала.

- Функциональные требования (содержит описание функциональных требований к системе).
- Требования к удобству использования (содержит требования к удобству использования системы. Например, время обучения обычного и опытного пользователя, среднее время выполнения типовых задач и т.д.).
- Требования к надежности (содержит требования к надежности системы. Например, её доступность (в %), среднее время между возникновением ошибок, среднее время восстановления работоспособности, точность, количество найденных критических ошибок и т.д.).
- Требования к производительности (содержит требования к производительности системы. Например, время ответа (максимальное и среднее), максимальное количество обрабатываемых транзакций в секунду, максимальное количество

одновременно работающих пользователей, использование ресурсов (память, дисковое пространство) и т.д.).

- Ограничения разработки (содержит все требования к процессу разработки. Например, используемый язык программирования, требования к процессу разработки (методологии), выбранные инструменты разработки, использование сторонних библиотек и т.д.).
- Интерфейсы (описывает интерфейсы, которые должна поддерживать система).
- Требования к лицензированию (описывает, по какой лицензии следует распространять разрабатываемый продукт).

ИСТОЧНИКИ:

1. https://sceweb.uhcl.edu/helm/RationalUnifiedProcess/process/artifact/ar_srs.htm

ВОПРОС №39

Артефакт Risk List (Список Рисков). Предназначение разделов. Управление рисками.

Список рисков и рекомендации к нему

Список рисков представляет собой список известных рисков для проекта, отсортированный в порядке убывания важности и связанный с конкретными действиями по смягчению или непредвиденным обстоятельствам.

Список рисков предназначен для отражения предполагаемых рисков для успеха проекта. Он определяет в порядке убывания приоритета события, которые могут привести к значительному отрицательному результату.

За ведение списка рисков ответственен менеджер проекта и контроль рисков ведется на протяжении всего проекта. Список создается на начальном этапе и постоянно обновляется по мере выявления новых рисков, а также снижения или устранения существующих рисков. Как минимум, он пересматривается в конце каждой итерации по мере ее оценки.

Следует отметить что оптимальный размер списка не должен превышать 20 записей, иначе за ними будет сложно смотреть.

Разделы списка рисков:

- **Введение**

Введение в Список рисков дает обзор всего документа. Он включает цель, описание сферы применения, определения, акронимы, аббревиатуры, ссылки и обзор этого Списка рисков.

- **Описание рисков**

Для каждого риска в проекте необходимо указать следующие пункты

- **Идентификатор риска**

Имя или специальный номер

- **Шкала оценки либо ранг риска**

Индикатор величины риска может быть назначен, чтобы помочь ранжировать риски

- **Краткое описание риска**

- **Влияние риска на проект**

Описать какой эффект на проект окажет риск при его наступлении

- **Признаки появления риска (индикаторы)**

Необходимо описать как будет отслеживаться появление риска

- **Стратегия смягчения**

Описываются действия выполняющиеся для предотвращения появления риска или снижения его последствия

- **Стратегия наступления**

Описываются действия, которые необходимо выполнить, чтобы снизить эффект уже наступившего риска

Управление рисками и план управления рисками

Лучше будет прочитать первый источник, тк там много и понятно написано

Управление рисками - комплекс мероприятий направленный на выявление возможных негативных факторов и оценку вероятности их возникновения.

- **Начало списка рисков**

Собрать команду проекта и, задав вопрос “что может пойти не так” выделить проблемы, которые могут повлиять на проект и которые сможем снизить или убрать. В частности, события, которые могут снизить вероятность того, что мы

сможем реализовать проект с нужными функциями, требуемым уровнем качества, вовремя и в рамках бюджета.

- **Анализ и приоритет рисков**

Методы количественного управления рисками рекомендуют устанавливать приоритеты в соответствии с общей подверженностью риску, которую он представляет для проекта. Чтобы определить подверженность каждому риску необходимо оценить следующую информацию:

- Влияние риска - отклонения графика, усилий или затрат от плана в случае возникновения риска
- Вероятность появления - вероятность того, что риск действительно возникнет
- Подверженность риску - рассчитывается путем умножения воздействия на вероятность возникновения

После определения подверженности для каждого риска вы можете отсортировать риски в порядке уменьшения подверженности, чтобы создать список 10 наиболее существенных рисков. Либо можно сгруппировать риски по категориям в зависимости от величины их воздействия на проект (например: по категориям низкий, средний или высокий).

- **Стратегии для избежания наступления рисков**

Риски могут быть вызваны нечетким определением области функциональности системы; в этом случае после устранения необязательных требований будут устраниены целые разделы списка рисков. Многие риски вызваны тем, что для выполнения работы выделено недостаточное количество ресурсов (включая время).

В других случаях можно использовать следующий прием уменьшения рисков, связанных с определенной функциональностью системы: один набор рисков (связанных с реализацией технологии собственными силами) заменяется на другой (зависимость от процессов, не находящихся под контролем коллектива разработки).

- **Стратегии для снижения влияния рисков**

Для рисков, над которыми проект имеет некоторую степень контроля, определите, какие действия будут предприняты для снижения вероятности риска или уменьшения его воздействия на проект (стратегии смягчения). Обычно риск возникает из-за недостатка информации; часто сама стратегия смягчения последствий заключается в дальнейшем исследовании темы с целью уменьшения неопределенности.

Существуют риски, в отношении которых можно предпринять определенные действия, чтобы либо материализовать риск, либо устранить его. В итеративном процессе разработки распределите такие действия по ранним итерациям, чтобы снизить риск как можно раньше. Противостоять рискам как можно раньше. если риск выражается в форме «X может не работать?», то запланируйте как можно скорее попробовать X.

- **Стратегии необходимые при наступлении риска (или непредвиденных ситуациях)**

В отношении каждого риска, вне зависимости от отсутствия или наличия плана его уменьшения, следует решить, какие действия нужно будет предпринять в случае реализации риска. Последовательность таких действий обычно называют резервным планом. Резервный план приводится в действие в том случае, если

избежать либо передать риск не удалось, смягчение риска было безуспешным, и теперь с ним следует работать напрямую.

Для этого в резервном плане необходимо учесть: индикаторы риска и действия для снижения эффекта.

Индикаторы

Появление некоторых рисков можно предугадать, отслеживая показатели процесса, тенденции и пороговые величины (пример: объем работ, которые необходимо переделать, остается слишком высоким); некоторые риски можно отслеживать, основываясь на требованиях и результатах тестирования проекта (например: время ответа системы на порядок превосходит требуемое). Существует множество других, менее четких индикаторов, которые не позволяют однозначно определить возникновение проблемы. Например, всегда существует риск ухудшения морального состояния в коллективе.

Действия по снижению эффекта риска

В простых случаях резервный план содержит альтернативные решения.

Воздействие риска обычно выражается в затратах финансов и времени, требуемых на то, чтобы аннулировать текущее решение и создать новое.

Реализация риска может быть вызвана тем, что попытка избежать или смягчить его была неудачной. В этом случае необходимо проверить список рисков, так как коллектив мог систематически упускать из виду какие-либо важные детали.

- **Проверка рисков в итерациях проекта**

Оценка рисков не просто должна проводиться через заданные интервалы времени - это непрерывный процесс. Следующие действия являются необходимыми:

- Еженедельно пересматривайте составленный список, отмечая изменения.
- Сделайте список десяти наиболее опасных рисков общедоступным и требуйте принятия соответствующих действий. Обычно список рисков следует добавлять в отчеты о состоянии проекта.

При завершении итерации сформулируйте цели следующей итерации с учетом списка рисков. В частности:

- Исключите из рассмотрения риски, которые были полностью смягчены.
- Добавьте информацию о рисках, обнаруженных в ходе итерации.
- Выполните заново оценку величины рисков и упорядочите их по этому показателю

ВОПРОС №40

Артефакт Business Case (Бизнес обоснование). Предназначение разделов.

Экономическое обоснование предоставляет необходимую информацию с точки зрения бизнеса, чтобы определить, стоит ли инвестировать в этот проект.

Для коммерческого программного продукта экономическое обоснование должно включать набор предположений о проекте и порядок величины возврата инвестиций (ROI), если эти предположения верны. Например, рентабельность инвестиций будет равна пяти, если завершится за один год, двум, если завершится за два года, и отрицательному числу после этого. Эти предположения снова проверяются в конце этапа проработки, когда объем и план определены с большей точностью.

Основная цель бизнес обоснования - разработать экономический план для реализации видения проекта, представленного в артефакте «Концепция» (Vision). После разработки экономическое обоснование используется для точной оценки рентабельности инвестиций (ROI), обеспечиваемой проектом. Оно обеспечивает обоснование проекта и устанавливает его экономические ограничения. Оно предоставляет лицам, принимающим экономические решения, информацию об экономической ценности проекта и используется для определения того, следует ли продвигать проект.

На критических этапах экономическое обоснование пересматривается, чтобы убедиться, что оценки ожидаемой прибыли и затрат по-прежнему точны, и следует ли продолжать проект.

Этот артефакт разрабатывается на начальном этапе, утверждается на этапах жизненного цикла и обновляется на разовой основе в результате некоторой оценки на последующих этапах.

За бизнес обоснование отвечает менеджер проекта.

Раздел «Введение»

Введение представляет собой обзор на весь документ в целом и включает в себя следующие разделы - назначение, область применения, определения и аббревиатуры, ссылки и обзор.

Раздел «Описание продукта»

Краткое описание разрабатываемого продукта, пояснения, какие задачи он решает и почему его стоит разрабатывать. Будет уместна ссылка на Концепцию проекта (Vision).

Раздел «Бизнес контекст»

Определение бизнес-контекста продукта - в какой сфере он будет применяться (банки, сети, малый бизнес и т.д.), на каком рынке продаваться, кто его потенциальные пользователи. Необходимо также указать, является ли продукт контрактной разработки или коммерческим решением.

Раздел «Цели продукта»

Указываются цели разработки продукта, описывается предварительный план их достижения и предварительная оценка рисков. Ясно выраженные цели являются хорошей основой для формирования вех разработки и создания задач, таким образом производится мониторинг прогресса продукта.

Раздел «Финансовый прогноз»

Данный раздел содержит в себе график окупаемости разработки продукта и его оценка. Например, если удалось осуществить возврат инвестиций за 1 год, то оценка окупаемости будет равна 5, за 2 года - 2, и отрицательной в иных случаях. Возврат средств определяется по оценке затрат и потенциальной прибыли. Следует пояснить затраты сметой за каждую фазу разработки продукта (и общей его стоимостью). Потенциальная прибыль рассчитывается как <(прибыль от реализации - ежемесячные затраты) * количество месяцев с момента внедрения системы>

Раздел «Ограничения»

Указываются ограничения, которые могут повлиять на стоимость и оценку рисков. Пример: необходимость интеграции с другими системами, следование стандартам, использование определенных технологий и т.д.

ИСТОЧНИКИ:

1. https://sceweb.uhcl.edu/helm/RationalUnifiedProcess/process/artifact/ar_bcase.htm

ВОПРОС №41

Модели жизненного цикла разработки программного продукта.

Модель жизненного цикла разработки ПП - это структура, определяющая последовательность выполнения и взаимосвязи процессов, действий, задач, выполняемых на протяжении жизненного цикла разработки ПП.

Модель жизненного цикла зависит от специфики и сложности выполняемого проекта, а также от условий, в которых создается и будет функционировать ПП.

Стандарт ISO/IEC 12207 - стандарт ISO, описывающий процессы жизненного цикла программного обеспечения. Он не предлагает конкретную модель.

Модель жизненного цикла любого конкретного ПП определяет характер процесса его создания, который представляет собой совокупность упорядоченных во времени, взаимосвязанных и объединенных в этапы работ, выполнение которых необходимо и достаточно для создания ПП, соответствующего заданным требованиям.

Под этапом разработки ПП понимается часть процесса создания ПП, ограниченная некоторыми временными рамками и заканчивающаяся выпуском конкретного продукта (*моделей ПП, программных компонентов, документации*), определяемого заданными для данной стадии требованиями.

Наиболее распространенные модели жизненного цикла разработки ПП:

- Waterfall model - каскадная модель “Водопад”;
- V-shaped model - V-образная модель;
- Prototype model - модель прототипирования;
- RAD model - модель быстрой разработки приложения;
- Incremental model - многопроходная модель;
- Spiral model - спиральная модель;

Название	Характеристика
Каскадная модель	Прямолинейная и простая в использовании. Необходим постоянный жесткий контроль за ходом работы. Разрабатываемое ПО не доступно для изменений.
V-образная модель	Простая в использовании. Особое значение придается тестированию и сравнению результатов фаз тестирования и проектирования.
Модель прототипирования	Создается “быстрая” частичная реализация системы до составления окончательных требований. Обеспечивается обратная связь между пользователями и разработчиками в процессе выполнения проекта. Используемые требования не полные.
Модель быстрой разработки приложений	Проектные группы небольшие (3 ... 7 человек) и составлены из высококвалифицированных специалистов.

	Уменьшенное время цикла разработки (<i>до 3 мес</i>) и улучшенная производительность. Повторное использование кода и автоматизация процесса разработки.
Многопроходная модель	Быстро создается работающая система. Уменьшается возможность внесения изменений в процессе разработки. Невозможен переход от текущей реализации к новой версии в течение построения текущей частичной реализации.
Сpirальная модель	Охватывает каскадную модель. Разделяет фазы на меньшие части. Позволяет гибко выполнять проектирование. Анализирует риски и управляет ими. Пользователи знакомятся с ПП на более раннем этапе благодаря прототипам.

Источники

А.В. Рудаков. Технология разработки программных продуктов. Стр 24-25.
http://lib.maupfib.kg/wp-content/uploads/1rudakov_a_v_tekhnologiya_razrabotki_programmnykh_produktov.pdf

ВОПРОС №42

Каскадная модель, особенности, область применения. Каскадная модель, особенности, область применения. (Рудаков 26, 27; Royce W. all)

Важной особенностью каскадного подхода является то, что переход на следующий этап осуществляется только после того, как работа над текущим этапом будет полностью завершена и возвратов на предыдущие этапы не подразумевается. Выходные данные этапа i , являются входными данными для этапа $i+1$. Требования к программному продукту строго документируются в ТЗ и фиксируются на весь этап разработки продукта. После выполнения каждого этапа выпускается документация достаточная для того, чтобы следующие этапы реализовывала уже другая команда.

Критерий качества - точность выполнения ТЗ.

Характеристики на которые обращают внимание разработчики - производительность, затрачиваемая память и тд.



Рис. 3.1. Каскадная модель

Преимущества:

- относительно прост
- на каждом этапе формируется законченный набор документации, отвечающий критериям полноты и согласованности
- выполняемые в логичной последовательности стадии позволяют планировать сроки завершения всех работ и соответствующие затраты.

Каскадный подход хорошо показал себя при написании систем, для которых в самом начале разработки можно достаточно точно сформулировать все требования, с целью предоставить разрабам свободу реализовать технически в лучшем ключе. Примеры таких систем: сложные системы с большим числом задач вычислительного характера, системы реального времени.

Но на практике реальная разработка не может полностью уложиться в эту схему. Чаще всего разработка программного продукта это итерационный процесс.

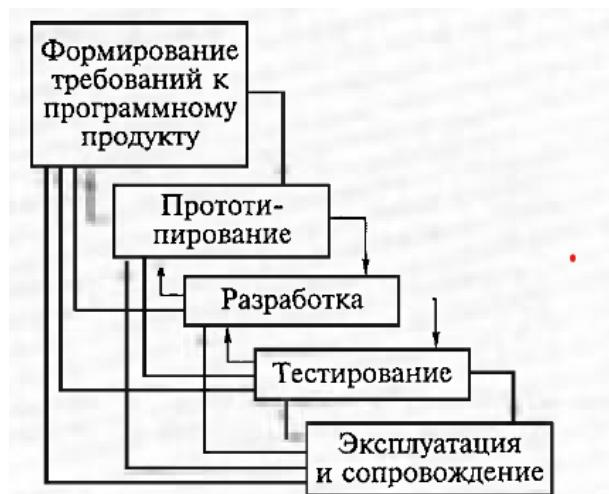


Рис. 3.2. Схема реального процесса разработки программного продукта

Схему представленную на рисунке выше называют еще моделью с промежуточным контролем.

Основными недостатками каскадного подхода являются существенное запаздывание с получение результатов и, как следствие, достаточно высоки риск создания системы, не удовлетворяющей изменившимся потребностям пользователей.

На практике, очень сложно на начальном этапе проекта полностью и точно сформулировать все требования к будущей системе.

Это объясняется тем, что пользователи не в состоянии сразу изложить все требования и не могут предвидеть их изменения в процессе разработки. Плюс за время разработки могут произойти изменения во внешней среде, которые также повлияют на требования к системе.

В рамках каскадного подхода ТЗ формируется один раз, а согласование резов с пользователями происходит после окончания этапов. Таким образом существенные замечания пользователи смогут внести только по завершению всей разработки.

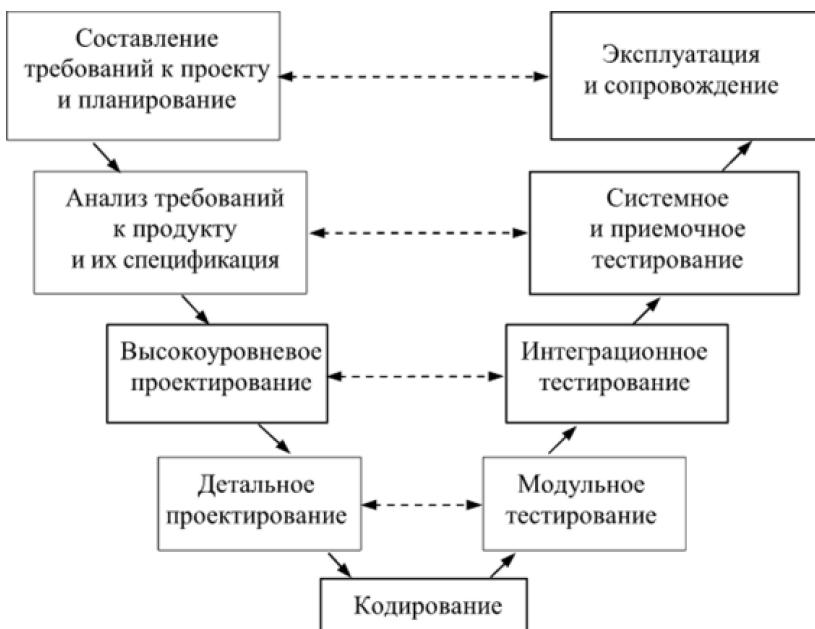
ВОПРОС №43

V-образная модель, особенности, область применения.

Особенности:

- простая в применении
- последовательная структура (следующая фаза начинается только после успешного завершения предыдущей)
- особое значение придается тестированию и сравнению результатов фаз тестирования и проектирования
- тестирование продукта обсуждается, проектируется и планируется, начиная с ранних этапов жизненного цикла разработки

Картинка



Фазы

Фаза	Описание
Составление требований к проекту и планирование	Определяются системные требования , выполняется планирование работ
Составление требований к продукту и их анализ	Составляется полная спецификация требования к программному продукту
Высокоуровневое проектирование	Определяется структура программного продукта, взаимосвязи между основными компонентами и реализуемые ими функции
Детальное проектирование	Определяется алгоритм работы каждого компонента
Кодирование	Пишем код. (Выполняется преобразование алгоритмов в готовое программное обеспечение)

Модульное тестирование	Выполняется проверка каждого компонента (в изоляции)
Интеграционное тестирование	Осуществляется интеграция программного продукта и его тестирование
Системное тестирование	Выполняется проверка функционирования программного продукта после помещения его в аппаратную среду в соответствии со спецификацией требований
Эксплуатация и сопровождение	Запуск программного продукта в производство . Возможны правки и модернизация

Преимущества

- ранее планирование, большое внимание тестированию
- тестирование не только программного продукта, но и полученных внутренних и внешних данных
- простой мониторинг хода разработки

Недостатки

- не учитываются итерации между фазами
- нельзя внести изменения на разных фазах [кажется модель бесполезна, прим. автора ответа]
- тестирование требований происходит поздно, увеличивая сроки разработки

Целесообразно использовать для: программных продуктов с высокими требованиями надежности.

ВОПРОС №44

Модель прототипирования, особенности, область применения.

Модель прототипирования (МП) позволяет создать прототип программного продукта (ПП) до или в течение этапа составления требований к ПП.

Жизненный цикл разработки ПП:

- разработка плана проекта;
- быстрый анализ;
- создание базы данных (по необходимости);
- создание пользовательского интерфейса;
- разработка необходимых функций.

Результат – документ, содержащий частичную спецификацию требований к ПП и являющийся основой для итерационного цикла быстрого прототипирования.

В результате прототипирования:

- демонстрация разработчиком готового прототипа и оценка пользователем его функционирования;
- определение проблем и их устранение, пока ПП не будет соответствовать требованиям пользователей;
- демонстрация прототипа пользователям для получения предложений по его совершенствованию, пока рабочая модель не окажется удовлетворительной;
- утверждение прототипа пользователем и выполнение окончательного его преобразования в готовый ПП.

Преимущества МП:

- вовлечение заказчика в разработку системы на раннем этапе;
- сведение к минимуму неточностей в требованиях;
- снижение искажения информации, создание более качественного ПП;
- возможность учета новых, неожиданных требований заказчика;
- гибкое выполнение проектирования и разработки, включая несколько итераций на всех фазах жизненного цикла разработки;
- сведение противоречий между заказчиками и разработчиками к минимуму;
- уменьшение числа доработок, и соответственно, снижение стоимости разработки;
- решение проблем на ранних стадиях, и соответственно, сокращение расходов на их устранение;
- участие заказчика на протяжении всего жизненного цикла разработки, и в итоге удовлетворенность результатами.

Недостатки МП:

- решение сложных задач может отодвинуться на будущее;
- заказчик может предпочесть прототип, а не полную версию ПП;
- прототипирование может неоправданно затянуться;
- неизвестно, сколько итераций придется выполнить.

МП применяется, когда:

- требования к ПП заранее неизвестны, не постоянны или неудачно сформулированы;
- требования необходимо уточнить;
- нужна проверка концепции;
- потребность в пользовательском интерфейсе;
- новая, не имеющая аналогов разработка;
- разработчики не уверены в том, какое решение следует выбрать.

ВОПРОС №45

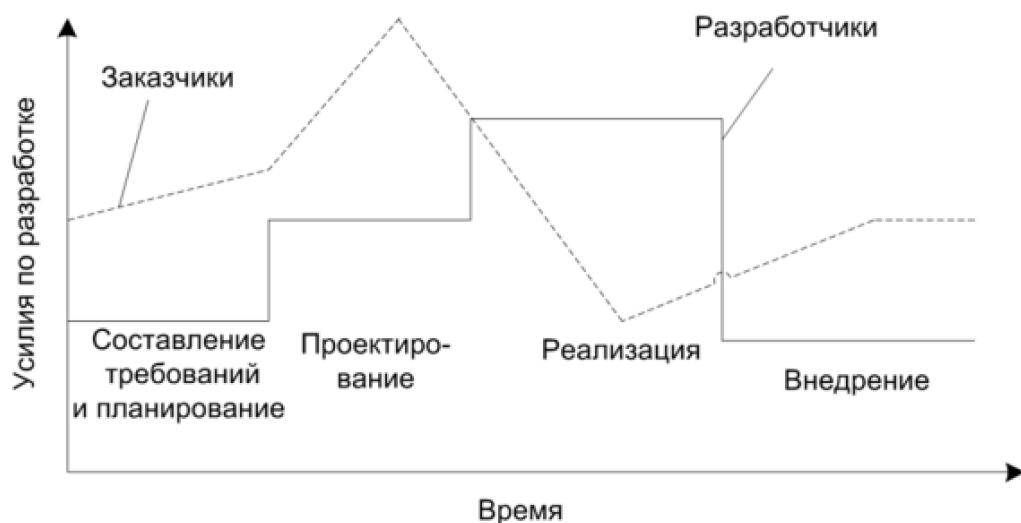
Вопрос: RAD-модель, особенности, область применения

RAD - rapid application development (быстрая разработка приложений).

Особенности:

- решающую роль играет конечные пользователи
- конечный пользователь участвует в формировании требований
- конечный пользователь взаимодействует с разработчиками
- конечный пользователь участвует в апробации прототипов
- благодаря активному участию пользователя на раннем этапе конечная система формируется быстрее
- большую часть работы составляют планирование и проектирование
- применяется автоматизация программирования и повторное использование компонентов существующих продуктов
- концепцию RAD часто связывают с концепцией визуального программирования

Картинка



Фазы

Фаза	Описание
Составление требований и планирование	Применяется метод совместного планирования требований -- выполняется структурный анализ, обсуждаются решаемые задачи, формируются требования и планируются работы
Проектирование (описание пользователя)	Проектирование программного продукта при непосредственном участии заказчика
Реализация	детальное проектирование, кодирование, тестирование, поставка заказчику
Внедрение (+ сопровождение)	Приемочные испытания, установка программного продукта, обучение пользователей

Достоинства:

- сокращение времени разработки
- минимизация риска недовольства заказчика конечным программным продуктом
- повторное использование компонентов существующих программ

Недостатки:

- негативное влияние на результат низкой вовлеченности заказчика
- требуются высококвалифицированные кадры
- риск заикливания работ

Целесообразно применять:

- программные продукты невысокой сложности, при высокой вовлеченности заказчика
- программные продукты, которые хорошо поддаются моделированию (хорошо известны требования), при высокой вовлеченности заказчика
- в составе другой модели для ускорения разработки прототипов

ВОПРОС №46

Многопроходная модель, особенности, область применения.

Многопроходная модель(ММ) - это несколько итераций построения прототипа ПП с добавление каждой следующей итерации новых функциональных возможностей или повышением эффективности ПП.

Предполагается, то на ранних этапах жизненного цикла разработки (планирование, анализ требований и разработка проекта) выполняется конструирование ПП в целом. Тогда же определяется и число необходимых инкрементов и относящихся к ним функций. Каждый инкремент затем проходит через оставшиеся фазы жизненного цикла (кодирование и тестирование). Сначала выполняются конструирование, тестирование и реализация базовых функций, составляющих основу ПП. Последующие итерации направлены на улучшение функциональных возможностей ПП.

Преимущества ММ:

в начале разработки требуется ср-ва только для разработки и реализации основных функций ПП;

после каждого инкремента получается функциональный продукт;

снижается риск неудачи и изменения требований;

улучшается понимание как разработчиками, так и пользователями ПП требований для более поздник итераций;

инкременты функциональных возможностей легко поддаются тестированию.

Недостатки ММ:

не предусмотрены итерации внутри каждого инкремента;

определение полной функциональности должно быть осуществлено в самом начале жизненного цикла разработки;

может возникнуть тенденция оттягивания решения трудных задач;

общие затраты на создание ПП не будут снижены по сравнению с другими моделями;

обязательным условием является наличие хорошего планирования и проектирования;

ММ может быть применена, если большинство требований к ПП буду сформированы заране, а для выполнения проекта будет выделен большой период времени.

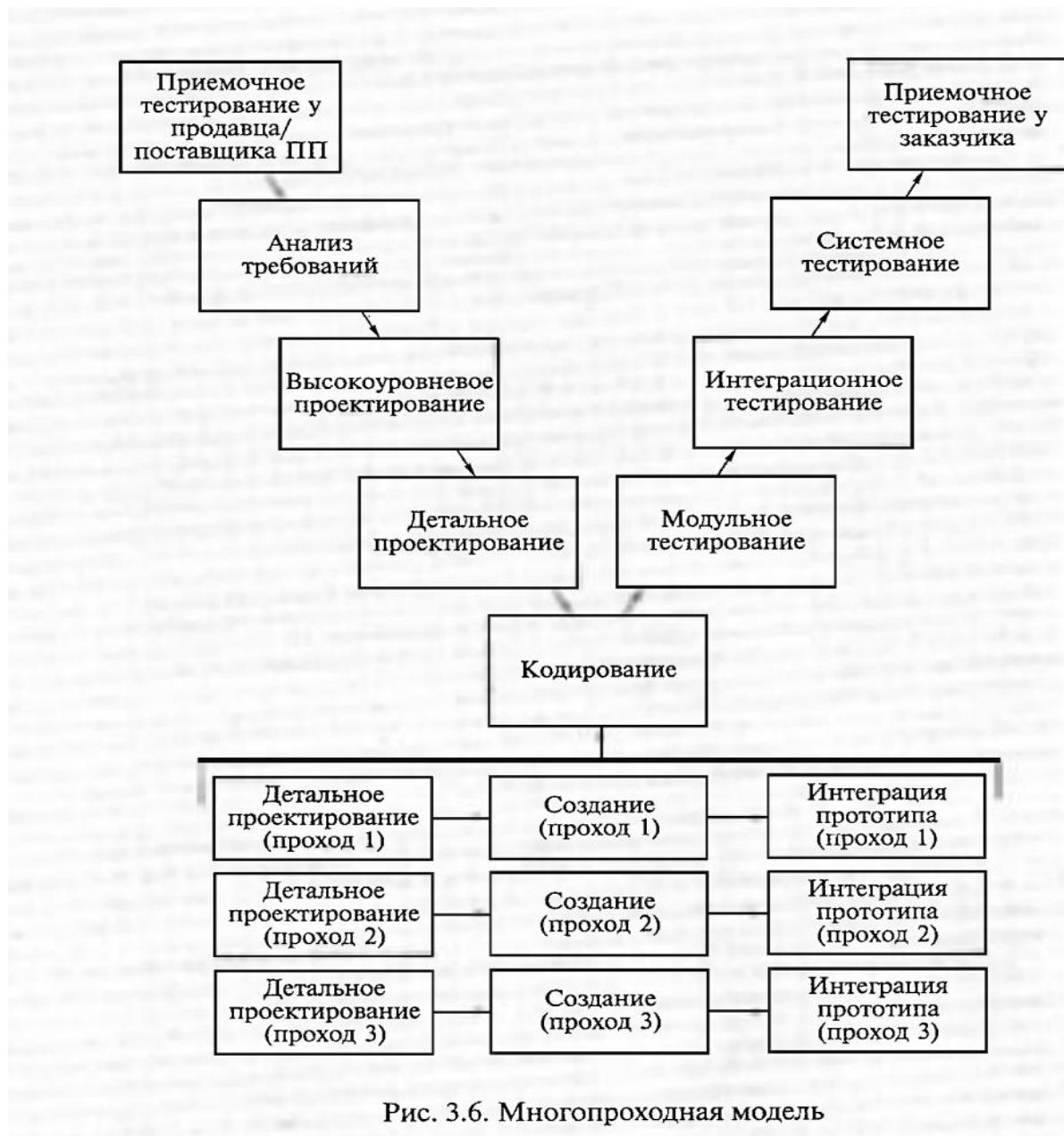


Рис. 3.6. Многопроходная модель

ВОПРОС №47

Сpirальная модель, особенности, область применения

Источник

<https://qalight.ua/ru/baza-znaniy/spiralnaya-model-spiral-model/>

Сpirальная модель

В спиральной модели жизненный путь разрабатываемого продукта изображается в виде спирали, которая, начавшись на этапе планирования, раскручивается с прохождением каждого следующего шага. Таким образом, на выходе из очередного витка мы должны получить готовый протестированный прототип, который дополняет существующий билд. Прототип, удовлетворяющий всем требованиям – готов к релизу.

Особенности

Главная особенность спиральной модели – концентрация на возможных рисках. Для их оценки даже выделена соответствующая стадия. Основные типы рисков, которые могут возникнуть в процессе разработки ПО:

- Нереалистичный бюджет и сроки;
- Дефицит специалистов;
- Частые изменения требований;
- Чрезмерная оптимизация;
- Низкая производительность системы;
- Несоответствие уровня квалификации специалистов разных отделов.

Плюсы и минусы спиральной модели:

- + улучшенный анализ рисков;
- + хорошая документация процесса разработки;
- + гибкость – возможность внесения изменений и добавления новой функциональности даже на относительно поздних этапах;
- + раннее создание рабочих прототипов.
- может быть достаточно дорогой в использовании;
- управление рисками требует привлечения высококлассных специалистов;
- успех процесса в большой степени зависит от стадии анализа рисков;
- не подходит для небольших проектов.

Область применения

Когда использовать спиральную модель:

- когда важен анализ рисков и затрат;
- крупные долгосрочные проекты с отсутствием четких требований или вероятностью их динамического изменения;
- при разработке новой линейки продуктов.

ВОПРОС №48

EUP. Фазы и дисциплины.

Изменения в дисциплинах RUP.

Единый корпоративный процесс – Enterprise Unified Process Единый корпоративный процесс.

Unified Process Enterprise (ОУП) представляет собой расширенный вариант унифицированного процесса и была разработана Скотт У. Амблер и Ларри Константина в 2000 году, в конце концов, переработан в 2005 году Амблер, Джон Налбон и Майкл Vizdos. Изначально EUP был введен для преодоления некоторых недостатков RUP, а именно отсутствия производства и возможного вывода из эксплуатации программной системы. Так были добавлены две фазы и несколько новых дисциплин. EUP рассматривает разработку программного обеспечения не как отдельную деятельность, а как часть жизненного цикла системы (которая должна быть построена, расширена или заменена), жизненного цикла ИТ предприятия и жизненного цикла организации / бизнеса самого предприятия. Он занимается разработкой программного обеспечения с точки зрения клиента. В 2013 году началась работа по развитию EUP, основанной на дисциплинированной гибкой доставке вместо унифицированного процесса.

Фазы

Единый процесс определяет четыре фазы проекта:

- Зарождение
- Проработка
- Строительство
- Переход

К этим EUP добавляет две дополнительные фазы:

- Производство
- Отставка

Дисциплины

Rational Unified Process определяет девять проектных дисциплин:

- Бизнес-моделирование
- Требования
- Анализ и дизайн
- Выполнение
- Тест
- Разворачивание
- Конфигурация и управление изменениями
- Управление проектом
- Среда

К ним EUP добавляет еще одну дисциплину проекта:

- Операции и поддержка и семь корпоративных дисциплин
- Бизнес-моделирование предприятия
- Управление портфелем
- Архитектура предприятия
- Стратегическое повторное использование
- Управление персоналом
- Администрация предприятия

- Улучшение программного процесса.

Лучшие практики EUP

EUP предоставляет следующие передовые практики:

1. Развивайте итеративно
2. Управляйте требованиями
3. Проверенная архитектура
4. Моделирование
5. Постоянно проверяйте качество
6. Управляйте изменениями
7. Совместная разработка
8. Не ограничивайтесь развитием
9. Регулярная поставка работающего программного обеспечения
10. Управляйте риском.

RUP – рациональный унифицированный процесс разработки программного обеспечения.

Рациональный унифицированный процесс разработки программного обеспечения (RUP – Rational Unified Process) является частным случаем унифицированного процесса (UP – Unified Process). В основу рационального унифицированного процесса положена итеративная разработка программного обеспечения. В рамках RUP разработка выполняется в виде нескольких краткосрочных итераций продолжительностью от 2 до 6 недель. Итерация по существу является мини-проектом фиксированной длительности, в результате которой расширяется и дополняется функциональность разрабатываемой системы. Поэтому унифицированный процесс разработки иногда называют итеративной и инкрементальной разработкой.

В результате каждой итерации получается работающая, но не полнофункциональная система, которая еще не является коммерческой и не подлежит распространению. Продолжительность создания коммерческой версии программной системы составляет 10 – 15 итераций.

Но результат каждой итерации нельзя рассматривать и в виде прототипа системы. Правильнее сказать, что в результате каждой итерации создается окончательная версия некоторой части всех системы.

Унифицированный процесс допускает внесение изменений требований пользователей к создаваемой программной системе. Таким образом, он является адаптивным процессом. Это достигается за счет итеративному процессу разработки и наличию ранней обратной связи. Благодаря обратной связи заказчик может оценить часть системы и высказать некоторые предложения по внесению изменений в ее функциональность. Здесь речь не идет о том, что функциональность совершенно не устраивает заказчика или пользователей, просто могут возникнуть идеи об ее улучшении или же возникнет новая ситуация, под которую необходимо адаптировать создаваемую систему. Таким образом, реализуется эволюционный процесс, в результате которого разрабатываемая система постоянно улучшается и все больше удовлетворяет требованиям пользователей.

Фазы рационального унифицированного процесса (RUP)

Название фазы	Содержание фазы
Начало (inception)	Определение начального видения проблемы, прецедентов, а также оценка сложности проекта.
Развитие (elaboration)	Формирование более полного видения проблемы, итеративная реализация базовой архитектуры системы, создание наиболее критичных компонентов (разрешение высоких рисков), определение основных требований и оформление их в виде системы прецедентов, получение более реалистичных оценок сложности проекта и сроков.
Конструирование (construction)	Итеративная реализация менее критичных и более простых элементов, подготовка к развертыванию системы.
Передача (transition)	Бета-тестирование и развертывание системы.

Основными свойствами унифицированного процесса являются:

- итеративная разработка;
- допустимость внесения изменений;
- адаптивность;
- оценка рисков;
- построение базовой архитектуры на ранних итерациях;
- разработка базируется на требованиях пользователей, заданных прецедентами;
- постоянная обратная связь и учет пожеланий пользователей;
- ориентирован на объектно-ориентированные технологии программирования;
- используется UML;
- постоянный контроль качества, раннее тестирование.

Рациональный унифицированный процесс является наиболее предпочтительным при создании систем автоматизированного проектирования (САПР), CASE-систем, систем с элементами искусственного интеллекта, а также, систем поддержки структурно-параметрического синтеза объектов.

ВОПРОС №49

EUP. Дисциплина продуктивного использования и поддержки

Основная задача дисциплины - организация поддержки и обслуживания системы после её запуска (на проде).

Планирование поддержки основано на следующих моментах:

1. Способы оказания поддержки: то, как клиент будет обращаться в тех.поддержку: по телефону, е-мейлу или факсу. А также как за неё будет происходить оплата: отдельно или в составе тарифа.
2. Контактные данные ответственного лица: необходимо четко указывать информацию о работе ответственного лица и по каким вопросам следует к нему обращаться.
3. Уведомление об ошибках и обработка запросов об улучшении: сотрудники поддержки первые, кто получают о таких запросах звонки и должны передавать такие запросы в специальную доску по управлению изменениями (Enterprise Change Control Board - ECCB)
4. SLA - Service-Level Agreement: соглашение нацелено на описание того, что поддержка занимается только вопросами по поддержке данного продукта. К примеру, погодой на марсе она не занимается. Там также указывается время на устранение проблем, их приоритезация и среднее время отклика на те или иные запросы.
5. Приоритезация ошибок и управление временем на их устранение: здесь рассчитываются те средства и затраты, которые требуются на устранение каждого кейса клиента.
6. Критерий эскалации неполадки: описывает то, в каких случаях и как указанное замечание передается выше исходя из тяжести ошибки.
7. Доставка исправленных версий клиентам: указание о том, как часто (по расписанию или по запросу), каким образом и в каком порядке клиентам рассылаются новые версии с исправлениями.

Поддержка пользователей - 2 стратегии:

1. “Персональный менеджер” - клиент общается по всем вопросам с одним сотрудником и тот ведёт до конца его запрос и сам общается с профильными сотрудниками. Эта стратегия приятна клиентам, но более затратна - нужен специалист, который немного разбирается во всех вопросах и его замена может быть трудна.
2. Многоуровневая поддержка - большинство запросов просты и могут быть обработаны быстро. Лишь небольшая часть передается на уровень выше, где обрабатывается профильными специалистами.

Поддержка работающих систем

Разделяют две основные роли:

1. Оператор - обслуживает систему, хранит и поднимает бэкапы, подкручивает параметры работы системы и проводит перенастройки по необходимости.
2. Разработчик поддержки - обычно член команды разработки, который занимается накатыванием хот-фиксовых на работающую систему.

Стратегия подготовки к катастрофе: описание шагов по непрерывному сохранению системы на случай внешних проблем и прекращения работы системы.

Стратегия восстановления после катастрофы: план по четкому и быстрому восстановлению системы после остановки ее работы. Обычно этим занимается оператор поддержки и он оказывает помощь с восстановлением доступов к системе других команд.

Использованные источники:

EUP. Enterprise Agile: Operations and Support

<http://www.enterpriseunifiedprocess.com/essays/operationsAndSupport.html>

ВОПРОС №50

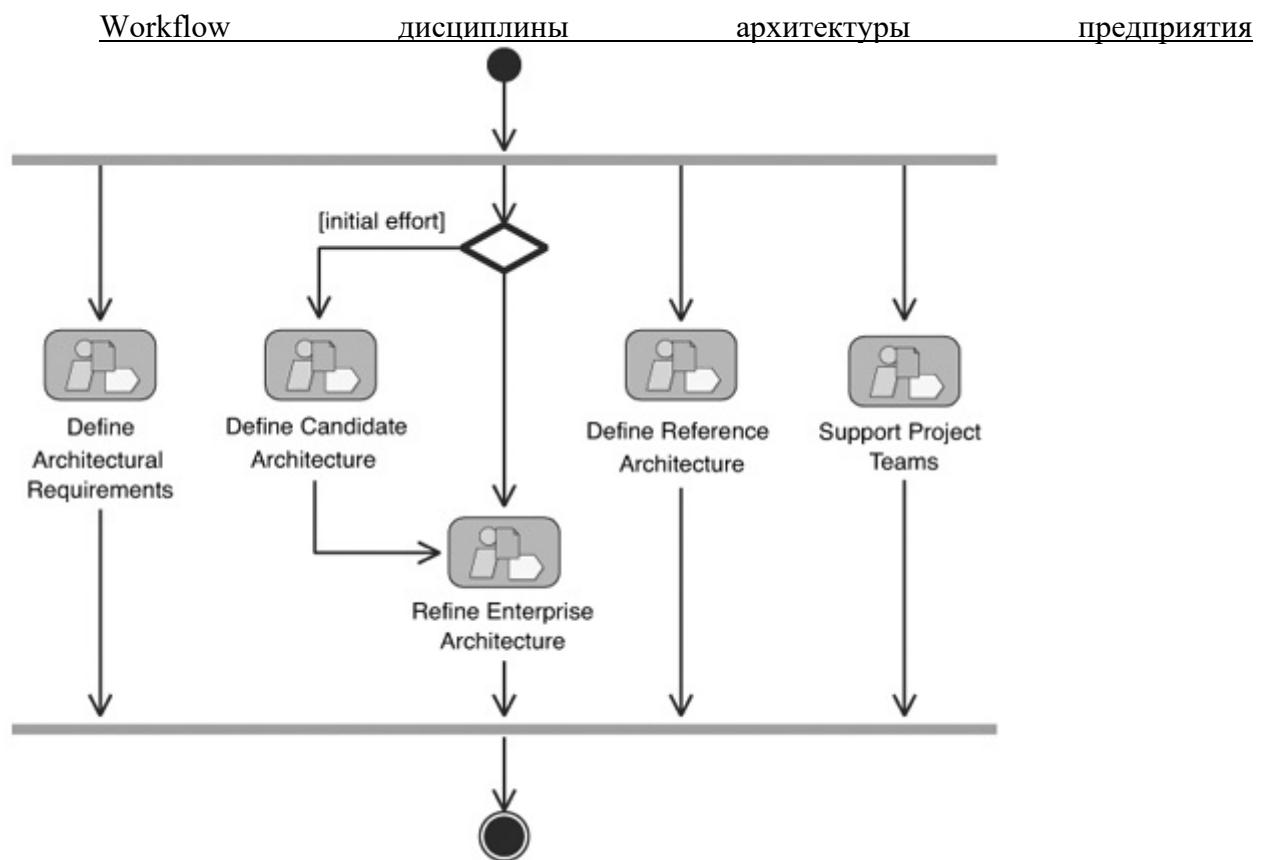
EUP. Дисциплины архитектуры предприятия, управления людьми

Дисциплина архитектуры предприятия (The Enterprise Architecture Discipline)

Дисциплина архитектуры предприятия описывает процесс создания корпоративной архитектуры, фреймворков, сетей, конфигураций развертывания, линейки продуктов домена и вспомогательной инфраструктуры, которые образуют техническую архитектурную инфраструктуру для выбранной организации. Корпоративные архитекторы также предоставляют эталонные архитектуры, технические требования, стандарты и руководящие принципы.

Архитекторы предприятия несут ответственность за разработку, проверку и поддержку архитектуры предприятия, определяют архитектурные решения, структуры, шаблоны и эталонные архитектуры для использования в нескольких системах внутри организаций.

Администраторы предприятия несут ответственность за данные, безопасность, сетевую и аппаратную инфраструктуры. Архитектура предприятия также будет создана командами разработчиков приложений.



Define Architectural Requirements

Необходимо определить технические (в т.ч. требования к производительности или доступности) и бизнес-требования (фундаментальные бизнес-аспекты, включая бизнес-процессы, бизнес-правила и сущности предметной области). Также необходимо указать предстоящие проекты, которые определены в настоящее время, а также ожидаемое время их старта.

Define Candidate Architecture

Архитектура-кандидат – это предлагаемое дополнение или изменение архитектуры предприятия, которое не было полностью детализировано или реализовано. Следующий шаг - показать, что подходящая архитектура действительно работает и вписывается в существующую корпоративную архитектуру.

Refine Enterprise Architecture

Имея на руках подходящую архитектуру-кандидат, архитекторы предприятия интегрируют ее в модель архитектуры выбранного предприятия. Модель архитектуры вашего предприятия должна отражать принятые руководящие принципы корпоративного администрирования (стандарты и рекомендации по безопасности и данным), руководство по моделированию и развитию предприятия.

Define Reference Architecture

Эталонная архитектура должна быть задокументирована таким образом, чтобы разработчикам приложений было легко использовать ее. Один из способов сделать это - создать стандартный документ архитектуры программного обеспечения RUP (SAD).

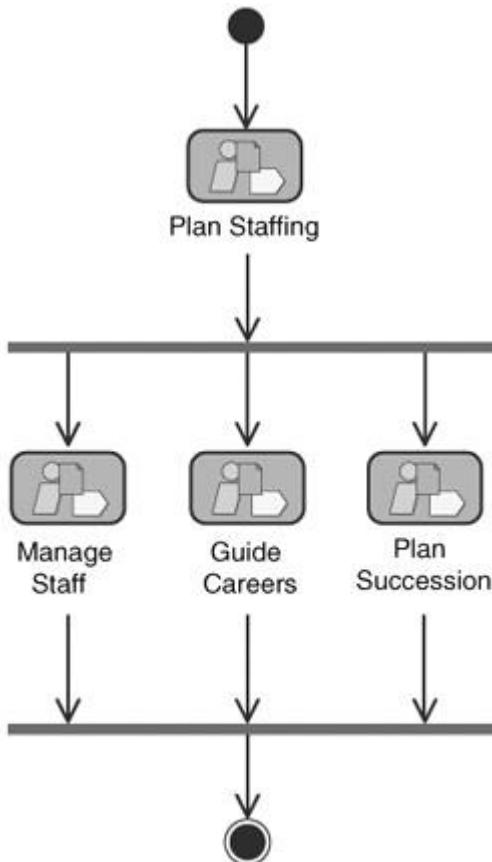
Support the Project Teams

Необходимо поддерживать проектные группы. Нет смысла определять отличную корпоративную архитектуру, если команды разработчиков не могут ее понять или не придерживаются ее.

Дисциплина управления людьми (The People Management Discipline)

Цели дисциплины управления людьми - управлять и повышать эффективность сотрудников в организации информационных технологий (ИТ) для успешного завершения проектов. Необходимо управлять сотрудниками, помогать им расти и поддерживать их взаимодействие с другими. Эта дисциплина описывает процесс организации, мониторинга, обучения и мотивации людей, чтобы они хорошо работали вместе и успешно вносили свой вклад в проекты организации.

Workflow дисциплины управления людьми



Plan Staffing

При планировании кадрового обеспечения ИТ-отдела важно оценить потребности бизнеса с точки зрения предприятия.

Manage Staff

Отдел кадров организации должен помогать в управлении персоналом, как и любой другой отдел в вашей организации. Для эффективного найма ИТ-специалистов необходимо потратить достаточное количество времени на планирование своих собеседований, чтобы научиться различать людей, которые говорят великие речи, и людей, которые могут сделать отличную работу.

Guide Careers

ИТ-отдел и отдел кадров должны работать вместе, чтобы гарантировать, что люди в различных командах будут руководить своей карьерой. Чтобы эффективно спланировать обучение и наставничество ИТ-персонала, необходимо разговаривать с ИТ-специалистами, чтобы узнавать, чем они хотят заниматься.

Succession Planning

Цель планирования преемственности - убедиться, что ваша организация готова к тому, что будущие поколения вашего ИТ-персонала возьмут на себя его работу, когда нынешний персонал больше не будет работать в вашей организации.

Timing

В начале каждого нового проекта выясняйте, каковы цели людей, и помогайте им в их достижении, получая необходимое обучение, образование и наставничество. Ближе к

концу проекта выясняйте, хотят ли они заниматься чем-то еще в компании, и начните помогать им в переходе в новую команду. Открытое и честное общение может дать отдачу, превосходящую любые заранее установленные ожидания.

Источники:

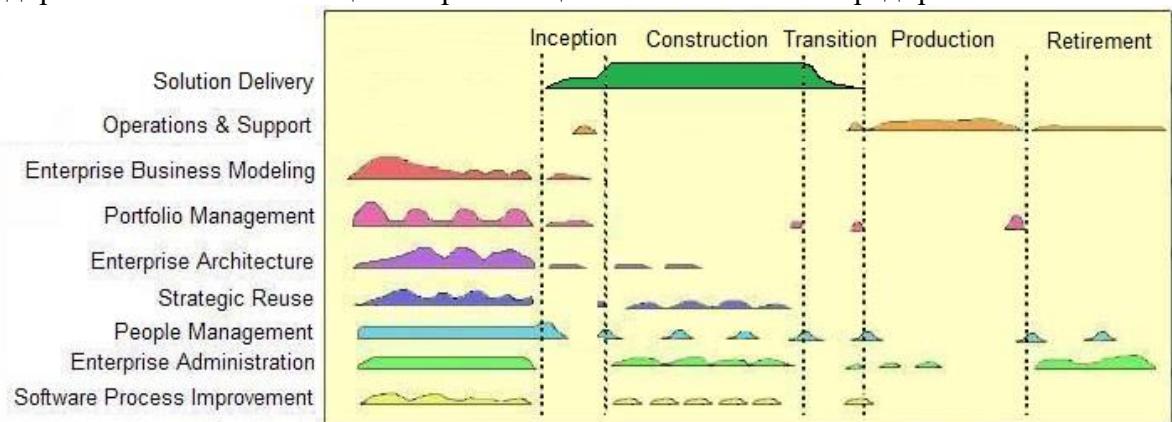
1. Книжка “The Enterprise Unified Process: Extending the Rational Unified Process“, Scott W. Ambler, John Nalbone, Michael Vizdos, глава 9, страницы 142-161 (дисциплина архитектуры предприятия) + глава 11, страницы 186-200 (дисциплина управления людьми). Доступна для скачивания по ссылке: [https://groups.google.com/group/master-of-computer-iauctb/attach/1631d7216d17a85f/The Enterprise Unified Process.pdf?part=0.1](https://groups.google.com/group/master-of-computer-iauctb/attach/1631d7216d17a85f/The%20Enterprise%20Unified%20Process.pdf?part=0.1)

ВОПРОС №51

EUP. Дисциплины управления портфолио, стратегического повторного использования.

Enterprise Unified Process (EUP)

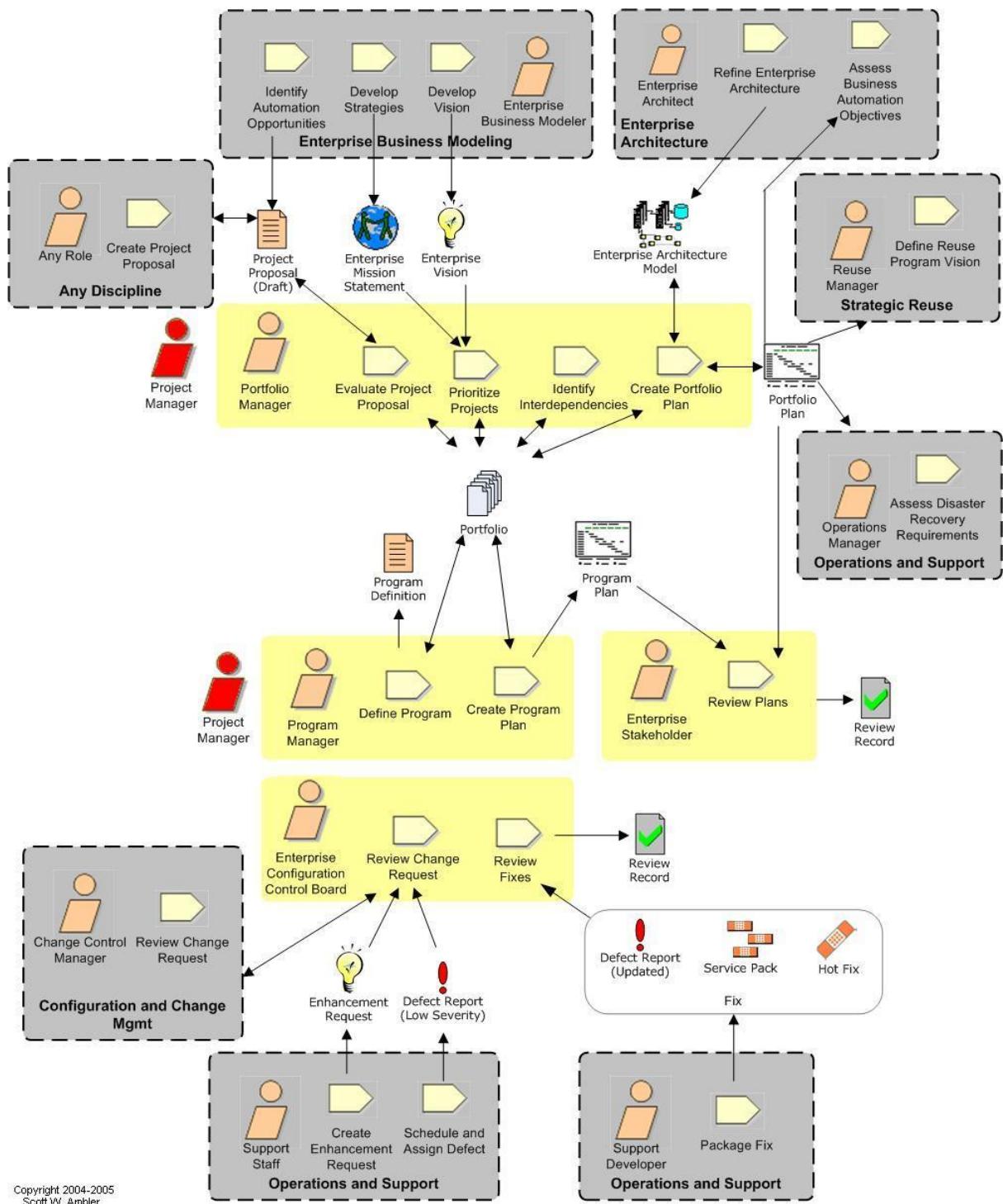
Enterprise Unified Process (EUP) - это расширенный вариант Unified Process.. Первоначально EUP был введен для преодоления некоторых недостатков RUP, а именно отсутствия производства и возможного вывода из эксплуатации программной системы. Так были добавлены две фазы и несколько новых дисциплин. EUP рассматривает разработку программного обеспечения не как отдельную деятельность, а как часть жизненного цикла системы (которая должна быть построена, расширена или заменена), жизненного цикла ИТ предприятия и жизненного цикла организации / бизнеса самого предприятия.



Дисциплины управления портфолио

Разработку плана портфолио ведет менеджер портфолио. Мониторинг существующих проектов является важным аспектом управления портфолио/программами. Он может потребовать гибкости со стороны менеджера портфолио, поскольку проектные группы могут следовать различным методологиям разработки программного обеспечения. Отдельные команды могут следовать DAD, XP, FDD или многим другим методам. Взаимозависимости между проектами должны быть определены, поняты и приняты в рамках портфолио, чтобы обеспечить поставку систем, отвечающих общим бизнес-целям организации. Зависимости могут включать в себя такие результаты, как общая архитектура (которая может возникнуть при объединении плана), важные даты выпуска, прибытие нового оборудования инфраструктуры, планы обновления базы данных или других систем или множество других вопросов, связанных с проектом. Также можно выявить перекрывающиеся или избыточные зависимости ресурсов и навыков. Еще одно преимущество заключается в том, что можно синхронизировать деятельность между проектами и даже программами.

Когда менеджер портфолио видит, что два проекта преследуют одну и ту же цель, то необходимо либо исключить один из проектов, либо объединить эти проекты. Такого рода проблемы не должны возникать, если управление портфолио эффективно; поэтому необходимо определить, как произошла эта ситуация и почему. Кроме того, если функциональность программы зависит от какого-либо внешнего проекта, и этот проект сталкивается с проблемами, основная программа должна быть соответствующим образом скорректирована.



Стратегия повторного использования

Одним из важнейших проектных факторов, который влияет как на архитектурные решения, так и на вопросы детализированного проектирования, является повторное использование элементов программы. Стратегия проектирования с повторным использованием предполагает максимальное применение уже имеющихся удачных программных или инструментальных решений, полученных при успешных разработках программных систем.

Реализация повторного использования зависит не только от применения объектно-ориентированной технологии и корректного описания классов, но и от многих технических, организационных и социальных аспектов разработки программных систем. Предпосылкой ускорения процесса перехода на компоненты многократного использования является

применение языка моделирования UML, обладающего свойствами расширяемости и хорошо определенной семантикой. В UML повторное использование (reuse) определяется как «использование уже существующего артефакта (artifact)». Среди образцов для повторного использования наиболее распространенными являются следующие категории: шаблоны анализа, архитектурные шаблоны, шаблоны проектирования, шаблоны кодирования.

ВОПРОС №52

Гибкие методологии разработки. Agile Manifesto. Принципы Agile

Источник

Вольфсон Б. Гибкие методологии разработки., стр. 5-6

Гибкие методологии разработки, Agile Manifesto

Гибкая методология разработки (англ. agile software development, agile-разработка) — обобщающий термин для целого ряда подходов и практик, основанных на ценностях Манифеста гибкой разработки программного обеспечения и 12 принципах, лежащих в его основе.

В более строгом варианте эти тезисы были сформулированы отцами-основателями гибких методологий в документе, который получил название Agile Manifesto:

- Люди и их взаимодействие важнее процессов и инструментов;
- Готовый продукт важнее документации по нему;
- Сотрудничество с заказчиком важнее жестких контрактных ограничений;
- Реакция на изменения важнее следования плану.

Принципы Agile

1. Наш высший приоритет – это удовлетворение заказчика с помощью частых и непрерывных поставок продукта, ценного для него.
2. Мы принимаем изменения в требования, даже на поздних этапах реализации проекта.
3. Гибкие процессы приветствуют изменения, что является конкурентным преимуществом для заказчика.
4. Поставлять полностью рабочее программное обеспечение каждые несколько недель, в крайнем случае, каждые несколько месяцев. Чем чаще, тем лучше.
5. Представители бизнеса и команда разработки должны работать вместе над проектом.
6. Успешные проекты строятся мотивированными людьми. Дайте им подходящую окружающую среду, снабдите всем необходимым и доверьте сделать свою работу.
7. Самый эффективный метод взаимодействия и обмена информацией – это личная беседа.
8. Рабочее программное обеспечение – главная мера прогресса проекта
9. Гибкие процессы способствуют непрерывному развитию. Все участники проекта должны уметь выдерживать такой постоянный темп.
10. Постоянное внимание к техническому совершенству и качественной архитектуре способствуют гибкости.
11. Простота необходима, как искусство максимизации работы, которую не следует делать.
12. Лучшая архитектура, требования, дизайн создается в самоорганизующихся командах.

ВОПРОС №53

Scrum. Основы процесса. Роли. Артефакты. (Вольфсон, стр. 8, 9, 16, 17)

Scrum - методология управления проектами.

Согласно Scrum-у, работа в организации ведётся в небольших кроссфункциональных командах, которые содержат всех необходимых специалистов. В каждой команде есть скрам-мастер.

Требования разбиваются на небольшие, ориентированные на пользователей, функциональные части, которые максимально независимы друг от друга - это беклог продукта. Затем элементы беклога упорядочиваются по их важности и производится относительная оценка объемов каждой истории. Дополнительно есть владелец продукта.

Вся работа ведётся короткими (от 1 до 4 недель) фиксированными итерациями – спринтами. В конце каждого спрингта поставляется инкремент продукта. Команда согласно своей скорости набирает задачи на спрингт. Каждый день проводится скрам-митинг, на котором команда синхронизирует свою работу и обсуждает проблемы. В процессе работы члены команды берут в работу элементы беклога спрингта согласно приоритету.

В конце каждого спрингта проводится обзор спрингта с целью получения обратной связи от владельца продукта и ретроспектива спрингта для оптимизации процессов Scrum-а. После этого владелец продукта может изменить требования и их приоритеты и запустить новый спрингт.

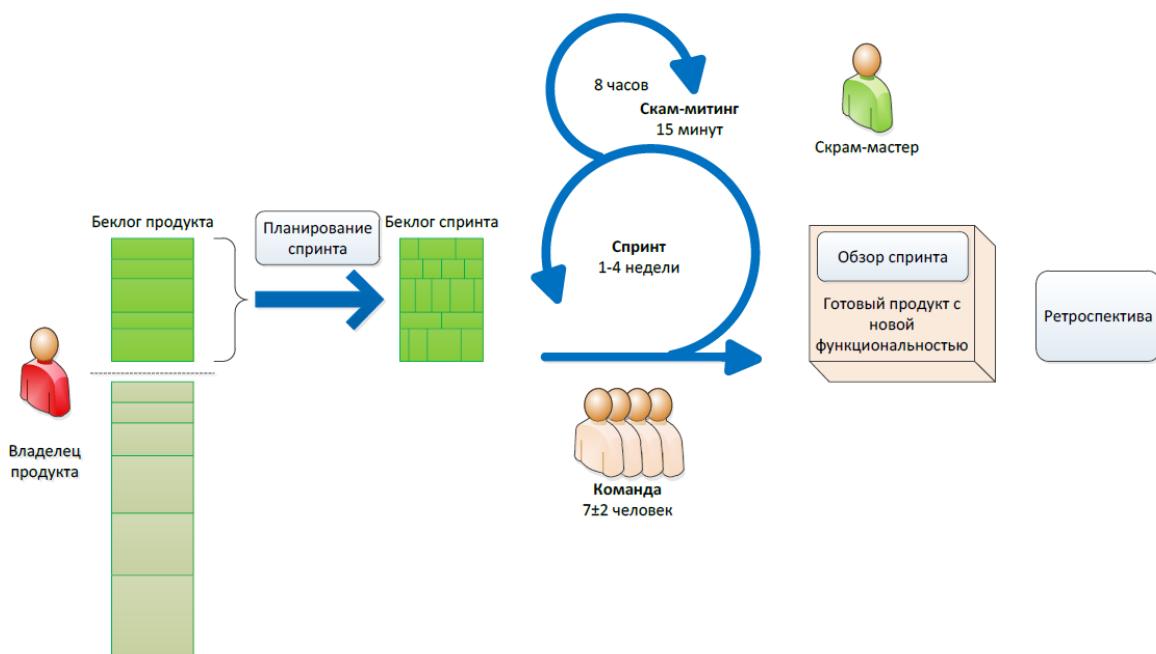


Рисунок 2. Общая схема Scrum

Роли:

1. Владелец продукта (Product owner) - человек, ответственный за приоритезацию требований и (часто) за их создание;
2. Скрам-мастер - член команды, который дополнительно отвечает за процессы (скрам-митинги, планирование спрингта, запуск спрингта, обзор спрингта, ретроспективу), координацию работы команды и поддержание социальной атмосферы в команде;
3. Команда - 7±2 человек, которые реализуют требования владельца продукта.

Артефакты:

1. Беклог продукта (Product Backlog) – приоритезированный список требований с оценкой трудозатрат. Обычно он состоит из бизнес-требований, которые приносят конкретную бизнес-ценность и называются элементы беклога;
2. Беклог спринта (Sprint Backlog) – часть беклога продукта, с самой высокой важностью и суммарной оценкой, не превышающей скорость команды, отобранная для спринта;
3. Инкремент продукта - новая функциональность продукта (которую можно при необходимости вывести на рынок), созданная во время спринта.

Источники:

1. Книга “Гибкие методологии разработки”, Б.Вольфсон, страницы 8-9, 16-17. Доступна для скачивания по ссылке: <https://www.itru.ru/wp-content/uploads/2019/04/%D0%91%D0%BE%D1%80%D0%B8%D1%81%D0%92%D0%BE%D0%BB%D1%8C%D1%84%D1%81%D0%BE%D0%BD%D0%93%D0%B8%D0%B1%D0%BA%D0%B8%D0%B5%D0%BC%D0%B5%D1%82%D0%BE%D0%B4%D0%BE%D0%BB%D0%BE%D0%B3%D0%B8%D0%B8%D0%B8.pdf>

ВОПРОС №54

Scrum. Процессы. Ретроспектива

Согласно словам Бориса Вольсона Scrum является самой популярной гибкой методологией разработки ПО:

- Scrum используют не только для разработки ПО, он отлично подходит для многих процессов по созданию продукта: от венчурных до маркетинговых продуктов.
- Scrum вовсе не методология, это гибкий управленческий фреймворк.
- Scrum обычно дополняют инженерными практиками из других гибких методологий (например, практики разработки из экстремального программирования, или практики анализа и сбора требований из ICONIX).

Классический Scrum состоит из следующих элементов:

Роли	Артефакты	Процессы
<ul style="list-style-type: none"> • Владелец продукта • Скрам-мастер • Команда 	<ul style="list-style-type: none"> • Беклог продукта • Беклог спринта • Инкремент продукта 	<ul style="list-style-type: none"> • Планирование спринта • Обзор спринта • Ретроспектива • Скрам-митинг • Спринт

Роли:

- **Владелец продукта** (Product owner, Product owner, Менеджер продукта) – это человек, ответственный приоритезацию требований и часто за их создание.
- **Скрам-мастер** – член команды, который дополнительно отвечает за процессы, координацию работы команды и поддержание социальной атмосферы в команде.
- **Команда** – 7±2 человек, которые реализуют требования владельца продукта.

Артефакты:

- **Беклог продукта** (Product Backlog) – приоритизированный список требований с оценкой трудозатрат. Обычно он состоит из бизнес-требований, которые приносят конкретную бизнес-ценность и называются элементы беклога.
- **Беклог спринта** (Sprint Backlog) – часть беклога продукта, с самой высокой важностью и суммарной оценкой, не превышающей скорость команды, отобранная для спринта.
- **Инкремент продукта** – новая функциональность продукта, созданная во время спринта спринта.

Процессы:

Большинство процессов Scrum носят характер встреч, так как данная методология основана на качественных коммуникациях.

- **Скрам-митинг** (Scrum meeting, скрам, ежедневный скрам, планерка) – собрание членов команды (с возможностью приглашения владельца продукта) для синхронизации деятельности команды и обозначения проблем. Каждый член команды отвечает на три вопроса:
 1. Что было сделано с предыдущего скрам-митинга?

2. Какие есть проблемы?
3. Что будет сделано к следующему скрам митингу?

Если первый и третий пункт служат для синхронизации деятельности команд, то второй пункт очень важен для выработки решений проблем: если проблема действительно небольшая, ее можно решить или выработать решение прямо на скрам-митинге, если серьезная и требует обсуждения, ее решить после скрам-митинга.

- **Планирование спрингта**

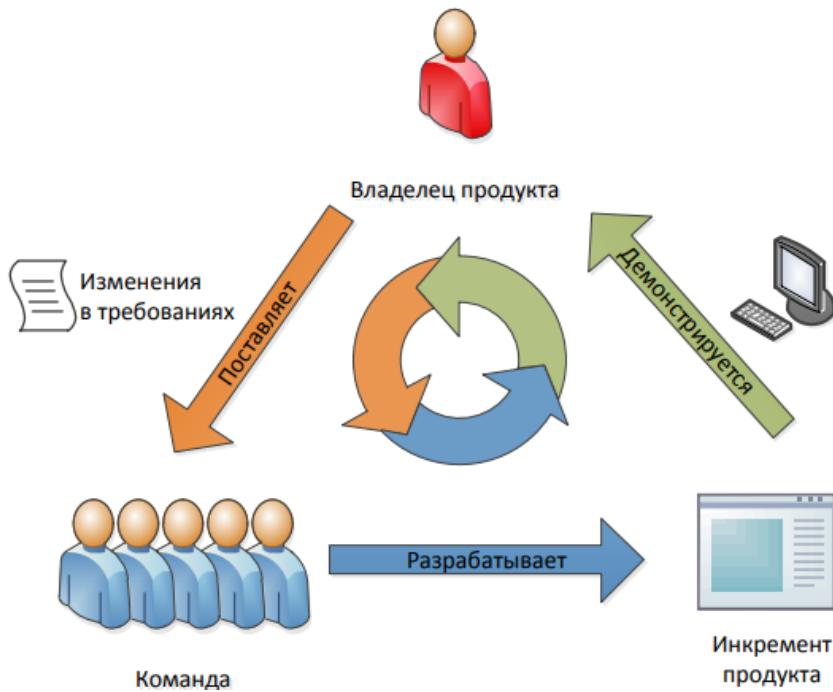
Для планирования спрингта необходимо иметь качественный беклог, что означает следующее:

- все элементы беклога должны иметь уникальную числовую важность;
- самые важные элементы беклога должны быть уточнены и понятны всей команде и владельцу продукта;
- владелец продукта должен четко представлять, что будет реализовано в рамках каждого элемента беклога.

Основным результатом планирования спрингта является беклог спрингта – список задач, которые команда планирует реализовать в рамках спрингта. Поскольку длина спрингта в Scrum жестко фиксирована, то команда определяет количество элементов беклога (объем работ), которые она может реализовать. Можно данную ситуацию отобразить на классическом «треугольнике управления проектами»:



- **Обзор спрингта** (также часто используется термин «демонстрация» или сокращенно «демо») – показ владельцу продукта (и заинтересованным лицам) работающего функционала продукта, сделанного за спрингт. Основная задача проведения обзора спрингта заключается в получении обратной связи, а общий цикл ее получения выглядит следующим образом:



Демонстрация результатов работы не только мотивирует команду, но и подталкивает реализовывать задачи полностью.

В обзоре спринта обязательно должно принимать участие вся команда, при этом возможны разные стратегии показа. Антипаттерном можно назвать демонстрацию функционала одним человеком, например, скрам-мастером.

- **Ретроспектива**

Ретроспективу традиционно проводят после обзора спринта спустя небольшое количество времени, чтобы оперативно получить фидбек. Скрам-мастер собирают всю команду для обсуждения результатов спринта. Рекомендуется на ретроспективу приглашать владельца продукта для получения дополнительной обратной связи.

В долгосрочном плане ретроспективы (или сокращенно «ретро») являются самой важной практикой Scrum: ведь именно они позволяют адаптировать и кастомизировать Scrum, делая из него по-настоящему гибкий фреймворк для управления проектами.

Структура ретроспектизы

Обычно ретроспектива занимает от 30 минут до 4 часов и ее продолжительность зависит от следующих факторов:

- Длина спринта: чем длиннее спринт, тем больше команда успевает сделать и тем больше материала для обсуждения;
- Размер команды: чем команда больше, тем больше надо времени, чтобы у каждого ее члена была возможность высказаться и тем больше функционала команда успевает сделать;
- Наличие проблем: со временем команда решает проблемы и ретроспективы сокращаются по времени.

В процентном соотношении принято выделять такую структуру:



Также традиционным является формат по сбору данных, который заключается в ответах каждого участника на три вопроса:

1. Что было сделано хорошо?
2. Что можно улучшить?
3. Какие улучшения будем делать?

Количество улучшений, которые команда берет в реализацию, не должно превышать 2-3, чтобы не снизить скорость реализации бизнес-функционала и не потерять фокус. Команда должна обязательно в том или ином виде составить план улучшений для контроля их исполнения.

Для максимальной открытости и прозрачности обсуждения необходимо использовать основное правило ретроспективы, которое можно озвучивать в начале:

«В независимости от того, что удастся выяснить в результате ретроспективы, каждый член команды сделал всё, чтобы добиться успеха»

Если у команды отсутствуют яркие проблемы, то желательно следующие темы обсудить на ретроспективе:

- скорость команды и ее изменение по сравнению с предыдущими спринтами;
- нереализованные истории пользователей и причины опоздания;
- дефекты и их причины;
- качество процессов (нарушения, отступления).

К паттернам можно отнести анализ сделанных улучшений за несколько прошлых спринтов. Такая «ретроспектива ретроспектив» может проводить раз в 4 спрингта и позволяет контролировать уровень сделанных улучшений.

ИСТОЧНИКИ:

1. [Борис Вольфсон. Гибкие методологии \(стр. 16-21\)](#)

ВОПРОС №55

Scrum. Этапы командообразования. Самоорганизация в командах, модель CDE.

Scrum (англ. *scrum* «схватка») — гибкая методология управления проектами, используемая не только в сфере разработки ПО, но и других производственных отраслях.

Scrum — минимально необходимый набор мероприятий, артефактов, ролей, на которых строится процесс scrum-разработки, позволяющий за фиксированные небольшие промежутки времени (спринты), предоставлять конечному пользователю работающий продукт с новыми бизнес-возможностями, для которых определён наибольший приоритет.

Гибкие методологии опираются на людей и взаимодействие между ними.

Команда — это небольшая группа людей, взаимодополняющих и взаимозаменяющих друг друга, которые собраны для совместного решения задач производительности и взаимной ответственности.

Чтобы группа людей превратилась в сплоченную команду, она должна пройти несколько этапов:

1. Формирование.

Происходит создание команды и постановка целей, распределение и закрепление ролей. Отдельные члены команды еще не очень понимают поставленную цель и задачи.

2. Бурление.

Участники осознают свои цели и определяют вектор движения.

Цель стала более четкой и понимаемой для команды. Возможны конфликты и противостояние между членами команды, поэтому возрастает роль скрам-мастера как модератора.

3. Нормализация.

Члены команды притираются друг к другу и начинают двигаться сонаправленно.

4. Функционирование.

Команда становится самоуправляемой и способной оптимизировать свою производительность. Однако команда не становится бесконтрольной. Руководство устанавливает контрольные точки, чтобы избежать нестабильности и хаоса, но избегает жесткого микроконтроля, который убивает креативность.

5. Расформирование.

Наступает, когда цели, поставленные перед командой, достигнуты.

Чтобы работать эффективно, команда должна находиться на этапе функционирования. Главной задачей команды является максимально быстрый переход между этапами. Самый лучший вариант - взять сыгранную команду.

Важным свойством команды является ее самоорганизация: команда сама определяет способ, которым сделает из элементов журнала пожеланий инкремент продукта. Эффективность можно измерить сроком, составом и ценой работ. При этом команде приходится адаптироваться к ограничениям в проекте, и условиям окружающей среды.

В гибких командах ведется параллельная работа: контроль и «власть» децентрализованы и распределены между членами команды. Решения, из которых складывается конечный результат, принимаются каждым членом команды, разделяя ответственность, но не размывая ее между всеми.

Модель CDE (Container/Differences/Exchanges) представляет собой простой фреймворк для развития самоуправляемых команд.

Можно влиять на самоорганизующиеся команды, изменяя:

- Контейнеры: формальные команды, неформальные команды и их ожидания
- Различия: увеличивая или уменьшая различия внутри или между контейнерами
- Обмен: добавляя новых людей, инструменты и техники

Ссылка:

Стр. 35-39

<https://strategium.space/wp-content/uploads/2018/07/Gibkie-metodologii.pdf>

ВОПРОС № 56**Scrum. Покер-планирование.**

Покер-планирование - оценка командой проблем и историй пользователей. Процесс заключается в независимом выставлении оценки по каждому стори-поинту. Причем обсуждение командной происходит до тех пор, пока у всех не будет единого мнения. Оценки выставляются с помощью заранее заготовленных карт, которые члены команды выкладывают рубашкой вверх при объявлении пользовательской истории.

Само по себе число относительно: для него выбирается эталонная задача, которая всем понятна, мала, проста и характерна для контекста проекта и ей присваивается коэффициент 1. Остальные задачи оцениваются относительно эталонной по логарифмической шкале (0, 0.5, 1,2, и т.д. до 100).

Таким образом, планирование происходит раундами, где участники оценивают сложность пользовательских историй. Происходит до тех пор, пока все участники не сойдутся в оценках.

Использованные источники:

Вольфсон Борис / Гибкие методологии разработки. Издание 1.2. Стр. 41 - 46.

Режим доступа:

<https://disk.yandex.ru/d/i-kJzGx2PLCW3>

ВОПРОС №57

Scrum. Диаграмма сгорания. Улучшенная диаграмма сгорания (Вольфсон, 48-50, Подвижная мишень и дрожащие руки, Дорофеев Максим)

Для мониторинга прогресса в Scrum используется специальный график – диаграмма сгорания (burndown diagram). По горизонтальной оси - дни спринта, а по вертикальной - количество оставшихся сторипоинтов и/или количество закрытых историй пользователей. Дополнительно строится идеальная диаграмма сгорания, которая показывает запланированный ход работ:

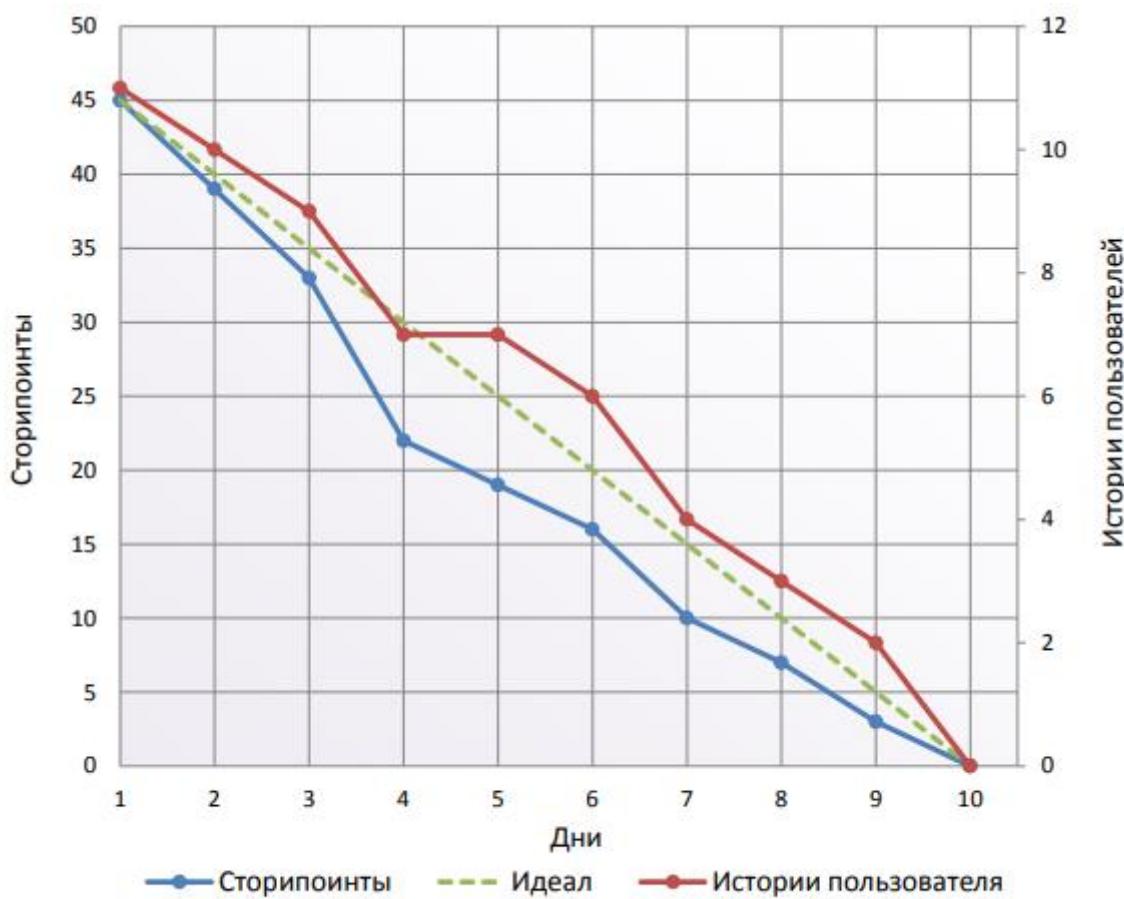


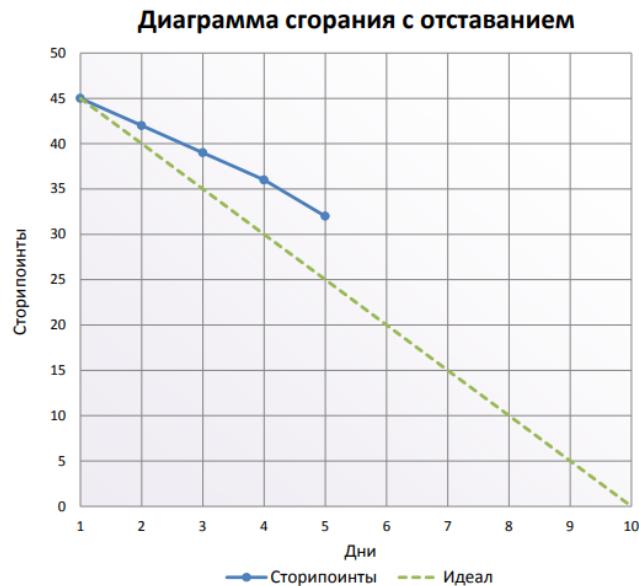
Рисунок 27. Диаграмма сгорания показывает, что спрент завершился в соответствии с планом

Анализ производится путем сравнения реального графика с идеальным: если реальный график выше идеального, значит, команда отстает от плана; если реальный график ниже идеального, значит, команда опережает план.

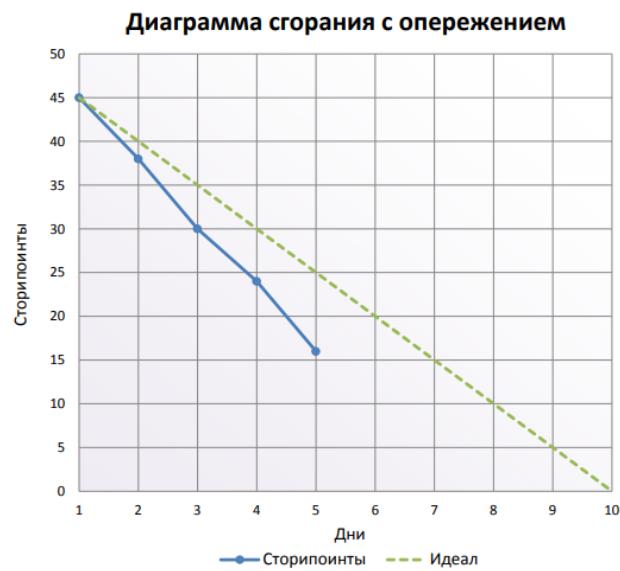
Основные причины отставания от плана:

- Сильная ошибка в планировании;
- Болезнь или иная причина отсутствия одного или нескольких членов команды;
- Недооценка и реализация рисков (обычно технологических)

При отставании необходимо, чтобы владелец продукта убрал из спрента одну или несколько историй пользователя с минимальным приоритетом.

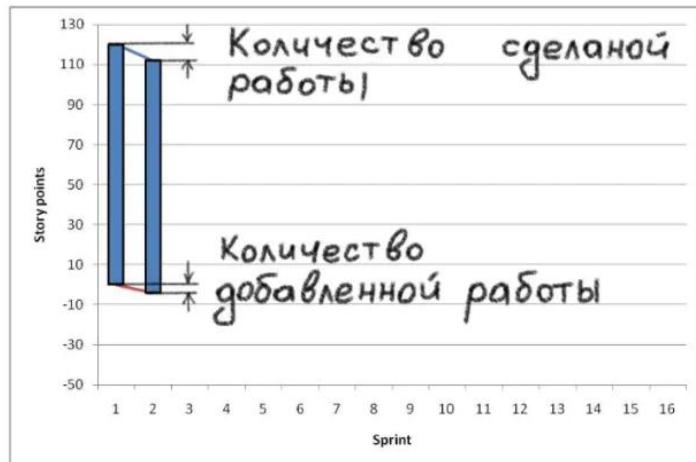


При опережении плана: команде необходимо в беклог спринта взять одну или несколько дополнительных историй пользователей с высшим приоритетом из тех, которые не вошли в спринт.

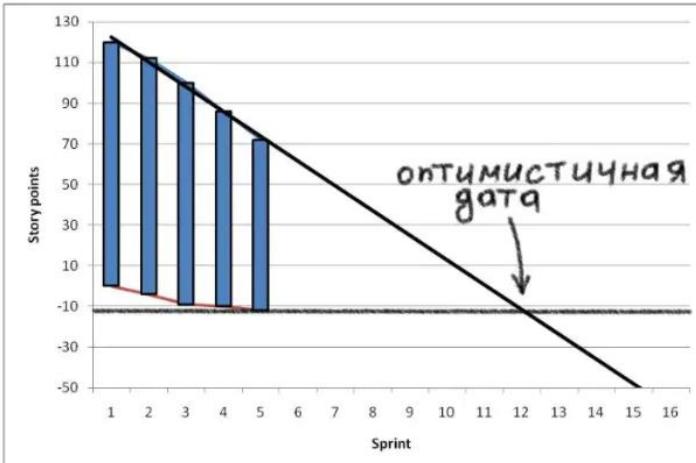


Улучшенная диаграмма сгорания

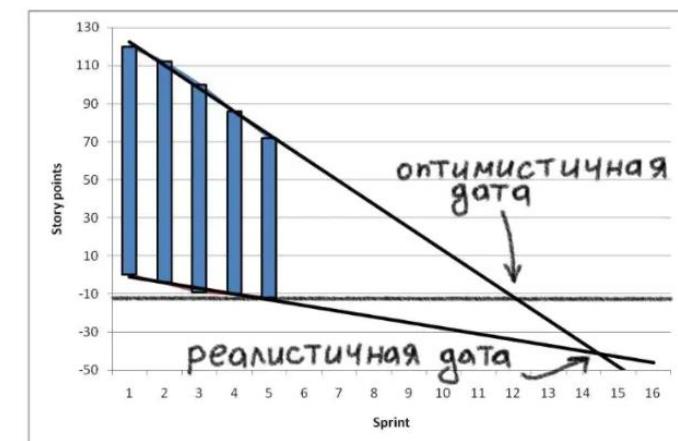
Диаграмма сгорания не учитывает добавление новых задач при условии, что какие-то задачи сделаны.



Если не учитывать добавление новой работы:

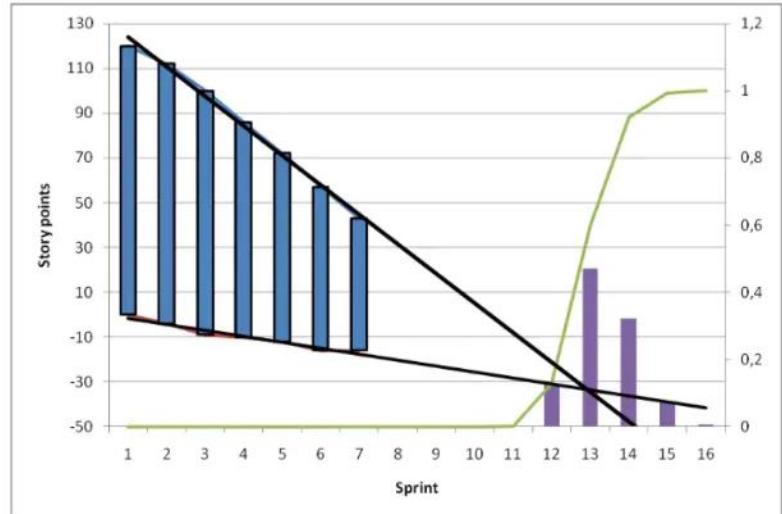
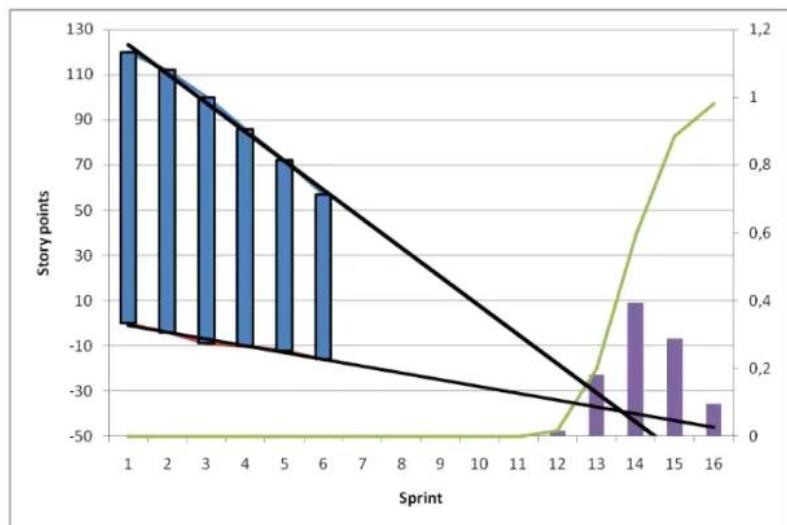
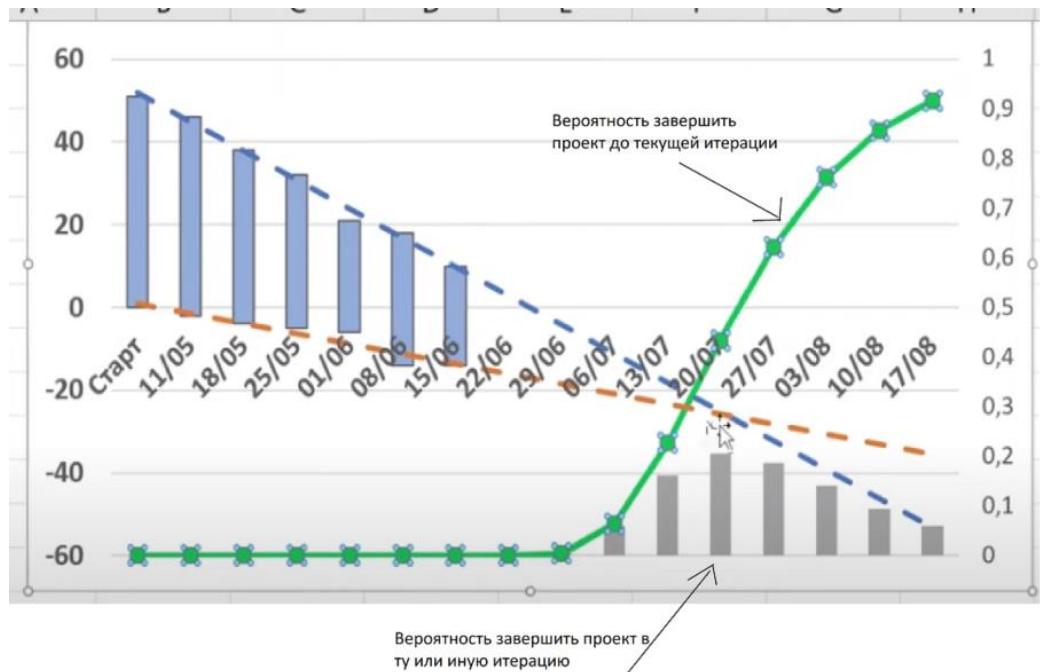


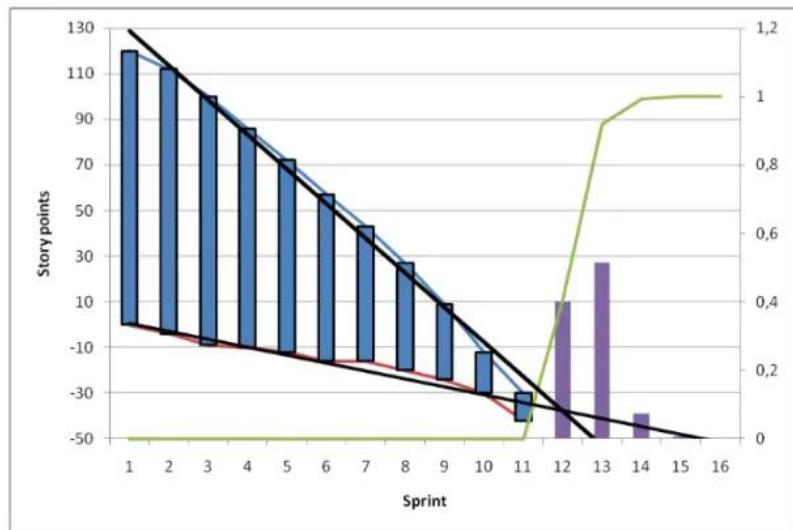
Если учитывать:



Добавление прогнозов на будущее.

Для добавленных и сделанных задач надо посчитать среднее значение и стандартное отклонение (считаем в условных единицах сложности).





Ссылки:

1. Вольфсон Б. Гибкие методологии разработки. СПб.: Питер. 2012. — 112 с.
2. <https://www.slideshare.net/Cartmendum/hitting-moving-target>
3. [Оценка проектов: Велосити и добавлясити \(расширенная диаграмма выгорания работ\)](#)

ВОПРОС №58

Scrum. Доска задач. Теория X и Y (Вольфсон, 50-54)

Доска задач - наглядный инструмент мониторинга и управления внутри спринта.

На стикерах указываются пользовательские истории (названия) и они двигаются по соответствующим состояниям во время спринта.

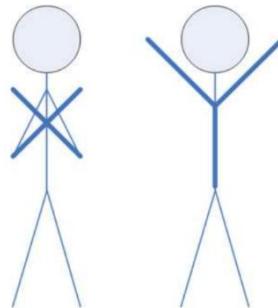
В начале спринта все карточки находятся в первом столбце (План), отсортированные по важности. По мере реализации меняются статусы (Аналитика, Разработка, Тестирование). К концу спринта все карточки находятся в последнем столбце (Готово). После все карточки удаляются.

В некоторых случаях слева добавляется ещё один столбец (Пользовательские истории), относительно большие (3 - 7 на спринт), которые разбиваются на продуктовые задачи (План), которые и передвигаются по доске.

Для наглядности:

Истории пользователей	План	Аналитика	Разработка	Тестирование	Готово
A			A2	A1	
B		B2	B1		
C	B3 C1 C2				

Важность ↑



Теории X и Y - теории мотивации людей (практически противоположные).

Теория X. Люди работают только для того, чтобы получить зарплату. Они не работают, а отрабатывают. Для них работа измеряется часами, а не результатом.

Руководителю не стоит отказываться от таких людей, их много, и среди них есть талантливые. Для них нужно тщательное планирование и контроль результатов. Это проверка для руководства, таких людей зажечь и увлечь работой будет не просто, но не стоит вырабатывать авторитарные стиль руководства.

Теория Y. Люди в принципе высокомотивированы на достижение результатов и сама работа приносит им удовольствие. Их производительность труда намного выше среднего.

Руководителю необходимо следить за мотивацией, понимать ожидания, поддерживать успехи, поощрять инициативы. При контроле необходимо ставить чёткие цели и добиваться понимания и разделения их командой.

X + Y. Обычно приходится работать на стыке, применяя то один подход, то другой. Надо стараться работать в соответствии с подходом Y, и только в крайнем случае, X.

Для работы в рамках Scrum приоритетны люди, соответствующие теории Y, для самоорганизации и эффективности команды.

ВОПРОС №59

Disciplined Agile 2.0 Введение в методологию. Процесс, фазы, дисциплины

Disciplined Agile Delivery (англ. дисциплинированная гибкая разработка), DAD — подход к гибкой разработке ИТ-решений, который ориентирован на обучение и в первую очередь учитывает человеческий фактор. Подход допускает масштабирование и может применяться в масштабах предприятий, а не только небольших команд. Жизненный цикл подхода построен на принципах «риск — ценность» и ориентирован на раннее достижение поставленных целей.

Согласно фреймворку Disciplined Agile 2.X, жизненный цикл проекта содержит три основные фазы:

Начало. Во время этой фазы происходит инициация проекта. Несмотря на то, что agile-сообщество не приветствует разбиение проектов на «фазы», в реальности подавляющее большинство команд выполняет определённый фронт работ в самом начале проекта. Не следует путать данную фазу с «нулевым спринтом», так как чаще всего эти активности занимают больше времени. Таким образом, DAD выделяет эти работы в отдельную фазу, целью которой является фиксирование границ проекта.

Конструирование. Эта фаза позволяет команде разработки создать потенциально используемое решение инкрементальным путём. Это может сделано как с использованием итераций, так и более непрерывным способом. Команда может применять различные практики из скрама, экстремального программирования и прочих гибких методологий разработки.

Передача. Согласно DAD, поставка программного обеспечения заинтересованным сторонам не является тривиальным процессом. Команды разработки, равно как и предприятие, получающее готовый результат, улучшают процессы доставки по мере жизни продукта, так чтобы данная фаза занимала минимальное количество времени, а в идеале и исчезла бы совсем.

Процессный каркас DAD строится на десяти **принципах**:

1. **Люди превыше всего.** DAD опирается на стратегию формирования многофункциональных команд, состоящих из специалистов широкого профиля. Участники команды поощряются к приобретению навыков во многих областях и к выполнению работы, относящейся не только к их основной области специализации.

2. **Ориентация на обучение.** Среда, поощряющая к обучению, должна решать три основные задачи. Первая — обучение по проблемной области: необходимо изучать и понимать требования заказчиков проекта, а также помогать в этом всей команде. Вторая задача — обучение в целях совершенствования самого процесса разработки на индивидуальном, командном и корпоративном уровне. Третья — это техническое обучение эффективному использованию инструментов и технологий, применяемых в подготовке решения по требованию заказчиков.

3. **Agile-принципы.** Процессный каркас DAD придерживается ценностей и принципов Agile Manifesto и дополняет их.

4. **Гибридность.** DAD перенимает и адаптирует проверенные стратегии таких методологий, как Scrum, Extreme Programming, Agile Modeling, Unified Process, Kanban, Agile Data и т. п. Таким образом, DAD представляет собой agile-методологию второго поколения, опирающуюся на все лучшее, что было создано ранее.

5. **Ориентация на ИТ-решения.** DAD переключает внимание с простой разработки ПО на создание именно решений, представляющих реальную ценность для бизнеса с учетом имеющихся экономических, культурных и технических ограничений.

6. Ориентация на выпуск. В отличие от DAD, agile-методы первого поколения обычно фокусировались на этапе сборки, а все остальное оставлялось на откуп разработчикам.

7. Ориентация на цели. Ориентированный на цели рекомендательный подход дает командам разработки достаточно указаний, но при этом является вполне гибким, чтобы процесс можно было адаптировать в соответствии с конкретными обстоятельствами.

8. Разработка в контексте предприятия. Команды скорой разработки действуют не в вакууме, и часто необходимо как минимум позаботиться о том, чтобы новое решение не влияло на существующие рабочие системы заказчика, а лучше, чтобы ваша система пользовалась уже имеющимися функциональностью и данными.

9. Акцент на риски и ценность. Процессный каркас DAD полагается на жизненный цикл, подразумевающий минимизацию рисков и повышение ценности и, по сути, представляющий собой «облегченный» вариант используемого в методологии Unified Process.

10. Масштабируемость. Процессный каркас DAD обеспечивает основу для масштабируемых «скорых» ИТ-проектов и является, в частности, важным элементом стратегии IBM Agility at Scale.

ВОПРОС №60

XP: Подход к разработке ПО

XP - гибкая методология разработки ПО, которая доводит использование многих общепринятых и широко используемых принципов программирования до экстремальных уровней.

Основные практики экстремального программирования:

- **Вся команда** - все участники проекта работают как одна команда. В нее обязательно входит представитель заказчика, который выдвигает требования к продукту и расставляет приоритеты в реализации функциональности.
- **Игра в планирование** - в рамках XP используется планирование по нарастающей, в результате общий план проекта возникает достаточно быстро, однако при этом подразумевается, что этот план эволюционирует в течение всего времени жизни проекта.
- **Частые релизы версий** - релизы выпускаются часто, но с небольшим функционалом. Во-первых, маленький объем функциональности легко тестировать и сохранять работоспособность всей системы. Во-вторых, каждую итерацию заказчик получает часть функционала, несущую бизнес-ценность.
- **Пользовательские тесты** - заказчик сам определяет автоматизированные приемочные тесты, чтобы проверить работоспособность очередной функции продукта. Команда пишет эти тесты и использует их для тестирования готового кода.
- **Коллективное владение кодом** - любой разработчик может править любой кусок кода.
- **Непрерывная интеграция** - новые части кода сразу же встраиваются в систему по несколько раз в день. Во-первых, сразу видно, как изменения влияют на систему. Во-вторых, команда всегда работает с последней версией продукта.
- **Стандарты кодирования** - важно принять единые стандарты оформления, чтобы код выглядел так, как будто он написан одним профессионалом.
- **Метафора системы** - это ее сравнение с чем-то знакомым, чтобы сформировать у команды общее видение.
- **Устойчивый темп** - команды работают на максимуме продуктивности, сохранив устойчивый темп. При этом экстремальное программирование негативно относится к переработкам и пропагандирует 40-часовую рабочую неделю.
- **Разработка через тестирование** - сначала пишутся тесты, затем реализуется логика, необходимая, чтобы тесты прошли. В таком случае тестами покрывается большая часть функционала.
- **Парное программирование** - код создается парами программистов, работающих за одним компьютером. Один из них работает с исходным кодом программы, а второй просматривает его работу и следит за общей картиной происходящего. (экстремальный уровень code review).
- **Простой дизайн** - реализуется только то, что нужно сейчас, не пытаясь угадать будущую функциональность. Простой дизайн и непрерывный рефакторинг дают синергетический эффект — когда код простой, его легко оптимизировать.
- **Рефакторинг** - это процесс постоянного улучшения дизайна системы, чтобы привести его в соответствие новым требованиям. XP предполагает постоянные рефакторинги, поэтому дизайн кода всегда остается простым.

Преимущества экстремального программирования:

- Заказчик получает именно тот продукт, который ему нужен, даже если в начале разработки сам точно не представляет его конечный вид.

- Команда быстро вносит изменения в код и добавляет новую функциональность за счет простого дизайна кода, частого планирования и релизов.
- Код всегда работает за счет постоянного тестирования и непрерывной интеграции.
- Команда легко поддерживает код, т.к. он написан по единому стандарту и постоянно рефакторится.
- Быстрый темп разработки за счет парного программирования, отсутствия переработок, присутствия заказчика в команде.
- Высокое качество кода.
- Снижаются риски, связанные с разработкой, т.к. ответственность за проект распределается равномерно и уход/приход члена команды не разрушит процесс.
- Затраты на разработку ниже, т.к. команда ориентирована на код, а не на документацию и собрания.

Недостатки экстремального программирования:

- Успех проекта зависит от вовлеченности заказчика, которой не так просто добиться
- Трудно предугадать затраты времени на проект, т.к. в начале никто не знает полного списка требований
- Успех XP сильно зависит от уровня программистов, методология работает только с senior специалистами
- Менеджмент негативно относится к парному программированию, не понимая, почему он должен оплачивать двух программистов вместо одного
- Регулярные встречи с программистами дорого обходятся заказчикам
- Требует слишком сильных культурных изменений, чтобы не контролировать каждую задачу
- Из-за недостатка структуры и документации не подходит для крупных проектов
- Т.к. гибкие методологии функционально-ориентированные, нефункциональные требования к качеству продукта сложно описать в виде пользовательских историй.

Кент Бек рекомендует внедрять XP постепенно для решения проблем в проекте. Команда выбирает самую насущную проблему и решает ее с помощью одной из практик экстремального программирования. Затем переходит к следующей проблеме, используя еще одну практику. Для внедрения в XP нужно освоить его методики в областях:

Тестирования - команда создает тесты перед написанием нового кода и постепенно перерабатывает старый код. Для старого кода тесты пишутся по мере необходимости: когда нужно добавить новую функциональность, исправить ошибку или переработать часть старого кода.

Проектирования - команда постепенно рефакторит старый код, обычно перед добавлением новой функциональности. При этом команде стоит сформулировать долгосрочные цели переработки кода и постепенно достигать их.

Планирования - команда должна перейти на тесное взаимодействие с заказчиком.

Менеджмента - роль менеджеров при переходе на XP — контролировать, чтобы все члены команды работали по новым правилам.

Разработки - команда должна программировать парами большую часть рабочего времени, как бы тяжело это не давалось разработчикам.

В настоящее время почти никто не использует экстремальное программирование в чистом виде, однако отдельные практики XP используются большинством компаний, работающих по гибким методологиям.

Источники:

- [Кент Бек - Экстремальное программирование](#), Введение.
- <https://worksection.com/blog/extreme-programming.html>

ВОПРОС №61

ХР: Проблемы и мотивации. Риски при разработке ПО. (Кент Бек, Глава 1.)

Основная проблема разработки программного обеспечения — это риск.

ХР — дисциплина разработки программного обеспечения, которая ориентирована на снижение степени риска на всех уровнях процесса разработки. ХР способствует существенному увеличению производительности и улучшению качества разрабатываемых программ.

Примеры рисков:

Смещение графиков - перенос (например на 6 мес) сроков сдачи проекта.

Снижение риска по ХР: предлагает использовать очень короткие сроки (несколько месяцев) выпуска каждой очередной версии. В связи с этим объемы работы в рамках каждой версии ограничен, что положительно сказывается на возможных смещениях графиков. Внутри каждой версии несколько итераций запрашиваемых заказчиком возможностей (1-4 недели). Заказчик получает гибкую и четкую обратную связь. Итерация делиться на задачи (1-3 дня). Позволяет команде находить и устранять проблемы во время итерации. Так же работают приоритеты, то есть задачи с высшим приоритетом решаются в первую очередь.

Закрытие проекта - после нескольких смещений графика и переносов даты сдачи проекта, проект закрывается.

Снижение риска по ХР: заказчик определяет минимальное набор возможностей, которыми должна обладать минимальная работоспособная версия программы, имеющая смысл с точки зрения решения бизнес-задач. Из этого следует, что программисту стоит приложить минимальное количество усилий для того, чтобы заказчик понял нужен ли ему этот проект.

Система теряет полезность - система успешно разворачивается, но по прошествию пары лет стоимость изменений и/или количество дефектов увеличивается настолько, что легче просто заменить на новую.

Снижение риска по ХР: из-за того что проводится много тестов, качество системы увеличивается, а значит дефектам просто не дают накапливаться.

Количество дефектов и недочетов - система устанавливается в реальной среде, однако количество дефектов и недочетов настолько велико, что системой не пользуются.

Снижение риска по ХР: реализуются тесты как со стороны программистов для функций так и со стороны заказчика для реализованной возможности.

Несоответствие решаемой проблеме - система устанавливается в реальной среде, но не решает проблему бизнеса для решения которой она разрабатывалась.

Снижение риска по ХР: заказчик является составной частью команды, проект постоянно перерабатывается и любые уточнения и открытия находят свое отражение в разработке.

Изменение характера бизнеса - система устанавливается в реальной среде, однако в течении 6 последних месяцев проблема для которой разрабатывалась система потеряла актуальность, а вместо нее бизнес столкнулся с более серьезной проблемой.

Снижение риска по ХР: цикл работы над версией существенно укорочен из-за чего бизнес не успевает претерпевать существенные изменения. В процессе разработки новой версии заказчик может отказаться от некоторой функциональности и добавить новые возможности. Разработчики могут и не заметить этого.

Недостаток возможностей - система имеет много потенциально интересных возможностей, которые было приятно разрабатывать, но ни одна из них не приносит заказчику достаточно много пользы.

Снижение риска по XP: в рамках XP осуществляется реализация только наиболее высокоприоритетных задач.

Текучка кадров - в течении двух лет, все хорошие программисты возненавидели разрабатываемую систему и ушли на другую работу.

Снижение риска по XP: XP предлагает программистам брать на себя ответственность самостоятельно определять объем работы и время, необходимое для выполнения этой работы. В рамках XP содержатся правила, определяющие, кто именно имеет право делать предварительные оценки и изменять их. XP стимулирует интенсивное общение между членами команды разработчиков (снижает одиночество). XP явно определяется модель смены кадров (постепенное увеличение ответственности).

ВОПРОС №62

Проблемы и мотивации. Модель четырех переменных.

см. (Кент Бек, Глава 4.)

В рамках данной модели (систему контролируемых переменных) разработка программного обеспечения определяется с использованием следующих четырех переменных:

- затраты (cost);
- время (time);
- качество (quality);
- объем работ (scope).

Внешние силы (заказчики, менеджеры) должны определить значения для любых трех переменных из указанного набора, при этом команда разработчиков должна выбрать результирующее значение для оставшейся переменной.

Взаимосвязь между переменными

- **Затраты** (cost) - чем больше денег, тем легче работать, однако слишком большое количество денег в самые кратчайшие сроки создаст больше проблем, чем требуется решить. С другой стороны, если вы выделяете на проект слишком мало денег, вы не сможете решить поставленные перед вами заказчиком проблемы
- **Время** (time) - с увеличением объема времени, выделяемого на выполнение проекта, у вас появляется возможность повысить качество разрабатываемой программы, а также расширить объем работ. Однако отзывы о системе, которая уже эксплуатируется в реальных рабочих условиях, гораздо ценнее, чем любые другие отзывы, поэтому, если вы выделите для выполнения проекта слишком много времени, проект может пострадать. Если вы выделите для проекта слишком маленькое время, пострадает качество.
- **Качество** (quality) - эта переменная контролируется хуже всего. Если вам необходимо достигнуть каких-либо краткосрочных целей (для достижения которых требуется несколько дней или недель), вы можете намеренно пожертвовать качеством, однако связанные с этим затраты — человеческие, деловые и технические — могут оказаться чрезмерными.
- **Объем работ** (scope) - сократив объем работ, вы можете повысить качество (при условии, конечно, что поставленная заказчиком задача решена). Сокращение объема работ позволяет также сократить время проекта и связанные с проектом затраты.

Между этими переменными **нет** простой зависимости.

Затраты являются наиболее ограничивающей переменной. Вы не можете напрямую менять деньги на качество, объем работ или скорость, с которой выпускаются промежуточные версии продукта. В начале работ инвестирование необходимо выполнять понемногу и затем, с течением времени, увеличивать объемы вложений. В процессе того, как проект будет развиваться, вы сможете эффективно тратить все большее и большее количество денег. С другой стороны, затраты тесно связаны с другими переменными. Если, действуя в допустимых пределах, вы увеличите объем инвестиций, вы можете расширить объем работ, или вы можете действовать с большей свободой и улучшить качество, или вы можете (до определенной степени) уменьшить время, необходимое для завершения работы. Затратив дополнительные деньги, вы также снижаете трение — вы получаете более быстрые

компьютеры, большее количество технических специалистов, более просторные и удобные офисы.

Ограничения, связанные с **временем** реализации проекта, как правило появляются извне. В качестве примеров внешних временных ограничений можно привести также конец года, начало следующего квартала, дата планируемого завершения работы старой системы, крупная торговая выставка. Зачастую время не поддается контролю со стороны менеджеров проекта — его контролируют заказчики.

Качество — это еще одна весьма странная переменная. Зачастую, настаивая на улучшении качества, мы можете завершить проект быстрее, чем запланировано. Или вы можете успеть сделать больше за заданный интервал времени. Существует весьма странная зависимость между внутренним и внешним качеством. Внешнее качество — это качество, измерением которого занимается заказчик. Внутреннее качество оценивается программистами.

Объем работ, связанный с производством программного продукта, — это наиболее важная переменная и сильно изменяющаяся переменная, с которой приходится иметь дело в производстве программного продукта. Если вы активно управляете показателем объема работ, вы можете предоставить менеджерам и заказчикам контроль над затратами, качеством и временем. Показатель объема работ является наиболее удобной в управлении переменной из четырех переменных. Так как этот показатель наименее четко очерчен, мы можем формировать его в соответствии с нашими собственными предпочтениями и предпочтениями заказчика — немного в эту сторону, немного в ту сторону. Вначале мы можем рассматривать объем работ как функцию от первых трех переменных, однако в дальнейшем мы получаем возможность постоянно корректировать объем работ с учетом складывающихся условий.

ВОПРОС №63

Вопрос: XP: Проблемы и мотивации. Стоимость изменений. (Кент Бек, Глава 5.)

Алгоритм внедрения методологии XP и процесс работы

Бек Кент рекомендует внедрять XP для решения проблем в проекте. Команда выбирает самую насущную проблему и решает ее с помощью одной из практик экстремального программирования. Затем переходит к следующей **проблеме**, используя еще одну практику. При таком подходе проблемы выступают **мотивацией** к применению XP и команда постепенно осваивает все инструменты методологии

Проблема: «Затраты, связанные с исправлением проблемы, обнаруженной внутри программного продукта, растут экспоненциально с течением времени. Проблема, для решения которой в процессе анализа требований потребовался бы доллар, может стоить вам нескольких тысяч долларов, если вы обнаружите ее в момент, когда система уже будет в производстве». (Как на рис. 1.)

Мотивация - хотим не тратить несколько тысяч долларов

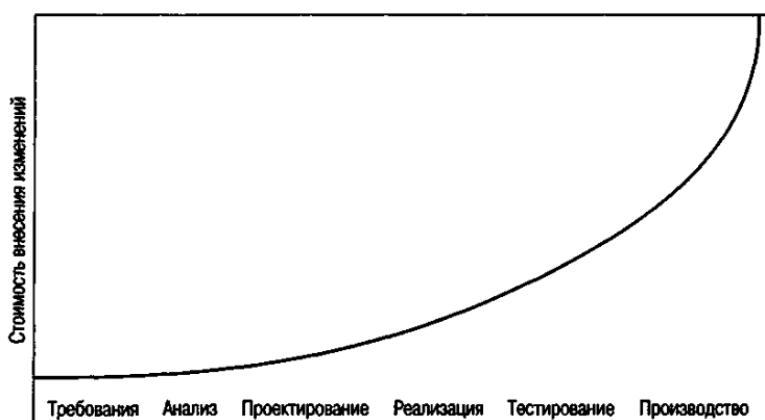


Рис. 1. С течением времени стоимость внесения изменений в программный продукт возрастает экспоненциально

Сохранение низкой стоимости изменений — это не какой-то магический трюк, оно достигается не просто так, а в результате применения технологий и методик, которые позволяют сделать программный продукт податливым и легко модернизируемым.

С технологической точки зрения ключевой технологией, позволяющей добиться этого, являются объекты. Обмен сообщениями между объектами позволяет существенно расширить спектр возможностей по изменению разрабатываемой системы. Каждое сообщение становится потенциальной точкой для внесения в систему грядущих модификаций, модификаций, которые могут вносится в систему, не затрагивая при этом существующий код.

Объектные базы данных переносят эту гибкость в пространство постоянной памяти. Благодаря использованию объектной базы данных вы получаете возможность с легкостью переносить информацию об объектах из одного формата в другой, так как в объектных базах данных код соединен с данными, а не отделен от них, как это было в более ранних базах данных. Даже если вы не можете найти способ выполнить миграцию объектов, вы можете обеспечить в рамках одной системы сосуществование двух разных реализаций

Чтобы упростить модификацию вашего кода даже спустя несколько лет после начала работы над проектом, вы должны учитывать следующие факторы:

- простой дизайн, в котором отсутствуют лишние элементы, — никаких идей, которые на текущий момент не используются, однако предположительно могут использоваться в будущем;
- автоматические тесты — благодаря им всегда с легкостью можно узнать о том, что в результате внесения в систему изменений ее поведение изменилось;
- постоянная практика в деле модификации дизайна системы — когда приходит время менять систему, вы не почувствуете страха перед этим.

Если мы заложим в основу нашей работы три перечисленных элемента — простой дизайн, автоматические тесты и опыт постоянного видоизменения системы, мы увидим, что кривая расходов, связанных с внесением в систему изменений, станет пологой, как на рис. 3.

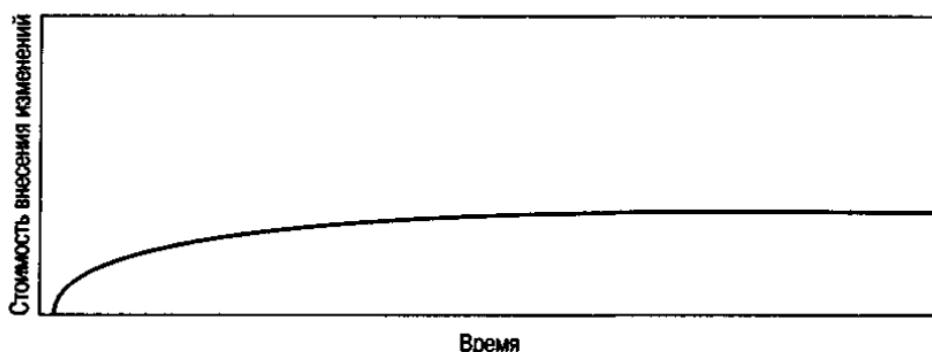


Рис. 3. Стоимость изменений со временем может увеличиваться существенно медленнее, чем экспонента

ВОПРОС № 64

Scrum & XP. Инженерные практики(Вольфсон, 63-67, Кент Бек, Часть 2.)

SCRUM - Самой популярной гибкой методологией разработки ПО. Кроме управления проектами по разработке ПО, SCRUM может также использоваться в работе команд поддержки программного обеспечения, как подход к управлению разработкой и сопровождению программ.

Scrum — вовсе не методология, это гибкий управленческий фреймворк. Под Agile будем подразумевать семейство гибких методологий, а Scrum будем рассматривать в качестве управленческого фреймворка, дополненного практиками из других гибких методологий.

Инженерные практики:

Рефакторинг

Рефакторинг – это изменения исходного кода без изменения функциональности для улучшения внутреннего качества (простота кода, гибкость архитектуры и так далее). Для проведения рефакторинга желательно знать «запахи кода» и непосредственно приемы рефакторинга (Фаулер, 2009):



Парное программирование

При парном программировании код пишется двумя разработчиками за одним компьютером, причем один из разработчиков играет роль «пилота», а второй роль «штурмана»:



Формальные инспекции кода

Для проведения формальной инспекции кода используют специальные чек-листы, в которых указаны правила, которые должен соблюдать программист при разработке кода. Отметим, что эти правила должны быть нетривиальными, и не стоит включать в этот список правила, которые можно проверить автоматически при сборке проекта

Простота архитектуры и метафора системы

Поскольку мы работаем итеративно, нам важно иметь максимально простую архитектуру, в которую будет возможно быстро и дешево внести изменения.

С другой стороны, мы можем для улучшения понимания системы придумать метафору, которая будет ее описывать. Или, в крайнем случае, подобрать понятие из предметной области нашего приложения. Хорошим примером здесь может служить «корзины» в Интернет-магазинах или «окна» в графическом интерфейсе операционных систем

Коллективное владение кодом и стандарт кодирования

Важным преимуществом такого подхода является быстрое распространение знаний между участниками команды.

Для реализации этой практики необходимо использовать стандарты кодирования, чтобы код, написанный разными участниками команды, был одинаков с точки зрения оформления

40-часовая рабочая неделя

Это гарантия для команды от перегрузок, одного из вида потерь в бережливом производстве. Надо очень четко понимать, что количество отработанных часов не равно количеству сделанного функционала, как и в любой интеллектуальной и инженерной деятельности.

ВОПРОС №65

OpenUP: Жизненный цикл, принципы, дисциплины.
[\(http://epf.eclipse.org/wikis/openup/\)](http://epf.eclipse.org/wikis/openup/)

OpenUP - это компактный унифицированный процесс, который применяет итеративный и инкрементальный подходы (легкий и гибкий RUP).

Проект организуется по микро-шагам - маленьким единицам работы. Поэтому в OpenUP чрезвычайно короткий цикл обратной связи, который способствует принятию адаптивных решений в рамках каждой итерации.

Жизненный цикл проекта

1. Начальная
2. Уточнение
3. Конструирование
4. Передача

Жизненный цикл проекта обеспечивает предоставление заинтересованным лицам и членам коллектива точек ознакомления и принятия решений на протяжении всего проекта. План проекта определяет жизненный цикл, а конечным результатом является окончательное приложение.

OpenUP делит проект на итерации: планируемые, ограниченные во времени интервалы, длительность которых обычно измеряется неделями. План итерации определяет, что именно должно быть сдано по окончании итерации, а результатом является работоспособная версия.

Основные принципы

1. Совместная работа с целью согласования интересов и достижения общего понимания
2. Развитие с целью непрерывного обеспечения обратной связи и совершенствования проекта
3. Концентрация на архитектурных вопросах на ранних стадиях для минимизации рисков и организации разработки
4. Выравнивание конкурентных преимуществ для максимизации потребительской ценности для заинтересованных лиц - максимизировать выгоду и соответствовать ограничениям

Дисциплины

1. Требования
2. Архитектура
3. Разработка
4. Тестирование
5. Управление проектами,
6. Управление конфигурацией и изменениями

Другие дисциплины и проблемные области были опущены (бизнес-моделирование, среда, расширенные инструменты управления требованиями). Эти дисциплины либо считаются ненужными для небольшого проекта, либо обрабатываются другими подразделениями организации, не входящими в проектную группу.

ВОПРОС №66

OpenUP: Артефакты, роли. Отличия от RUP.

OpenUp - это итеративно-инкрементальный метод разработки ПО.
Позиционируется как легкий и гибкий вариант RUP.

Основные навыки, необходимые для небольших групп, работающих в одном месте, представлены **ролями** OpenUP:

- Стейхолдер
- Аналитик
- Архитектор
- Разработчик
- Тестировщик
- Менеджер проекта
- Любая дополнительная роль

Артефакт - это то, что создается, модифицируется или используется задачей. Роли отвечают за создание и обновление артефактов. Артефакты подлежат контролю версий на протяжении всего жизненного цикла проекта.

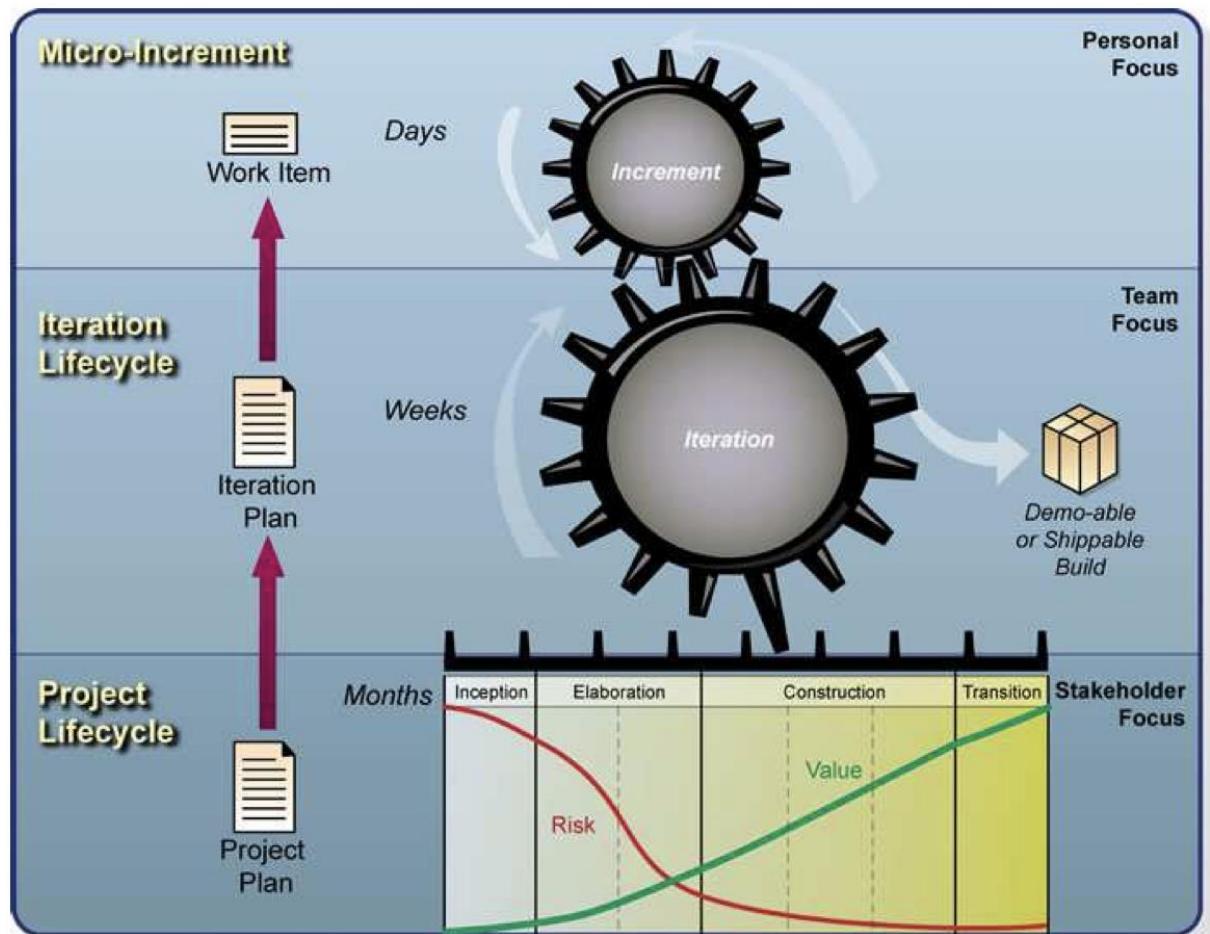
17 артефактов в OpenUP считаются важными артефактами, которые проект должен использовать для сбора информации о продукте и проекте. Нет никаких обязательств по сбору информации в формальных артефактах. Информация может быть неофициально зафиксирована на доске (например, для дизайна и архитектуры), в заметках о встречах (например, для оценки статуса) и т. д. Тем не менее, шаблоны предоставляют стандартный способ сбора информации. Проекты могут использовать артефакты OpenUP или заменять их собственными.

OpenUp как и RUP использует 4 фазы:

- **Начальная фаза.** Согласованы ли масштаб и задачи проекта; следует ли продолжить работу над проектом?
- **Фаза уточнения.** Согласована ли архитектура исполнения, которая будет использоваться для разработки приложения? Являются ли созданная на данный момент потребительская ценность и риск приемлемыми?
- **Фаза конструирования.** Получили ли мы достаточно близкое к завершению приложение; можно ли переключить внимание коллектива на настройку, окончательную отделку и обеспечение гарантии успешного развертывания?
- **Фаза передачи.** Готово ли приложение к выпуску?

Жизненный цикл проекта предоставляет заинтересованным сторонам механизмы надзора, прозрачности и управления для управления финансированием проекта, масштабом, подверженностью рискам, предоставляемой стоимостью и другими аспектами процесса.

Но в отличие от RUP OpenUP предназначен для маленьких проектов и организован в двух различных взаимосвязанных измерениях: method content и process content. В content method определяются элементы method (а именно роли, задачи, артефакты и рекомендации), независимо от того, как они используются в жизненном цикле проекта. Content process - это то, где элементы метода применяются во временном смысле. Многие разные жизненные циклы для разных типов проектов могут быть созданы из одного и того же набора элементов метода. Контент OpenUP обращается к организации работы на личном уровне, уровне команды и заинтересованных сторон, как показано на рисунке 1.



На личном уровне члены команды в проекте OpenUP вносят свой вклад в свою работу с микроинкрементами, которые обычно представляют собой результат от нескольких часов до нескольких дней работы. Приложение развивается по одному микрошагу за раз, и прогресс эффективно виден каждый день. Члены команды открыто делятся своим ежедневным прогрессом на микро-шагах, что повышает видимость работы, доверие и командную работу.

Проект разделен на итерации: запланированные, временные интервалы, обычно измеряемые неделями. OpenUP помогает команде надлежащим образом сосредоточить свои усилия на жизненном цикле итерации, чтобы обеспечить дополнительную ценность для заинтересованных сторон предсказуемым образом - полностью протестированную демонстрационную или поставляемую сборку (приращение продукта) в конце каждой итерации.

Источники:

1. <https://www.eclipse.org/epf/general/OpenUP.pdf>
2. <https://ru.wikipedia.org/wiki/OpenUP>

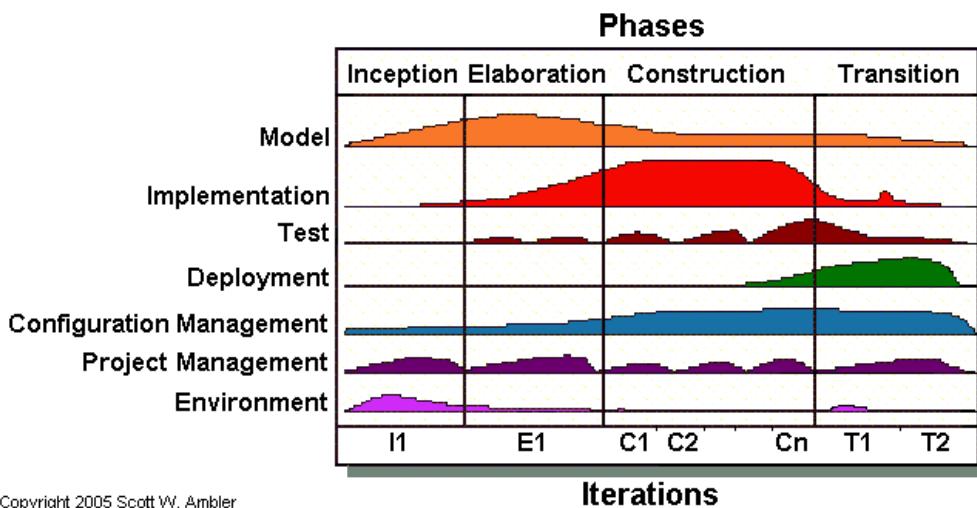
ВОПРОС №67

AUP: Жизненный цикл, философия, инкрементальные выпуски.

Источник: <http://www.amblysoft.com/unifiedprocess/agileUP.html>

AUP -- упрощенная версия унифицированного процесса UP.

Жизненный цикл:



Особенности:

1. Дисциплина “модель” охватывает дисциплины “бизнес-моделирование”, “управления требованиями” и “анализ и проектирование” из RUP
2. “Configuration Management” включает в себя Configuration and Change Management из RUP.

Философия:

1. Члены команды сами знают свою работу. Люди не любят погружаться в изучение детальной документации. Вместо этого они предпочитают краткие инструкции и интерактивное обучение. AUP содержит короткое высокоуровневое описание процесса, ознакомления с которым достаточно для работы. Кроме того, имеется детальное описание процесса.
2. Простота. Все аспекты процесса описываются сжато.
3. Гибкость. AUP соответствует принципам и ценностям гибкой методологии и Agile Alliance.
4. Фокус на высокоуровневой деятельности. Процесс описывает деятельность в рамках проекта на высоком уровне абстракции. Автор не пытается подробно описать все мелкие детали.
5. Независимость от конкретных инструментов. AUP работает вместе с любым набором инструментов, который привычен команде.
6. AUP может быть адаптирован для ваших личных нужд.

Инкрементальные выпуски:

Вместо подхода "big bang", при котором вы предоставляете программное обеспечение сразу, вы вместо этого выпускаете его в производство по частям. Команды AUP обычно доставляют релизы разработки в конце каждой итерации в промежуточные области подготовки производства. **Релиз разработки** – это то, что потенциально может быть запущено в производство, если оно проходит через процессы QA, тестирования и развертывания.

ВОПРОС №68

AUP: дисциплины, артефакты, роли. Отличия от RUP.

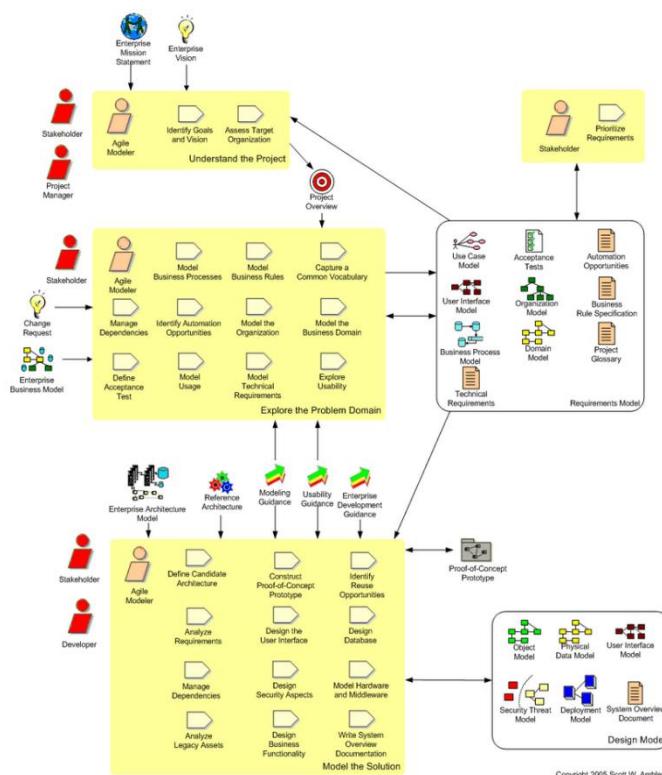
Гибкий унифицированный процесс (Agile UP) — это упрощенный подход к разработке программного обеспечения, основанный на рациональном унифицированном процессе IBM (RUP). AUP применяет гибкие методы, включая разработку через тестирование (TDD), гибкое моделирование (AM), гибкое управление изменениями и рефакторинг базы данных для повышения производительности. Жизненный цикл Agile UP является последовательным и итеративным, обеспечивая постепенные выпуски с течением времени.

Дисциплины выполняются итеративным способом, определяя действия, которые выполняют члены команды разработчиков для создания, проверки и предоставления рабочего программного обеспечения, отвечающего потребностям их заинтересованных сторон.

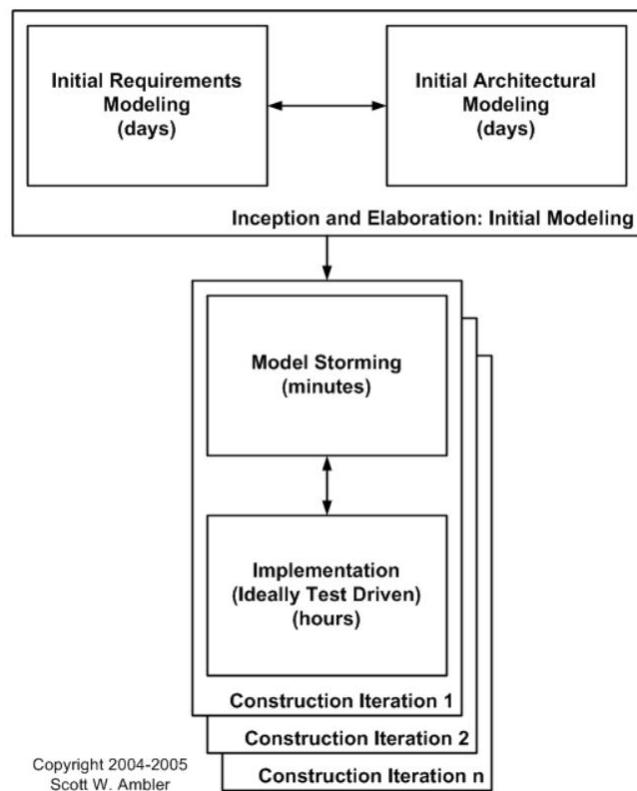
В отличие от RUP, AUP содержит всего **семь** дисциплин:

1. Модель - цель этой дисциплины заключается в определении бизнес-организации, выявлении её проблемной области, а также жизнеспособного решения.

Рабочий процесс



Жизненный цикл разработки на основе гибкой модели (AUP)

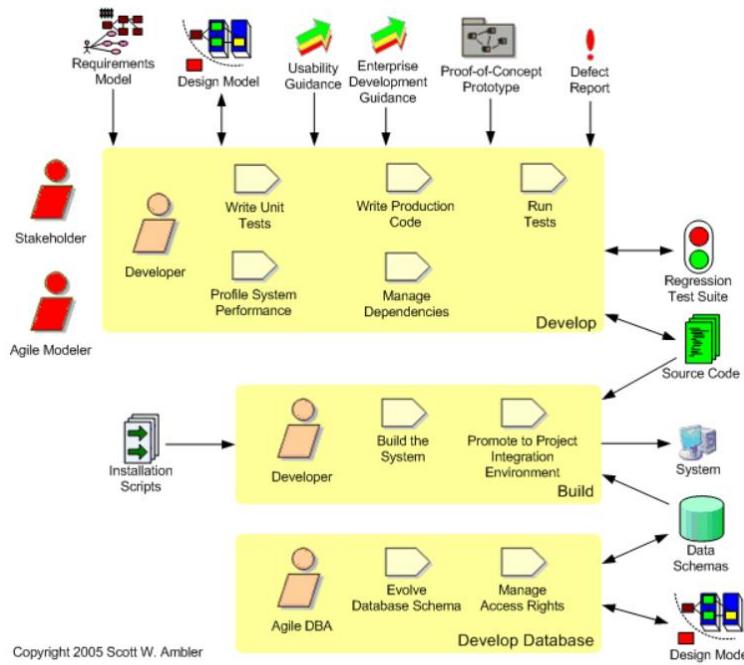


Этапы

Этап	Деятельность
Inception	Определение жизнеспособной архитектурной стратегии, включает: написание сценариев; создание диаграмм потоков данных (DFD); выявление основных бизнес-сущностей и их взаимоотношений; определение основных бизнес-правил и технических требований; написание глоссария; определение политической структуры.
Elaboration	Определение технических рисков и их устранение. Архитектурное моделирование. Прототипирование пользовательского интерфейса.
Construction	Анализ модели с учётом взаимодействия с заинтересованными сторонами проекта. Дизайн-модель проекта: UML-диаграммы, модель развёртывания, диаграмма классов, модель угроз безопасности, физические модели данных.
Transition	Моделирование. Завершение подготовки документации по обзору системы.

2. Реализация – преобразования модели в исполняемый код и выполнение первичного тестирования.

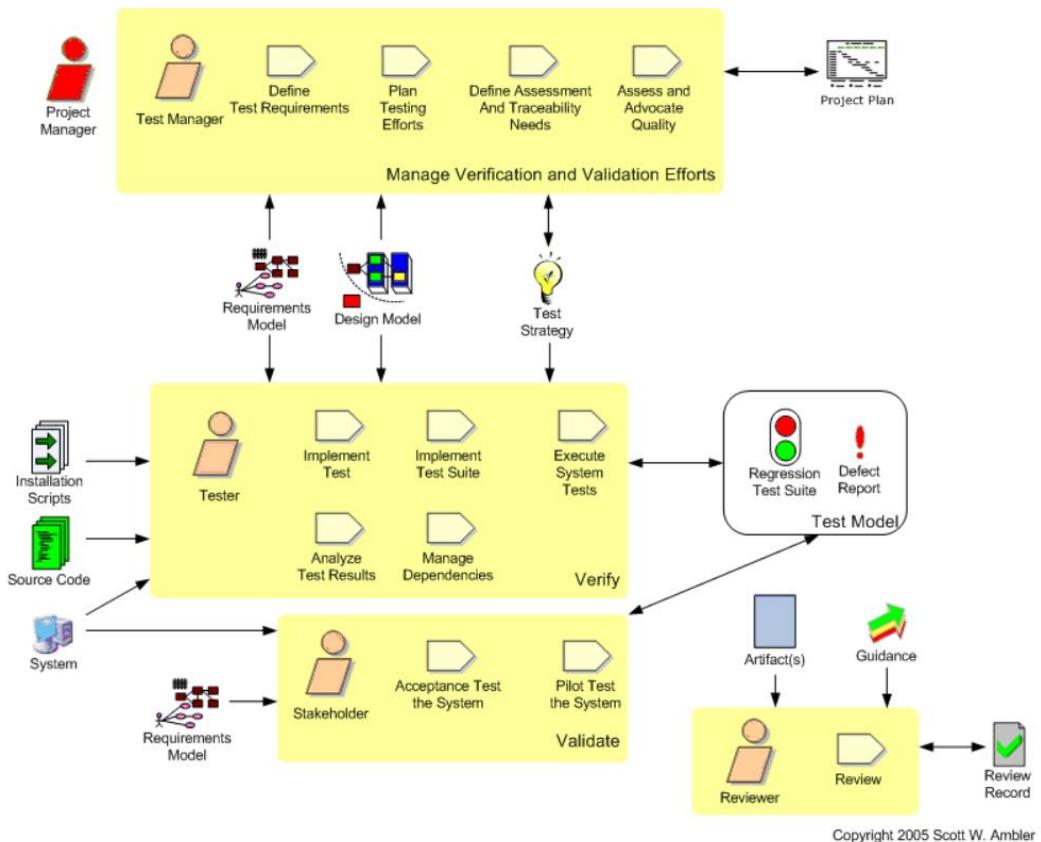
Рабочий процесс



Фаза	Деятельность
Inception	Техническое прототипирование. Прототипирование пользовательского интерфейса
Elaboration	Определение архитектуры
Construction	Тестирование, наращивание логики предметной области, пользовательского интерфейса, схем данных, разработка интерфейсов устаревшего функционала / данных, организация доступа к устаревшим данным.
Transition	Исправление дефектов, обнаруженных после тестирования.

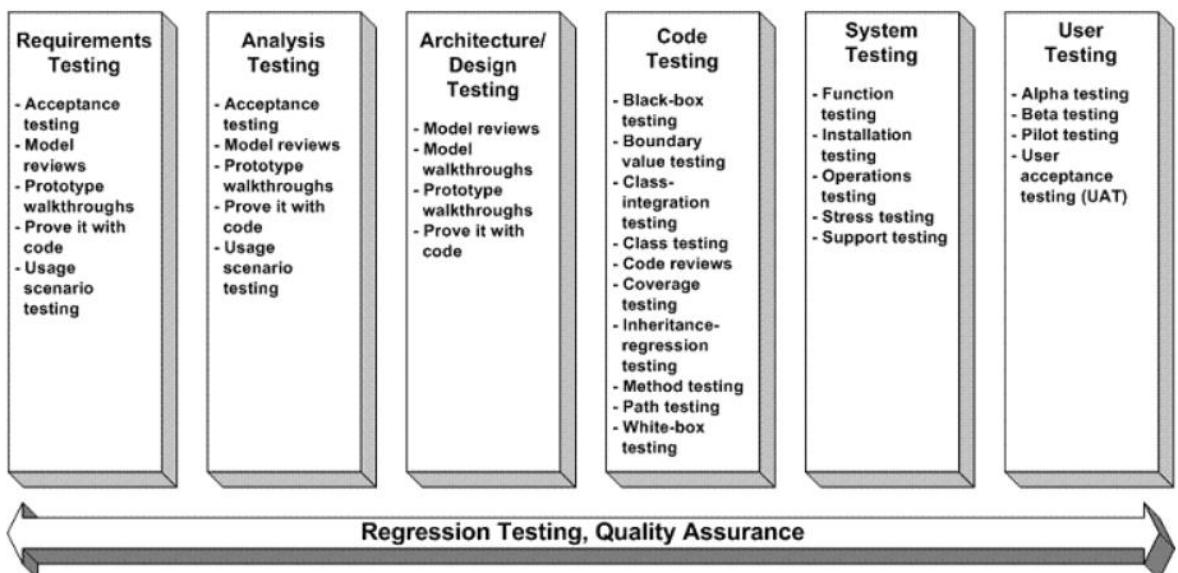
3. Тест – проведение объективной оценки качества. Поиск дефектов, проверка того, что система работает согласно заявленным требованиям.

Рабочий процесс



Copyright 2005 Scott W. Ambler

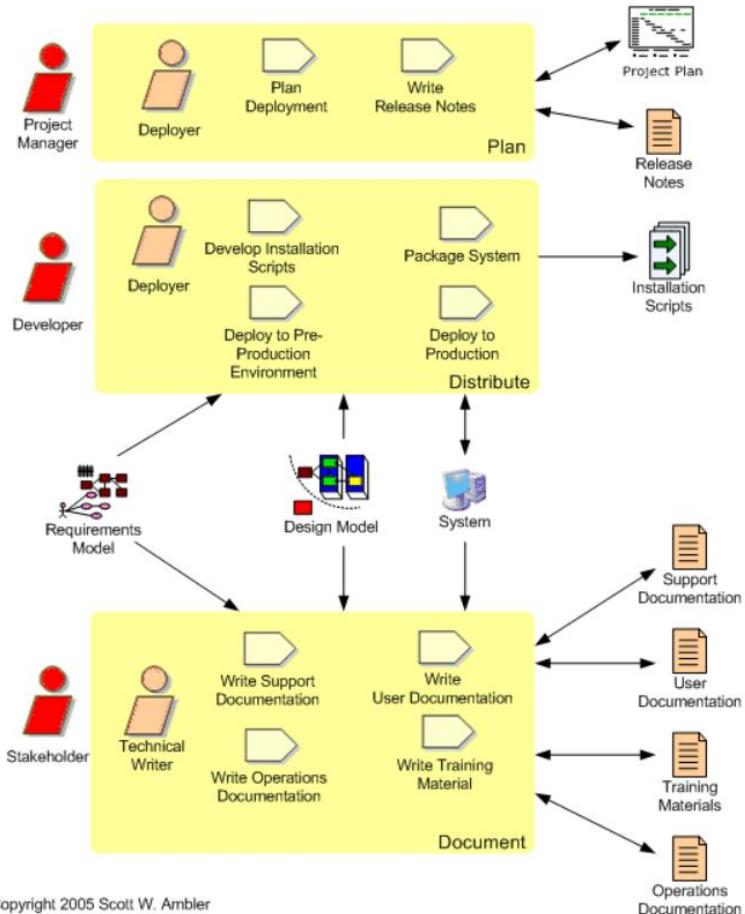
The FLOOR lifecycle



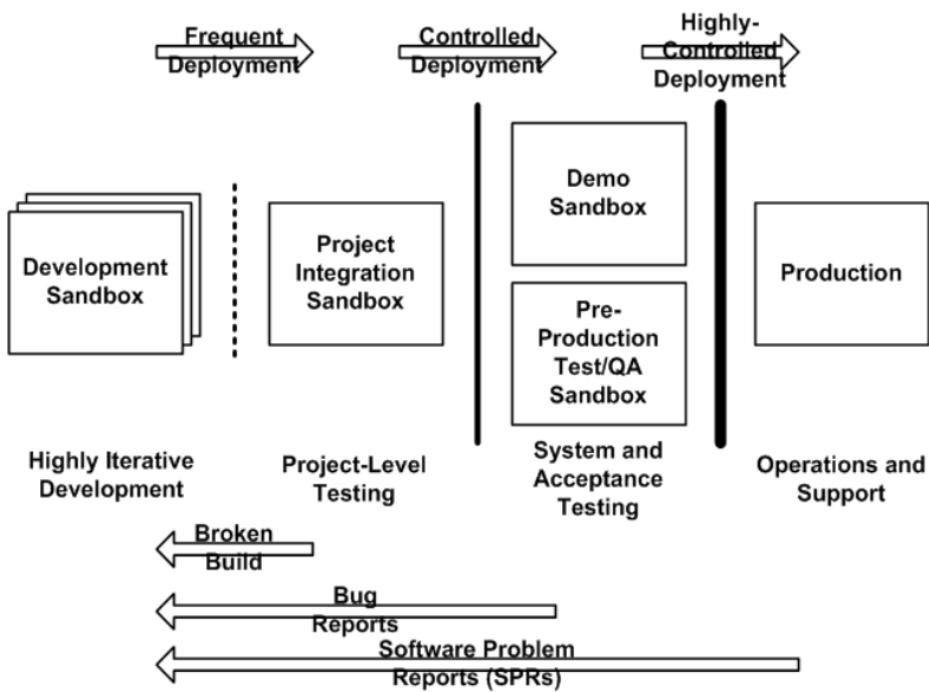
Фаза	Деятельность
Inception	Планирование первоначального тестирования, менеджмент продукта, обзор исходных моделей.
Elaboration	Валидация архитектуры. Разработка тестовой модели системы.
Construction	Тестирование программного обеспечения
Transition	Проверка системы, документации, доработка тестовой модели.

4. Развёртывание – интеграция системы, обуславливающая доступность для конечных пользователей.

Рабочий процесс



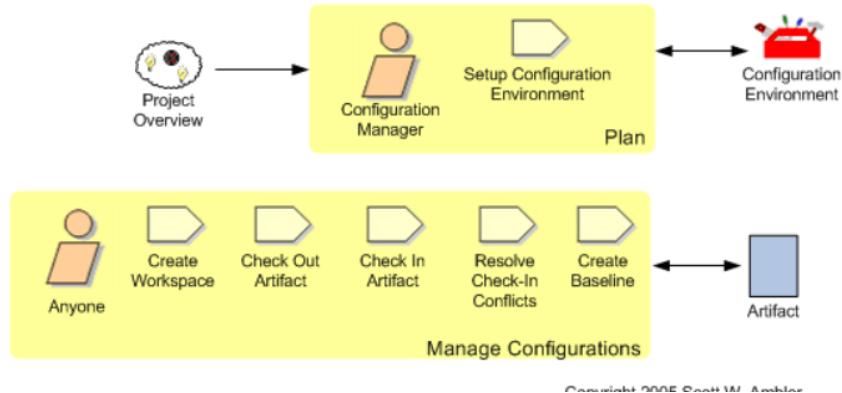
Sandboxes



Фаза	Деятельность
Inception	Определение даты выпуска системы. Высокоуровневое планирование развёртывания системы - дата выпуска, определение аудитории.
Elaboration	Корректирование плана развёртывания системы. Важной частью определения архитектуры является её конфигурация.
Construction	Кодирование - тестирование сценариев установки и удаления. Написание примечаний к выпуску системы. Разработка документации. Развёртывание системы в предпроизводственной среде.
Transition	Завершение этапов развёртывания. Обучение клиентов проекта. Развёртывание системы в производстве.

5. Управление конфигурацией – управление доступом к продуктам проекта, отслеживание рабочих версий, их контроль и управление изменениями.

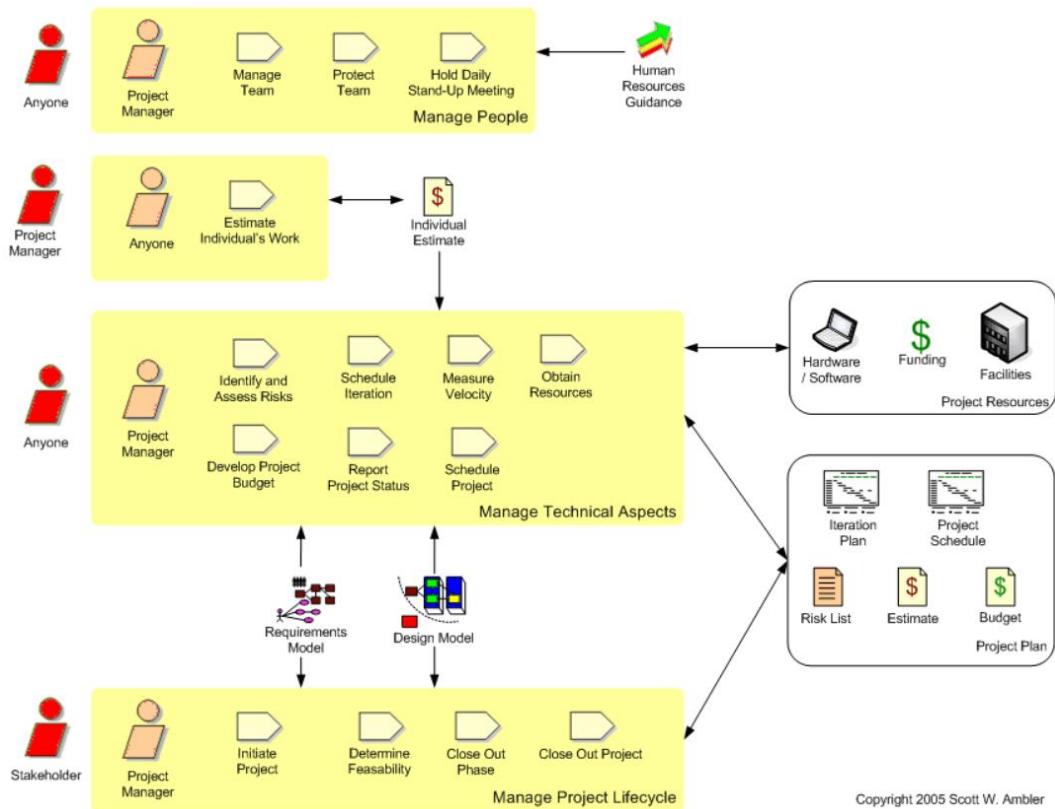
Рабочий процесс



Фаза	Деятельность
Inception	Настройка среды конфигурации - установка репозиторий СМ, структуризация исходных данных, доступ членам команды к папкам проекта. Контроль версий.
Elaboration	=
Construction	=
Transition	=

6. Управление проектом – включает управление рисками, организация коллектива, координация работы людей и систем с целью отслеживания прогресса реализации проекта.

Рабочий процесс

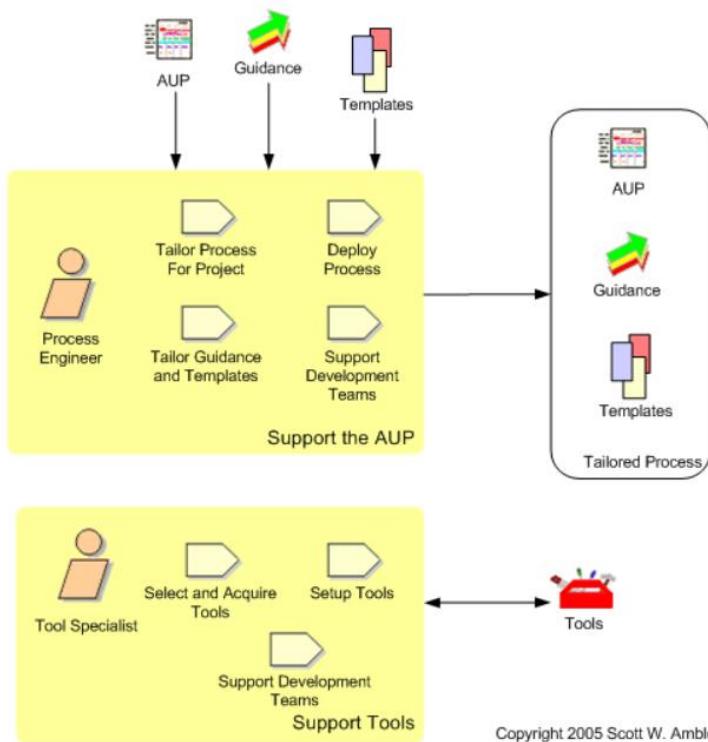


Copyright 2005 Scott W. Ambler

Фаза	Деятельность
Inception	Построение команды проекта, отношений с заинтересованными сторонами. Определение готовности проекта - с финансовой, технической, функциональной и политической точек зрения. Разработка общего графика для всего проекта. Разработка подробного плана итераций. Менеджмент рисков. Получение поддержки и финансирования заинтересованных сторон. Проведение обзора этапов жизненного цикла (LCO) - формализование поддержки проекта инвесторами.
Elaboration	Организационные вопросы команды - создание, снабжение всем необходимым. Проведение обзора этапов архитектурного жизненного цикла (LCA).
Construction	Контрольная проверка операционных возможностей (IOC) - показывает, что ваша команда разработала систему, которая потенциально готова к развертыванию в производственной среде.
Transition	Инициирование следующего проектного цикла.

7. Окружающая среда – организация доступности инструментария (аппаратного и программного обеспечения и т.д.).

Рабочий процесс



Copyright 2005 Scott W. Ambler

Фаза	Деятельность
Inception	Настройка рабочей среды - ПО для рабочих станций. Определение категории проекта.
Elaboration	Tailor the process materials.
Construction	Развитие команды, обучение конечных пользователей.
Transition	Setup operations and/or support environments. Recover software licenses.

Роли

Роль	Описание	Дисциплина
Agile DBA	Администратор базы данных (DBA), который совместно с членами проектной группы работает над проектированием, тестированием, развитием и поддержкой схем (схем) данных приложения.	Implementation
Agile Modeler	Создаёт и улучшает модели эскизы, учетные карточки или сложные файлы инструментов CASE.	Model Implementation
Anyone	Любой человек в любой другой роли.	Configuration Management Project Management
Configuration Manager	Отвечает за предоставление всей инфраструктуры СМ и среды для команды разработчиков.	Configuration Management
Deployer	Отвечает за развертывание системы в предпроизводственной и производственной средах.	Deployment

Developer	Пишет, тестирует и создает программное обеспечение.	Model Implementation Deployment
Process Engineer	Разрабатывает, адаптирует и поддерживает материалы по процессам программного обеспечения вашей организации.	Environment
Project Manager	Управляет членами команды, выстраивает отношения с заинтересованными сторонами.	Model Test Deployment Project Management
Reviewer	Оценивает рабочие продукты проекта.	Test
Stakeholder	Заинтересованная сторона проекта - это любой, кто является прямым пользователем, косвенным пользователем, менеджером пользователей, старшим менеджером, сотрудником отдела операций, сотрудником службы поддержки (справочной службы), разработчиками, работающими над другими системами, которые интегрируются или взаимодействуют с системой, находящейся в стадии разработки или обслуживания.	Model Implementation Test Deployment Project Management
Technical Writer	Несут ответственность за создание документации для заинтересованных сторон.	Deployment
Test Manager	Несут ответственность за успех тестирования, включая планирование, управление и поддержку действий по тестированию и обеспечению качества.	Test
Tester	Несут ответственность за написание, проведение и фиксирование результатов тестирования.	Test
Tool Specialist	Несут ответственность за выбор, приобретение, настройку и поддержку инструментов.	Environment

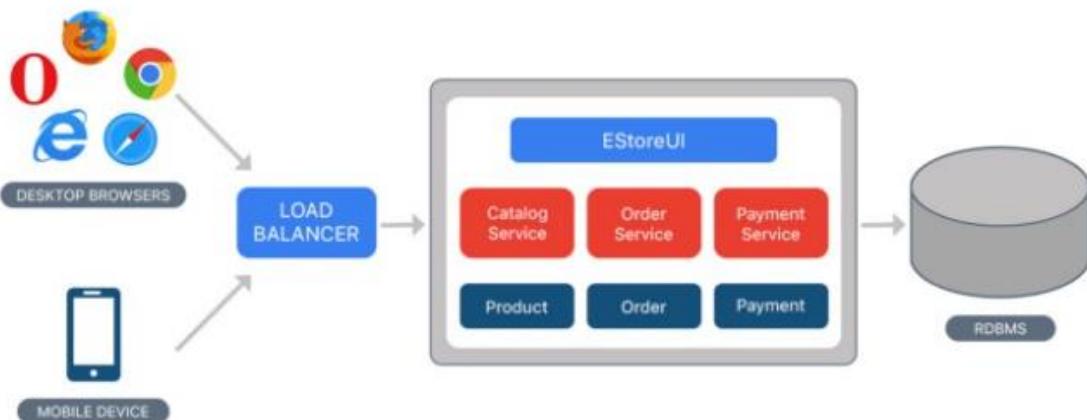
ВОПРОС №69

Архитектура монолитных приложений. Достоинства и недостатки.

В программной инженерии монолитная модель относится к единой неделимой единице. Концепция монолитного программного обеспечения заключается в том, что различные компоненты приложения объединяются в одну программу на одной платформе. Обычно монолитное приложение состоит из базы данных, клиентского пользовательского интерфейса и серверного приложения. Все части программного обеспечения унифицированы, и все его функции управляются в одном месте. Монолитная архитектура удобна для работы небольших групп, поэтому многие стартапы выбирают этот подход при создании приложения. Компоненты монолитного программного обеспечения взаимосвязаны и взаимозависимы, что помогает программному обеспечению быть самодостаточным. Эта архитектура является традиционным решением для создания приложений, но некоторые разработчики считают ее устаревшей.

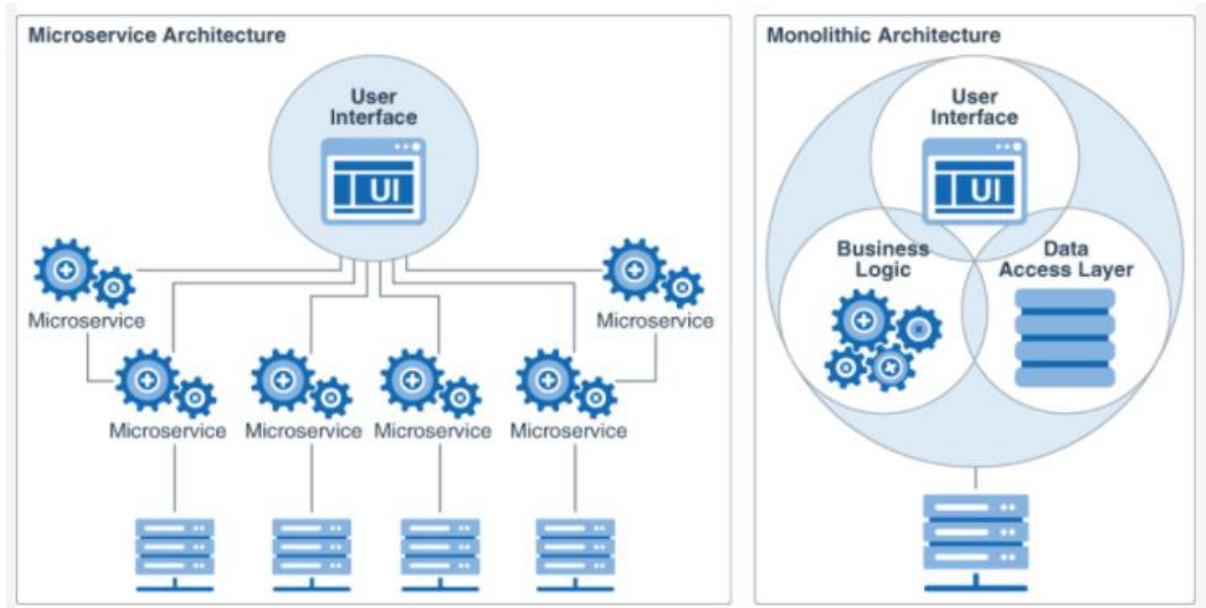
Пример монолитного подхода

Рассмотрим пример приложения электронной коммерции, которое авторизует клиента, принимает заказ, проверяет инвентарь продуктов, авторизует платеж и отправляет заказанные продукты. Это приложение состоит из нескольких компонентов, включая пользовательский интерфейс электронного магазина для клиентов (веб-представление магазина), а также некоторые серверные службы для проверки товарных запасов, авторизации и начисления платежей и заказов на доставку.



Monolithic Architecture (for E-Commerce Application)

Несмотря на наличие разных компонентов / модулей / сервисов, **приложение создается и развертывается как одно приложение для всех платформ** (например, для настольных ПК, мобильных устройств и планшетов) с использованием СУБД в качестве источника данных.



Различие микросервисной и монолитной архитектур.

Преимущества:

- Простота разработки - в начале проекта намного проще использовать монолитную архитектуру.
- Простота тестирования. Можно реализовать сквозное тестирование (end to end [E2E]), в монолитной архитектуре его легче выполнить.
- Простота развертывания. Нужно только скопировать упакованное приложение на сервер. Можно использовать скрипт, загружающий модуль и запускающий приложение.
- Простое масштабирование. Оно достигается путем размещения Loadbalancer перед несколькими экземплярами приложения.
- Лучшая производительность. При правильной сборке монолитные приложения обычно более производительны, чем приложения на основе микросервисов. Обеспечивают более быструю связь между программными компонентами благодаря общему коду и памяти.

Недостатки:

- Сопровождение - если приложение слишком велико и сложно для полного понимания, сложно быстро и правильно вносить изменения.
- Размер приложения может замедлить время запуска.
- При каждом обновлении необходимо повторно развертывать все приложение.
- Монолитные приложения также сложно масштабировать, когда разные модули имеют противоречивые требования к ресурсам.
- Надежность - ошибка в любом модуле (например, утечка памяти) может потенциально вывести из строя весь процесс. Более того, поскольку все экземпляры приложения идентичны, эта ошибка влияет на доступность всего приложения.
- Независимо от того, насколько легкими могут показаться начальные этапы, монолитные приложения испытывают трудности с внедрением новых и передовых технологий. Поскольку изменения в языках или фреймворках влияют на все приложение, требуются усилия для тщательной работы с деталями приложения, следовательно, это требует больших затрат времени и усилий.

Архитектура монолитного программного обеспечения может быть полезной, если ваша команда находится на начальной стадии, вы создаете непроверенный продукт и не имеете опыта работы с микросервисами. Монолит идеально подходит для стартапов, которым

необходимо как можно быстрее запустить продукт в эксплуатацию. Однако некоторые проблемы, упомянутые выше, идут рука об руку с монолитной архитектурой.

Ссылки:

1. <https://medium.com/koderlabs/introduction-to-monolithic-architecture-and-microservices-architecture-b211a5955c63>
2. <https://habr.com/ru/company/otus/blog/476024/>
3. <https://proplib.io/p/monolitnaya-vs-mikroservisnaya-arhitektura-2019-09-16>

ВОПРОС №70

Двухуровневая архитектура клиент-сервер. Достоинства и недостатки.

Клиент-серверная архитектура - это собирательное понятие, обозначающее класс систем, в которых работа с данными разделена между несколькими вычислительными машинами. Хранение и отображение данных происходит на разных машинах.

Клиенты - это машины на которых происходит представление и отображение данных. Они получают некую услугу или данные.

Серверы - это машины, на которых происходит хранение данных. Они предоставляют услугу или данные

По месту обработки данных системы клиент сервер разделяют на два типа:

- **тонкий клиент, толстый сервер** - обработка и хранение данных происходит на сервере, а клиент занимается исключительно представлением и отображением данных
- **толстый клиент, тонкий сервер** - сервер занимается только хранением данных. Клиент занимается обработкой, представлением и отображением данных.

Чаще всего при упоминании клиент-серверной архитектуры подразумевается вариант тонкий клиент, толстый сервер.

Для двухуровневой архитектуры характерно расположение базы данных и приложений, выполняющих бизнес-логику на одном сервере, а при использовании трехуровневой архитектуры выделяются отдельно сервер баз данных и сервер приложений.

Обычно система клиент-сервер устроена следующим образом:

На сервере развернута база данных, которая хранит все данные, имеющиеся в системе. Там же располагается приложение, реализующее бизнес-логику системы. Оно предоставляет REST API, с помощью которого клиенты обращаются к серверу и запрашивают необходимые данные или действия (например сохранение новых данных или обновление существующих).

Клиент представляет собой приложение, которое делает запросы к серверу через REST API и отображает данные, которые возвращает сервер, в человекочитаемом виде. Также клиент предоставляет пользователю интерфейс для управления данными.

Двухуровневая архитектура используется, когда в системе не очень много бизнес логики, например для сайта интернет-магазина. Если логики становится больше, то более разумно будет использовать трехуровневую архитектуру.



Преимущества:

- Систему просто развернуть и поддерживать
- Быстрое время ответа, поскольку данные и логика их обработки сосредоточены на одной машине
- Проще защищать данные, так как они сосредоточены на одной машине. Необходимо защищать доступ только к одной машине и, пока она не взломана, все данные в безопасности.
- Проще организовать сетевое взаимодействие и централизованный доступ к данным, чем в распределенной системе.
- Снижаются требования к клиентским машинам, поскольку вся обработка данных сосредоточена на сервере

Недостатки:

- Неработоспособность сервера выводит из строя всю вычислительную сеть.
- Поддержка работы данной системы требует системного администратора.
- Данные уязвимы, потому что они сосредоточены на одной машине. Если не предприняты достаточные меры по защите данных, то один удачный взлом защиты может привести к краже или повреждению всех данных.
- Система показывает меньшую производительность, чем трехуровневая архитектура. Более того, при резком росте числа клиентов производительность сильно падает.
- Меньшая масштабируемость по сравнению с трехуровневой архитектурой

Источники

- <https://www.geeksforgeeks.org/difference-between-two-tier-and-three-tier-database-architecture/>
- <https://web.archive.org/web/20110406121920/http://java.sun.com/developer/Books/jdbc/ch07.pdf>
- <https://itelon.ru/blog/arkhitektura-klient-server/>
- <https://medium.com/@fleviankanaiza/two-tier-three-tier-architecture-8b02536d3482>
- <https://www.collegenote.net/curriculum/web-technology-csit/84/468/>

ВОПРОС №71

Многоуровневая архитектура, особенности применения. (Software Architecture Pattern, гл. 1)

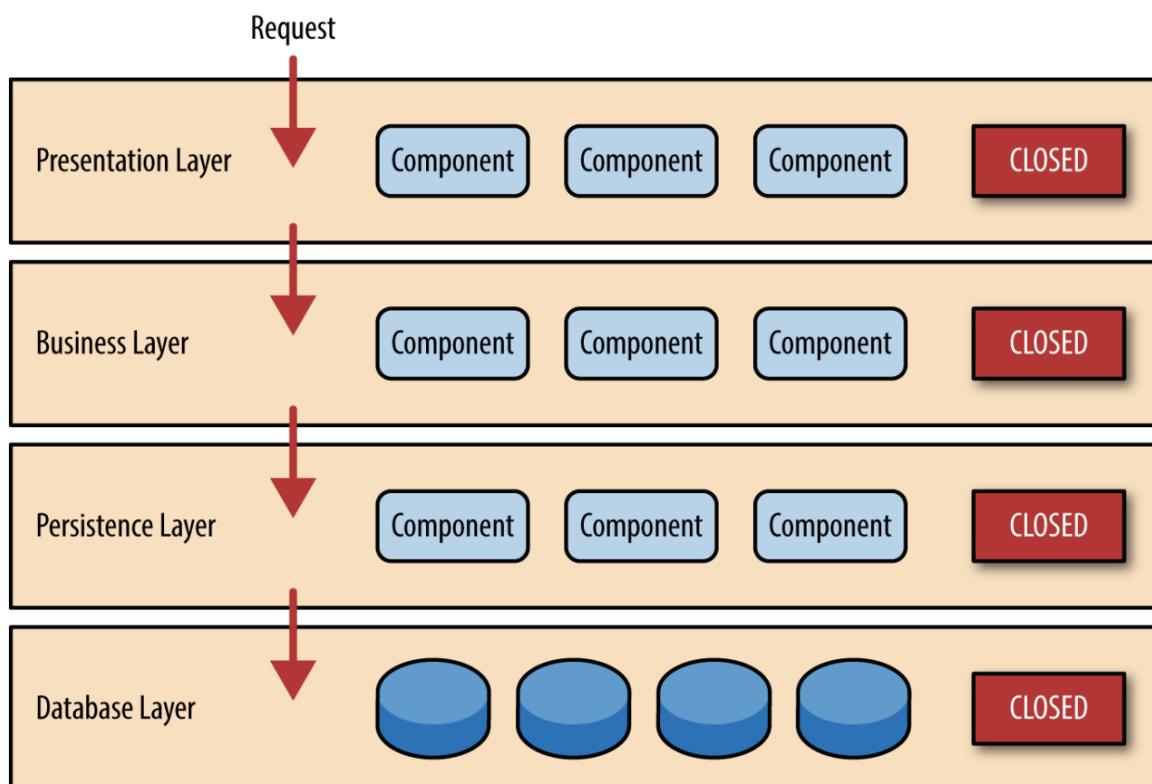
Компоненты в шаблоне многоуровневой архитектуры организованы в горизонтальные уровни, каждый из которых выполняет определенную роль в приложении (например, логику представления или бизнес-логику).

Четыре стандартных уровня архитектуры:

- уровень представления содержит пользовательский интерфейс и отвечает за обеспечение хорошего пользовательского опыта;
- бизнес-логики — как следует из названия, содержит бизнес-логику приложения. Он отделяет UI/UX от вычислений, связанных с бизнесом;
- персистентный уровень;
- уровень базы данных.

Основные идеи многоуровневой архитектуры

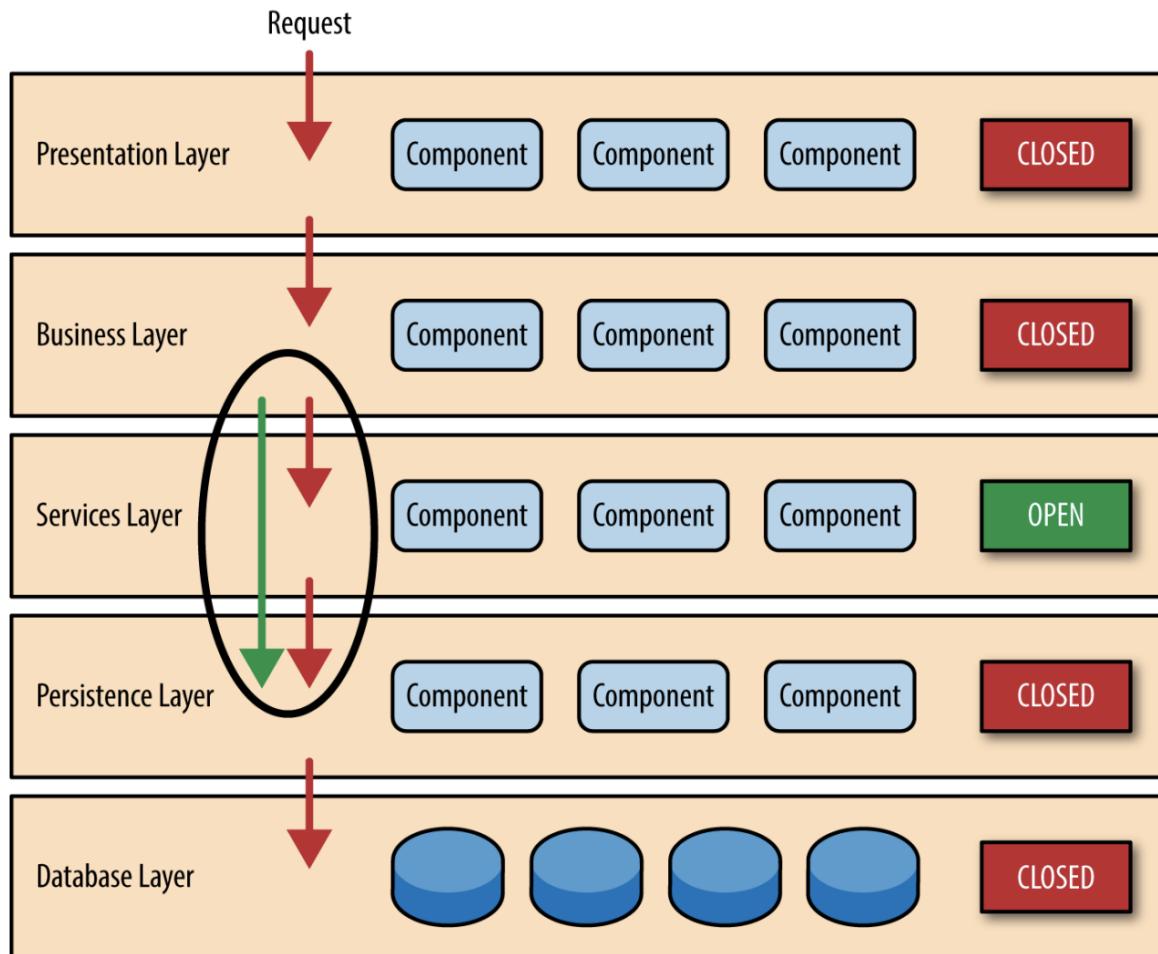
Каждый уровень в такой архитектуре должен быть “закрытым”. Это очень важная концепция в шаблоне многоуровневой архитектуры. Закрытый уровень означает, что по мере того, как запрос перемещается от уровня к уровню, он должен пройти через слой прямо под ним, чтобы перейти к следующему уровню ниже этого. Например, запрос, исходящий от уровня представления, должен сначала пройти через бизнес-уровень, а затем на уровень персистентности, прежде чем, наконец, попасть на уровень базы данных.



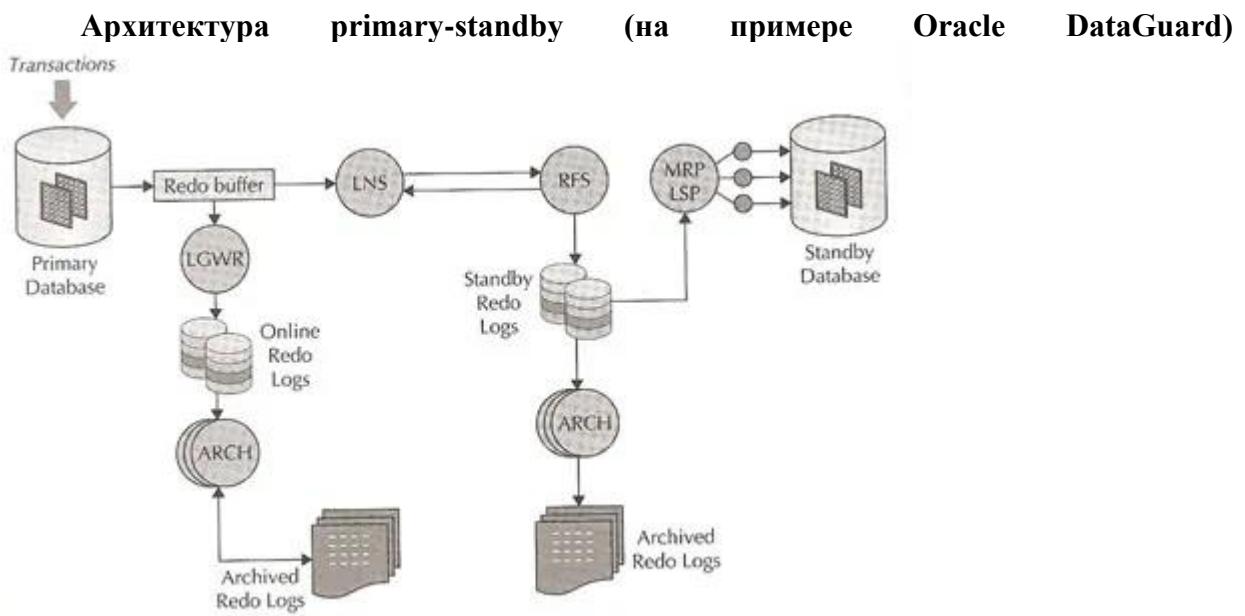
Следующей важной концепцией многоуровневой архитектуры является концепция изолированности каждого уровня. Эта концепция означает, что изменения, внесенные на одном уровне архитектуры не должны влиять на компоненты на других уровнях. Если разрешить уровню представления сразу же вносить изменения на уровень базы данных, то такие изменения также будут иметь влияние и на уровень логики и на уровень

персистентности. Таким образом будут созданы очень тесные связи между компонентами разных уровней. Такой тип архитектуры впоследствии становится очень сложным и дорогостоящим в поддержке и изменениях.

Иногда бывает полезно добавлять открытые уровни сервисов.



ВОПРОС №72



Standby база данных имеет следующие основные цели:

- Защита от стихийных бедствий
- Защита от повреждения данных
- Дополнительная отчетность

Экземпляр БД Oracle содержит следующие виды файлов:

- Управляющие файлы (Control files) — содержат служебную информацию о самой базе данных. Без них не могут быть открыты файлы данных и поэтому не может быть открыт доступ к информации базы данных.
- Файлы данных (Data files) — содержат информацию базы данных.
- Оперативные журналы (Redo logs) — содержат информацию о всех изменениях, произошедших в базе данных.

Эта информация может быть использована для восстановления состояния базы при сбоях.

Существуют другие файлы, которые формально не входят в базу данных, но важны для успешной работы БД.

- Файл параметров — используется для описания стартовой конфигурации экземпляра.
- Файл паролей — позволяет пользователям удаленно подсоединяться к базе данных для выполнения административных задач.
- Архивные журналы — содержат историю созданных экземпляром оперативных журнальных файлов (их автономные копии).

Эти файлы позволяют восстановить базу данных. Используя их и резервы базы данных, можно восстановить потерянный файл данных.

Главная идея при создании standby экземпляра состоит в том, чтобы с помощью выполнения транзакций, сохраненных в оперативных или архивных журналах основной БД, поддерживать резервную БД в актуальном состоянии (такой механизм для Oracle называется Data Guard).

Отсюда следует первое требование к основной базе — она должна быть запущена в archivelog mode.

Вторым требованием является наличие файла паролей. Это позволит удаленно подключаться к нашей БД в административном режиме.

Третье требование — это режим force logging. Этот режим нужен для принудительной записи транзакций в redo logs даже для операций, выполняемых с опцией NOLOGGING. Отсутствие этого режима может привести к тому, что на standby базе будут повреждены некоторые файлы данных, т.к. при «накатке» архивных журналов из них нельзя будет получить данные о транзакциях, выполненных с опцией NOLOGGING.

Также необходимо отметить, что если используется Oracle ниже 11g, то необходимо, чтобы сервера для основной базы и для standby имели одинаковую платформу. Т.е., если основная база работает на Linux-сервере, то standby-сервер не может быть под управлением Windows.

Архитектуру Data Guard можно резюмировать следующим образом. Dataguard перемещает записи повтора, созданные в основной базе данных, которая называется primary базой данных, в резервную базу данных (standby), которую называют резервной базой данных или аварийной базой данных, и эти данные повтора применяются к резервной базе данных и создают самую последнюю резервную копию основной базы данных.

Данные, которые мы называем данными повтора, представляют собой полный набор транзакций, генерируемых в первичной базе данных (вставка, обновление, удаление, создание, изменение и т. д.). Перемещение транзакций из основной базы данных в резервную означает применение тех же транзакций к резервной стороне. Data Guard также проверяет переносимые данные повтора перед их применением в резервной базе данных, а также на предмет проблем с повреждением блока.

Источники:

1. <https://habr.com/ru/post/120495/>
2. <https://ittutorial.org/oracle-dataguard-standby-architecture-1/>

ВОПРОС №73

Кластеры высокой готовности (на примере SolarisCluster или VeritasCluster)

Отказоустойчивый кластер (High-Availability cluster) - кластер (группа серверов), спроектированный в соответствии с методиками обеспечения высокой доступности и гарантирующий минимальное время простоя за счет аппаратной избыточности.

Отказоустойчивая кластеризация обеспечивает устойчивость к отказам на уровне системы с помощью процесса, называемого подхватом функций. Когда какая-нибудь система или узел в кластере выходит из строя или перестает отвечать на запросы клиентов, кластеризованные службы или приложения, которые функционировали на этом конкретном узле, переводятся в автономный режим и перемещаются на другой узел, где снова делаются функциональными и полностью доступными. В большинстве реализаций отказоустойчивые кластеры требуют доступа к общему хранилищу данных

Балансировка сетевой нагрузки (Network Load Balancing) – обеспечивает отказоустойчивость за счет того, что вынуждает каждый сервер в кластере индивидуально запускать сетевые службы и приложения и тем самым исключает вероятность появления одиночных точек отказа.

Кластер – это группа независимых серверов (узлов), позволяющая получать доступ к входящим в нее серверам и представлять их в сети так, будто бы они являются единой системой.

Узел – это отдельный сервер, который является членом кластера.

Кластерный ресурс – это служба, приложение, IP-адрес, диск или сетевое имя, которое определяется и управляется с помощью кластера. Внутри самого кластера кластерные ресурсы объединяются и управляются вместе за счет использования групп кластерных ресурсов, которые теперь называются группами ролей

Группа ролей. Кластерные ресурсы, содержащиеся внутри кластера в виде логического набора, теперь называются группами ролей, хотя часто употребляется и название «кластер группы служб и приложений». Эти группы представляют собой своего рода единицы подхвата функций в кластере. В случае выхода одного кластерного ресурса из строя и невозможности осуществить его автоматический перезапуск, группа ролей, частью которой является данный ресурс, будет переведена в автономный режим, перемещена на другой узел в кластере, а затем снова возвращена в оперативный режим.

Активный узел (active node) – узел в кластере, на котором в текущий момент функционирует хотя бы одна группа ролей. Группа ролей может быть активной только на одном узле в каждый момент времени и потому все остальные узлы, на которых она также может находиться, считаются по отношению конкретно к ней пассивными.

Пассивный узел (passive node) – узел, на котором в текущий момент не функционируют никакие группы ролей

Кворум кластера (cluster quorum) отвечает за отличительные конфигурационные данные кластера и текущее состояние каждого узла, каждой группы ролей, каждого ресурса и сети в кластере. Более того, при считывании данных кворума каждый узел определяет на основании извлекаемой информации то, должен ли он оставаться доступным, завершить работу кластера или активизировать конкретные группы ролей на локальном узле.

Требования к архитектуре приложения

- Должен быть относительно простой способ запуска, остановки, принудительной остановки, и проверки состояния приложения. На практике это означает, что приложение должно иметь интерфейс командной строки или скрипты для управления им, в том числе для работы с несколькими запущенными экземплярами приложения.
- Приложение должно уметь использовать общее хранилище данных (NAS / SAN).

- Очень важно, что приложение должно хранить в неразрушающем общем хранилище максимально возможное количество данных о своём текущем состоянии. Важна возможность приложения быть перезапущенным на другом узле в состоянии, предшествующем сбою, с использованием данных о состоянии из общего хранилища.
- Приложение не должно повреждать данные при падении или восстановлении из сохраненного состояния.

Схемы построения

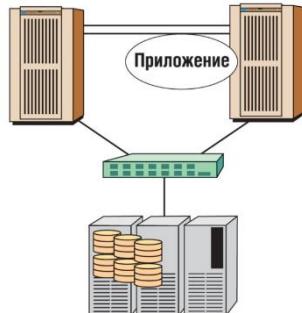


Рис.1 - Общая схема кластера высокой доступности

Чаще всего встречаются двухузловые НА-кластеры – это минимальная конфигурация, необходимая для обеспечения отказоустойчивости. Но часто кластеры содержат намного больше, иногда десятки узлов. Все эти конфигурации, как правило, могут быть описаны одной из следующих моделей:

Кластер типа «активный-активный» – это такой кластер, в котором на каждом узле активно обслуживается или функционирует хотя бы одна группа ролей. Такая конфигурация является типичной в случаях, когда в одном отказоустойчивом кластере развертывается множество групп ролей для обеспечения максимального использования серверных или системных ресурсов. Ее недостаток состоит в том, что в случае выхода одной активной системы из строя оставшейся системе или системам нужно обслуживать все группы и предоставлять доступ к службам ролей в кластере всем необходимым клиентам.

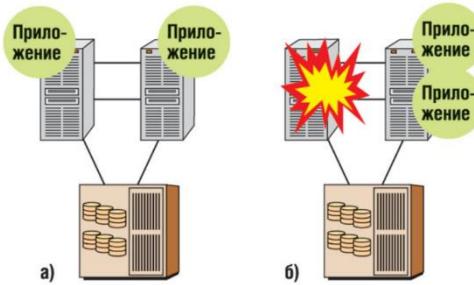


Рис.2 - Активный-активный, а - нормальный режим, б - аварийный режим

Кластер типа «активный-пассивный» – это такой кластер, в котором имеется хотя быть один узел с функционирующей группой ролей, и несколько дополнительных узлов, которые также могут обслуживать эту группу, но пока что находятся в состоянии ожидания. При отказе главного узла переходит на резервный узел, затем резервный узел становится главным узлом. Резервный узел полностью избыточен, что приводит к некоторой трате.

Конфигурация N+1 имеет один полноценный резервный узел, к которому в момент отказа переходит роль отказавшего узла. В случае гетерогенной программной конфигурации основных узлов дополнительный узел должен быть способен взять на себя роль любого из основных, за резервирование которых он отвечает. Такая схема применяется в кластерах, обслуживающих несколько разнородных сервисов, работающих одновременно; в случае единственного сервиса такая конфигурация вырождается в Active/Passive. N+1 эффективная по соотношению сложности и эффективности использования оборудования.

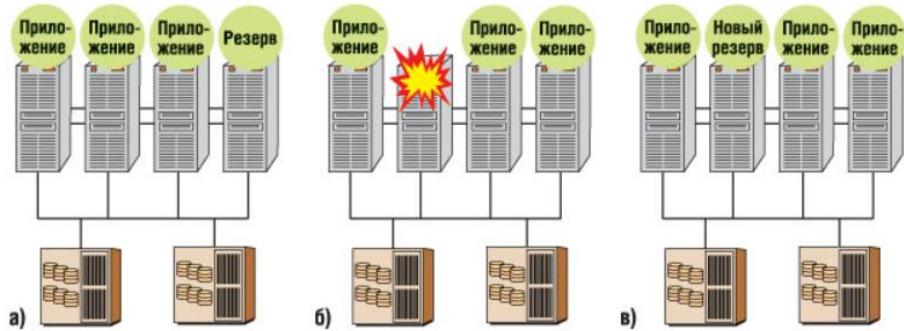


Рис 3. N+1 а - нормальный режим, б - аварийный режим, в - восстановленный режим
Разновидность N+1 - менее гибкая **конфигурация N к 1**, когда резервный узел всегда остается постоянным для всех рабочих узлов. В случае выхода из работы активного сервера сервис переключается на резервный, и система остается без резерва до тех пор, пока не будет активирован вышедший из строя узел.

Конфигурация N+M, N активных узлов и M резервных узлов. Если один кластер обслуживает несколько сервисов, включение в него единственного резервного узла (N+1) может оказаться недостаточным для надлежащего уровня резервирования. В таких случаях в кластер включается несколько резервных серверов, количество которых является компромиссом между ценой решения и требуемой надежностью. Все узлы кластера должны обладать некоторой избыточной мощностью сверх минимально необходимой. При проектировании такой системы необходимо учитывать совместимость приложений, их связи при «переезде» с узла на узел, загрузку серверов, пропускную способность сети и многое другое. Эта конфигурация наиболее сложна в проектировании и эксплуатации, но она обеспечивает максимальную отдачу от оборудования при использовании кластерного резервирования.

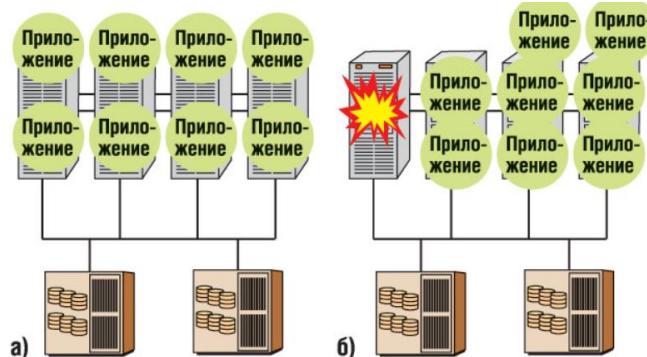


Рис.4 - N+M, а - нормальный режим, б - аварийный режим

Реализации кластеров

Oracle Solaris Cluster (ранее Sun Cluster или SunCluster) - кластерное программное обеспечение для операционной системы Solaris, разработанное корпорацией Sun Microsystems. Оно используется для увеличения доступности приложений (например, баз данных, коммерческих веб-сайтов). Sun Cluster дает возможность удаленным компьютерам и узлам работать вместе; при отказе одного из них другие продолжат предоставлять требуемый сервис. Узлы могут быть расположены в одном data-центре или на разных континентах. В качестве ядра SC выступает ОС Solaris с надстроенной оболочкой (**High-Availability Framework**), обеспечивающей функцию высокой доступности. Далее **глобальные компоненты**, которые предоставляют свои службы, полученные от кластерного ядра и, **пользовательские компоненты**.

HA framework - это компонент, расширяющий ядро Solaris для предоставления кластерных служб. Задача фреймворка начинается с инициализации кода, загружающего узел в

кластерный режим. Основные задачи - межузловое взаимодействие, управление состоянием кластера и членством в нем.

Модуль межузлового взаимодействия передает сообщения heartbeat между узлами (короткие сообщения, подтверждающие отклик соседнего узла). Взаимодействием данных и приложений также управляет HA framework как частью межузлового взаимодействия. Кроме того, фреймворк управляет целостностью кластерной конфигурации и при необходимости выполняет задачи восстановления и обновления. Целостность поддерживается через кворум-устройство; при необходимости выполняется реконфигурация. Назначать кворум-устройство можно вне кластерной системы, т. е. использовать дополнительный сервер на платформе Solaris, доступный по TCP/IP. Неисправные члены кластера выводятся из конфигурации. Элемент, который вновь оказывается работоспособен, автоматически включается в конфигурацию.

Функции глобальных компонентов вытекают из HA framework. Сюда относятся:

- глобальные устройства с общим пространством имен устройств кластера;
- глобальная файловая служба, организующая доступ к каждому файлу системы для каждого узла так, как будто он находится в своей локальной файловой системе;
- глобальная сетевая служба, предоставляющая балансировку нагрузки и возможность получать доступ к кластерным службам через единый IP.

Пользовательские компоненты управляют кластерной средой на верхнем уровне прикладного интерфейса. Есть возможность вести администрирование как через графический интерфейс, так и через командную строку. Модули, которые отслеживают работу приложений, запускают и останавливают их, называются агентами. Существует библиотека готовых агентов для стандартных приложений; с каждым релизом этот список пополняется.

Veritas Cluster Server

Межузловое взаимодействие в VCS основано на двух протоколах - LLT и GAB. Для поддержки целостности кластера VCS использует внутреннюю сеть.

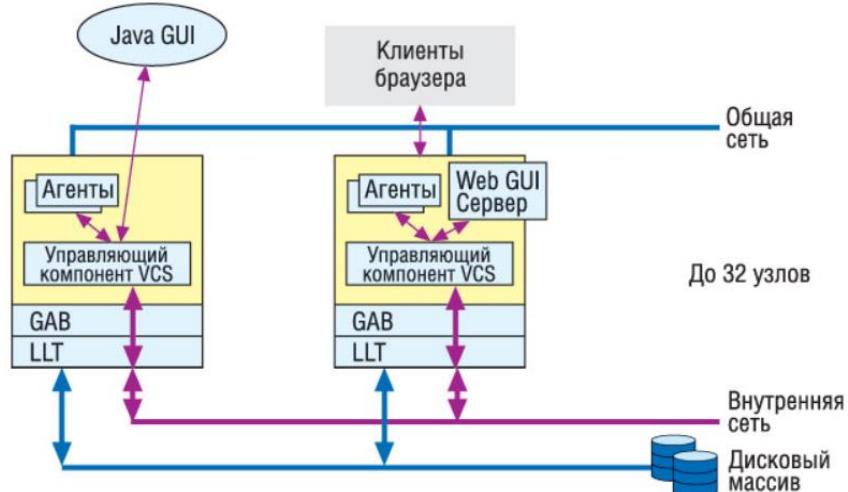


Рис. 5 Двухузловой Veritas Cluster Server

LLT (Low Latency Transport) - это разработанный Veritas протокол, функционирующий поверх Ethernet как высокоэффективная замена IP-стека и используемый узлами во всех внутренних взаимодействиях. Для требуемой избыточности в межузловых коммуникациях требуется как минимум две полностью независимые внутренние сети. Это необходимо, чтобы VSC мог различить сетевую и системную неисправность.

Протокол LLT выполняет две основные функции: распределение трафика и отправку heartbeat. LLT распределяет (балансирует) межузловое взаимодействие между всеми доступными внутренними связями. Такая схема гарантирует, что весь внутренний трафик случайно распределен между внутренними сетями (их может быть максимум восемь), что повышает производительность и устойчивость к отказу. В случае неисправности одного

линка данные будут перенаправлены на оставшиеся другие. Кроме того, LLT отвечает за отправку через сеть heartbeat-трафика, который используется GAB.

GAB (Group Membership Services/Atomic Broadcast) - это второй протокол, используемый в VCS для внутреннего взаимодействия. Он, как и LLT, ответственен за две задачи. Первая - это членство узлов в кластере. GAB получает через LLT heartbeat от каждого узла. Если система долго не получает отклика от узла, то она маркирует его состояние как DOWN - нерабочий.

Вторая функция GAB - обеспечение надежного межкластерного взаимодействия. GAB предоставляет гарантированную доставку бродкастов и сообщений «точка-точка» между всеми узлами.

Управляющая составляющая VCS - VCS engine, или **HAD** (High Availability daemon), работающая на каждой системе. Она отвечает за:

- построение рабочих конфигураций, получаемых из конфигурационных файлов;
- распределение информации между новыми узлами, присоединяемыми к кластеру;
- обработку ввода от администратора (оператора) кластера;
- выполнение штатных действий в случае сбоя.

HAD использует агенты для мониторинга и управления ресурсами. Информация о состоянии ресурсов собирается от агентов на локальных системах и передается всем членам кластера. HAD каждого узла получает информацию от других узлов, обновляя свою собственную картину всей системы. HAD действует как машина репликации состояния (replicated state machine RSM), т. е. ядро на каждом узле имеет полностью синхронизированную со всеми остальными узлами картину состояния ресурсов.

Кластер VSC управляет либо через Java-консоль, либо через Web.

SC написан Sun под собственную ОС и глубоко с ней интегрирован. VCS - продукт многоплатформенный, а следовательно, более гибкий.

Таблица 2. Возможности кластерных решений Sun Cluster Server (SC) и Veritas Cluster Server (VCS)

Параметр	VCS	SC
Двухузловой, симметричный/асимметричный	+	+
N к 1	+	+
N+1	+	-
N к N	+	-
Глобальный кластер	+	+
DR-тестирование	+	-
Интерфейс GUI/Web	+	+
Поддержка приложений	+	+
Гетерогенность	+	-
Сложность написания собственных агентов	Несложно	Несложно
Простота инсталляции	Очень просто	Очень просто
Поддержка единого управления несколькими кластерами	+	-
Кластерная файловая система	+	+

Рис 6. Возможности кластерных решений SC и VCS

Ссылки:

- <https://ru.wikipedia.org/wiki/%D0%9E%D1%82%D0%BA%D0%B0%D0%B7%D0%BE%D1%83%D1%81%D1%82%D0%BE%D0%B9%D1%87%D0%B8%D0%B2%D1%8B%D0%B9%D0%BA%D0%BB%D0%B0%D1%81%D1%82%D0%B5%D1%80>
https://ru.wikipedia.org/wiki/Solaris_Cluster
https://docs.oracle.com/cd/E37745_01/html/E37723/cacfifdd.html
https://en.wikipedia.org/wiki/Veritas_Cluster_Server
https://www.bytemag.ru/articles/detail_print.php?ID=6326&PRINT=Y
<https://drive.google.com/drive/folders/1JII2XEUdKVZGM9QRftM92YixLcG4r6Sn>

ВОПРОС №74

Высокопроизводительные кластеры (на примере Univa Grid Engine)

Кластер – это набор соединенных друг с другом компьютеров, используемых как общий вычислительный ресурс в едином административном пространстве.

Кластер состоит из двух или более серверов, соединяющих их сетей, общего дискового пространства и управляющего программного обеспечения.

Кластеры могут быть использованы в решениях, требующих масштабируемости большей, чем одна система: High Performance Computing (HPC, высокопроизводительные вычисления). Кластеры могут быть использованы для упрощения управления группой индивидуальных систем в решении по консолидации серверов (Server Consolidation). Кластеры могут быть использованы для повышения доступности приложений - High Availability (решение по обеспечению высокой доступности).

Грид-система – это "виртуальный суперкомпьютер", состоящий из кластеров и слабо связанных разнородных компьютеров, работающих вместе для выполнения огромного количества задач. С точки зрения сетевой организации, грид – это согласованная, открытая и стандартизированная среда, обеспечивающая гибкое, безопасное и скоординированное разделение вычислительных и запоминающих ресурсов, являющихся частью этой среды в рамках одной виртуальной организации. Отличительными особенностями являются грид-безопасность и географически распределенные узлы, которые могут быть расположены в любом месте и обычно подключены через Интернет. Например, компьютеры, расположенные в разных частях мира, могут работать вместе над одной и той же задачей. Существует несколько типов грид-систем: добровольная грид (основанная на использовании свободно предоставляемого бесплатного ресурса персональных компьютеров), научная грид и коммерческая грид. Многие считают, что главным отличием облачных вычислений от грид-технологий является виртуализация. В то время как грид-системы обеспечивают высокое использование вычислительных ресурсов путем распределения одной сложной задачи на несколько вычислительных узлов, облачные вычисления следуют пути выполнения нескольких задач на одном сервере в качестве виртуальных машин. Виртуальная среда позволяет разделить такие ресурсы, как центральный процессор (CPU), память, ввод-вывод и сеть из одной хост-системы на несколько сред.

Univa Grid Engine – это система управления распределенными ресурсами, которая оптимизирует ресурсы в центрах обработки данных путем прозрачного выбора ресурсов. Она предназначена для работы в грид системах

Особенности:

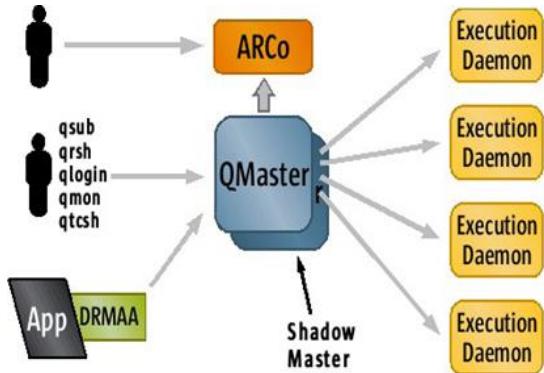
- Масштабируемость в облаках до 1М ядер
- Повышенная пропускная способность рабочей нагрузки
- Ускорение времени достижения результатов
- Облачное тестирование
- Снижение общей стоимости владения
- Включение крупномасштабного машинного обучения

Продукт Univa Grid Engine предназначен, прежде всего, для сетей среднего размера, охватывающих отдел или небольшое предприятие. Этот продукт предназначен для сетей класса Cluster Grid и доступен бесплатно.

Пакет позволяет объединить несколько серверов или рабочих станций в единый вычислительный ресурс, который может быть использован как для пакетных задач, так и для высокопроизводительных пакетных вычислений.

Администратор вычислительной сети может получать данные мониторинга и статистики, и на их основе оптимизировать уровень использования ресурсов. Административный интерфейс позволяет задавать различные параметры вычислительных задач, такие, как приоритеты, требуемые ресурсы оборудования, лицензии на программное обеспечение, временное окно выполнения, права пользователей на доступ к тем или иным ресурсам.

На диаграмме показаны компоненты Univa Grid Engine кластера.



В центре диаграммы – qmaster. Этот центральный компонент Univa Grid Engine управляет кластером, принимая поступающие задания от пользователей, назначая задания на ресурсы, контролируя текущий статус кластера, и обрабатывая команды управления. qmaster – многопоточный демон, который работает на одном хосте в вычислительном кластере. Чтобы уменьшить незапланированный простой кластера, один или более shadow masters могут выполнить задачи на дополнительных узлах в кластере. В случае неудачного завершения задачи qmaster'ом или хостом, задача передается новому qmaster, запуская нового qmaster демона. Каждый хост в кластере, который должен выполнить задания, должен будет запустить соответствующий демон. Демон получает задания от qmaster и выполняет их в определённом месте на своем хосте. Программное обеспечение Univa Grid Engine не устанавливает ограничений на число заданий, которые может распределить демон, но в большинстве случаев число заданий определено числом ядер центрального процессора доступных на хосте. Когда задание завершено, демон сообщает qmaster, что он может планировать новое задание.

В стандартном режиме каждый демон шлет сообщение о своем состоянии qmaster'у. Если qmaster неудачно завершает одну из задач, получая несколько последовательных сообщений от демона, то qmaster не зарегистрирует этого хоста и все его ресурсы как недоступные и удалит его из списка планировщика как недоступного. Задания посыпаются в qmaster разнообразными путями. DRMAA обеспечивает программный интерфейс для приложения, чтобы обеспечить запуск и контроль задания. Программное обеспечение Univa Grid Engine работает с C и Java, позволяя использовать DRMAA для широкого диапазона приложений. qmon – это графический пользовательский интерфейс Univa Grid Engine. Через qmon пользователи и администраторы могут запускать, контролировать и управлять заданиями, а также управлять всеми функциями кластера. qsub – это командная строка утилита для того, чтобы запускать очереди, пакеты и параллельные задания. Последний компонент, показанный на диаграмме – это ARCo (Accounting and Reporting Console), интернет приложение, обеспечивающее доступ к Univa Grid Engine для учёта информации, хранившейся в базе данных. Используя ARCo, конечные пользователи и администраторы могут создавать и выполнять запросы по учёту работы кластера.

ВОПРОС №75

Обмен сообщениями. Очереди сообщений. (на примере ApacheMQ и MQTT)

ActiveMQ

ActiveMQ лучше всего описать, как классическую систему обмена сообщениями. Она была написана в 2004 году, восполняя потребность в брокере сообщений с открытым исходным кодом. ActiveMQ была разработана как реализация спецификации Java Message Service (JMS). Система обмена сообщениями (или промежуточное ПО, ориентированное на сообщения, как ее иногда называют), реализующая спецификацию JMS, состоит из следующих компонентов:

- **Брокер**

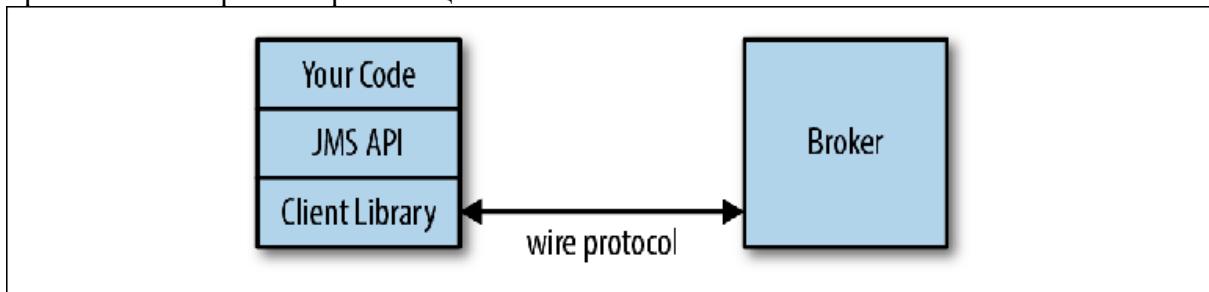
Центральная часть промежуточного программного обеспечения, распределяющая сообщения.

- **Клиент**

Часть программного обеспечения, которая перебрасывает сообщения с помощью брокера. Она, в свою очередь, состоит из следующих артефактов:

- Код, использующий API JMS.
- JMS API — это набор интерфейсов для взаимодействия с брокером в соответствии с гарантиями, изложенными в спецификации JMS.
- Клиентская библиотека системы, которая обеспечивает реализацию API и взаимодействует с брокером.

Клиент и брокер общаются друг с другом через протокол прикладного уровня, также известный, как *протокол взаимодействия*. Спецификация JMS оставила детали этого протокола конкретным реализациям.



JMS использует термин *провайдер* для описания реализации вендором системы обмена сообщениями, лежащей в основе API JMS, которая включает брокер, а также ее клиентские библиотеки.

В спецификации изложены четкие указания по обязанностям клиента системы обмена сообщениями и брокера, с которым он общается, отдавая предпочтение обязательству брокера распределять и доставлять сообщения. Основная обязанность клиента — взаимодействовать с адресатом (очередью или топиком) отправляемых им сообщений. Сама спецификация направлена на то, чтобы сделать взаимодействие API с брокером относительно простым.

MQTT

Упрощенный сетевой протокол, работающий поверх TCP/IP, ориентированный на обмен сообщениями между устройствами по принципу издатель-подписчик. Может использоваться в качестве протокола связи между клиентом и брокером в ActiveMQ.

Как работают очереди

Модель работы ActiveMQ:

одна часть отвечает за прием сообщений от продюсера, а другая отправляет эти сообщения потребителям. На самом деле, отношения являются более сложными для целей оптимизации производительности, но модель достаточна для базового понимания.

Отправка сообщений в очередь

Давайте рассмотрим взаимодействие, которое происходит при отправке сообщения. Figure 2-3 показывает нам упрощенную модель процесса, с помощью которого сообщения принимаются брокером. Он не полностью соответствует поведению в каждом случае, но вполне подходит, чтобы получить базовое понимание.

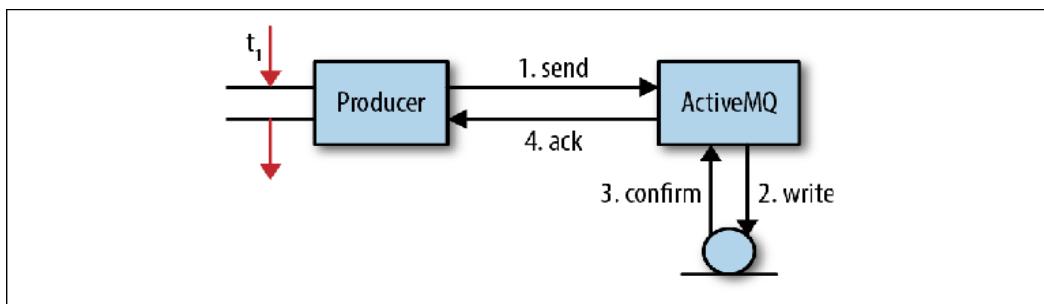


Figure 2-3. Отправка сообщений в JMS

В клиентском приложении поток получает указатель на MessageProducer. Он создает Message с предполагаемым пейлоадом сообщения и вызывает MessageProducer.send («orders», message), при этом конечным адресатом сообщения является очередь. Поскольку программист не хочет потерять сообщение, если бы сломался брокер, то заголовок сообщения JMSDeliveryMode был установлен в значение PERSISTENT (поведение по умолчанию).

На этом этапе (1) отправляющий поток вызывает клиентскую библиотеку и маршализирует сообщение в формат OpenWire. Затем сообщение отправляется брокеру.

В брокере, принимающий поток снимает сообщение с линии и анмаршаллит его во внутренний объект. Затем объект-сообщение передается персистенс-адаптеру, который маршализирует сообщение, используя формат Google Protocol Buffers, и записывает его в хранилище (2).

После записи сообщения в хранилище персистенс-адаптер должен получить подтверждение того, что сообщение действительно было записано (3). Это, как правило, самая медленная часть всего взаимодействия.

Как только брокер удостоверится, что сообщение сохранено, он отправит клиенту ответ-подтверждение (4). После чего, поток клиента, первоначально вызвавший операцию send (), может продолжить свою работу.

Это ожидание подтверждения персистентных сообщений является базой гарантии, предоставляемой JMS API — если вы хотите, чтобы сообщение было сохранено, для вас, вероятно, также важно, было ли сообщение принято брокером в первую очередь. Существует ряд причин, по которым это может оказаться невозможным, например, достигнут предел памяти или диска. Вместо сбоя, брокер либо приостанавливает операцию отправки, заставляя продюсера ждать, пока не появятся достаточно системных ресурсов для обработки сообщения (процесс называется Producer Flow Control), либо он отправит негативное подтверждение продюсеру, бросая исключение. Точное поведение настраивается для каждого брокера.

В этой простой операции происходит значительное количество взаимодействий ввода-вывода: две сетевые операции между продюсером и брокером, одна операция

сохранения и шаг подтверждения. Операция сохранения может быть простой записью на диск или еще одним сетевым переходом на сервер хранения.

Вычитка сообщений из очереди

Процесс считывания сообщений начинается, когда потребитель выражает готовность их принять либо настраивая MessageListener для обработки сообщений по мере их поступления, либо вызывая метод MessageConsumer.receive () (Figure 2-5).

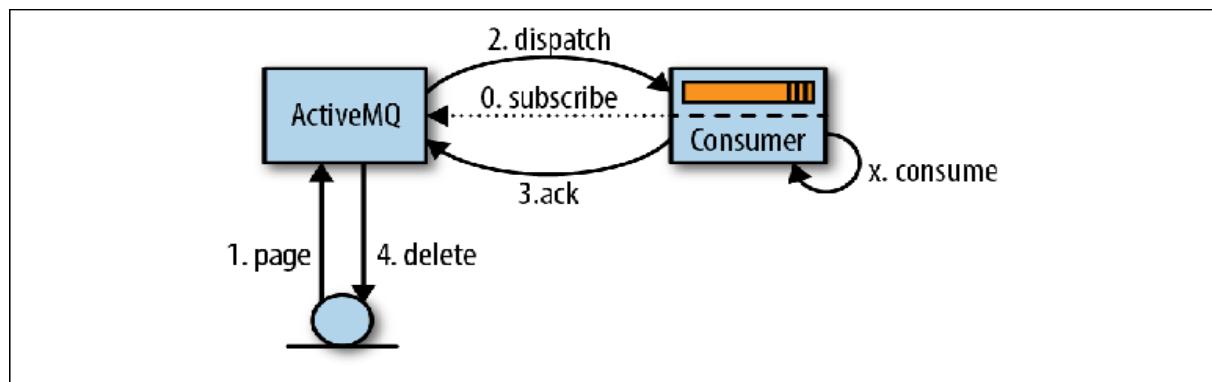


Figure 2-5. Вычитывание сообщений посредством JMS

Когда ActiveMQ становится известно о консюмере, он (ActiveMQ) постранично читает (pages) сообщения из хранилища в память для распространения (1). Затем эти сообщения перенаправляются (dispatched) консюмеру (2), часто несколькими частями для снижения объема сетевого взаимодействия. Брокер отслеживает, какие сообщения были перенаправлены и какому консюмеру.

Сообщения, полученные консюмером, не обрабатываются сразу приложением, а помещаются в область памяти, известную как *буфер предварительной выборки (prefetch buffer)*. Цель этого буфера состоит в том, чтобы выровнять поток сообщений, чтобы брокер мог выдавать сообщения консюмеру по мере того, как они становятся доступными для отправки, в то время, как консюмер мог получать их упорядоченно, по одному.

В какой-то момент после попадания в буфер предварительной выборки, сообщения вычитываются логикой приложения (X) и брокеру отправляется подтверждение о вычитке (3).

Как только брокер принимает подтверждение доставки сообщения, оно удаляется из памяти и из хранилища сообщений (4). Термин «удаление» в некоторой степени вводит в заблуждение, так как в действительности в журнал записывается запись о подтверждении и увеличивается указатель в индексе. Фактическое удаление файла журнала, содержащего сообщение, будет выполнено Сборщиком мусора (Garbage collector) в фоновом потоке на основе этой информации.

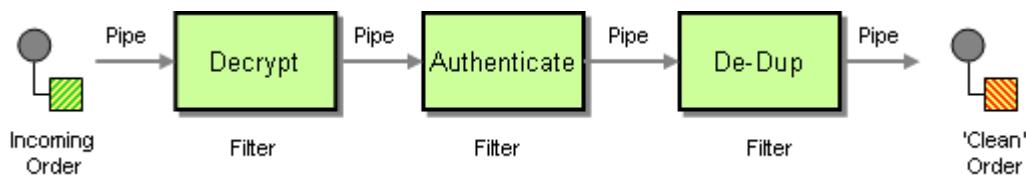
Описанное выше поведение является упрощением для облегчения понимания. В действительности, ActiveMQ не просто постранично читает (page) данные с диска, а вместо этого использует механизм курсора между принимающей и перенаправляющей частями брокера для минимизации взаимодействия с хранилищем брокера везде, где это возможно. Постстраничное чтение, как описано выше, является одним из режимов, используемым в этом механизме. Курсоры можно рассматривать, как кэш уровня приложения, который необходимо поддерживать в синхронизированном состоянии с хранилищем брокера.

Источники:
<https://3-info.ru/post/3789>

ВОПРОС №76

Фильтры и конвейерная архитектура (на примере GStreamer)

Конвейерная архитектура представляет из себя надежную и достаточно мощную архитектуру. Она состоит из множества фильтров, которые фильтруют или обрабатывают информацию, прежде чем передать другим компонентам.



Составляющие

Фильтр — преобразует или фильтрует данные, которые он получает через каналы, с которыми он связан. Фильтр может иметь любое количество входных каналов и любое количество выходных каналов.

Канал — это соединитель, который передает данные от одного фильтра к другому. Это направленный поток данных, который обычно реализуется буфером данных для хранения всех данных до тех пор, пока следующий фильтр не успеет их обработать.

Продюсер — является источником данных. Это может быть статический текстовый файл или устройство ввода с клавиатуры, постоянно создающее новые данные.

Приемник или потребитель является целью данных. Это может быть другой файл, база данных или экран компьютера.

Ключевое преимущество конвейерной структуры заключается в том, что в ней можно параллельно выполнять экземпляры медленных фильтров, позволяя распределить нагрузку и повысить пропускную способность в системе.

Пример с GStreamer

GStreamer — фреймворк для построения мультимедийных приложений, который позволяет создавать приложения различной сложности и реализует данную конвейерную архитектуру.

Архитектура GStreamer

В GStreamer есть несколько основных компонентов:

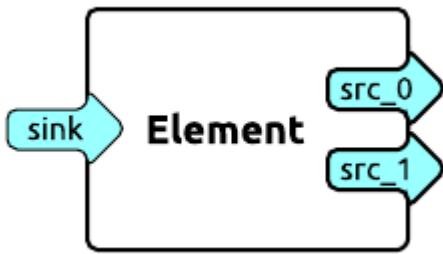
- Элементы
- Pads
- Контейнеры bin и pipeline
- Bus

Элементы

Element

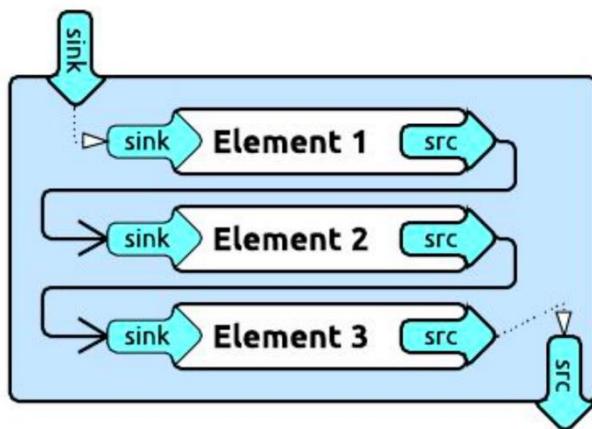
Практически все в GStreamer является элементом. Все, начиная от обычных источников потоков (filesrc, alsasrc, и т. п.), обработчиков потоков (демультиплексоры, декодеры, фильтры, и т. п.) и заканчивая конечными устройствами вывода (alsasink, fakesink, filesink, и т. п.).

Pads



Pad — это некая точка подключения элемента к другому элементу, если более просто — это входы и выходы элемента. Обычно они именуются «**sink**» — вход и «**src**» — выход. Элементы всегда имеют как минимум один pad. Элементы из разряда «**filters**» (те, которые как-то трансформируют поток) имеют две и более точек подключения. Например, элемент **volume** имеет pad с именем «**sink**», на который поступает поток, внутри этого элемента трансформируется (изменяется громкость), и через pad с названием «**src**» уже продолжает свой путь.

Контейнеры



Внутри контейнеров элементы проводят свой жизненный цикл. Контейнер управляет рассылкой сообщений от элемента к элементу, управляет статусами элементов.

Контейнеры делятся на два вида:

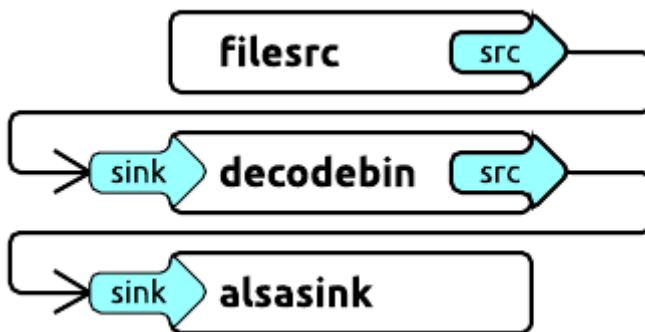
- **Bin**
- **Pipeline**

Bin - простой контейнер, который управляет рассылкой сообщений от элемента к элементу, которые находятся внутри него. Bin обычно используется для создания группы элементов, которые должны совершать какое-либо действие.

Pipeline является реализацией **Bin**. **Pipeline** является контейнером верхнего уровня, он управляет синхронизацией элементов, рассыпает статусы. Например, если pipeline установить статус PAUSED, этот статус будет автоматически разослан всем элементам которые находятся внутри него.

Bus: Интерфейс сообщений, который позволяет асинхронно взаимодействовать с активным Pipeline.

Примерная схема примитивного плеера:



Источник потоков (filesrc), обработчик потоков (декодер), конечное устройство вывода.

Источник 1: <https://stefaniuk.website/all/pipeline-architecture/>

Источник 2: <https://habr.com/ru/post/178813/>

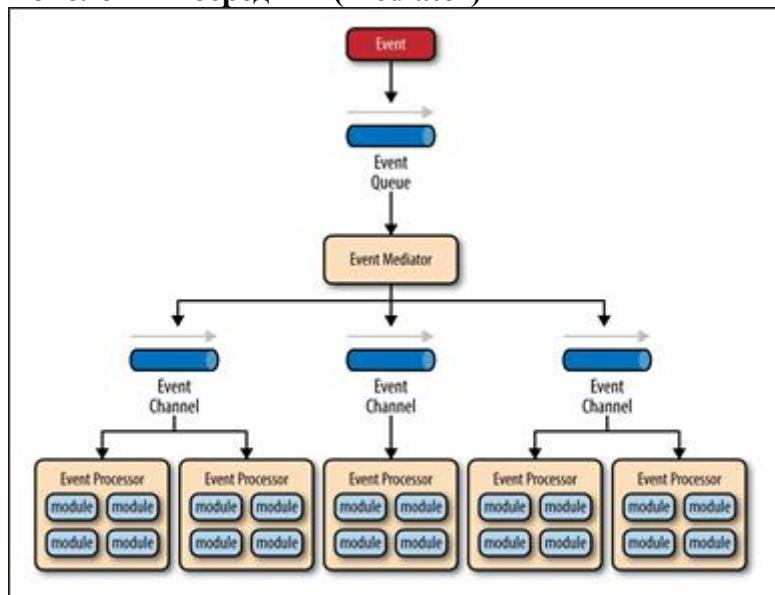
ВОПРОС №77

Архитектура, основанная на событиях (Software Architecture Patern, гл. 2)

Паттерн событийно-ориентированной архитектуры (Event-Driven Architecture) - это популярный шаблон распределенной асинхронной архитектуры, используемый для создания высоко масштабируемых приложений. Архитектура, управляемая событиями, состоит из сильно связанных однотиповых компонентов обработки событий, которые асинхронно получают и обрабатывают события.

Паттерн управляемой событиями архитектуры состоит из двух основных топологий, посредника (Mediator) и брокера (Broker). Топология посредника обычно используется, когда вам нужно организовать несколько шагов в рамках события через центрального посредника, тогда как топология брокера используется, когда вы хотите объединить события без использования посредника. Поскольку характеристики архитектуры и стратегии реализации в этих двух топологиях различаются, важно понимать каждую из них, чтобы знать, какая из них лучше всего подходит для вашей конкретной ситуации.

Топология посредник (Mediator)

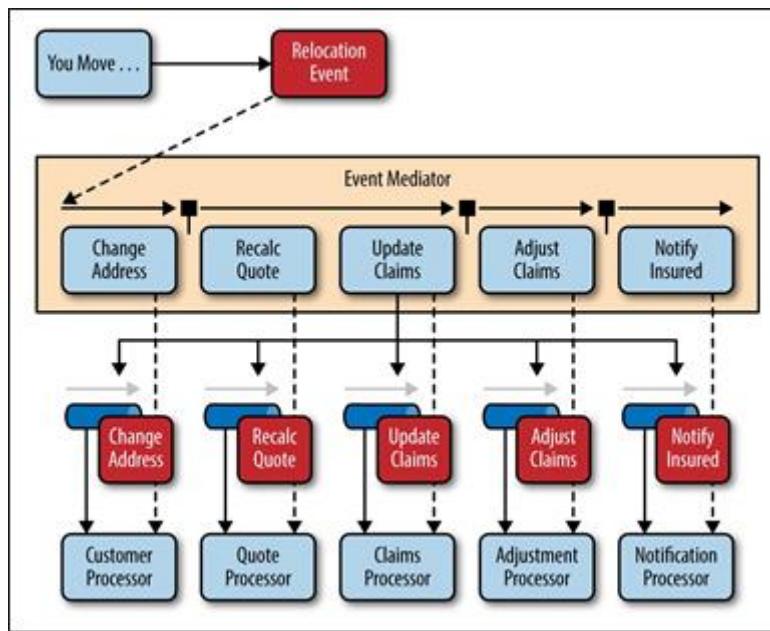


В топологии посредника есть четыре основных типа компонентов архитектуры: очередь событий (event queues), посредник событий (event mediator), каналы событий (event channels) и обработчики событий (event processors). Поток событий начинается с того, что клиент отправляет событие в очередь событий, которая используется для передачи события посреднику событий. Посредник событий получает начальное событие и управляет этим событием, отправляя дополнительные асинхронные события в каналы событий для выполнения каждого шага процесса. Обработчики событий, которые прослушивают каналы событий, получают событие от посредника событий и выполняют определенную бизнес-логику для обработки события.

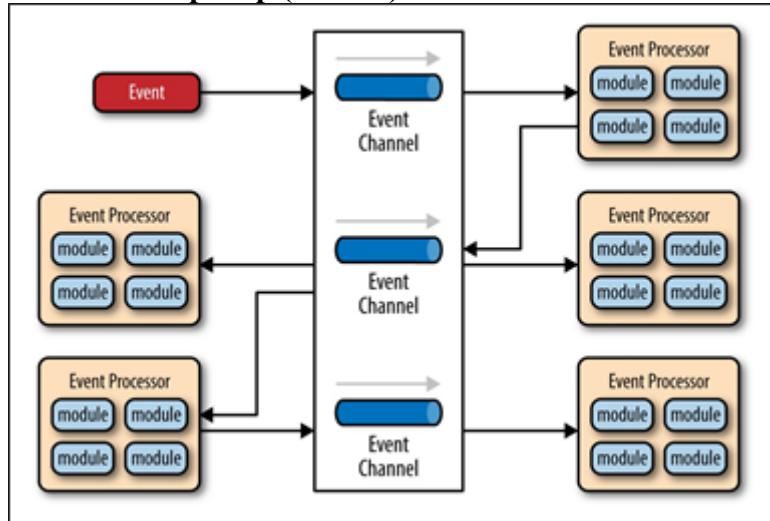
Другими словами и кратко: сообщения из очереди сообщений переходят к посреднику, который разделяет шаги сообщения по каналам в зависимости от операции, которой к этим шагам необходимо применить, и обработчики событий их выполняют.

Иллюстрация события переезда при наличии страховки.

(Красным цветом обозначаются события)



Топология брокер (Broker)

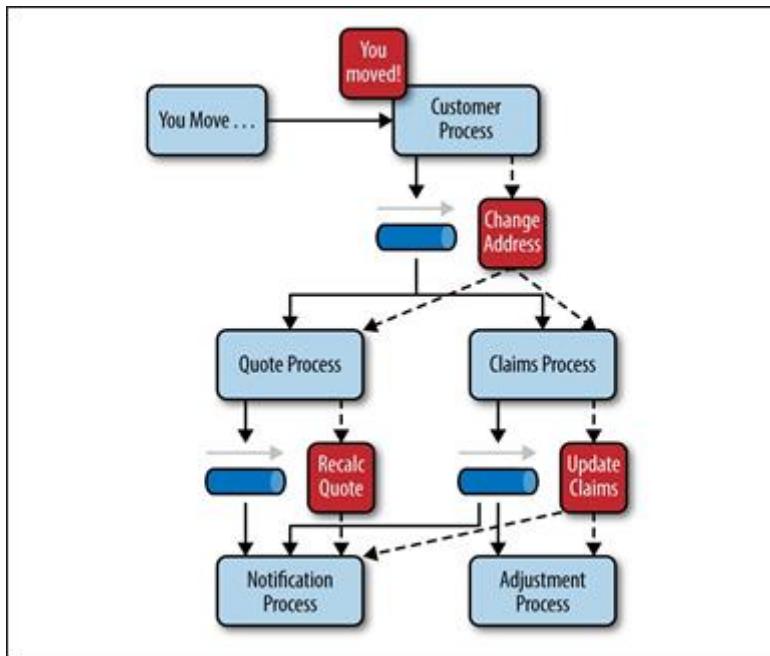


Топология брокера отличается от топологии посредника тем, что в ней отсутствует центральный посредник событий; скорее, поток сообщений распределяется по компонентам обработчика событий цепочкой через брокер сообщений. Эта топология полезна, когда у вас относительно простой поток обработки событий и вы не хотите (или не нуждаетесь) в централизации событий.

В топологии брокера есть два основных типа компонентов архитектуры: компонент брокера и компонент обработчика событий. Компонент брокера может быть централизованным или объединенным и содержать все каналы событий, которые используются в потоке событий.

Каждый компонент обработчика событий отвечает за обработку события и публикацию нового события, указывающего на только что выполненное действие.

Иллюстрация события переезда при наличии страховки.



Рекомендации

Шаблон управляемой событиями архитектуры является относительно сложным, в первую очередь из-за его асинхронно-распределенной природы. При реализации этого шаблона вы должны решить различные проблемы распределенной архитектуры, такие как доступность удаленных процессов, отсутствие реакции и логика повторного подключения брокера в случае отказа брокера или посредника.

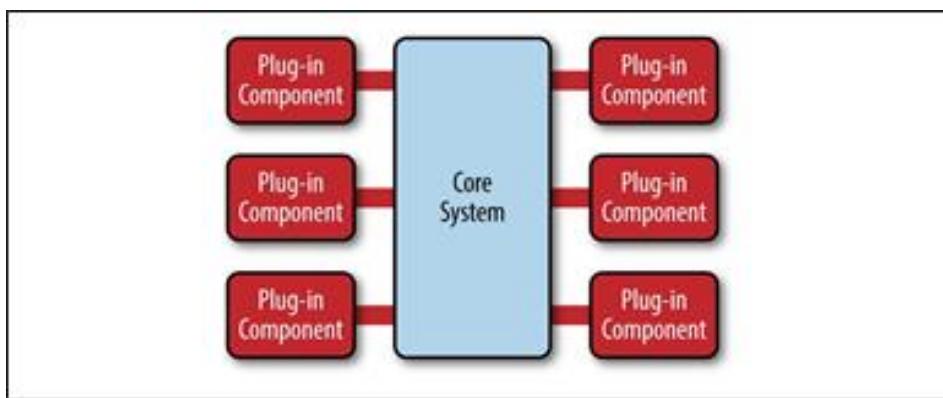
Одна из рекомендаций, которое следует учитывать при выборе этого шаблона архитектуры, это отсутствие атомарных транзакций для одного бизнес-процесса. Поскольку компоненты обработчика событий сильно разделены и распределены, очень сложно поддерживать транзакционную единицу работы между ними. По этой причине при разработке приложения с использованием этого шаблона вы должны постоянно думать о том, какие события могут и не могут выполняться независимо, и соответственно планировать степень детализации ваших обработчиков событий.

Возможно, одним из самых сложных аспектов шаблона управляемой событиями архитектуры является создание, обслуживание и управление контрактами компонента обработчика событий. С каждым событием обычно связан определенный контракт (например, значения данных и формат данных передающихся в обработчик событий). При использовании этого шаблона жизненно важно выбрать стандартный формат данных (например, XML, JSON, Java Object и т. д.) И с самого начала установить политику управления версиями контракта.

ВОПРОС №78

Архитектура микроядра (Software Architecture Pattern, гл. 3)

Паттерн архитектуры микроядра (иногда называемый шаблоном архитектуры плагина) является естественным шаблоном для реализации приложений на основе продукта. Приложение на основе продукта (product-based application) - это приложение, которое упаковано и доступно для загрузки в версиях, как типичный продукт стороннего производителя. Шаблон архитектуры микроядра позволяет добавлять дополнительные функции приложения в виде подключаемых модулей к основному приложению, обеспечивая расширяемость, а также разделение и изоляцию функций.



Шаблон архитектуры микроядра состоит из двух типов компонентов архитектуры: **основная система (core system)** и **подключаемых модулей (plug-in module)**. Логика приложения разделена между независимыми подключаемыми модулями и основной системой.

Основная система традиционно содержит только минимальную функциональность, необходимую для обеспечения работоспособности системы. Многие операционные системы реализуют шаблон архитектуры микроядра, отсюда и название этого шаблона.

Подключаемые модули представляют собой автономные независимые компоненты, которые содержат специализированную обработку, дополнительные функции и настраиваемый код, предназначенный для улучшения или расширения основной системы. Подключаемые модули должны быть независимыми от других подключаемых модулей, но это не является ограничением. В любом случае важно свести к минимуму обмен данными между подключаемыми модулями, чтобы избежать проблем с зависимостями.

Основная система должна знать, какие подключаемые модули доступны и как к ним добраться. Один из распространенных способов реализации этого - создание реестра подключаемых модулей. Подключаемые модули могут быть подключены к базовой системе различными способами, включая обмен сообщениями, веб-сервисы или даже прямое двухточечное связывание (т. е. создание экземпляра объекта). Шаблон не определяет деталей реализации, поэтому тип соединения зависит от типа приложения и потребностей к нему.

Примером данного шаблона могут служить IDE или браузеры, к которым можно установить дополнения для увеличения их функциональности.

Рекомендации

В шаблоне архитектуры микроядра замечательно то, что он может быть встроен или использоваться как часть другого шаблона архитектуры. Например, если с помощью этого шаблона реализована определенная часть архитектуры, но реализовать всю архитектуру только на этом не выйдет, то его можно будет встроить в другой вами используемый шаблон (например, многоуровневую архитектуру).

Шаблон архитектуры микроядра обеспечивает отличную поддержку эволюционного проектирования и инкрементальной разработки. Сначала вы можете создать надежную базовую систему, а по мере постепенного развития приложения добавлять функции и возможности без внесения значительных изменений в основную систему.

Для приложений на основе продукта шаблон архитектуры микроядра всегда должен быть первым выбором в качестве начальной архитектуры, особенно для тех продуктов, где вы будете выпускать дополнительные функции с течением времени и хотите контролировать, какие пользователи получают какие функции.

ВОПРОС №79

Архитектура микросервисов

Основные отличительные особенности архитектуры:

- **Раздельно развертываемые модули** (каждый из компонентов архитектуры представляет собой отдельную сущность и развертывается сам по себе, что упрощает процесс развертывания и повышает масштабируемость).
- **Понятие сервисного компонента** (вся архитектура представляется не как набор сервисов, а как набор сервисных компонентов. Компоненты состоят из произвольного количества модулей и могут различаться по размеру от одного модуля до большей части приложения. Каждый из модулей внутри реализует одну конкретную задачу или является представлением независимой части приложения. Правильное определение компонентов, модулей и их размеров является основной и наиболее сложной задачей при проектировании микросервисной архитектуры).
- **Распределенность** (все компоненты архитектуры полностью отделены друг от друга и связываются исключительно посредством каких-либо протоколов удаленного доступа (JMS, REST, SOAP и т.д.). Данная особенность позволяет повысить масштабируемость и упростить развертывание).

Дополнительные сведения:

- Возникла в качестве ответа на недостаточную масштабируемость и сложность дополнения монолитной архитектуры и чрезмерную сложность и дороговизну сервисно-ориентированной архитектуры.
- **Главная сложность – верно определить нужный размер компонентов** (если они будут слишком маленькими, то архитектура станет слишком сложной и приблизится к сервис-ориентированной, если они будут слишком большими, то архитектура потеряет гибкость с масштабируемостью и приблизится к монолитной).
- **Сервисы максимально отделены друг от друга**, поэтому обмен информацией между сервисами как правило реализуется с помощью общей БД, общая функциональность часто копируется из сервиса в сервис (хотя это и считается в других случаях плохой практикой), чтобы избежать обращений.
- **Централизация управления исключена**, если ее не выходит избежать, то это означает, что архитектура не подходит.

Преимущества: гибкость, масштабируемость, легко разрабатывать, тестировать и разворачивать.

Недостатки: невысокая производительность.

Вариантов реализации архитектуры множество, но наиболее популярны следующие три:

- **API REST** – обычно используется для небольших сайтов, предоставляющих простые услуги через какие-либо API. Компоненты очень маленькие, содержат по 1-2 модуля, реализующих конкретные функции, независимые от остальных. Доступ к компонентам производится через отдельно развернутый REST-интерфейс, реализованный в виде слоя API.
- **Application REST** – обычно используется для небольших и средних по размеру бизнес-приложений с простой архитектурой. Компоненты больше по размеру,

представляют собой отдельные логические части приложения. Доступ к компонентам производится через отдельно развернутое веб-приложение, контактирующее с ними через простые REST-интерфейсы.

- **Centralized Messaging** – по размерам и сути компонентов аналогична Application REST, однако вместо веб-приложения и REST-интерфейсов используется централизованный брокер сообщений (ActiveMQ, HornetQ и т.д.), который, однако, в отличие от SOA-архитектур, не берет на себя функции управления и изменения. Обычно используется в больших по размеру бизнес-приложениях, т.к. предоставляет более сложные функции (асинхронный обмен сообщениями, наблюдение, обработку ошибок и т.д.).

ВОПРОС №80

Архитектура облака (Software Architecture Patern, гл. 5)

Space-Based Architecture шаблон (шаблон облачной архитектуры) получил свое название от концепции пространства кортежей (реализация парадигмы ассоциативной памяти для параллельных/распределенных вычислений). Высокая масштабируемость достигается за счет устранения ограничений центральной базы данных и использования вместо этого реплицированных сеток данных в памяти. Данные приложения хранятся в памяти и реплицируются между всеми активными процессорами. Процессоры могут динамически запускаться и выключаться по мере увеличения и уменьшения нагрузки пользователя, тем самым решая проблему переменной масштабируемости. Поскольку центральной базы данных нет, узкое место базы данных устраниется, обеспечивая почти бесконечную масштабируемость в приложении.

В этом шаблоне архитектуры есть два основных компонента: процессор и виртуализированное промежуточное ПО. На рис 1. показан базовый шаблон архитектуры на основе пространства и его основные компоненты архитектуры.

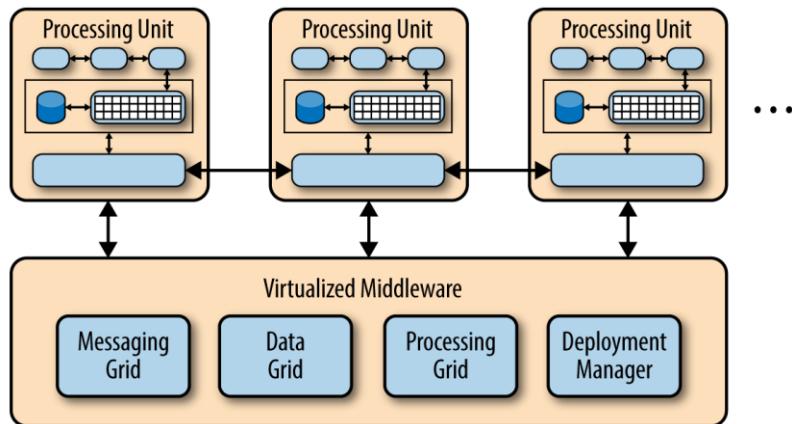


Рис. 1 - Основные компоненты архитектуры облака

Компонент модуля обработки (processing-unit component) содержит компоненты приложения (или части компонентов приложения), которые включают в себя веб-компоненты, а также внутреннюю бизнес-логику. Содержимое блока обработки зависит от типа приложения - небольшие веб-приложения будут развернуты в одном блоке обработки, крупные приложения могут разделить функциональность приложения на несколько блоков обработки в зависимости от функциональных областей приложения. Блок обработки содержит модули приложения, сетку данных в памяти, дополнительное асинхронное хранилище постоянства для переключения при отказе и механизм репликации данных, который используется **виртуализированным промежуточным ПО (virtualized middleware)** для репликации изменений данных, внесенных одним процессором в другие активные процессоры.

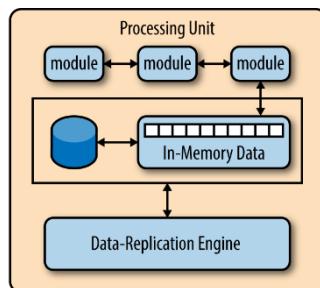


Рис. 2 - Компонент модуля обработки

Компонент виртуализированного промежуточного ПО обеспечивает обслуживание и обмен данными. Он содержит компоненты, управляющие различными аспектами синхронизации данных и обработки запросов. В виртуализированное промежуточное ПО включены сетка обмена сообщениями, сетка данных, сетка обработки и менеджер развертывания.

Сетка обмена сообщениями (Messaging Grid) управляет потоком входящей транзакции, а также обменом данными между службами. Когда запрос поступает в компонент виртуализированного промежуточного ПО, определяет, какие активные компоненты обработки доступны для получения запроса, и пересыпает запрос одному из этих блоков обработки.

Сетка данных (Data Grid) взаимодействует с механизмом репликации данных в каждом блоке обработки для управления репликацией данных между блоками обработки при обновлении данных. Поскольку сетка обмена сообщениями может пересыпать запрос на любой из доступных модулей обработки, важно, чтобы каждый модуль обработки содержал точно такие же данные в своей сетке данных в памяти.

Сетка обработки (Processing Grid) управляет распределенной обработкой запросов при наличии нескольких блоков обработки, каждый из которых обрабатывает часть приложения. Обеспечивает параллельную обработку событий между различными сервисами.

Менеджер по развертыванию (Deployment Manager) управляет динамическим запуском и остановкой процессоров в зависимости от условий нагрузки. Этот компонент постоянно отслеживает время отклика и пользовательские нагрузки, а также запускает новые блоки обработки при увеличении нагрузки и выключает блоки обработки при уменьшении нагрузки.

Анализ характеристик архитектуры.

Общая гибкость: высокая

Общая гибкость - это способность быстро реагировать на постоянно меняющуюся среду. Поскольку блоки обработки (развернутые экземпляры приложения) можно быстро включать и выключать, приложение хорошо реагирует на изменения, связанные с увеличением или уменьшением пользовательской нагрузки (изменения среды).

Легкость развертывания: высокая

Анализ: Сложные облачные инструменты позволяют легко «проталкивать» приложения на серверы, упрощая развертывание.

Тестируемость: низкая

Достижение очень высоких пользовательских нагрузок в тестовой среде требует больших затрат и времени, что затрудняет тестирование масштабируемости приложения.

Производительность: высокая

Высокая производительность достигается за счет встроенных в этот шаблон механизмов доступа к данным в памяти и кэширования.

Масштабируемость: высокая

Высокая масштабируемость обусловлена тем, что централизованная база данных практически не зависит от нее, что, по существу, устраняет это ограничивающее узкое место.

Легкость разработки: низкая

Сложные продукты для кэширования и сетки данных в памяти делают этот шаблон относительно сложным для разработки, в основном из-за недостаточного знакомства с инструментами и продуктами, используемыми для создания такого типа архитектуры. Кроме того, следует проявлять особую осторожность при разработке этих типов архитектур, чтобы убедиться, что ничто в исходном коде не влияет на производительность и масштабируемость.

Источники:

<https://book.huihoo.com/pdf/software-architecture-patterns.pdf> гл.5