**Вопросы к экзамену по курсу**

**«Объектно-ориентированные технологии программирования и стандарты проектирования ч.2»**

1.Особенности CASE-технологии. История развития CASE-средств.

2.Системная модель CASE-средств.

3.Критерии развития CASE-средств.

4.Понятие проекта. Масштаб проекта. Общие принципы управления проектом.

5.Три составляющие программного проекта: система обозначений, процесс и инструмент. Их роль и значение для проекта.

6.Типы и особенности современных программных проектов.

7. Задачи и категории современных методологий создания программных проектов.

8.Взаимосвязь между методологией, размером задачи и командой разработчиков.

9. Целесообразность использования различных методологий для различных типов программных проектов.

10. Унифицированный процесс разработки программных средств.

11. Основные и дополнительные элементы объектно-ориентированного подхода.

12. История появления, особенности и назначение унифицированного языка моделирования UML.

13. Сравнительный анализ CASE-средств *Rational Rose*, *Rational XDE*, *Enterprise Architect*.

14. Назначение, особенности и построение диаграммы *Use Case*.

15. Назначение, особенности и построение диаграммы *Deployment*.

16. Назначение, особенности и построение диаграммы *Statechart*.

17. Назначение, особенности и построение диаграммы *Activity*.

18. Назначение, особенности и построение диаграммы *Sequence*.

19. Назначение, особенности и построение диаграммы *Collaboration*.

20. Принцип модульности. Декомпозиция системы на подсистемы и модули.

21. Связность модуля. Типы связностей.

22. Сцепление модулей. Типы сцеплений.

23.Назначение, особенности и построение диаграммы *Component*.

24.Назначение, особенности и построение диаграммы *Class*, виды и особенности связей между классами на диаграммах.

25. Понятие шаблонов проектирования и их классификация. Шаблоны в нотации языка UML.

26. Шаблон “Фасад” и его обозначение в нотации языка UML.

27. Шаблон “Наблюдатель” и его обозначение в нотации языка UML.

28. Создание модели предметной области программной системы с помощью диаграммы классов.

29. Создание модели анализа с помощью диаграммы классов. Различные стереотипы для классов и их назначение.

30. Декомпозиция системы. Правила выделения подсистем.

31. Особенности создания шаблона приложения в среде *Rational Rose* с использованием библиотеки *MFC*. Структура и классы приложения.

32. Функциональные возможности *Rational Rose*: модуль *Component Assignment Too*l, компонент *Model Assistant*, обновление кода по модели и модели по коду.

33.Особенности генерации исходного кода в среде *Rational XDE*. Способы синхронизации модели.

34.Сравнительный анализ процедур генерации исходного кода в *Rational Rose*, *Rational XDE*, *Enterprise Architect*.

35. Назначение, возможности, особенности использования модуля *Web Modeler* в *Rational Rose*.

36. Возможности и особенности построения *Web*-модели в среде *Rational XDE*.

37. Понятие проектных рисков. Действия по управлению рисками.

38. Статический и динамический аспекты *Rational Unified Process (RUP)*.

39. Принципы и стадии разработки ПС в технологии *Rational Unified Process.*

40. Содержание и результаты первой и второй стадий в технологии *Rational Unified Process.*

41. Содержание и результаты третей и четвертой стадий в технологии *Rational Unified Process*.

42. Этапы создания программных средств в технологии *Oracle*.

43. Процессы создания программных средств в технологии *Oracle.*

44.Классический и быстрый подходы к разработке ПС в технологии *Oracle*. Факторы, определяющие выбор подхода.

45. Этапы разработки ПС в технологии *Borland*.

46. Сравнительный анализ технологий создания ПС *Rational Unified Process*, *Oracle*, *Borland*.

47. Понятия CASE-средство, CASE-система, CASE-технология, CASE-индустрия и различия между ними.

48. Проектирование информационных систем с применением UML. Разработка моделей бизнес-прецедентов и бизнес-объектов.

49. Проектирование информационных систем с применением UML. Разработка концептуальной модели данных и требований к системе.

50. Проектирование физической реализации информационной системы с применением UML.

51. Требования к программному обеспечению. Виды требований.

52. Формирование требований. Анализ требований.

53. Основы технологии сетевого планирования и управления проектами.

54. Размерно-ориентированные и функциональные метрики оценивания программных продуктов.

**1. Особенности CASE-технологии. История развития систем автоматизированной разработки ПС.**

Причина появления систем автоматизированного проектирования, явился дисбаланс между производительностями труда в сфере производства и в сфере обработки информации. Причем разница была не в пользу последней. Сначала проблему пытались решить путем перевода людей из одной области труда в другую, что привело к падению общих темпов роста производительности труда и экономики. Кроме этого, ситуацию осложняли следующие факторы:

1.Число разных классов тех. систем удваивалось в среднем через каждые 10 лет;

2.Сложность изделий, вызванная увеличением кол-ва их комплектующих, удваивалась через 10 лет;

3.Объем научно-технической информации удваивался через каждые 8 лет; Период создания новых изделий уменьшался в 2 раза через 10 лет, и при этом сокращалось время их морального старения.

В связи с этим были начаты работы по автоматизации процессов проектирования. До недавнего времени концепция автоматизации труда базировалась на принципах геометрического моделирования и компьютерной графики, что позволяло охватить лишь стадии технического и рабочего проектирования.

Системы компьютеризации труда конструкторов, технологов, программистов и менеджеров развивались автономно. В современных условиях необходима комплексная компьютеризация инженерной деятельности на всех этапах ЖЦ ПП. Комплексный характер САПР заключается в том, что в них выполняются и сложные вычисления, и обработка большого объема данных. Это определяет направление дальнейшего развития САПР как интегрированных интеллектуальных систем.

Современные САПР делят на категории:

а) тяжелые

б) средние

в) легкие.

Категории определяются средствами, вложенными в систему, или усилиями, потраченными на ее освоение. Считается, что успех САПР во многом зависит от того, на какие информационные ресурсы она ориентируется:

а) активные

б) пассивные.

Активные информационные ресурсы составляет информация, доступная для автоматизированного хранения, поиска и обеспечивающая обработку данных. Иные формы информации являются пассивными ресурсами.

САПР должна взаимодействовать только с активными ресурсами.

CASE-средства – средства обеспечивающие поддержку много-ных технологий проектирования информационных систем, охватывая всевозможные средства автоматизации и весь жизненный цикл программного обеспечения (ПО).

CASE-технология включает в себя методологию **анализа, проектирования, разработки и сопровождения сложных систем ПО.**

Основная цель CASE-подхода – разделить и максимально автоматизировать все этапы разработки ПС. Большинство CASE-средств основано на парадигме *методология / метод / нотация / средство.*

**Методология** определяет шаги работы и их последовательность, а также правила распределения и назначения методов.

**Метод** – это систематическая процедура генерации описаний компонентов ПО. **Нотация** предназначена для описания структур данных, порождающих систем и метасистем.

**Средства** – это инструментарий для поддержки методов на основе принятой нотации.

Преимущества применения CASE-средств:

- улучшение качества ПО за счет автоматического контроля проекта;

- возможность быстрого создания прототипа будущей системы, что позволяет уже на ранних стадиях разработки оценить результат;

- ускорение процессов проектирования и программирования;

- освобождение разработчиков от выполнения рутинных операций;

- возможность повторного использования ранее созданных компонентов.

**2. Системная модель CASE-средств, объектно-ориентированных CASE­-средств.**

Любая техническая система, включая CASE-средство, может быть представлена следующим набором характеристик: S={Ind, P, Atr, Inp, Out, Str}, Ind – обозначение и наименование системы; P – цели системы; Atr – общесистемные характеристики; Inp – входы системы; Out – выходы системы; Str – структура системы; Str={E, R}, где E – компоненты системы, R – связи между компонентами.

**Обозначение и наименование системы**. Каждая коммерческая система должна иметь зарегистрированный товарный знак, который в совокупности с обозначением версии и модификации системы представляет собой обозначение системы. Наименование включает в себя ее функциональное описание.

Цели системы достигаются за счет ее технических функций, которые характеризуют способность преобразовывать входную информацию в выходную. Чаще всего выступают такие характеристики, как:

* **трудоемкость,**
* **себестоимость,**
* **длительность цикла процесса,**
* **качество продукта**.

Уменьшение трудоемкости проектирования дос-ся за счет:

* автоматизации оформления документации,
* информационной поддержки и автоматизации принятия решений,
* параллельного проектирования.

*Снижение себестоимости* разработки происходит благодаря разумной экономии всех ресурсов.

*Сокращение ЖЦ* достигается за счет параллельного проектирования.

*Улучшение качества* результатов проектирования обеспечивается путем:

* использования автоматизированного поискового и многовариантного проектирования,
* применения математических методов оптимизации параметров и структур объектов и процессов,
* привлечения стратегического проектирования.

**Общесистемные характеристики** участвуют в классификации CASE-средств по следующим признакам:

* прикладная область объектов проектирования,
* сложность проектируемых объектов,
* уровень автоматизации:
  + менее 25% от общего количества автомат-ых процедур – низкоавтоматизированные CASE;
  + от 25% до 50% автоматизированных процедур – среднеавтоматизированные CASE;
  + более 50% автоматизированных процедур – высокоавтоматизированные CASE),
  + комплексность автоматизации проектирования (зависит от стадий проектирования, которые охватывает система, здесь выделяют: одноэтапные, многоэтапные и комплексные),
  + возможность работы в сетевом режиме и в Internet.

**Структура системы** включает в себя ее функциональные составные части и связи между ними, а также зависит от комплексности CASE-средств.

**Входы и выходы системы** зависят от ее функционального назначения и описываются в техническом задании на разработку.

**3. Критерии развития CASE-средств.**

Критерии развития – группа свойств, которые определяют меру совершенства и прогрессивности данной системы. Критерии, важные для CASE-средств, можно разделить на четыре группы: **функциональные, технологические, экономические, эргономические**.

**Функциональный критерий** рассматривается как интегральный показатель, зависящий от ряда частных функциональных критериев: **скорости обработки информации, интенсивности обработки информации, степени автоматизации труда, непрерывности процесса проектирования.**

*Скорость обработки информации* характеризуется двумя величинами: КНП – натуральным критерием производительности CASE-средства и КИП – информационным критерием производительности CASE-средства.

*С интенсивностью* обработки информации в CASE-средства связан информационный критерий эффективности (КИП). Он представляет собой усредненное отношение объема выходной информации к суммарному объему входной и выходной информации. Предельным минимумом для этого является техническое задание.

*Критерий совмещения функциональных операций* имеет смысл для интерактивных CASE-средств и определяет допустимое число одновременно выполняемых на одной рабочей станции функциональных операций.

*Критерий автоматизации* – критерий, характеризующий степень автоматизации CASE-средства.

*Критерий непрерывности* процесса проектирования.

**Технологические критерии** связаны с настоящими и будущими затратами на стадиях создания, развития и адаптации системы к решению конкретных задач.

Критерий технологических возможностей отражает простоту разработки CASE-средства и подготовки ее к эксплуатации.

**Экономический критерий** CASE-средства служит для комплексного стоимостного учета положительного эффекта от автоматизации проектирования и основных затрат. В качестве этого показателя принято использовать КЭ – величину годового экономического эффекта от использования CASE-средства. (может быть, как положительный, так и отрицательный)

**Критерий эргономичности** CASE-средства равен отношению реализуемой эффективности системы к максимально возможной эффективности этой системы. Он представляет собой зависящую от времени функцию, стремящуюся к 1. Данный критерий можно трактовать как КПД человека в системе.

**4. Понятие проекта. Масштаб проекта. Общие принципы управления проектом.**

**Под проектом подразумевают** некоторую работу (или группу работ), которую необходимо выполнить за определенный промежуток времени с соответствующим качеством, не выходя за рамки сметы.

В проекте должны быть четко определены:

1) Цели и запланированные результаты (масштаб или область охвата) ;

2) Уровень качества (качество);

3) Этапы и сроки выполнения работ (время);

4) Бюджет (или смета) по срокам и видам работ (стоимость).

Перечисленные выше параметры проекта – масштаб, качество, время и стоимость – тесно связаны между собой. Масштаб, время, и стоимость образуют так называемый треугольник проекта. Качество находится в центре этого треугольника.

Изменение любой из сторон треугольника влечет за собой изменения других параметров проекта и оказывает влияние на качество. Увеличение масштаба может привести к увеличению времени. Дополнительное время позволит повысить качество отдельных работ всего проекта. С другой стороны, увеличение времени или масштаба предполагает дополнительные расходы.

В процессе своего развития проект проходит через несколько этапов, которые составляют жизненный цикл (ЖЦ) проекта.

Полный цикл работы над проектом должен состоять, как минимум, из четырех основных этапов:

1) Постановка задачи,

2) Планирование проекта,

3) Реализация проекта,

4) Завершение проекта

**Принципы управления проектами.**

Управление проектом подразумевает не только планирование, но и координацию и контроль работ по проекту для достижения его целей в рамках заданного бюджета и сроков и с надлежащим качеством. Только постоянный контроль за выполнением работ позволяет своевременно выявить отклонения текущего состояния проекта от базового плана.

Во время управления осуществляется:

● Общее управление изменениями – определение, согласование, утверждение, и принятие к выполнению корректирующих воздействий и координация изменений по всему проекту

● Управление ресурсами – внесение изменений в состав исполнителей и назначение ресурсов на работы проекта

● Управление масштабом – корректировка целей проекта по результатам анализа

● Управление качеством – разработка мероприятий по устранению причин неудовлетворительного выполнения работ

К вспомогательным процессам управления относятся:

● Управление рисками – реагирование на события и изменение рисков в процессе выполнения проекта

● Управление контрактами – координация работы подрядчиков, корректировка контрактов, разрешение конфликтов.

**Современное профессиональное управление** проектами базируется на следующих основных принципах: ­

- четкое определение целей, результатов и работ проекта с учетом возможных приемлемых рисков;

-­ определение центров ответственности за проект в целом и отдельные его части;

- создание системы комплексного и прогнозирующего планирования работ и параметров проекта;

- создание системы контроля и регулирования хода выполнения проекта; ­

- создание команды проекта и управление ею с целью объединения и координации усилий всех исполнителей, вовлеченных в проект.

**5. Три составляющие программного проекта: система обозначений, процесс и инструмент. Их роль и значение для проекта.**

Успех любого программного проекта зависит от трех составляющих:

- системы обозначений (нотации, языка),

- процесса

- и инструмента.

**Одинаково нужны все три составляющие**.

**Система обозначений** важна в любой модели. Это связующее звено между всеми составляющими процесса разработки проекта. Примером полной и надежной системы обозначений может служить унифицированный язык моделирования (Unified Modeling Language – UML). С его помощью можно описывать модели на любом этапе разработки ПС: от анализа требований до проектирования и реализации.

**Процесс.** Современные проекты должны разрабатываться с помощью эффективных средств в соответствии со строгим графиком и обеспечивать возможность внесения изменений и адаптации к конкретным условиям и требованиям. ЖЦ проекта должен быть управляемым, что позволит гарантировать его непременное завершение. Примером процесса, подходящего для современных IT-проектов, является унифицированный процесс. Его методология основана на языке UML и находит поддержку в современных инструментах.

**Инструмент.** Ни одна из современных технологий разработки ПС не обходится без какого-либо инструмента. Сегодня на рынке представлен достаточно широкий спектр инструментов, использующих нотацию UML и объектноориентированный подход к разработке ПО.

**6. Типы и особенности современных программных проектов.**

**Типы современных программных проектов**:

• *Проект для постоянного заказчика*. Ситуация, при которой команда разработчиков в течение длительного времени обслуживает единственного заказчика.

• *Продукт под заказ*. Ситуация, при которой команда разработчиков находит стороннего заказчика и договаривается с ним о разработки программного продукта, призванного решить те или иные проблемы заказчика.

• *Тиражируемый продукт*. Ситуация, при которой команда разработчиков либо вообще не имеет конкретных заказчиков, либо довольно большое количество заказчиков желают иметь один и тот же продукт.

• *Аутсорсинг*. Это одна из наиболее новых моделей производства ПО. Суть ее состоит в том, что между крупной (обычно) фирмой по производству ПО и другой иностранной фирмой заключается договор о субподряде.

В настоящее время наиболее распространенным является **итерационный процесс создания ПО**. Он представляет собой набор итераций, в рамках каждой из которых проект проходит через все этапы ЖЦ. Итеративный цикл позволяет выявлять проблемы на самых ранних этапах разработки, управлять рисками, раньше начинать тестирование и т.п. Поэтому такой процесс становится более динамичным и управляемым.

**7. Задачи и категории современных методологий создания программных проектов**

Под методологией понимается набор методов, практик, метрик и правил, используемых в процессе производства ПО. Согласно Алистеру Кобурну, методология нужна для того, чтобы:

1. Облегчить процедуру введения новых людей в курс процесса производства
2. Обеспечить взаимозаменяемость людей
3. Распределить ответственность
4. Произвести впечатление на спонсора/заказчика
5. Демонстрировать видимый прозрачный процесс
6. Создать учебную базу для своих сотрудников

По определению Джима Хайсмита «Настоящее назначение методологий – это увеличение производительности, обеспечивающее решения для заказчиков »

Методологии можно условно разбить на 3 категории:

1. тяжелые
2. легкие
3. средние

Упрощенно, каждая из категорий предназначена для работы в условиях, соответственно, больших, малых и средних проектов. Однако это не совсем так.

Согласно Алистеру Кобурну:

Размер методологии – это количество элементов управления, включающих в себя промежуточные релизы, стандарты, виды деятельности, вехи, метрики качества и т.д.

Плотность методологии – это уровень детальности и целостности, требуемый для элементов методологии.

Вес проекта – это размер методологии, умноженный на ее плотность.

Размер проекта – это количество людей, вовлеченных в реализацию проекта.

Все вышеприведенные определения носят концептуальный характер, поскольку не существует количественных метрик для данных величин.

Первая категория(тяжелые) методологий появилась на свет раньше всех и является неотъемлемой частью моделей качества программного обеспечения. Тяжелые методологии отличаются тем, что охватывают все аспекты деятельности компании, производящей программное обеспечение – от управления требованиями и планирования процесса до регламентирования взаимоотношений с субподрядчиками и описания требований к вспомогательным процессам. Все методологии данной категории нетерпимы к изменениям и рассматривают людей как обычный ресурс. Примеры: CMM, ISO9000, SPICE.

Вторая категория(легкие) методологий появилась на свет в качестве некоторой совокупности методов и практик, применявшихся небольшими командами разработчиков в успешных проектах. Здесь огромное значение уделяется взаимоотношениями между людьми внутри команды, учитываются вопросы терпимости и другие психологические аспекты. Все процессы данной категории предусматривают итерационный жизненный цикл разработки ПО. В частности, во всех легких методологиях предусмотрен лишь необходимый минимум документов, т.к. отдается должное принципу «Документация – это не есть понимание». Существенным отличием данных методологий от методологий первого типа является отношение к планированию. Примеры: SCRUM, XP (eXtremal Programming), Crystal Clear.

В третью(средние) категорию методологий попадают так называемые «универсальные» процессы. Самым ярким и известным представителем данной категории является RUP. Основной характеристикой подобных процессов является масштабируемость – т.е. процесс может быть настроен как на работу в малой команде над небольшим проектом, так и в большой команде над большим и серьезным проектом.

**8. Взаимосвязь между методологией, размером задачи и командой разработчиков.**

Методологии можно условно разбить на три категории: **тяжелые, легкие и средние**.

Упрощенно каждая из них предназначена для работы в условиях больших, малых и средних проектов соответственно.

**Тяжелая категория** методологий появилась раньше других и служит неотъемлемой частью моделей качества ПО. Тяжелые методологии охватывают все аспекты деятельности компании, производящей ПО, – от управления требованиями и планирования процесса до регламентирования отношений с заказчиком. Все методологии данной категории нетерпимы к изменениям и рассматривают людей как ресурс. К этой категории относятся ISO9001, CMM, SPICE.

**Легкая категория методологий** – это некоторая совокупность методов и практик, применявшихся небольшими командами разработчиков в небольших проектах. Все процессы данной категории предусматривают итерационный ЖЦ разработки ПО. Идея легких методологий – обеспечение максимальной скорости и качества разработки ПО при минимуме ограничений. Во всех легких методологиях предусмотрен лишь необходимый минимум документов. Примеры легких методологий - SCRUM, ICONIX, XP, Crystal Clear.

**Средняя категория** методологий включает в себя так называемые универсальные процессы. Наиболее популярным представителем этой категории является методология рационального унифицированного процесса – Rational Unified Process (RUP). Основная характеристика этой методологии – масштабируемость, т.е. процесс может быть настроен на работу как в малой команде над небольшим проектом, так и в большой команде над большим проектом.

**9. Целесообразность использования различных методологий для различных типов программных проектов.**

**Проект для постоянного заказчика**.

Самый благоприятный тип проекта для внедрения *легких методологий*, поскольку заказчик всегда доступен и не предъявляет сверхтребований к ПО. Однако необходимо учитывать количество разработчиков и степень их распределенности. Как правило, у таких команд не бывает необходимости в сертификации.

**Продукт под заказ**.

Самый уязвимый тип проекта. Фирма целиком зависит от количества заключенных договоров. Постоянно идет поиск новых заказчиков. В таких условиях, конечно же, желательно наличие сертификата ISO. Сертификацию целесообразно проводить лишь при достижении определенной численности персонала, которой будет достаточно для внедрения *тяжелой или средней технологии*. Альтернативный вариант – одна из легких методологий.

**Тиражируемый продукт**.

Самый устойчивый тип проекта. Выпуск такого проекта всегда характеризуется более низкими затратами на его производство по сравнению с выпуском единичных экземпляров. В данных условиях невозможно использование легкой методологии в чистом виде, т.к. нет возможности постоянно работать с заказчиком. В этом случае все зависит от способа управления командой, тактических и стратегических целей.

**Аутсорсинг**.

Данный вид проектов характеризуется распределенной структурой и начальными предпосылками к утяжелению процесса, поскольку общение с заказчиком происходит в виде документов установленного образца. Если сторона фирмы-заказчика предоставляет команде свой технологический процесс, то у нее нет свободы выбора. В противном случае стоит остановить свой выбор на каком-либо из процессов *средней тяжести.*

**10. Унифицированный процесс разработки программных средств.**

**Процесс** – множество различных видов деятельности, необходимых для преобразования требований пользователей в программную систему.

**Унифицированный процесс** – это обобщенный каркас процесса, который может быть специализирован для любых программных систем.

Его основными принципами являются:

• итерационный и инкрементный подход к созданию ПО;

• управление вариантами использования;

• построение системы на базе архитектуры ПО.

В соответствии с ним разработка системы выполняется в виде нескольких краткосрочных минипроектов фиксированной длительности (от 2 до 6 недель), называемых *итерациями*. Каждая итерация включает в себя свои собственные этапы анализа требований, проектирования, реализации, тестирования, интеграции и завершается созданием работающей системы.

Итерационный цикл основывается на постоянном расширении и дополнении системы в процессе нескольких итераций с периодической обратной связью и адаптацией добавляемых модулей к существующему ядру системы.

**Вариант использования** – это часть функциональности системы, необходимая для получения пользователем значимого для него, ощутимого и измеримого результата.

Варианты использования обеспечивают функциональные требования.

Все варианты использования в совокупности составляют модель вариантов использования, которая описывает полную функциональность системы. Варианты использования управляют архитектурой системы, которая, в свою очередь, оказывает влияние на их выбор. И архитектура системы, и варианты использования развиваются по мере хода жизненного цикла.

Созданная архитектура является основой всей дальнейшей разработки. В будущем неизбежны незначительные изменения в деталях архитектуры, однако серьезные изменения маловероятны.

**11. Основные и дополнительные элементы объектно-ориентированного подхода.**

Главное отличие объектного подхода от структурного заключается в объектной декомпозиции системы. При этом статическая структура системы описывается в терминах объектов и связей между ними, а поведение системы – в терминах обмена сообщениями между объектами.

Главные элементы объектной модели: абстрагирование, инкапсуляция, модульность, иерархия. 3 дополнительных элемента: типизация, параллелизм, устойчивость.

**Абстрагирование** – это выделение существенных характеристик некоторого объекта, которые отличают его от всех других видов объектов и, таким образом, четко определяют его концептуальные границы относительно дальнейшего рассмотрения и анализа. Абстрагирование позволяет отделить существенные особенности поведения объекта от деталей их реализации.

**Инкапсуляция** – это процесс отделения друг от друга элементов объекта, определяющих его устройство и поведение. Инкапсуляция служит для того, чтобы разделить интерфейс и внутреннюю реализацию объекта.

**Модульность** – это свойство системы, связанное с ее декомпозицией на ряд внутренне сильно связанных, но слабо связанных между собой модулей. Инкапсуляция и модульность создают барьеры между абстракциями.

**Иерархия** – это ранжированная или упорядоченная система абстракций. Иерархия по номенклатуре – это структура классов, а иерархия по составу – это структура объектов.

**Типизация** – это ограничение, накладываемое на класс объектов и препятствующее взаимозаменяемости различных классов.

**Параллелизм** – это свойство объектов находиться в активном или пассивном состоянии и различать между собой активные и пассивные объекты.

**Устойчивость** – это свойство объекта существовать во времени и в пространстве вне зависимости от процесса, породившего данный объект.

**Полиморфизм** можно интерпретировать, как способность класса принадлежать более чем одному типу.

**Наследование** означает построение новых классов, на основе существующих с возможностью добавления или переопределения данных и методов.

Благодаря применению абстрагирования, модульности и полиморфизма на всех стадиях разработки ПС существует согласованность между моделями всех этапов ЖЦ, когда модели ранних стадий могут быть сравнены с моделями реализации.

**12. История появления, особенности и назначение унифицированного языка моделирования UML.**

**UML** — язык графического описания для объектного моделирования в области разработки ПО, моделирования бизнес­процессов, системного проектирования и отображения организационных структур.

UML позволяет разработчикам ПО достигнуть соглашения в графических обозначениях для представления общих понятий (класс, компонент, обобщение, агрегация, поведение), больше сконцентрироваться на проектировании и архитектуре.

*В 1994* Гради Буч и Джеймс Рамбо, из компании Rational Software, объединились для создания нового языка объектно-ориентированного моделирования. За основу были взяты методы моделирования Object­Modeling Technique и Booch. OMT был ориентирован на анализ, а Booch — на проектирование ПС.

*В октябре 1995* была выпущена предварительная версия 0.8 унифицированного метода.

*Осенью 1995* к компании Rational присоединился Ивар Якобсон, автор метода Object­Oriented Software Engineering — OOSE. OOSE обеспечивал возможности для спецификации бизнес-процессов и анализа требований при помощи сценариев использования. OOSE был также интегрирован в унифицированный метод.

В настоящее время UML является общепризнанным стандартом моделирования. В UML ­модели есть два аспекта: ­ **статическая структура** – описывает, какие типы объектов важны для моделирования системы и как они взаимосвязаны; ­ **динамическое поведение** – описывает ЖЦ этих объектов и то, как они взаимодействуют для обеспечения требуемой функциональности системы.

**Основные цели создания унифицированного языка моделирования:**

1. Предоставить пользователям готовый к применению выразительный язык визуального моделирования, для разработки осмысленных моделей и обмена ими.

2. Предусмотреть механизмы для расширения базовых концепций.

3. Обеспечить независимость UML от конкретных ЯП и процесса разработки.

4. Создать формальную основу для понимания языка моделирования.

5. Стимулировать рост рынка объектно-ориентированных инструментальных средств.

6. Интегрировать лучший практический опыт.

Семантика UML представляет собой некоторую метамодель, которая определяет абстрактный синтаксис и семантику понятий объектного моделирования на языке UML. Семантика определяется для двух видов объектных моделей: *структурных и поведения*. Структурные (статические) описывают структуру сущностей или компонентов некоторой системы, включая классы, интерфейсы, атрибуты и отношения. Модели поведения (динамические) описывают поведение или функционирование объектов системы, включая их методы, взаимодействие и сотрудничество между ними, а также процесс изменения состояний отдельных компонентов и системы в целом. Для решения широкого диапазона задач моделирования разработана полная семантика для всех компонентов графической нотации. Требования семантики языка UML конкретизируются при построении отдельных видов диаграмм.

**13. Сравнительный анализ CASE-средств Rational Rose, Rational XDE, Enterprise Architect.**

Rational Rose – это средство автоматизированного проектирования ПС. В его основе лежит CASE-­технология, комплексный подход и использование единой унифицированной нотации на всех этапах ЖЦ создания ПС. Графические возможности продукта позволяют решать задачи, связанные с проектированием, на различных уровнях абстракции: от общей модели процессов предприятия до конкретной модели класса в создаваемом ПО. В среде Rational Rose проектировщик и программист работают в тандеме: 1ый создает логическую модель системы, а 2ой дополняет ее моделями классов на конкретном ЯП. В настоящее время ПП обеспечивает генерацию кода по модели на ряде ЯП.

*Преимущества от применения Rational Rose:*

1. сокращение цикла разработки приложения “заказчик-­программист-­заказчик”;
2. увеличение продуктивности работы программиста;
3. улучшение потребительских качеств создаваемых программ за счет ориентации на пользователей и бизнес;
4. способность вести большие проекты и группы проектов;
5. возможность повторного использования уже созданного программного обеспечения за счет упора на разбор их архитектуры и компонентов.

Главное отличие ​Rational XDE от своего предшественника, программы Rational Rose, это полная интеграция с платформой Microsoft Visual Studio.NET, позволяющая в одной оболочке работать как с моделями программной системы, так и с кодом. За счет этого заметно сократилось время синхронизации модели и программного кода, в результате чего повысилась производительность рабочей станции. Программа потеряла часть своей универсальности по сравнению с Rational Rose.

*Возможности Rational XDE:*

1) представляет собой единый процесс проектирования и программирования. Таким образом, на этапах бизнес­-моделирования и кодирования может использоваться одна и та же модель. Причем, аналитики могут работать в традиционной Rose, а разработчики в XDE, не мешая друг другу;

2) позволяет проводить прямое и обратное проектирование, а также синхронизацию кода и модели, что позволяет структурировать проект любого типа сложности;

3) имеет возможность по созданию шаблонов проектирования, так называемых паттернов, используя которые разработчики могут быстро формировать основные элементы проекта, основываясь на уже устоявшихся подходах;

4) поддерживает моделирование в свободной форме, с различными настройками под предметную область, с учетом традиций компании-­разработчика;

5) имеет высокоэффективные механизмы поиска определенных моделей или их свойств, что позволяет легко ориентироваться в сложных системах;

6) поддерживает одновременную совместную работу разных моделей с помощью общих представлений и различных связей;

7) поддерживает все возможности по управлению версиями.

Одним из отличий Rational XDE от предыдущей версии явилось использование окна Toolbox, содержащего дополнительные инструменты для работы над проектом. Строки инструментов (Toolbar) также остались, но их роль сократилась до управления основными режимами, а создание новых элементов теперь производится с использованием Toolbox. Аналогично тому, как это происходило в Rational Rose, в среде Rational XDE все действия над объектами выполняются посредством контекстного меню. Оно зависит от набора функций, доступных для применения к конкретному объекту.

**14. *Use Case.***

Этот вид диаграмм позволяет создать список операций, которые выполняет система. Часто Use case называют диаграммой функций, так как на основе набора таких диаграмм создается список требований к системе и определяется множество выполняемых ею функций. Данный тип диаграмм используется при определении требований к будущей программной системе.

Диаграмма Use Case состоит из 10 значков, главными из которых являются:

* **Use Case** - варианты использования;
* **Actor -** действующие лица.

Use Case и Actor объединяются при помощи соответствующих связей.

Рекомендуется придерживаться следующих правил, создавая диаграмму Use Case:

**1.** Не моделировать связи между Actors, так как по определению они находятся вне сферы действия системы. Следовательно, связи между ними также не относятся к ее компетенции.

**2.** Не соединять непосредственно два Use Case, поскольку данная диаграмма только перечисляет варианты

использования, доступные системе, а не указывает порядок их выполнения.

**3.** Каждый вариант использования инициируется действующим лицом, поэтому должна быть связь, начинающаяся на действующем лице и заканчивающаяся на варианте использования.

**15. *Deployment*.**

Этот вид диаграмм предназначен для анализа аппаратной части системы, то есть «железа», а не программ. В прямом переводе с английского Deployment означает «развертывание», но также может переводиться как «топология».

Для каждой модели создается только одна такая диаграмма, отображающая процессоры (Processor), устройства (Device) и их соединения.

Обычно этот тип диаграмм используется в самом начале проектирования системы для анализа аппаратных средств, на которых она будет эксплуатироваться.

При создании диаграммы Deployment строка инструментов насчитывает 7 кнопок, главными из которых являются Device, Processor и Connection, которые объединяют компоненты диаграммы.

Разработка диаграммы развертывания начинается с идентификации всех аппаратных, механических и других типов устройств, которые необходимы для выполнения системой всех своих функций. В первую очередь специфицируются вычислительные узлы системы, обладающие памятью и/или процессором. При этом используются имеющиеся в языке UML стереотипы, а в случае отсутствия последних, разработчики могут определить новые стереотипы. Отдельные требования к составу аппаратных средств могут быть заданы в форме ограничений, свойств и помеченных значений.

Дальнейшее построение диаграммы развертывания связано с размещением всех исполняемых компонентов диаграммы по узлам системы. Если отдельные исполняемые компоненты оказались не размещенными, то подобная ситуация должна быть исключена введением в модель дополнительных узлов, содержащих процессор и память.

**16. *Statechart***​

Состояние(state) - ситуация в жизненном цикле объекта, во время которой он удовлетворяет некоторому условию, выполняет определенную деятельность или ожидает какого-­то события. Состояние объекта определяется значениями некоторых его атрибутов и присутствием или отсутствием связей с другими объектами.

Диаграмма состояний Statechart предназначена для изучения состояний объектов и условий переходов между ними. Модель состояний позволяет представить поведение объекта при получении им сообщений и взаимодействии с другими объектами. Диаграмма состояний полезна при моделировании жизненного цикла объекта. От других диаграмм диаграмма состояний отличается тем, что описывает процесс изменения состояний только одного экземпляра определенного класса ­ одного объекта, причем объекта *реактивного*, то есть объекта, поведение которого характеризуется его реакцией на внешние события.

Имеется ряд обозначений.

Скругленные прямоугольники представляют состояния, через которые проходит объект в течение своего жизненного цикла. Стрелками показываются переходы между состояниями, которые вызваны выполнением методов описываемого диаграммой объекта. Существует также *два* вида псевдосостояний: начальное, в котором находится объект сразу после его создания (обозначается сплошным кружком), и конечное, которое объект не может покинуть, если перешел в него (обозначается кружком, обведенным окружностью).

Этапы построения:

1. Создание точки начала работы (**Initialize point**).
2. Обычно следующим состоянием системы после начала ее работы является ожидание наступления событий. После чего в нашем случае создается состояние (**State**), которое соединяется стрелкой State Transition с начальной точкой. Для повышения информативности состояниям и событиям, переводящим объекты из одних состояний в другие, присваиваются имена. С состоянием объекта могут быть связаны события и действия. Разница между ними заключается в том, что действие осуществляется самим классом, для которого строится диаграмма, а посылка сообщения направлена на объект другого класса, чей метод вызывается при помощи сообщения.
3. Построение диаграммы завершается добавлением на нее значка – **End State**, который отражает окончание работы.

**17. Activity.**

Данный тип диаграмм является разновидностью диаграмм состояний. Главное различие между Activity и Statechart заключается в том, что первая характеризует действия, а вторая – статичные состояния. При этом Activity больше подходит для моделирования последовательности действия, а Statechart – для моделирования дискретных состояний объекта. Диаграммы деятельности могут создаваться на всех стадиях разработки ПО. Перед началом работ с их помощью моделируются важные рабочие процессы предметной области, с целью определения структуры и динамики бизнеса. На этапе обсуждения требований диаграммы деятельности используются для описания процессов и событий выявленных прецедентов. В течение фазы анализа и проектирования Activity помогает моделировать процесс операций объектов.

Диаграмма Activity является разновидностью диаграмм состояний и характеризует **действия**. Activity больше используется для моделирования последовательности действий.

Этапы построения:

1. Activity начинается значком **Start State**
2. **State** означает выполнение определенных действий в течение жизни объекта. В отличие от обычно обозначающего ожидание какого-либо события, показывает непосредственное действие.
3. Завершается значком **End State**.

Деятельности соединяются на диаграмме значком – **State Transition (переход состояния)**. Кроме этого, показывает получение и обработку сообщения объектом. Переход состояния может происходить между Action-Action, State-State, State-Action, Action-State. Возможна установка нескольких переходов между двумя состояниями или действиями. Каждый такой переход уникален и показывает реакцию объекта на определенное сообщение. Поэтому нельзя создать несколько переходов между двумя состояниями с указанием одного и того же сообщения.

**18. *Sequence***.

После того как в системе изучено поведение каждого объекта, необходимо точно представить взаимодействие этих объектов между собой, определить клиентов, серверы и порядок обмена сообщениями между ними. Диаграмма Sequence позволяет получить отражение процесса обмена сообщениями во времени. В течение работы системы объекты, являющиеся клиентами, посылают друг другу различные сообщения, а объекты серверы обрабатывают их. В простейшем случае можно рассматривать сообщение как вызов метода какого-­либо класса, в более сложных случаях сервер имеет обработчик очереди сообщений, и сообщения им обрабатываются асинхронно, т.е. сервер накапливает несколько сообщений в очереди, если не может обработать их сразу.

- **Object** (объект) позволяет включить объект в диаграмму. Каждый из них является реализацией какого-нибудь класса, поэтому в объекте можно указать соответствующий ему класс.

- **Message** (сообщение) предназначен для передачи сообщения от одного объекта к другому. Классы должны позволять отправку или прием сообщений.

- **Message to self** (сообщение самому себе) показывает, что отправитель сообщения является одновременно и его получателем.

*Для реализации объектов в окне параметров доступна опция Persistence, отражающая время жизни объекта*. Это время от его создания до уничтожения.

Можно настроить следующие параметры жизни объекта:

∙ **Persistent** – область видимости объекта больш время жизни.

∙ **Static** – элемент существует на всем протяжении работы программы.

∙ **Transient** – время жизни объекта и области видимости совпадают.

Свойства сообщений задаются в окне спецификации с помощью двух групп кнопок.

*1. Frequency*​ ​ определяет частоту обмена сообщениями:

**Periodic** – сообщения поступают от клиента с заданной периодичностью;

**Aperiodic** – сообщения поступают от клиента нерегулярно.

*2****.Synchronization*** определяет порядок обмена сообщениями:

· **Simple** – простая посылка сообщения.

· **Synchronous** – операция происходит только в том случае, когда клиент посылает сообщение, а сервер может принять сообщение клиента.

· **Timeout** – клиент отказывается от выдачи сообщения, если сервер в течение определенного времени не может его принять.

· **Balking** – операция происходит только в том случае, когда сервер готов немедленно принять сообщение, иначе клиент не выдает сообщение.

· **Procedure Call** – клиент вызывает процедуру сервера и полностью передает ему управление.

· **Return** – возврат из процедуры.

· **Asynchronous** – клиент выдает сообщение, и, не ожидая ответа сервера.

**19.** **Collaboration**

Collaboration ­ диаграмма кооперации. Она акцентирует внимание на структуре обмена сообщениями между отдельными объектами. Данная диаграмма отличается от Sequence тем, что здесь не акцентируется внимание на последовательности передачи сообщений, а отражается наличие взаимосвязей между клиентами и серверами вообще. Поскольку на Collaboration для демонстрации сообщений не применяется временная шкала, диаграмма получается более компактной и оптимально подходит для представления взаимодействий сразу всех объектов. Однако такое представление является мгновенным снимком системы в некотором состоянии, так как объекты создаются и уничтожаются на всем протяжении работы программы. В связи с этим появляются такие понятия, как время жизни и область видимости объектов.

Главная особенность диаграммы кооперации заключается в возможности графически представить не только последовательность взаимодействия, но и все структурные отношения между объектами, участвующими в этом взаимодействии.

Прежде всего, на диаграмме кооперации в виде прямоугольников изображаются участвующие во взаимодействии объекты, содержащие имя объекта, его класс и, возможно, значения атрибутов. Далее, как и на диаграмме классов, указываются ассоциации между объектами в виде различных соединительных линий. При этом можно явно указать имена ассоциации и ролей, которые играют объекты в данной ассоциации. Дополнительно могут быть изображены динамические связи ­ потоки сообщений. Они представляются также в виде соединительных линий между объектами, над которыми располагается стрелка с указанием направления, имени сообщения и порядкового номера в общей последовательности инициализации сообщений.

В отличие от диаграммы последовательности, на диаграмме кооперации изображаются только отношения между объектами, играющими определенные роли во взаимодействии. На этой диаграмме не указывается время в виде отдельного измерения. Поэтому последовательность взаимодействий и параллельных потоков может быть определена с помощью порядковых номеров. Следовательно, если необходимо явно специфицировать взаимосвязи между объектами в реальном времени, лучше это делать на диаграмме последовательности.

Удобной возможностью работы в Rational Rose является то, что на основе Sequence ­диаграммы можно создавать Collaboration и наоборот.

Rational XDE не поддерживает диаграмму *Collaboration* в том виде, в котором она присутствует в среде Rational Rose. *Collaboration* нельзя создать как отдельную диаграмму. Проектирование ведется с использованием только диаграммы *Sequence*.

**20. Принцип модульности. Декомпозиция системы на подсистемы и модули.**

*Модульность* – это свойство системы, связанное с ее декомпозицией на ряд внутренне сильно связанных, но слабо связанных между собой модулей. Инкапсуляция и модульность создают барьеры между абстракциями.

*Декомпозиция системы* – возможность представления проекта в виде удобных для обработки частей-модулей.

*Мо́дульное программи́рование* — это организация программы как совокупности небольших независимых блоков, называемых модулями, структура и поведение которых подчиняются определенным правилам. Использование модульного программирования позволяет упростить тестирование программы и обнаружение ошибок. Аппаратно-зависимые подзадачи могут быть строго отделены от других подзадач, что улучшает мобильность создаваемых программ.

*Мо́дуль* — функционально законченный фрагмент программы. Во многих языках (но далеко не обязательно) оформляется в виде отдельного файла с исходным кодом или поименованной непрерывной её части. Некоторые языки предусматривают объединение модулей в пакеты.

Принцип модульности является средством упрощения задачи проектирования программного обеспечения (ПО) и распределения процесса разработки между группами разработчиков. При разбиении ПО на модули для каждого модуля указывается реализуемая им функциональность, а также связи с другими модулями. Удобство использования модульной архитектуры заключается в возможности обновления (замены) модуля, без необходимости изменения остальной системы.

Количество модулей в комплексе должно определяться декомпозицией поставленной задачи на независимые подзадачи. В предельном случае модуль может использоваться даже для заключения в него всего лишь одной процедуры, если необходимо, чтобы выполняемое ею локальное действие было гарантировано независимым от влияния других частей программы при любых изменениях.

Роль модулей могут играть структуры данных, библиотеки функций, классы, сервисы и др. программные единицы, реализующие некоторую функциональность и предоставляющие интерфейс к ней.

Программный код часто разбивается на несколько файлов, каждый из которых компилируется отдельно от остальных. Такая модульность программного кода позволяет значительно уменьшить время перекомпиляции при изменениях, вносимых лишь в небольшое количество исходных файлов, и упрощает групповую разработку. Также это возможность замены отдельных компонентов (таких как jar-файлы, so или dll библиотеки) конечного программного продукта, без необходимости пересборки всего проекта (например, разработка плагинов к уже готовой программе).

Одним из методов написания модульных программ является объектно-ориентированное программирование. ООП обеспечивает высокую степень модульности благодаря таким свойствам, как инкапсуляция, полиморфизм и позднее связывание.

**21. Связность модуля. Типы связностей.**

Связность, или прочность (англ. cohesion, module strength), — мера силы взаимосвязанности элементов внутри модуля; способ и степень, в которой задачи, выполняемые некоторым программным модулем, связаны друг с другом.

Макконнелл объясняет понятие связности на примере класса (как частного случая модуля): «связность характеризует то, насколько хорошо все методы класса или все фрагменты метода соответствуют главной цели, — иначе говоря, насколько сфокусирован класс».

Связность обычно противопоставляется зацеплению. Слабое зацепление является признаком хорошо структурированной и хорошо спроектированной системы, и, когда она комбинируется с сильной связностью, соответствует общим показателям хорошей читаемости и сопровождаемости.

**Типы связности:**

Случайная связность.

Тип связности, при котором задачи, выполняемые программным модулем, не имеют функциональной взаимосвязи друг с другом.

Коммуникационная связность.

Тип связности, при котором задачи, выполняемые программным модулем, используют одни и те же входные данные или участвуют в формировании одних и тех же выходных данных.

Функциональная связность.

Тип связности, при котором все задачи, выполняемые программным модулем, вносят вклад в выполнение одной и той же функции.

Логическая связность.

Тип связности, при котором задачи, программным модулем, реализуют логически сходные функции (например, одинаково обрабатывают разные типы входных данных).

Процедурная связность.

Тип связности, при котором все задачи, выполняемые программным модулем, участвуют в некоторой программной процедуре.

Последовательностная связность.

Тип связности, при котором выходные данные одной задачи, выполняемой программным модулем, служат входным данными для другой задачи, выполняемой этим же модулем.

Временна́я связность.

Тип связности, при котором все задачи, выполняемые программным модулем, требуются для некоторой фазы выполнения программы (например, модуль содержит все задачи для инициализации).

**22. Сцепление модулей. Типы сцеплений.**

Зацепление, сцепление (англ. coupling) — способ и степень взаимозависимости между программными модулями; сила взаимосвязей между модулями; мера того, насколько взаимозависимы разные подпрограммы или модули.

Сильное зацепление рассматривается как серьезный недостаток, поскольку затрудняет понимание логики модулей, их модификацию, автономное тестирование, а также переиспользование по отдельности. Слабое зацепление, напротив, является признаком хорошо структурированной и хорошо спроектированной системы, и, когда оно комбинируется с сильной связностью, соответствует общим показателям хорошей читаемости и сопровождаемости.

**Типы зацепления:**

Зацепление по общей области.

Тип зацепления, при котором два программных модуля совместно используют общую область данных.

Зацепление по содержимому.

Тип зацепления, при котором некоторые или все программные модули включены в некоторый модуль как составные части.

Зацепление по управлению.

Тип зацепления, при котором один программный модуль обменивается данными с другим модулем с явной целью повлиять на его последующее выполнение.

Зацепление по данным.

Тип зацепления, при котором выходные данные одного программного модуля служат входными данными другого модуля.

Смешанное зацепление.

Тип зацепления, при котором различные подмножества значений некоторого элемента данных используются в нескольких программных модулях для разных и несвязанных целей.

Патологическое зацепление.

Тип зацепления, при котором один программный модуль зависит от деталей внутренней реализации другого модуля или влияет на них.

***Методы уменьшения зацепления***

Существуют различные методы уменьшения зацепления (англ. decoupling). Как правило, они описаны в виде шаблонов проектирования. Одним из ключевых методов является инверсия управления (фреймворки), и, в частности, внедрение зависимости.

Снизить зацепление также помогает использование многослойной архитектуры приложений, например Model-View-Controller, Model-View-Presenter, Model-View-ViewModel и т. п.

**23.** **Назначение, особенности и построение диаграммы Component.**

Этот тип диаграмм предназначен для распределения классов и объектов по компонентам при физическом проектировании системы. Часто данный тип диаграмм называют диаграммами модулей.

Диаграмма компонентов описывает особенности физического представления системы. Она позволяет определить архитектуру разрабатываемой системы, установив зависимости между программными компонентами, в роли которых может выступать исходный и исполняемый код.

Основными графическими элементами диаграммы компонентов являются:

* компоненты
* интерфейсы
* зависимости между ними.

Диаграмма компонентов разрабатывается для следующих целей:

* визуализации общей структуры исходного кода программной системы;
* спецификации исполняемого варианта программной системы;
* обеспечения многократного использования отдельных фрагментов программного кода;
* представления концептуальной и физической схем баз данных.

Для представления физических сущностей в языке UML применяется специальный термин - компонент. Компонент реализует некоторый набор интерфейсов и служит для общего обозначения элементов физического представления модели. Для графического представления компонента используется специальный символ ­ прямоугольник со вставленными слева двумя более мелкими прямоугольниками. Внутри большого прямоугольника записывается имя компонента и, при необходимости, некоторая дополнительная информация. Изображение этого символа может незначительно варьироваться в зависимости от характера ассоциируемой с компонентом информации.

**24. Назначение, особенности и построение диаграммы *Class,* виды и особенности связей между классами на диаграммах.**

Диаграмма классов – основная для создания кода приложения. С ее помощью строится внутренняя структура системы. Обычно данная диаграмма строится для всех классов, становясь логической моделью системы. Rational Rose позволяет на основе Class diagram создавать исходный код приложения на любом ЯП, который поддерживается генератором кода Rational Rose.

Главная диаграмма классов (Main) уже присутствует во вновь созданной пустой модели, но возможно создание дополнительных диаграмм.

Rational Rose позволяет устанавливать значительное количество свойств класса, которые влияют на генерацию его кода.

***Спецификация класса (вкладки):***

* **General** - задаются главные свойства класса: имя, тип, стереотип и доступ к нему, когда класс находится в пакете, а также документация к классу.
* **Detail -** позволяет указать дополнительные характеристики класса:
* **Multiplicity** – ожидаемое количество объектов, которые будут созданы на основе данного класса;
* **Space** – количество оперативной памяти, необходимой для создания объекта, учитывая накладные расходы на его создание плюс размер всех объектов, входящих в данный; Persistence – признак, указывающий время жизни объекта;
* **Concurrency** – поведение элемента в многопотоковой среде;
* **Abstract adornment** обозначает, что класс является абстрактным, т.е. базовым, который должен быть наследован подклассами;
* **Formal Arguments** заполняется только для параметризированных классов и утилит классов.
* ­**Components**: отражает компоненты, с которыми ассоциирован класс.
* **Attributes**: позволяет добавлять, удалять, редактировать атрибуты класса. Можно задать тип хранения атрибута в классе: By Value – по значению; By Reference – по ссылке; Unspecified – не указано. Кроме того, можно указать, что атрибут является Static (статическим) или Derived (производным).
* **Operations**: предназначена для добавления, удаления, редактирования операции класса.
* **Relations**: позволяет добавлять, удалять, редактировать связи класса.

**Назначение и виды связей в диаграмме Class**

* Однонаправленная ассоциация - один класс включается в другой как атрибут по ссылке или по значению;
* Aggregate – один класс содержит другой;
* Зависимость - один класс использует объекты другого. Это может осуществляться при передаче параметров или вызове операций класса;
* Ассоциированный класс - отображение свойства ассоциации; предназначена для задания дополнительных атрибутов у связи; обозначает, что некоторый класс со своими атрибутами включается как элемент в два других;
* Наследование - один класс является родительским по отношению к другому;
* Реализация - один класс является реализацией, т.е. создан на основе шаблона другого; используется понятие параметризованный класс.

**25. Понятие шаблонов проектирования и их классификация. Шаблоны в нотации языка UML.**​

Встречаются ситуации, когда решение проблем в различных проектах имеют сходные структурные черты. Попытки выявить похожие схемы или структуры в рамках ​*объектно­-ориентированного анализа и проектирования*​ привели к появлению понятия паттерна. Паттерны различаются степенью детализации и уровнем абстракции. Классификация паттернов по категориям их применения:

***Архитектурные паттерны***​­ - множество предварительно определенных подсистем со спецификацией их ответственности, правил и базовых принципов установления отношений между ними. Эти паттерны относятся к уровню системы и подсистем, но не к уровню классов. Как правило, формулируются в обобщенной форме, используют обычную терминологию и не зависят от области приложения.

***Паттерны проектирования*** ​­- специальные схемы для уточнения структуры подсистем или компонентов программной системы и отношений между ними. Описывают общую структуру взаимодействия элементов программной системы, которые реализуют исходную проблему проектирования в конкретном контексте. Эти​ ​паттерны не зависят от языка реализации, но их реализация зависит от области приложения.

***Паттерны анализа*** ​­- специальные схемы для представления общей организации процесса моделирования. Относятся к одной или нескольким предметным областям и описываются в терминах ​*предметной области*​. В дальнейшем *паттерны*​ *анализа* ​конкретизируются в типовых моделях с целью выполнения аналитических оценок или имитационного моделирования бизнес­-процессов.

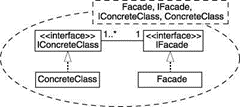
***Паттерны тестирования*** ​­ специальные схемы для представления общей организации процесса тестирования программных систем. К нему относятся : тестирование черного ящика, белого ящика, отдельных классов, системы.

***Паттерны реализации*** - совокупность компонентов и других элементов реализации, используемых в структуре модели при написании программного кода. Его подкатегории: паттерны организации программного кода, паттерны ​*оптимизации программного кода*​, паттерны устойчивости кода, паттерны разработки графического интерфейса пользователя и др. *Паттерн*​ проектирования в контексте языка ​*UML*​ представляет собой параметризованную кооперацию вместе с описанием базовых принципов ее использования.

При изображении паттерна используется обозначение параметризованной кооперации языка ​*UML*​, которая обозначается пунктирным эллипсом. В правый верхний угол эллипса встроен пунктирный ​*прямоугольник*​, в котором перечислены параметры кооперации, которая представляет тот или иной ​*паттерн*​. В последующем параметры паттерна могут быть заменены различными классами, чтобы получить реализацию паттерна в рамках конкретной кооперации. Эти параметры специфицируют используемые классы в форме ролей классов в рассматриваемой подсистеме. При связывании или реализации паттерна любая линия помечается именем параметра паттерна, которое является именем роли соответствующей ассоциации. В ​*дополнение*​ к диаграммам кооперации особенности реализации отдельных паттернов представляются с помощью диаграмм последовательности.

**26. ​Паттерн Фасад и его обозначение в нотации языка UML**

*Паттерн*​ Фасад предназначен для замены нескольких разнотипных интерфейсов доступа к определенной подсистеме некоторым унифицированным интерфейсом.



Пунктирная линия со стрелкой в форме треугольника служит для обозначения отношения реализации.

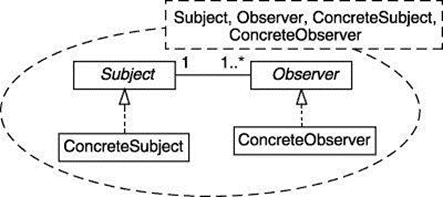
Пример для выполнения операций по заданию и считыванию адресов из ​*базы данных*​ сотрудников. Фрагмент соответствующей *диаграммы классов*​ содержит 2 класса: *Адрес*​ и ​*интерфейс*​ к операциям этого класса IАдрес. Очевидно, отслеживать при каждом обращении правильность выполнения этих последовательностей операций неудобно. С этой целью к данному фрагменту следует добавить еще один ​*интерфейс*​, реализацию паттерна Фасад для рассматриваемой ситуации. Соответствующий фрагмент модифицированной ​*диаграммы классов*​ будет содержать 4 класса, изображенные таким образом, чтобы иллюстрировать реализацию параметрической кооперации.

При задании адреса нового сотрудника в этом случае достаточно обратиться к интерфейсу IФасад и выполнить единственную операцию: задатьАдрес(), используя в качестве аргумента идентификационный номер нового сотрудника. Для получения информации об адресе сотрудника также достаточно обратиться к этому интерфейсу и выполнить единственную операцию: прочитатьАдрес(), используя в качестве аргумента идентификационный номер интересующего сотрудника. Реализацию данных операций следует предусмотреть в классе Фасад. Взаимодействие объектов этих классов может быть представлено с помощью диаграммы последовательности.

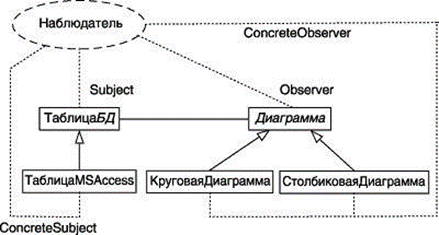
Фасад обеспечивает для клиента не только простоту доступа к информации об адресах, но и независимость представления объектов класса ​*Адрес*​ от запросов клиентов. Это актуально при изменении формата представления информации или смене соответствующей ​*базы данных*​. В этом случае потребуется внести изменения только в реализацию операций класса Фасад.

**27. ​Паттерн Наблюдатель и его обозначение в нотации языка UML**

*Паттерн*​ Наблюдатель предназначен для контроля изменений состояния объекта и передачи информации об изменении этого состояния множеству клиентов. В общем случае ​*паттерн*​ Наблюдатель также может быть изображен в виде параметризованной кооперации.



Пример для отслеживания изменений в таблице ​*БД*​ и отражении этих изменений на диаграммах. Для определенности можно использовать таблицу ​*БД*​ MS Access и две диаграммы ­ круговую и столбиковую. Фрагмент соответствующей ​*диаграммы классов*​ содержит 5 классов:



В этом случае за субъектом Таблицей MS Access может "следить" ​*произвольное*​ число наблюдателей, причем их добавление или удаление не влияет на ​*представление*​ информации в ​*БД*​. ​*Класс*​ ​*Таблица*​ MS Access реализует ​*операции*​ по отслеживанию изменений в соответствующей таблице, и при их наличии сразу информирует абстрактного наблюдателя. Тот в свою ​*очередь*​ вызывает ​*операции*​по перерисовке соответствующих диаграмм у конкретных наблюдателей, в качестве которых выступают классы Круговая ​*Диаграмма*​и Столбиковая ​*Диаграмма*​.

Использование паттерна Наблюдатель упрощает взаимодействие между объектами соответствующих классов и позволяет вносить изменения в реализацию операций классов субъекта и наблюдателей независимо друг от друга.

**28. Создание модели предметной области программной системы с помощью диаграммы классов.**

**Диаграмма классов** — диаграмма, демонстрирующая классы системы, их атрибуты, методы и взаимосвязи между ними. Входит в UML.

**Классом** называется именованное описание совокупности объектов с общими атрибутами, операциями, связями и семантикой. **Атрибутом** класса называется именованное свойство класса, описывающее множество значений, которые могут принимать экземпляры этого свойства. **Методом** класса называется именованная услуга, которую можно запросить у любого объекта этого класса. Метод – это абстракция того, что можно делать с объектом. Существуют разные точки зрения на построение диаграмм классов в зависимости от целей их применения. Для описания модели предметной области с помощью диаграммы классов используется **Концептуальная точка зрения**. В этой модели присутствуют только классы прикладных объектов. Концептуальная точка зрения используется на начальных этапах моделирования и разработки.

В концептуальной точке зрения диаграммы классов служат для представления понятий изучаемой предметной области. Эти понятия будут соответствовать реализующим их классам, но прямое соответствие может отсутствовать. *Концептуальная модель* может иметь слабое отношение или вообще не иметь никакого отношения к реализующему ее программному обеспечению, поэтому ее можно рассматривать без привязки к какому-­то языку программирования. Диаграмма Классов будет отображать классы без атрибутов и операций, а также отношения между классами**. Эти диаграммы полезны, когда проект находится в стадии анализа.**

**На этапе проектирования** диаграмма классов используется для проектирования подсистем и иерархии классов.

Одна или несколько диаграмм классов описывают классы верхнего уровня.

При включении на диаграмму пакетов в модель добавляются диаграммы классов, описывающие содержимое пакетов. Пакеты обычно используют для группирования классов по подсистемам. Кроме классов, в подсистемы могут включаться реализации вариантов использования, интерфейсы и другие подсистемы.

Разделение на подсистемы значительно упрощает параллельную разработку, конфигурирование и инсталляцию конечного продукта.

Создание подсистем позволяет проще устанавливать различные категории доступа к информации для пользователей, а также отделить алгоритмы для организации связи с внешними продуктами.

**29. Создание модели анализа с помощью диаграммы классов. Различный стереотипы для классов и их назначение.**

*Анализ требований.* После определения требований и контекста, в котором будет работать система, наступает этап анализа полученных данных. В процессе анализа создается аналитическая модель, которая подводит разработчиков к архитектуре будущей системы. Аналитическая модель – это взгляд на систему изнутри, в отличие от модели прецедентов, которая показывает, как система будет выглядеть снаружи. Эта модель позволяет понять, как система должна быть спроектирована, какие в ней должны быть классы и как они должны взаимодействовать между собой. Основное ее назначение – определить направление реализации

функциональности, выявленной на этапе сбора требований и сделать набросок архитектуры системы.

Модель анализа является в большей степени концептуальной моделью и только приближает разработчиков к классам реализации. Для отображения модели анализа при помощи UML используется диаграмма классов с образцами поведения (стереотипами) «граничный класс», «сущность», «управление», а для детализации используется Collaboration Diagram.

Диаграмма Class выполняет целый ряд функций: используется для создания иерархии классов; на ее основе строятся модели данных, и проектируется структура Web приложений; на этапе анализа и проектирования используется для создания диаграмм реализации прецедентов; с ее помощью создается модель предметной области, которая используется на этапе анализа. Кроме того, широко применяются стереотипы классов, позволяющие адаптировать стандартную UML диаграмму для конкретных целей, расширяя ее возможности. Рассмотрим основные этапы разработки системы с помощью диаграммы классов.

Для создания модели используются три стереотипа классов, которые определяют их назначение: *граничный класс* (boundary), *сущность* (entity), *управление* (control).

Стереотип *граничный класс* показывает, что класс предназначен для взаимодействия с внешними актерами и стоит на границе системы, поэтому и называется граничным. Такой класс, получая сообщение от внешнего актера, транслирует их внутрь системы, генерируя и передавая соответствующие сообщения другим классам.

Классы *сущности* используются для моделирования классов, которые отвечают за хранение определенной информации. Эти классы реализуют возможности по получению, изменению и сохранению информации в базе данных. Классы сущности обычно не отражаются ни на одной диаграмме прецедентов, но требуются для выполнения внутреннего хранения данных.

Классы *управления* используются для координации работы других классов приложения. Поведение этих классов обычно реализует один или несколько прецедентов, показанных на диаграммах моделирования. Классы управления реализуют поведение системы при помощи потоков управления. Они являются промежуточными звеньями между граничными классами и классами сущностями.

**30. Декомпозиция системы. Правила выделения подсистем.**

Метод проектирования удовлетворяет критерию Декомпозиции, если он помогает разложить задачу на несколько менее сложных подзадач, объединяемых простой структурой, и настолько независимых, что в дальнейшем можно отдельно продолжить работу над каждой из них.

Такой процесс часто будет циклическим, поскольку каждая подзадача может оказаться достаточно сложной и потребует дальнейшего разложения.

Следствием требования декомпозиции является разделение труда (division of labor): как только система будет разложена на подсистемы, работу над ними следует распределить между разными разработчиками или группами разработчиков. Это трудная задача, так как необходимо ограничить возможные взаимозависимости между подсистемами:

* Необходимо свести такие взаимозависимости к минимуму; в противном случае разработка каждой из подсистем будет ограничиваться темпами работы над другими подсистемами.
* Эти взаимозависимости должны быть известны: если не удастся составить перечень всех связей между подсистемами, то после завершения разработки проекта будет получен набор элементов программы, которые, возможно, будут работать каждая в отдельности, но не смогут быть собраны вместе в завершенную систему, удовлетворяющую общим требованиям к исходной задаче.

Типичным контрпримером (counter-example) является любой метод, предусматривающий включение в разрабатываемую систему модуля глобальной инициализации. Многие модули системы нуждаются в инициализации - открытии файлов или инициализации переменных.

Каждый модуль должен произвести эту инициализацию до начала выполнения непосредственно возложенных на него операций. Могло бы показаться, что все такие действия для всех модулей системы неплохо сосредоточить в одном модуле, который проинициализирует сразу все для всех. Подобный модуль будет обладать хорошей "согласованностью во времени" (temporal cohesion) в том смысле, что все его действия выполняются на одном этапе работы системы. Однако для получения такой "согласованности во времени", придется нарушать автономию других модулей. Придется модулю инициализации дать право доступа ко многим структурам данных, принадлежащим различным модулям системы и требующим специфических действий по их инициализации. Это означает, что автор модуля инициализации должен будет постоянно следить за структурами данных других модулей и взаимодействовать с их авторами. А это несовместимо с критерием декомпозиции.

В объектно-ориентированном методе каждый модуль должен самостоятельно инициализировать свои структуры данных.

Наиболее очевидным примером, удовлетворяющим критерию декомпозиции, является метод нисходящего (сверху вниз) проектирования (top-down design). В соответствии с этим методом разработчик должен начать с наиболее абстрактного описания функции, выполняемой системой. Затем последовательными шагами детализировать это представление, разбивая на каждом шаге каждую подсистему на небольшое число более простых подсистем до тех пор, пока не будут получены элементы с настолько низким уровнем абстракции, что становится возможной их непосредственная реализация. Этот процесс можно представить в виде дерева.

**31.Особенности создания шаблона приложения в среде Rational Rose с использованием библиотеки MFC.Структура и классы приложения.**

Будем рассматривать процесс создания приложения, как логическое продолжение построения кода класса. Поэтому в нашем распоряжении находятся язык VC++ и библиотека MFC. Мастер создания приложений VC++ (AppWizard) может строить несколько типов приложений:

**Single document** – приложение работает с одним документом; **Multiple document** – приложение работает с несколькими документами;

**Dialog based** – приложение основано на окне диалога. Для приложения, работающего с одним документом, мастер строит код следующих классов: главный класс приложения C\*\*\*App; класс документа C\*\*\*Doc; класс просмотра C\*\*\*View; класс для окна «О программе» CAboutDlg; класс основного окна программы CMainFrame.

Все приложения VC++ **MFC являются объектами**. Поэтому приложение – это главный класс, который включает в себя все необходимые для работы классы. Соблюдая соглашение об именах, мастер создает главный класс приложения с именем проекта, прибавляя к нему в начале букву С, а в конце App (в нашем случае это – C\*\*\*App, где \*\*\* – имя проекта). C\*\*\*App наследуется из библиотечного класса CWinApp. Класс документа C\*\*\*Doc, в котором должна проходить обработка данных, наследуется из библиотечного класса CDocument. Класс просмотра C\*\*\*View, отображающий данные на экране компьютера, наследуется из библиотечного класса CEditView. Таким образом, на основе стандартных классов документа, предоставляемых MFC, строится приложение, в котором необходимо будет только *добавить функциональность*

Ассоциация проекта Rational Rose с проектом VC++ выполняется аналогично тому, как это делалось для одного класса. С помощью Component Assigment Tool в строку VC++ перетаскиваются все необходимые классы. Для простоты работы все классы, для которых необходимо создание исходного кода, помещаются в один компонент, заключенный в проект \*\*\*. Библиотека классов MFC импортируется в модель путем следующих действий: Menu:Tools=>Visual C++=>Quick Import MFC 6.0. Затем в модель Rational Rose можно загрузить перечисленные ранее классы, созданные в VC++. Для того чтобы они появились в проекте, необходимо обновить проект по готовому коду, выполнив действия: Menu:Tools => Visual C++=> Update Model From Code. После обновления в модели Rational Rose и в проекте VC++ содержатся одинаковые наборы классов. Теперь можно запустить Visual Studio, перейти в нужный проект, откомпилировать его и получить результат работы программы.

Одно из неоспоримых преимуществ Rational Rose – обратное проектирование, поскольку разработчику и проектировщику важно увидеть перед изменениями уже работающую систему в нормальном графическом представлении. Проект, подвергшийся обратному проектированию, может быть доработан и вновь сгенерирован (а впоследствии и скомпилирован). Для осуществления обратного проектирования в Rational Rose предусмотрен мощный модуль – Analyzer, чье основное предназначение, вытекающее из названия, анализ программ, написанных на С и С++. Далее файл можно спокойно открыть для модификации из Rational Rose уже в визуальном режиме. Analyzer представляет собой отдельный программный файл, вызываемый как из самой Rose, так и обычным способом. Модуль входит не во все поставки Rational Rose, а только в Enterprise, Professional и RealTime.

**32.Функциональные возможности** ​***Rational Rose***​**: модуль *Component Assignment Too***​**l, компонент** ​***Model Assistant***​**, обновление кода по модели и модели по коду.**

*Component Assignment Too*​l

Для того чтобы использовать класс в программном проекте, необходимо его ассоциировать с выбранным языком, в нашем случае с VC++. Для этого проделаем следующее

Menu: Tools=>Visual C++=>Component Assigned Tools Активизирует диалоговое окно назначения классов в компоненты и назначения языка для класса. Это окно предоставляет возможность создания новых компонентов в модели, ассоциации компонентов с проектами на конкретных языках программирования и назначения классов в компоненты. Для того чтобы получить преимущества использования данного инструмента, необходимо создавать компоненты здесь, а не через окно Browser или в диаграмме компонентов. При этом созданные компоненты будут содержать всю необходимую информацию для генерации кода на выбранном языке программирования. Данное средство позволяет просмотреть классы, которые еще не назначены в компоненты, что уменьшает вероятность ошибки. Component Assignment Tool может быть открыт как посредством меню Tools, так и из контекстного меню компонента в окне Code Update Tool.

Model Assistant позволяет обновлять и конкретизировать классы в модели, используя дополнительные ключевые слова C++ для необходимой генерации кода. Model Assistant представляет собой окно, позволяющее создавать атрибуты и операции и изменять их свойства, причем значительно проще и нагляднее, чем это происходит для C++.

Инструмент Model Assistant Tool отображает элементы моделирования из программы Rational Rose на конструкции языков Visual С++ или Visual Basic. Для языка Visual Basic программа Model Assistant Tool способна создавать и определять константы, операторы, события, типы, перечисления, свойства, методы и параметры методов. Она позволяет создавать процедуры типа Get, Let и Set для свойств класса и ролей ассоциативной связи, а также определенный пользователем класс для наборов и списков. На языке Visual С++ программа Model Assistant Tool может создавать и определять операции для классов, такие как конструкторы, деструкторы и операции доступа для атрибутов и отношений.

В поле Preview (Предварительный просмотр) отображается код, который будет получен для выбранного элемента. Это позволяет увидеть, как повлияют на код настройки параметров генерации. Возвратное проектирование

После внесения изменений и дополнений в код на Visual С++ или Visual Basic необходимо обновить модель, чтобы отразить в ней сделанные изменения. Это выполняется с помощью мастера Model Update Tool. Его можно использовать и для создания новой модели по исходному коду.

Если в проект добавлены классы, например, при помощи Class Wizard, и они еще не отражены в модели Rational Rose. В этом случает необходимо провести обновление кода при помощи функции Update Code (обновить код).

По завершении возвратного проектирования мастер Model Update Tool отображает диалоговое окно с итоговым отчетом. Вкладка Summary содержит сведения об обновленной модели, а все ошибки, возникшие при обновлении, можно увидеть на вкладке Log.

**33. Особенности генерации исходного кода в среде Rational XDE. Способы синхронизации модели.**

Генерация по готовым диаграммам сокращает количество ошибок на стадии преобразования готовой модели в программный код. В любой программе, написанной при помощи среды Microsoft .NET, можно провести обратное проектирование и разобраться в архитектуре, представленной графически в виде иерархии классов.

Rational XDE предоставляет большое количество настроек для автоматической и ручной генерации кода приложения. Все установки по ­разному влияют на генерацию кода и призваны повышать продуктивность работы. Для работы с исходным кодом в доступна следующая группа пунктов меню:

* **Generate Code** – генерация кода по созданной модели;
* **Synchronize** – синхронизация кода и модели. При этом все изменения, внесенные в код, отражаются в модели, а изменения модели переносятся в исходный код;
* **Browse Code** – позволяет переключаться в режим просмотра созданного кода. Для работы с одним ­двумя классами этого вполне достаточно, но при работе с большими моделями удобнее использовать синхронизацию в автоматическом режиме. При создании архитектуры классов приложения используются различные типы связей. Одни не требуют настройки, другие обладают значительным количеством свойств, влияющих на исходный код. При создании исходного кода ассоциативные связи создают одинаковый код независимо от того, направленная это ассоциация или нет, отражает ли связь агрегацию или композицию классов. При генерации кода будет создана переменная класса, заданная в параметрах связи.

При этом направление связи не играет роли, а важно задание имени переменной. При создании связи Directed Association Rational XDE автоматически создает переменную класса того типа, который находится в окончании стрелки связи. Эта переменная включается в класс, из которого стрелка исходит.

Установки, созданные при построении связи, несложно изменить с помощью окна Properties. ​*Для настройки*

*параметров автоматической синхронизации модели необходимо выбрать из главного меню пункт:*

Tools => Options => Rational XDE => Round Trip Engineering => Synchronization Setting.

После выбора эл­тов в Automatic Synchronization становятся доступными следующие варианты синхронизации:

– When Saving Model Files – синхронизация в момент сохранения модели;

– When Model gets Focus – синхронизация в момент активизации модели;

– When saving Code Files – синх. происх. в момент сохр. кода после его изменения;

– When Code gets Focus – синх. происходит в момент активизации окна кода.

Каждый вариант синхронизации удобен для своего случая. На этапе анализа и проектирования, когда основную работу по моделированию выполняет аналитик, синхронизация в момент сохранения модели позволит программистам, работающим в проекте, пользоваться самой последней версией структуры приложения. На этапе реализации, когда основная работа ложится на программистов, установка синхронизации после сохранения внесения изменений в файлы кода позволит поддерживать модель системы в актуальном состоянии. Когда же разработка закончена или создается новая версия уже работающей системы, чтобы не нарушить работу программы, синхронизация может быть вообще отключена.

**34. Сравнительный анализ процедур генерации исходного кода в Rational Rose, Rational XDE, Enterprise Architect.**

Дело в том, что традиционные решения на базе Rationаl Rose позволяют проводить прямое и обратное проектирование кодов для многих языков программирования, т. е. возможна прямая генерация кода, обратная (так называемый реверс­инжиниринг) и постоянная модификация кода и модели с поддержанием их в актуальном состоянии (round­trip engineering). Однако применение этих инструментов ограничивалось тем, что они не допускали генерацию в реальном масштабе времени: код можно было получать, но при этом нужно было выполнить определенные дополнительные действия по встраиванию его в систему разработки. Кроме того, проблемы возникали и при постановке сгенерированных (перегенерированных) модулей под версионное управление.

Новый инструмент XDE Developer решает эти задачи, реализуя функцию поддержания кода и модели в актуальном состоянии при работе в реальном масштабе времени. Таким образом, нарисовав диаграмму классов с иерархической структурой, можно тут же сгенерировать код со всеми связями, атрибутами и методами. Принципиальное отличие XDE от Rose Professional заключается в том, что XDE работает со всеми типами диаграмм из UML 1.3. В то же время XDE Developer пока не покрывает полностью функциональность Rose, которая, в частности, предоставляет возможность работы с более широким кругом языков программирования, обладает более развернутыми функциями проектирования данных.

**35. Назначение, возможности, особенности использования модуля** ​***Web Modeler***​ **в *Rational Rose***​​**.**

Подключаемые модули (Add­Ins) позволяют значительно расширить применение Rational Rose не только для моделирования поведения приложений, но и для создания модели и разработки структуры Web сайтов. Web сайту присущи многие атрибуты настольного приложения. Для моделирования Internet приложений может использоваться обычная диаграмма классов.

Web­Modeler обеспечивает возможность строить диаграммы классов при проектировании WEB­ приложений с учетом специфики WEB ­программирования (классами являются страницы и скрипты, предусмотрены отношения передачи данных из формы, генерации страницы скриптом и т,д,). Дает возможность для предусмотренных платформ ASP, JSP:

* построить модель классов по коду;
* создать код по модели классов.

Для того чтобы начать работу с Web Modeler, необходимо установить его при помощи Add­Ins=>Add­Ins Manager=>Rose

Web Modeler=ON.

После этого в меню Tools появится новый пункт Web Modeler.

В подменю Web Modeler есть 2 подпункта:

1. «Установки пользователя» служит для настройки генератора кодов при построении кода по модели. Данный пункт предназначен для изменения установок модели, принятых по умолчанию для конкретного пользователя, что позволяет нескольким пользователям работать с одной Web моделью, но со своими настройками. Эти установки позволяют контролировать построение модели по коду, создание кода по модели, генерировать структуру каталога для ссылок, связей и HTML форм. При вызове этого пункта активизируется окно настройки параметров.

Установки позволяют настроить действия генератора для тегов HTML и скриптов.

2. «Reverse Engineer a New Web Application(Построение модели по коду)» позволяет создать модель по коду. При его активизации запускается мастер Reverse Engineer. Мастер позволяет задать платформу ASP (Active Server Pages) или JSP(Java Server Pages) и каталог, в котором находится исходный код. После того как мастер закончит свою работу в текущей модели, образуется структура выбранных классов, в которой отражаются связи, атрибуты и стереотипы. И то, и другое выполняется для указанных выше платформ. Для моделирования Web приложения используется диаграмма классов. ​Классы имеют специальные стереотипы, которые влияют не только на изображение класса на диаграмме, но и дополняют контекстное меню класса еще одной строкой спецификаций и пунктом Web Modeler. Создавать классы необходимо в Web нотации.

Для того, чтобы создавать классы в Web нотоции:

Menu:Tools=> Oplions=>Notation=>Default Language=Web

Notation.

Rational Rose предоставляет ряд стереотипов классов:

– **Server page (скрипты, выполняемые на сервере)**

**– Client page (использование скриптов на стороне клиента)**

**– HTML Form. (позволяют взаимодействовать пользователю с веб приложением, заполненная форма отправляется на сервер тот возвращает html-страницу)**

Для связей страниц и форм Web Modeler предоставляет дополнительные стереотипы, которые позволяют отразить характер связей. **- Link** показывает гиперссылку одной страницы на другую и может быть двунаправленной. **- Submit** определяет, что форма взаимодействует со страницей на сервере и передает ей данные. **- Build** показывает, что страница создается сервером. **- Redirect** определяет передачу управления одной серверной странице от другой и используется для страниц ASP. **- Includes** означает, что одна страница включается в другую**. - Forward** похожа на связь Redirect, только используется для страниц JSP

**36. Возможности и особенности построения** ​***Web***​**­ модели в среде** ​***Rational XDE***​**.**

В Rational XDE для создания структуры системы, ориентированной на работу в Web, используется диаграмма классов, в которой при создании архитектуры приложения учитываются ограничения реализации Web­ приложения. В спецификации UML не предусмотрен отдельный тип диаграмм для выполнения этой задачи. Вполне достаточно диаграммы классов с дополнительными стереотипами для работы с Web­ элементами.

Рассмотрим возможности Rational XDE по созданию Web­модели.

**Client Page** позволяет создать на диаграмме отображение простых страниц HTML, не имеющих собственного поведения. Обычно такие страницы предоставляют пользователям определенную, заранее заданную информацию. Страницы Client Page так же, как и классы, могут содержать атрибуты и операции, которые добавляются посредством контекстного меню элемента, после чего код обновляется автоматически или вручную.

**Link Relation** позволяет отобразить связи между страницами в том случае, когда на одной странице есть ссылка на другую. Rational XDE не знает, куда добавлять созданную ссылку, и вставляет ее в конец файла.

**HTML Form** позволяет отобразить формы ввода, которые присутствуют на страницах HTML. Форма не может существовать сама по себе, она включается на страницу при помощи агрегирования. Поэтому для ее разработки начинают с создания страницы, на которой будет находиться форма. Сначала форма соединяется со страницей связью Link, а затем посредством пункта Properties Window из контекстного меню связи значение свойства UML=>Kind изменяется на Agregation. Для добавления полей в форму можно воспользоваться пунктом ее контекстного меню Add Web. В случае необходимости отразить обработку данных, передаваемых из формы клиентской или серверной странице, используется значок связи Submit Relation (отношение предоставления). Для этого используются Server Page, которые и реализуют генерацию страниц для пользователя, что отображается при помощи связи Build Relation. Таким образом, Server Page являются связующим звеном между классами приложения и их визуальным отображением. При помощи Server Page with Code­Behind создается набор элементов, связанных между собой и содержащих необходимые элементы для создания ASP.NET приложения. Для отражения связей между страницами ASP используется значок NET­Link Relation. Если форма, расположенная на ASP странице, использует элементы управления, созданные пользователем, то значок NETRegister Relation позволяет отразить связи между страницей ASP и элементом управления пользователя.

Для отражения передачи управления другой странице используется NETTransfer Relation. При генерации кода создается директива Transfer, которая позволяет передавать управление другой странице с сохранением доступа к внутренним объектам исходной страницы.

NETExecute позволяет отразить передачу управления другой странице, но при генерации кода создается директива Execute, позволяющая не только передать управление с сохранением доступа к внутренним объектам, но и по завершении вернуть управление вызывающей странице. Для отражения простой переадресации с одной страницы на другую используется связь при помощи значка Redirect Relation. При этом не сохраняется доступ к внутренним объектам, как это происходит при использовании связей NETTransfer и NETExecute. Такая переадресация используется при необходимости активизации страницы, чье изображение зависит от установленного языка или возможностей браузера.

**37. Понятие проектных рисков. Действия по управлению рисками**

Проектный риск – это осознанная и зафиксированная возможность возникновения ущерба и потерь. Влияние риска вычисляют по выражению RE = P(UO) x L(UO), где:

RE — *показатель риска,* подверженность риску;

P(UO) — *вероятность неудачного результата*

L(UO) — *потеря при неудачном результате.*

**Управление рисками** — процесс принятия и выполнения управленческих решений, направленных на снижение вероятности возникновения неблагоприятного результата и минимизацию возможных потерь проекта, вызванных его реализацией.

Управление риском включает шесть действий:

1. **Идентификация риска** — выявление элементов риска в проекте.
2. **Анализ риска** — оценка вероятности и величины потери по каждому элементу риска.
3. **Ранжирование риска —** упорядочение элементов риска по степени их влияния.
4. **Планирование управления риском** — подготовка к работе с каждым элементом риска.
5. **Разрешение риска** — устранение или разрешение элементов риска.
6. **Наблюдение риска** — отслеживание динамики элементов риска, выполнение корректирующих действий. Первые три действия относят к этапу оценивания риска, последние три действия — к этапу контроля риска.

Выделяют три категории источников риска: проектный риск, технический риск, коммерческий риск.

**В ходе анализа** оценивается вероятность возникновения Рi и величина потери Li для каждого выявленного i­го элемента риска. В результате вычисляется влияние REi i­го элемента риска на проект. Вероятности определяются с помощью экспертных оценок или на основе статистики, накопленной за предыдущие разработки.

Итоги анализа сводятся в таблицу, которая содержит след. поля:

*элемент риска,*

*вероятность,*

*потери,*

*влияние риска*

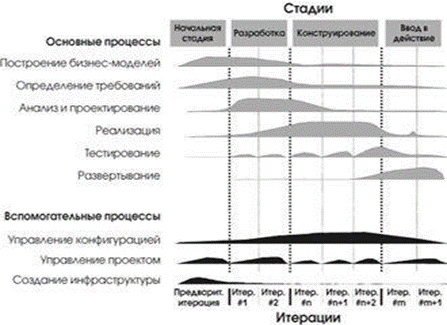
**Ранжирование** заключается в назначении каждому элементу риска приоритета, который пропорционален влиянию элемента на проект. Это позволяет выделить категории элементов риска и определить наиболее важные из них.

Для больших проектов количество элементов риска может быть очень велико (30­40 элементов). В этом случае управление риском затруднено. Поэтому к элементам риска применяют принцип Парето 80/20. Опыт показывает, что 80% всего проектного риска приходятся на долю 20% от общего количества элементов риска. В ходе ранжирования определяют эти 20% элементов риска (их называют существенными элементами). В дальнейшем учитывается влияние только существенных элементов риска.

**38. Статический и динамический аспекты** ​***RUP***​**.**

RUP организует работу над проектом в терминах статических и динамических аспектов процесса.

К статическим аспектам относятся *последовательности действий, продуктов деятельности, исполнителей*, а к динамическим – *циклы, фазы, итерации и временные отметки завершения определенных этапов в создании ПО*. В графическом виде процесс можно представить следующим образом. Вдоль горизонтальной оси будем откладывать его динамические аспекты, а вдоль вертикальной оси – статические аспекты процесса.



Каждая из фаз процесса разработки состоит из нескольких **итераций**, целью которых является последовательное осмысление стоящих проблем, наращивание эффективных решений и снижение риска потенциальных ошибок в проекте.

**Статический аспект RUP** представлен четырьмя основными элементами:

* *Роли* - поведение и ответственность личности или группы личностей, составляющих проектную команду. Одна личность может играть в проекте много различных ролей;
* *виды деятельности* - единица выполняемой им работы;
* *рабочие продукты*;
* *Дисциплины* - технологического процесса и представляет собой последовательность действий, приводящую к получению значимого результата.

В рамках RUP определены шесть основных дисциплин: построение бизнес-моделей; определение требований; анализ и проектирование; реализация; тестирование; развертывание. Имеется три вспомогательные дисциплины: управление конфигурацией и изменениями; управление проектом; создание инфраструктуры.

**39. Принципы и стадии разработки ПС в технологии Rational Unified Process**

Rational Unified Process (RUP) — это **методология** создания ПО, оформленная в виде размещаемой на Web базы знаний, которая снабжена поисковой системой.

В основе RUP лежит ЖЦ создания ПС. Рассмотрим, как реализуются основные этапы ЖЦ в рамках RUP.

**Определение требований.** Унифицированный процесс – это процесс, управляемый прецедентами, которые отражают сценарии взаимодействия пользователей. Фактически, это взгляд пользователей на программную систему снаружи. Таким образом, одним из важнейших этапов разработки, согласно RUP, является этап *определения требований*, сбор всех возможных пожеланий к работе системы (используют диаграммы Use Case и Activity).

Определение требований к системе напрямую связано с понимание контекста (части предметной области), в котором будет работать система. Для этого создаются **модель предметной области** (Class Diagram) и бизнес-модель. Различия этих моделей в том, что первая описывает важные понятия, с которыми будет работать система и их связи между собой.

**Анализ требований**. После определения требований и контекста, в котором будет работать система, наступает этап анализа полученных данных. В процессе анализа создается аналитическая модель, которая подводит разработчиков к архитектуре будущей системы. Аналитическая модель – это взгляд на систему изнутри. Модель показывает, как система будет выглядеть снаружи. Эта модель позволяет понять, как система должна быть спроектирована, какие в ней должны быть классы и как они должны взаимодействовать между собой (Сollaboration diagram, Sequence)

**Проектирование**. Следующим этапом в процессе создания системы будет проектирование, в ходе которого на основании моделей, созданных ранее, строится модель проектирования. Она отражает физическую реализацию системы и описывает создаваемый продукт на уровне классов и компонентов. В отличие от модели анализа, модель проектирования имеет явно выраженную зависимость от условий реализации, применяемых языков программирования и компонентов. (Сlass,Activity, Сollaboration)

**Реализация**. Основная задача процесса реализации – создание системы в виде набора компонентов. Это могут быть исходные тексты программ, сценарии, двоичные файлы, исполняемые модули и т.д. Модель реализации строится в виде диаграммы компонентов (Component Diagram)

**Тестирование**. В процессе тестирования проверяются результаты реализации. Для данного процесса создается модель тестирования, которая состоит из тестовых примеров, процедур тестирования, тестовых компонентов, однако не имеет отображения в виде UML диаграмм.

**40. Содержание и результаты первой и второй стадий в технологии Rational Unified Process.**

Согласно RUP, ЖЦ ПО разбивается на отдельные циклы, в каждом из которых создается новое поколение продукта. Каждый цикл, в свою очередь, разбивается на четыре последовательные стадии:

* начальная стадия (inception);
* стадия разработки (elaboration);
* стадия конструирования (construction);
* стадия ввода в действие (transition).

Каждая стадия завершается в четко определенной контрольной точке (milestone). В этот момент времени должны достигаться важные результаты и приниматься критически важные решения о дальнейшей разработке.

**Первая стадия (начальная стадия)**

Начальная стадия может принимать множество разных форм. Для крупных проектов она связана с всесторонним изучением всех возможностей реализации проекта. В это же время вырабатывается бизнес-­план проекта: определяется, сколько приблизительно он будет стоить и какой доход принесет. Кроме того, выполняется начальный анализ для оценки размеров проекта.

Результатами начальной стадии являются: ­

* общее описание системы:
* основные требования к проекту, его характеристики и ограничения;
* начальная модель вариантов использования;
* начальный проектный глоссарий;
* начальный бизнес­план;
* план проекта, отражающий стадии и итерации;
* один или несколько прототипов.

**Вторая стадия (стадия разработки)**

На стадии разработки выявляются более детальные требования к системе, выполняется высокоуровневый анализ предметной области и проектирование для построения базовой архитектуры системы, создается план конструирования и устраняются наиболее рискованные элементы проекта.

Результатами стадии разработки являются:

* завершенная модель вариантов использования (use case), определяющая функциональные требования к системе;
* перечень дополнительных требований, включая требования нефункционального характера и требования, не связанные с конкретными вариантами использования;
* описание базовой архитектуры будущей системы;
* работающий прототип;
* уточненный бизнес­план;
* план разработки всего проекта, отражающий итерации и критерии оценки для каждой итерации.

Самым важным результатом стадии разработки является **описание базовой архитектуры** будущей системы. Созданная архитектура является основой всей дальнейшей разработки. Стадия разработки занимает около пятой части общей продолжительности проекта. *Основными признаками ее завершения являются следующие:*

* разработчики в состоянии оценить, сколько времени потребуется на реализацию каждого варианта использования; ­
* идентифицированы все наиболее серьезные риски и степень понимания наиболее важных из них такова, что известно, как справиться с ними.

**41. Содержание и результаты третей и четвертой стадий в технологии** ​***Rational Unified Process***​**.**

Согласно RUP, ЖЦ ПО разбивается на отдельные циклы, в каждом из которых создается новое поколение продукта. Каждый цикл, в свою очередь, разбивается на четыре последовательные стадии:

* начальная стадия (inception);
* стадия разработки (elaboration);
* стадия конструирования (construction);
* стадия ввода в действие (transition).

Каждая стадия завершается в четко определенной контрольной точке (milestone). В этот момент времени должны достигаться важные результаты и приниматься критически важные решения о дальнейшей разработке.

**Стадия конструирования** напрямую связана с проработкой итераций. Они на стадии конструирования являются одновременно инкрементными и повторяющимися. Инкрементность связана с добавлением *новых конструкций к вариантам использования*, реализованным во время предыдущих итераций. Повторяемость относится к разрабатываемому коду: на каждой итерации некоторая часть существующего кода переписывается с целью сделать его более гибким.

Результатом стадии конструирования является продукт, готовый к передаче конечным пользователям и содержащий следующее:

* ПО, интегрированное на требуемых платформах;
* руководства пользователя;
* описание текущей реализации.

**Стадия ввода в действие** связана с передачей готового продукта в распоряжение пользователей. Она включает в себя:

* бета-тестирование, позволяющее убедиться, что новая система соответствует ожиданиям пользователей;
* параллельное функционирование с существующей системой, которая подлежит постепенной замене;
* конвертирование баз данных;
* оптимизацию производительности;
* обучение пользователей и специалистов службы сопровождения.

**42. Этапы создания ПС в технологии** ​***Oracle***​**.**

В технологии Oracle существует понятие Custom Development Method (CDM) – разработка прикладного ПО.

Метод CDM оформлен в виде консалтингового продукта CDM Advantage – библиотеки стандартов и руководств. Он представляет собой развитие достаточно давно созданного Oracle CASE Method, известного по использованию CASE­-средств фирмы Oracle и книгам Р. Баркера. По существу, CDM является методическим руководством по разработке прикладного ПО с использованием инструментального комплекса Oracle Developer Suite, а сам процесс проектирования и разработки тесно связан с Oracle Designer и Oracle Forms.

В соответствии с CDM ЖЦ ПО формируется из определенных этапов (фаз) проекта и процессов, каждый из которых выполняется в течение нескольких этапов (рис. 2):

* стратегия (определение требований);
* анализ (формулирование детальных требований к системе);
* проектирование (преобразование требований в детальные спецификации системы);
* реализация (написание и тестирование приложений);
* внедрение (установка новой прикладной системы, подготовка к началу эксплуатации);
* эксплуатация.

**На этапе ​*стратегии*** определяются цели создания системы, приоритеты и ограничения, разрабатывается системная архитектура и составляется план разработки.

**На этапе анализа** строится модель информационных потребностей (диаграмма «сущность-связь»), диаграмма функциональной иерархии (на основе функциональной декомпозиции системы), матрица перекрестных ссылок и диаграмма потоков данных.

**На этапе ​*проектирования*** разрабатывается подробная архитектура системы, проектируются схема реляционной БД и программные модули, устанавливаются перекрестные ссылки между компонентами системы для анализа их взаимного влияния и контроля за изменениями.

**На этапе *реализации* создается БД**, строятся прикладные системы, производится их тестирование, проверка качества и соответствия требованиям пользователей. Создается

системная документация, материалы для обучения и руководства пользователей.

**На этапах ​*внедрения* ​и ​*эксплуатации***​анализируются производительность и целостность системы, выполняется​ поддержка и, при необходимости, модификация системы.

**43. Процессы создания ПС в технологии** ​***Oracle.***

Методическую основу ТС ПО корпорации Oracle составляет метод Oracle (Oracle Method) комплекс методов, охватывающий большинство процессов ЖЦ ПО. В состав комплекса входят:

● CDM (Custom Development Method) ­ разработка прикладного ПО;

● PJM (Project Management Method) ­ управление проектом;

● AIM (Application Implementation Method) ­ внедрение прикладного ПО;

● BPR (Business Process Reengineering) ­ реинжиниринг бизнес­процессов;

● OCM (Organizational Change Management) ­ управление изменениями, и др.

В соответствии с CDM ЖЦ ПО формируется из определенных этапов (фаз) проекта и процессов, каждый из которых выполняется в течение нескольких этапов

● стратегия (определение требований);

● анализ (формулирование детальных требований к системе);

● проектирование (преобразование требований в детальные спецификации системы);

● реализация (написание и тестирование приложений);

● внедрение (установка новой прикладной системы, подготовка к началу эксплуатации);

● эксплуатация.

Процессы CDM:

* определение бизнес-­требований, или постановка задачи;
* исследование существующих систем (Existing Systems Examination);
* определение технической архитектуры(Technical Architecture);
* проектирование и реализация базы данных (Database Design and Build);
* проектирование и реализация модулей (Module Design and Build);
* конвертирование данных (Data Conversion);
* документирование (Documentation);
* тестирование (Testing);
* обучение (Training);
* внедрение, или переход к новой системе (Transition);
* поддержка и сопровождение (Post­System Support).

PJM ­ это определенная дисциплина ведения проекта, позволяющая гарантировать, что цели проекта, четко определенные в его начале, остаются в центре внимания на протяжении всего проекта. В основе PJM лежит метод, ориентированный на выполнение самостоятельных процессов (под процессом понимается набор связанных задач, выполнением которых достигается определенная цель проекта). Так же, как и CDM, метод руководства проектом представляется в виде четко определенной операционной схемы, в которой выделяются процессы, этапы, задачи, результаты решения задач и зависимости между задачами:

● Управление проектом и предоставление отчетности (Control and Reporting).

● Управление работой (Work Management).

● Управление ресурсами (Resource Management).

● Управление качеством (Quality Management).

● Управление конфигурацией (Configuration Management).

Цикл решения задач PJM состоит из отдельных этапов. Количество этапов зависит от выбранного подхода к разработке. Задачи PJM можно распределить внутри каждого процесса по трем группам ­ задачи планирования, управления и завершения, и по уровням ­ отнести задачу на уровень проекта или на уровень отдельного этапа.

**44. Классический и быстрый подходы к разработке ПС в технологии Oracle. Факторы, определяющие выбор подхода.**

Разработка прикладного ПО(Custom Development method – CDM) предоставляет возможность выбрать требуемый подход к разработке. Это возможно, поскольку каждый процесс базируется на известных зависимостях между задачами одного типа и не зависит от того, на какие этапы будет разбит проект.

При определении подхода к разработке оценивается масштаб, степень сложности и критичность будущей системы. При этом учитываются стабильность требований, сложность и количество бизнес-­правил, количество автоматически выполняемых функций, разнообразие и количество пользователей, степень взаимодействия с другими системами, критичность приложения для основного бизнес­-процесса компании и целый ряд других.

В соответствии с этими факторами в CDM выделяются два основных подхода к разработке:

**Классический подход (Classic).** Этапы данного подхода ­ стратегия, анализ, проектирование, реализация, внедрение, эксплуатация. Классический подход применяется для наиболее сложных и масштабных проектов, он предусматривает последовательный и детерминированный порядок выполнения задач. Для таких проектов характерно большое количество реализуемых бизнес­-правил, распределенная архитектура, критичность приложения. Применение классического подхода также рекомендуется при нехватке опыта у разработчиков, неподготовленности пользователей, нечетко определенной задачи. Много времени отводится на стадию анализа и проектирования. Работы ведутся линейно, итеративность не приветствуется. Пользователь привлекается к работам в основном на этапах анализа и внедрения​. На протяжении всего проекта генерируется мощный поток документов о ходе работ, призванных убедить заказчика, что работы идут нормально. Продолжительность таких проектов от 8 до 36 месяцев. Но данный подход неэффективен, когда ​за время выполнения проекта изменяются требования заказчика к функционированию системы и когда невозможно получить постановку задачи от пользователя, предварительно не показав ему, как программа может работать.

Подход быстрой разработки (Fast Track). Данный подход, в отличие от каскадного классического, является итерационным и основан на методе DSDM (Dynamic Systems Development Method – метод для разработки динамических систем). Его цель ­- сдать готовый проект вовремя и уложиться в бюджет, но, в то же время, регулируя изменения требований к проекту во время его разработки.​

В этом подходе четыре этапа:

* стратегия;
* моделирование требований;
* проектирование и генерация системы;
* внедрение в эксплуатацию.

Преимущества:

* Непосредственное интенсивное взаимодействие разработчиков с пользователями на протяжении всего проекта.
* Жесткое соблюдение временных рамок работ при регулярно проводимой переориентации требований, позволяющей добавлять более приоритетные новые требования в обмен на менее приоритетные старые.
* Уменьшение времени на бумажную постановку.

Подход используется для реализации небольших и средних проектов с несложной архитектурой системы, гибкими сроками и четкой постановкой задач.

Продолжительность проекта от 4 до 16 месяцев.

В полном виде неэффективен у нас, т.к. наши ГОСТы также не приветствуют итеративность и к тому же требуют в Техническом проекте описания всех форм и отчетов системы.

**45. Этапы разработки ПС в технологии *Borland***​**.**

Компания Borland в результате развития собственных разработок и приобретения целого ряда компаний представила интегрированный комплекс инструментальных средств, реализующих управление полным жизненным циклом приложений (Application Life Cycle Management, ALM). *​*В соответствии с технологией Borland процесс создания ПО включает в себя пять основных этапов:

1. определение требований;
2. анализ и проектирование;
3. разработка;
4. тестирование и профилирование;
5. развертывание (это основное).

Выполнение всех этапов координируется процессом управления конфигурацией и изменениями.

**Определение требований** реализуется с помощью системы управления требованиями CaliberRM, которая стала частью семейства продуктов Borland в результате покупки компании Starbase. CaliberRM сохраняет требования в базе данных, документы с их описанием создаются с помощью встроенного механизма генерации документов MS Word на базе заданных шаблонов. Система обеспечивает экспорт данных в таблицы MS Access и импорт из MS Word. CaliberRM поддерживает различные методы визуализации зависимостей между требованиями, с помощью которых пользователь может ограничить область анализа, необходимого в случае изменения того или иного требования. Имеется модуль, который использует данные требования для оценки трудозатрат, рисков и расходов, связанных с реализацией требований.

**Средство анализа и проектирования** Together ControlCenter разработано компанией TogetherSoft. В основе его применения лежит один из вариантов подхода "Быстрой разработки ПО" под названием Feature Driven Development (FDD)

Together ControlCenter ­ интегрированная среда проектирования и разработки, поддерживающая визуальное моделирование на UML с последующим написанием приложений для платформ J2EE (Java) и .Net (С#, C++ и Visual Basic). Кроме базовой версии, имеется уменьшенный вариант системы для индивидуальных разработчиков и небольших групп (Together Solo), а также редакции для платформы IBM WebSphere и среды разработки Jbuilder.

**Разработка.** В системе реализована технология LiveSource, которая обеспечивает синхронизацию между проектом приложения и изменениями ­ при внесении изменений в исходные тексты меняется модель программы, а при изменении модели надлежащим образом изменяется текст на языке программирования. Контроль версий осуществляется благодаря функциональной интеграции Together и системы StarTeam. Поддерживается также интеграция с системой управления конфигурацией Rational ClearCase.

**Тестирование** обеспечивается с помощью инструментальных средств Optimizeit Suite 5, Optimizeit Profiler for .NET и Optimizeit ServerTrace. Первые две системы позволяют выявить потенциальные проблемы использования аппаратных ресурсов: памяти и процессорных мощностей на платформах J2EE и .Net соответственно. Интеграция Optimizeit Suite 5 в среду разработки Jbuilder, а Optimizeit Profiler – в C#Builder и Visual Basic .Net реализует проводить контрольные испытания приложений по мере разработки и ликвидировать узкие места производительности. Система Optimizeit ServerTrace предназначена для управления производительностью серверных J2EE-приложений с точки зрения достижения заданного уровня обслуживания и сбора контрольных данных по виртуальным Java-машинам

В технологии Borland выделяется 3 уровня интеграции:

**Функциональная** (touch­point) интеграция позволяет обратиться из одной системы к функциям другой, выбрав соответствующий пункт меню. Такая интеграция дает возможность разделять информацию между системами, но не обеспечивает единого рабочего пространства, вынуждает пользователя переключать окна и приводит к дублированию процессов управления структурой проекта.

**Встроенная** (embedded) интеграция обеспечивает работу с одной системой непосредственно в среде другой. Например, не выходя из среды разработки Jbuilder, можно просматривать графики производительности, которые создает система Optimizeit.

Самый высокий уровень интеграции ­ **синергетический** (synergistic), позволяющий сочетать функции двух различных продуктов незаметно для разработчиков. Для большинства продуктов Borland и других поставщиков синергетическая интеграция пока остается делом будущего, однако ее принципы уже начинают реализовываться.

**46. Сравнительный анализ технологий создания ПС *Rational Unified Process*, *Oracle*, *Borland*.**

**47. Понятие CASE-средство, CASE-система, CASE-технология, CASE-индустрия и различия между ними.**

**CASE-средство** - это инструмент, который позволяет автоматизировать один или несколько этапов процесса разработки информационной системы и программного обеспечения.

Примеры - основные компоненты CASE-систем:

* репозиторий - главное CASE-средство. Его задача - обеспечить сохранность вариантов проекта и его определенных компонентов, синхронизацию приема информации от разных разработчиков в процессе групповой разработки, проверка метаданных на полноту и непротиворечивость;
* средства разработки приложений, с использованием языков [4GL](https://www.tadviser.ru/index.php?title=4GL&action=edit&redlink=1) и генераторов кодов;
* средства тестирования;
* средства документирования;
* графические средства анализа и проектирования, которые дают возможность создавать и редактировать иерархически связанные диаграммы (например, [DFD](https://www.tadviser.ru/index.php?title=DFD&action=edit&redlink=1), [ER-диаграмма](https://www.tadviser.ru/index.php?title=ER-%D0%B4%D0%B8%D0%B0%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B0&action=edit&redlink=1) и др.), создающие модели информационных систем;
* средства реинжиниринга;
* средства конфигурационного управления;
* средства управления проектом.

**CASE-система** - совокупность различных, взаимодополняющих друг друга CASE-средств из одной области для решения объемных задач, которые невозможно решить только одним CASE-средством. Примеры: Rational Rose, EA Architect.

**CASE-технология** - CASE-система + технические знания по её применению.

CASE-технология включает в себя методологию анализа, проектирования, разработки и сопровождения сложных систем ПО, поддержанную комплексом взаимосвязанных средств автоматизации. CASE-технология - именно то, что мы использовали при написании лабораторных.

**CASE-индустрия** - применение CASE-технологии для создания коммерческих продуктов и получения прибыли.

**48. Проектирование информационных систем с применением UML. Разработка моделей бизнес-прецедентов и бизнес-объектов.**

**Проектирование информационных систем с применением UML**

**Для работы с Rational Rose необходим UML** (Unified Modeling Language — унифицированный язык моделирования) – графический язык описания архитектуры системы. Программы на UML представляются в виде диаграмм, состоящих из объектов и связей между ними или из этапов процесса проектирования. UML обеспечивает поддержку всех этапов жизненного цикла ИС и предоставляет для этих целей ряд графических средств — диаграмм. На этапе создания концептуальной модели для описания бизнес ­деятельности используются модели бизнес ­прецедентов и диаграммы видов деятельности, для описания бизнес ­объектов – модели бизнес объектов и диаграммы последовательностей.

**На этапе создания логической** *модели ИС описание требований к системе задается в виде модели и описания системных прецедентов, а предварительное проектирование осуществляется с использованием диаграмм классов, диаграмм последовательностей и диаграмм состояний.*

**На этапе создания физической модели** *детальное проектирование выполняется с использованием диаграмм классов, диаграмм компонентов, диаграмм развертывания.*

**Разработка модели бизнес прецедентов**

Модель бизнес­ прецедентов описывает бизнес ­процессы с точки зрения внешнего пользователя, т.е. отражает взгляд на деятельность организации извне. Проектирование системы начинается с изучения и моделирования бизнес­ деятельности организации. На этом этапе вводится и отображается в модели ряд понятий, свойственных объектно­-ориентированному подходу. Исполнитель (Действующее лицо, Actor) ­ личность, организация или система, взаимодействующая с ИС; различают *внешнего исполнителя* (который использует или используется системой, т.е. порождает прецеденты деятельности) и *внутреннего исполнителя* (который обеспечивает реализацию прецедентов деятельности внутри системы). На диаграмме исполнитель представляется стилизованной фигуркой человека.

**Прецедент** - законченная последовательность действий, инициированная внешним объектом (личностью или системой), которая взаимодействует с ИС и получает в результате некоторое сообщение от ИС. На диаграмме представляется овалом с надписью, отражающей содержание действия.

**Разработка модели бизнес ­объектов**

Следующим этапом проектирования ИС является разработка модели бизнес ­объектов, которая показывает выполнение бизнес ­процессов организации ее внутренними исполнителями. Основными компонентами моделей бизнес-­объектов являются внешние и внутренние исполнители, а также бизнес ­сущности, отображающие все, что используют внутренние исполнители для реализации бизнес ­процессов. Появляется новое действующее лицо – отправитель запроса. Понятие " Отправитель запроса " служит для обобщенного представления всех действующих лиц при описании прецедента " Ответ на запрос ". " Отправитель запроса " становится суперклассом по отношению к обобщаемым понятиям (подклассам). Для детального описания выполнения бизнес­ процессов обычно используются диаграммы последовательностей. Основными элементами диаграммы последовательностей являются обозначения объектов (прямоугольники), вертикальные линии, отображающие течение времени при деятельности объекта, и стрелки, показывающие выполнение действий объектами. Результатом этого этапа являются согласованные с заказчиком и достаточно подробные описания действий специалистов организации, внедряющей ИС, необходимые для обеспечения исполнения ее функций.

**49. Проектирование информационных систем с применением UML. Разработка концептуальной модели данных и требований к системе**

**Разработка концептуальной модели данных**.

На основе информации, выявленной на этапах бизнес ­ моделирования, выполняется разработка концептуальной модели данных, которые будут использоваться в разрабатываемой системе. Этот этап завершает процедуры бизнес ­ моделирования и позволяет представить команде проектировщиков в едином формате ту информацию, которая будет необходима для создания системы. Разработанные диаграммы являются отправной точкой в процессах проектирования баз данных и приложений системы, обеспечивают согласованность действий бизнес ­ аналитиков и разработчиков в процессе дальнейшей работы над системой. Эти диаграммы, конечно же, будут претерпевать изменения в процессе последующего проектирования, однако эти изменения будут фиксироваться в формате, уже привычном для всей команды разработчиков, и будут автоматически отражаться в последующих моделях.

**Разработка требований к системе**

Основные задачи этапа:

1. определить проект системы, который будет отвечать всем бизнес ­требованиям;
2. разработать общий предварительный проект для всех команд разработчиков (проектировщиков баз данных, разработчиков приложений, системных архитекторов и пр.)

**Основным инструментом на данном этапе являются диаграммы классов системы**, которые строятся на основе разработанной модели системных прецедентов. Одновременно на этом этапе уточняются диаграммы **последовательностей выполнения** отдельных прецедентов, что приводит к изменениям в составе объектов и диаграммах классов. Это естественное отражение средствами UML итеративного процесса разработки системы. Диаграммы классов системы заполняются объектами из модели системных прецедентов. Они содержат описание этих объектов в виде классов и описание взаимодействия между классами. Таким образом, в результате этого этапа проектирования появляется достаточно подробное описание состава и функций проектируемой системы, а также информации, которую необходимо использовать в базе данных и в приложениях. Поскольку диаграммы классов строятся на основе разработанных ранее бизнес-­моделей, появляется уверенность в том, что разрабатываемая система будет действительно удовлетворять исходным требованиям заказчика.В то же время, благодаря своему синтаксису, диаграммы классов оказываются хорошим средством структурирования и представления требований к функциональности, интерфейсам и данным для элементов проектируемой системы.

**50. Проектирование физической реализации системы**

На этом этапе проектирования модели баз данных и приложений дополняются обозначениями их размещения на технических средствах разрабатываемой системы. Основными понятиями UML, которые используются на данном этапе, являются следующие:

* *компонент* – самостоятельный физический модуль системы;
* *зависимость* – связь между двумя элементами, при которой изменения в одном элементе вызывают изменения другого элемента;
* *устройство* – узел, не обрабатывающий данные;
* *процессор* – узел, выполняющий обработку данных;
* *соединение* – связь между устройствами и процессорами.

Диаграммы развертывания позволяют отобразить на единой схеме различные компоненты системы (программные и информационные) и их распределение по комплексу технических средств.

Таким образом, при проектировании сложной ИС она разделяется на части, и каждая из них затем исследуется и создается отдельно.

В настоящее время используются два различных способа такого разбиения ИС на подсистемы:

**структурное** (или функциональное) разбиение

**объектная** (компонентная) декомпозиция.

С позиций проектирования ИС суть *функционального разбиения* может быть выражена известной формулой: " Программа = Данные + Алгоритмы". При функциональной декомпозиции программной системы ее структура описывается **блок­схемами**, узлы которых представляют собой "обрабатывающие центры" (функции), а связи между узлами описывают движение данных.

При объектном разбиении в системе выделяются "активные сущности" – объекты (или компоненты), которые взаимодействуют друг с другом, обмениваясь сообщениями и выполняя соответствующие функции (методы) объекта.

Если при проектировании ИС разбивается на объекты, то для ее визуального моделирования следует использовать **UML.**

Если в основу проектирования положена функциональная декомпозиция ИС, то UML не нужен и следует использовать рассмотренные ранее структурные нотации.

В то же время, при выборе подхода к разработке ИС следует учитывать, что визуальные модели все более широко используются в существующих технологиях управления проектированием систем, сложность, масштабы и функциональность которых постоянно возрастают.

Они хорошо приспособлены для решения таких часто возникающих при создании систем задач как:

* физическое перераспределение вычислений и данных,
* обеспечение параллелизма вычислений,
* репликация БД,
* обеспечение безопасности доступа к ИС,
* оптимизация балансировки нагрузки ИС,
* устойчивость к сбоям и т.п.

Визуализированные средствами UML модели ИС позволяют наладить плодотворное взаимодействие между заказчиками, пользователями и командой разработчиков. Они обеспечивают ясность представления выбранных архитектурных решений и позволяют понять разрабатываемую систему во всей ее полноте.

51. Требования к программному обеспечению. Виды требований.

52. Формирование требований. Анализ требований.

Требования к ПО:

В рамках ЖЦ ПС работа с требованиями является обязательным этапом.

Требования – это описание функциональных возможностей и ограничений накладываемых На ПС. Обычно требования выражают то, что должна выполнять система. При этом на этапе требований не пытаются сформулировать то, как система должна их выполнять.

Две категории требований:

* Заказчика
* Разработчика

Первичные и вторичные различаются между собой степенью проработки.

Первичные требования документируются желания и потребности заказчика и пишутся на языке, понятном заказчику.

Детализированные требования документируются в специальной, структурной форме, они детализированы, по отношению к первичным требованиям.

Работа по созданию первичных требований – сбор требований. Она проявляется на этапе подготовки ЖЦ разработки.

Работа по созданию детальных требований – анализ требований. Она проявляется на этапе моделирования ЖЦ разработки.

Виды требований:

* Функциональные – описывают поведение системы и функций, которые она должна выполнять. Исходят из всестороннего анализа предметной области. Рассматривают разнообразные варианты, определяющие различные варианты использования и описание окружающей среды.
* Нефункциональные – требования, относящиеся к характеристикам системы и ее внешнему окружению.

Функциональные требования задают работу которая должна выполнять ПС. Не связаны напрямую с функциями системы. Многие из этих требований относятся к системе в целом. Ошибки в функциональных требованиях может снизить качество работы системы. Ошибки в нефункциональных требованиях может привести к отключению системы.

Выделяют 3 группы нефункциональных требований:

* К ПС;
* Организации требований;
* Внешние требования (учитывают фактор внешней среды).

Шаги формирования требований:

* Определить представление заказчика;
* Проведение опроса представителя заказчика;
* Документация результата опроса;
* Проверка требований.

Организация первичных требований:

· По режиму;

· По категории пользователей;

· По объектам;

· По свойствам;

· По стимулам;

· По откликам;

· По иерархии функций;

Преобразование первичных требований в детальные требования. Обычно одно требование заказчика преобразуется в несколько детальных требований, хотя возможен вариант один ко одному.

Аттестация детальных требований должна подтверждать, что требование определяет ту систему, которая нужна заказчику.

В состав аттестации входит:

* Проверка правильности требований
* Проверка на противоречия
* Проверка на полноту
* Проверка на выполнимость

Верификация идет после аттестации. Это более детальная проверка.

В ходе создания детальных требований ориентируются на формирование набора желательных характеристик. Методика связывания с отображением каждого проекта и системы.

Возможность пошагового прослеживания означает, что каждое детальное требование связано с конкретным элементом ПС, а также с тестом элемента.

**53. Основы технологии сетевого планирования и управления проектами.**

**54. Размерно-ориентированные и функциональные метрики оценивания программных продуктов.**

Размерно-ориентированные метрики прямо измеряют программный продукт и процесс его разработки. Основываются размерно-ориентированные метрики на LOC-оценках (Lines Of Code). LOC-оценка — это количество строк в программном продукте. Исходные данные для расчета этих метрик сводятся в таблицу по мере накопления опыта организации.

Достоинства размерно-ориентированных метрик:

1) широко распространены;

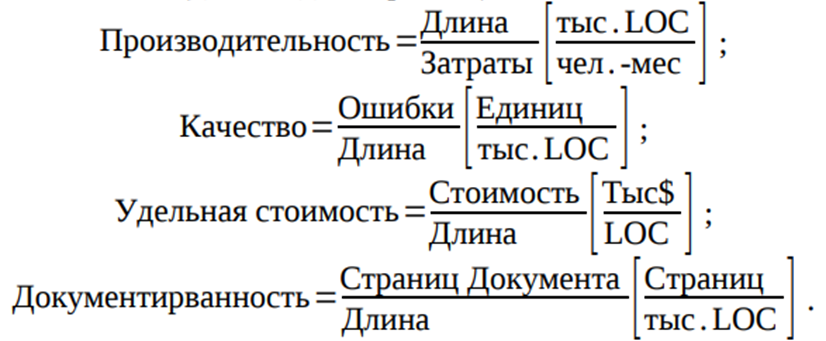
2) просты и легко вычисляются.

Недостатки размерно-ориентированных метрик:

1) зависимы от языка программирования;

2) требуют исходных данных, которые трудно получить на начальной стадии проекта;

3) не приспособлены к непроцедурным языкам программирования.



**Функционально-ориентированные метрики**

Функционально-ориентированные метрики косвенно измеряют программный продукт и процесс его разработки. Вместо подсчета LOC-оценки при этом рассматривается не размер, а функциональность или полезность продукта.

Используется 5 информационных характеристик.

1. Количество внешних вводов. Подсчитываются все вводы пользователя, по которым поступают разные прикладные данные. Вводы должны быть отделены от запросов, которые подсчитываются отдельно.

2. Количество внешних выводов. Подсчитываются все выводы, по которым к

пользователю поступают результаты, вычисленные программным приложением. В этом контексте выводы означают отчеты, экраны, распечатки, сообщения об ошибках. Индивидуальные единицы данных внутри отчета отдельно не подсчитываются.

3. Количество внешних запросов. Под запросом понимается диалоговый ввод, который приводит к немедленному программному ответу в форме диалогового вывода. При этом диалоговый ввод в приложении не сохраняется, а диалоговый вывод не требует выполнения вычислений. Подсчитываются все запросы — каждый учитывается отдельно.

4. Количество внутренних логических файлов. Подсчитываются все логические файлы (то есть логические группы данных, которые могут быть частью базы данных или отдельным файлом).

5. Количество внешних интерфейсных файлов. Подсчитываются все логические файлы из других приложений, на которые ссылается данное приложение.

Вводы, выводы и запросы относят к категории транзакция. Транзакция — это элементарный процесс, различаемый пользователем и перемещающий данные между внешней средой и программным приложением. В своей работе транзакции используют внутренние и внешние файлы. Приняты следующие определения.

Внешний ввод — элементарный процесс, перемещающий данные из внешней среды в приложение. Данные могут поступать с экрана ввода или из другого приложения. Данные могут использоваться для обновления внутренних логических файлов. Данные могут содержать как управляющую, так и деловую информацию. Управляющие данные не должны модифицировать внутренний логический файл.

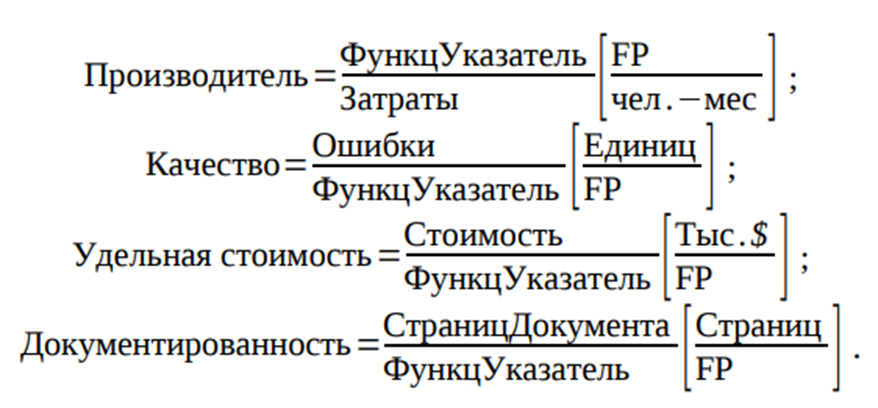
Внешний вывод — элементарный процесс, перемещающий данные, вычисленные в приложении, во внешнюю среду. Кроме того, в этом процессе могут обновляться внутренние логические файлы. Данные создают отчеты или выходные файлы, посылаемые другим приложениям. Отчеты и файлы создаются на основе внутренних логических файлов и внешних интерфейсных файлов. Дополнительно этот процесс может использовать вводимые данные, их образуют критерии поиска и параметры, не поддерживаемые внутренними логическими файлами. Вводимые данные поступают извне, но носят временный характер и не сохраняются во внутреннем логическом файле.

Внешний запрос — элементарный процесс, работающий как с вводимыми, так и с выводимыми данными. Его результат — данные, возвращаемые из внутренних логических файлов и внешних интерфейсных файлов. Входная часть процесса не модифицирует внутренние логические файлы, а выходная часть не несет данных, вычисляемых приложением (в этом и состоит отличие запроса от вывода).

Внутренний логический файл — распознаваемая пользователем группа логически связанных данных, которая размещена внутри приложения и обслуживается через внешние вводы.

Внешний интерфейсный файл — распознаваемая пользователем группа логически связанных данных, которая размещена внутри другого приложения и поддерживается им. Внешний файл данного приложения является внутренним логическим файлом в другом приложении.

Каждой из выявленных характеристик ставится в соответствие сложность. Для этого характеристике назначается низкий, средний или высокий ранг, а затем формируется числовая оценка ранга.



После заполнения таблицы по формуле, приведенной выше, вычисляется значение указателя свойств. Для сложных систем реального времени это значение на 25-30% больше значения, вычисляемого по таблице для количества функциональных указателей.

Достоинства функционально-ориентированных метрик:

1. Не зависят от языка программирования.

2. Легко вычисляются на любой стадии проекта.

Недостаток функционально-ориентированных метрик:

результаты основаны на субъективных данных, используются не прямые, а косвенные измерения. FP-оценки легко пересчитать в LOC-оценки, есть специальные таблицы