1. Как можно охарактеризовать структуру сложных систем.

Система— множество элементов, находящихся в отношениях и связях друг с другом, которое образует определённую целостность, единство.

Сложные системы– имеют разветвленную структуру и взаимосвязанные, взаимодействующие элементы (простые подсистемы)

Структура системы – это совокупность функциональных составляющих системы и их отношений, необходимых и достаточных для достижения системой заданной цели.

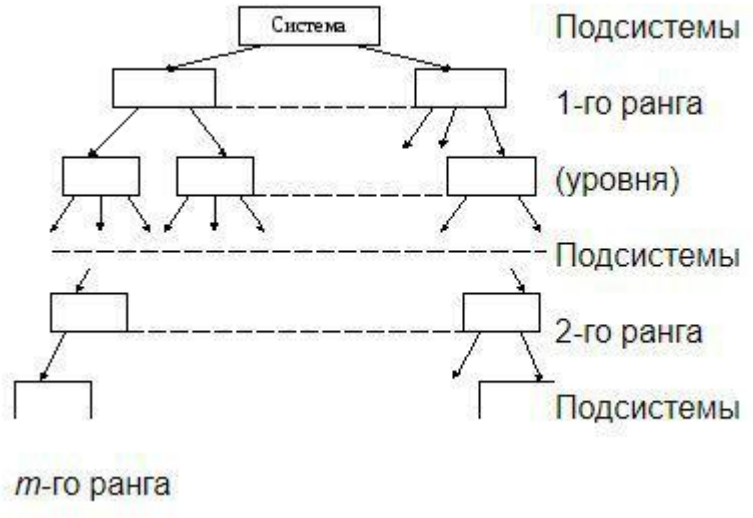
Функциональные составляющие, приведенные в определении структуры системы, носят названия: *подсистемы*и *элементы*.

- *Элемент*(в формализованной схеме системы) – это объект (часть системы), не подлежащий (при данном рассмотрении системы) дальнейшему разбиению на части. Внутренняя структура элемента при этом не является предметом изучения. Должны быть известны только те свойства элемента, которые определяют его взаимодействие с другими элементами системы и оказывают влияние на свойства системы в целом.

- *Подсистема*– совокупность элементов системы вместе со связями между ними.

Таким образом, деление системы весьма условно и зависит от уровня, на котором рассматривается система. То есть процесс расчленения системы может продолжаться до тех пор, пока дальнейшее разбиение окажется нецелесообразным. Число подсистем и их порядок может быть любым. Важно лишь, чтобы подсистемы, действующие совместно, обеспечивали выполнение всех функций следующей, высшей по уровню подсистемы.

!Таким образом, сложные системы имеют, как правило, иерархическую структуру!



Иерархические структуры бывают идеальные и неидеальные. Идеальные иерархические структуры должны иметь пять признаков:

- Многоуровневость (этажность);

- Ветвистость(элемент уровня связан с элементом верхнего или несколькими элементами нижнего уровня);
- Пирамидальность(на самом верхнем уровне имеется один элемент);
- Субординация внутренних связей (элементы уровня связаны с ближним верхним или ближним нижним уровнями);
- Субординация внешних связей, которые контролируются верхними подсистемами(элементы каждого уровня могут иметь связи с внешней средой, однако эти связи контролируются элементами ближайшего верхнего уровня).

Неидеальные характеризуются следующими признаками:

- Элемент уровня связан только с одним элементом верхнего уровня
- Элемент уровня связан более, чем с одним элементом верхнего уровня
- Элемент уровня связан с элементом внешних уровней, минуя ближайший верхний уровень
- На самом верхнем уровне имеется несколько элементов
- Элементы уровня связаны между собой

2. Как можно охарактеризовать сложность программного обеспечения.

Сложность вызывается пятью основными причинами:

1).Сложностью реальной предметной области, из которой исходит заказ на разработку

Проблемы, которые мы пытаемся решить с помощью программного обеспечения, часто неизбежно содержат сложные элементы, а к соответствующим программам предъявляется множество различных, порой взаимоисключающих требований;

Большие системы имеют тенденцию к эволюции в процессе их использования:

• ***Сопровождение***- устранение ошибок;

• ***Эволюция***- внесение изменений в систему в ответ на изменившиеся требования к ней;

• ***Сохранение***- использование всех возможных и невозможных способов для поддержания жизни в дряхлой и распадающейся на части системе.

2).Трудностью управления процессом разработки;

Основная задача разработчиков состоит в создании иллюзии простоты, в защите пользователей от сложности описываемого предмета или процесса.

Сегодня обычными стали программные системы, размер которых исчисляется десятками тысяч или даже миллионами строк на языках высокого уровня. Поэтому такой объем работ потребует привлечения команды разработчиков, в идеале как можно меньшей по численности. Но какой бы она ни была, всегда будут возникать значительные трудности, связанные с организацией коллективной разработки. Чем больше разработчиков, тем сложнее связи между ними и тем сложнее координация, особенно если участники работ географически удалены друг от друга, что типично в случае очень больших проектов. Таким образом, при коллективном выполнении проекта главной задачей руководства является поддержание единства и целостности разработки.

3).Необходимостью обеспечить достаточную гибкость программы;

Программирование обладает предельной гибкостью, и разработчик может сам обеспечить себя всеми необходимыми элементами, относящимися к любому уровню абстракции.

Такая гибкость заставляет разработчика создавать своими силами все базовые строительные блоки будущей конструкции, из которых составляются элементы более высоких уровней абстракции. Поэтому программные разработки остаются очень трудоемким делом.

4). Неудовлетворительными способами описания поведения больших дискретных систем.

Внутри большой прикладной программы могут существовать сотни и даже тысячи переменных и несколько потоков управления. Полный набор этих переменных, их текущих значений, текущего адреса и стека вызова для каждого процесса описывает состояние прикладной программы в каждый момент времени.

Так как исполнение нашей программы осуществляется на цифровом компьютере, мы имеем систему с дискретными состояниями. Аналоговые системы, такие, как движение брошенного мяча, напротив, являются непрерывными.

Мы стараемся проектировать системы, разделяя их на части так, чтобы одна часть минимально воздействовало на другую. Однако переходы между дискретными состояниями не могут моделироваться непрерывными функциями. Каждое событие, внешнее по отношению к программной системе, может перевести ее в новое состояние, и, более того, переход из одного состояния в другое не всегда детерминирован. При неблагоприятных условиях внешнее событие может нарушить текущее состояние системы

из-за того, что ее создатели не смогли предусмотреть все возможные варианты.

Представим себе пассажирский самолет, в котором система управления полетом и система электроснабжения объединены. Было бы очень неприятно, если бы от включения пассажиром, сидящим на месте 38J, индивидуального освещения самолет немедленно вошел бы в глубокое пике. В непрерывных системах такое поведение было бы невозможным, но в дискретных системах любое внешнее событие может повлиять на любую часть внутреннего состояния системы. Это, очевидно, и является главной причиной обязательного тестирования наших систем; но дело в том, что за исключением самых тривиальных случаев, всеобъемлющее тестирование таких программ провести невозможно.

5)Последствия неограниченной сложности

"Чем сложнее система, тем легче ее полностью развалить". Строитель едва ли согласится расширить фундамент уже построенного 100-этажного здания.

Но что удивительно, пользователи программных систем, не задумываясь, ставят подобные задачи перед разработчиками. Это, утверждают они, всего лишь технический вопрос для программистов.

Неумение создавать сложные программные системы проявляется в проектах, которые выходят за рамки установленных сроков и бюджетов и к тому же не соответствуют начальным требованиям. К сожалению, это приводит к разбазариванию человеческих ресурсов и к существенному ограничению возможностей создания новых продуктов.

3. Как можно охарактеризовать сложность предметной области.

Проблемы, которые мы пытаемся решить с помощью программного обеспечения, часто неизбежно содержат сложные элементы, а к соответствующим программам предъявляется множество различных, порой взаимоисключающих требований.

Рассмотрим необходимые характеристики электронной системы многомоторного самолета, сотовой телефонной коммутаторной системы и робота. Достаточно трудно понять, даже в общих чертах, как работает каждая такая система. Теперь прибавьте к этому дополнительные требования (часто не формулируемые явно), такие как удобство, производительность,

стоимость, выживаемость и надежность! Сложность задачи'и порождает ту сложность программного продукта, о которой пишет Брукс.

Эта внешняя сложность обычно возникает из-за «нестыковки» между пользователями системы и ее разработчиками: пользователи с трудом могут объяснить в форме, понятной разработчикам, что на самом деле нужно сделать. Бывают случаи, когда пользователь лишь смутно представляет, что ему нужно от будущей программной системы. Это в основном происходит не из-за ошибок с той или иной стороны; просто каждая из групп специализируется в своей области, и ей недостает знаний партнера. У пользователей и разработчиков разные взгляды на сущность проблемы, и они делают различные выводы о возможных путях ее решения. На самом деле, даже если пользователь точно знает, что ему нужно, мы с трудом можем однозначно зафиксировать все его требования. Обычно они отражены на многих страницах текста, «разбавленных» немногими рисунками. Такие документы трудно поддаются пониманию, они открыты для различных интерпретаций и часто содержат элементы, относящиеся скорее к дизайну, чем к необходимым требованиям разработки.

Дополнительные сложности возникают в результате изменений требований к программной системе уже в процессе разработки. В основном требования корректируются из-за того, что само осуществление программного проекта часто изменяет проблему. Рассмотрение первых результатов — схем, прототипов, — и использование системы после того, как она разработана и установлена, заставляют пользователей лучше понять и отчетливей сформулировать то, что им действительно нужно. В то же время этот процесс повышает квалификацию разработчиков в предметной области и позволяет им задавать более осмысленные вопросы, которые проясняют темные места в проектируемой системе.

Большая программная система — это крупное капиталовложение, и мы не можем позволить себе выкидывать сделанное при каждом изменении внешних требований. Тем не менее даже большие системы имеют тенденцию к эволюции в процессе их использования: следовательно, встает задача о том, что часто неправильно называют сопровождением программного обеспечения. Чтобы быть более точными, введем несколько терминов: под сопровождением понимается устранение ошибок; под эволюцией — внесение изменений в систему в ответ на изменившиеся требования к ней; под сохранением — использование всех возможных и невозможных способов для

4. Как можно охарактеризовать сложность дискретных систем.

Когда мы кидаем вверх мяч, мы можем достоверно предсказать его траекторию, потому что знаем, что в нормальных условиях здесь действуют известные физические законы. Мы бы очень удивились, если бы, кинув мяч с чуть большей скоростью, увидели, что он на середине пути неожиданно остановился и резко изменил направление движения. В недостаточно отлаженной программе моделирования полета мяча такая ситуация легко может возникнуть.

Внутри большой прикладной программы могут существовать сотни и даже тысячи переменных и несколько потоков управления. Полный набор этих переменных, их текущих значений, текущего адреса и стека вызова для каждого процесса описывает состояние прикладной программы в каждый момент времени. Так как исполнение нашей программы осуществляется на цифровом компьютере, мы имеем систему с дискретными состояниями.

Аналоговые системы, такие, как движение брошенного мяча, напротив, являются непрерывными. С другой стороны, дискретные системы по самой своей природе имеют конечное число возможных состояний, хотя в больших системах это число в соответствии с правилами комбинаторики очень велико. Мы стараемся проектировать системы, разделяя их на части так, чтобы одна часть минимально воздействовало на другую. Однако переходы между дискретными состояниями не могут моделироваться непрерывными функциями.

Каждое событие, внешнее по отношению к программной системе, может перевести ее в новое состояние, и, более того, переход из одного состояния в другое не всегда детерминирован. При неблагоприятных условиях внешнее событие может нарушить текущее состояние системы из-за того, что ее создатели не смогли предусмотреть все возможные варианты.

Представим себе пассажирский самолет, в котором система управления полетом и система электроснабжения объединены. Было бы очень неприятно, если бы от включения пассажиром, сидящим на месте 38J, индивидуального освещения самолет немедленно вошел бы в глубокое пике.

В *непрерывных системах* такое поведение было бы невозможным, но в дискретных системах любое внешнее событие может повлиять на любую часть внутреннего состояния системы. Это, очевидно, и является главной причиной обязательного тестирования наших систем; но дело в том, что за

исключением самых тривиальных случаев, всеобъемлющее тестирование таких программ провести невозможно.

5. Перечислите признаки сложной системы.

1). "Сложные системы часто являются иерархическими и состоят из взаимозависимых подсистем, которые в свою очередь также могут быть разделены на подсистемы, и т.д., вплоть до самого низкого уровням. Архитектура складывается из компонентов, и из иерархических отношений этих компонентов. Выбор, какие компоненты в данной системе считаются элементарными, относительно произволен и в большой степени оставляется на усмотрение исследователя.

2). Одной из важнейших характеристик сложной системы является непредсказуемость. Образ поведения системы складывается как результат взаимодействия и взаимоотношений между ее компонентами.

3). "Внутрикомпонентная связь обычно сильнее, чем связь между компонентами. Это обстоятельство позволяет отделять "высокочастотные" взаимодействия внутри компонентов от "низкочастотной" динамики взаимодействия между компонентами". Это различие внутрикомпонентных и межкомпонентных взаимодействий обуславливает разделение функций между частями системы и дает возможность относительно изолированно изучать каждую часть.

4). Отношения не являются линейными; следовательно, небольшое возмущающее воздействие может вызвать заметный эффект, и наоборот, большой воздействующий импульс может оказаться нерезультативным.

5). Отношения между компонентами могут включать обратные связи, причем как положительные (раскачивающие систему), так и отрицательные (демпфирующие) ее.

6). "Иерархические системы обычно состоят из немногих типов подсистем, по-разному скомбинированных и организованных". Т.е., разные сложные системы содержат одинаковые структурные части. Эти части могут использовать общие более мелкие компоненты.

7). "Любая работающая сложная система является результатом развития работавшей более простой системы. Сложная система, спроектированная "с нуля", никогда не заработает. Следует начинать с работающей простой системы". В процессе развития системы объекты, первоначально

рассматривавшиеся как сложные, становятся элементарными, и из них строятся более сложные системы.

8). Сложная система является открытой; ее границы в зависимости от природы системы должны быть проницаемы или для информации, или для энергии. По этой причине она подвергается изменениям, однако средствами управления ее можно удерживать в стабильном состоянии. Ни один из элементов не обладает полнотой информации о системе в целом.

Примеры: 1. Структура персонального компьютера. Большинство ПК состоит из одних и тех же основных элементов: системной платы, монитора, клавиатуры и устройства внешней памяти. Можно взять любую из этих частей и разложить ее в свою очередь на составляющие. Это пример сложной иерархической системы. Персональный компьютер нормально работает благодаря четкому совместному функционированию всех его составных частей. Вместе эти части образуют логическое целое. Можно понять, как работает компьютер, только потому, что можно рассматривать отдельно каждую его составляющую. Т.о., можно изучать устройства монитора и жесткого диска независимо друг от друга.

2. Структура растений и животных. Растения - это сложные многоклеточные организмы. В результате совместной деятельности различных органов растений происходят такие сложные типы поведения, как фотосинтез и всасывание влаги. Растение состоит из трех основных частей: корни, стебли и листья. Каждая из них имеет свою особую структуру. Так же, как у компьютера, части растения образуют иерархию, каждый уровень которой обладает собственной независимой сложностью.

При изучении морфологии растения мы не выделяем в нем отдельные части, отвечающие за отдельные фазы единого процесса, например, фотосинтеза. Фактически не существует централизованных частей, которые непосредственно координируют деятельность более низких уровней. Вместо этого мы находим отдельные части, которые действуют как независимые посредники, каждый из которых ведет себя достаточно сложно и при этом согласованно с более высокими уровнями. Только благодаря совместным действиям большого числа посредников образуется более высокий уровень функционирования растения. Поведение целого сложнее, чем поведение суммы его составляющих.

Многоклеточные животные, как и растения, имеют иерархическую структуру: клетки формируют ткани, ткани работают вместе как органы, группы органов определяют систему (например, пищеварительную) и так далее. Основной строительный блок всех растений и животных - клетка. Это пример общности в разных сферах.

6. В чем заключается выбор элементарных компонентов.(доп. в 7)

Декомпозиция— разделение целого на части. Также декомпозиция — это научный метод, использующий структуру задачи и позволяющий заменить решение одной большой задачи решением серии меньших задач, пусть и взаимосвязанных, но более простых.

Декомпозиция, как процесс расчленения, позволяет рассматривать любую исследуемую систему как сложную, состоящую из отдельных взаимосвязанных подсистем, которые, в свою очередь, также могут быть расчленены на части. В качестве систем могут выступать не только материальные объекты, но и процессы, явления и понятия.

Исходная система располагается на нулевом уровне. После её расчленения получаются подсистемы первого уровня. Расчленение этих подсистем или некоторых из них приводит к появлению подсистем второго уровня и т. д.

Выбор элементарных компонентов

Степень подробности описания и количество уровней определяются требованиями обозримости и удобства восприятия получаемой иерархической структуры, её соответствия уровням знания работающему с ней специалисту.

Обычно в качестве нижнего (элементарного) уровня подсистем берут такой, на котором располагаются подсистемы, понимание устройства которых или их описание доступно исполнителю (руководителю группы людей или отдельному человеку). Таким образом, иерархическая структура всегда субъективно ориентирована: для более квалифицированного специалиста она будет менее подробна.

Элементарная система (в данном случае компонент большой системы) – это формализованное представление системы, рассматриваемой как единое целое, с целью удобства ее исследования.

Элементарную систему можно представить в виде схемы «вход – процесс – выход»

7. Роль декомпозиции.

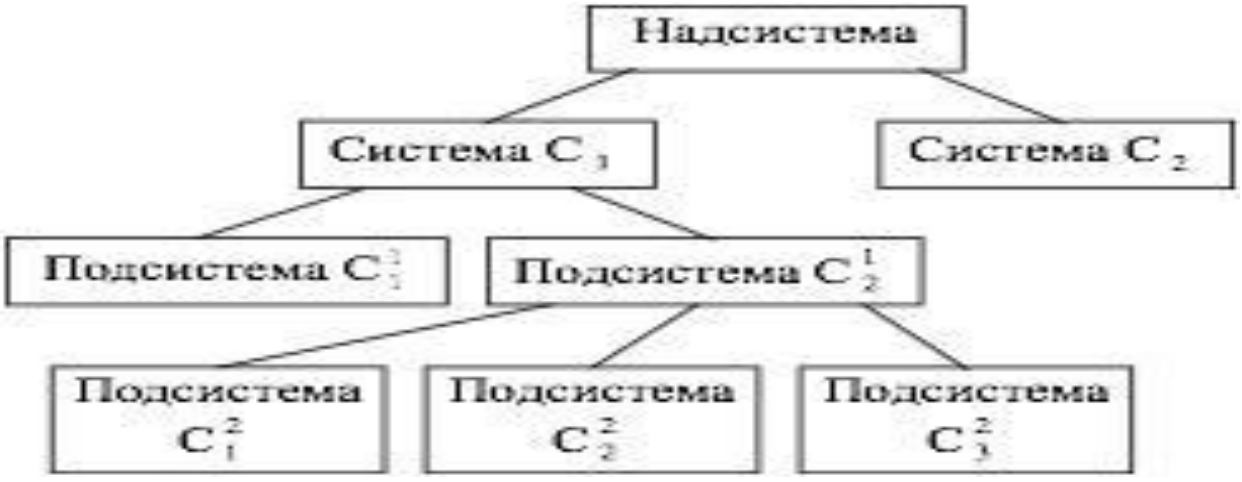
Декомпозиция— разделение целого на части. Также декомпозиция — это научный метод, использующий структуру задачи и позволяющий заменить решение одной большой задачи решением серии меньших задач, пусть и взаимосвязанных, но более простых.

Декомпозиция, как процесс расчленения, позволяет рассматривать любую исследуемую систему как сложную, состоящую из отдельных взаимосвязанных подсистем, которые, в свою очередь, также могут быть расчленены на части. В качестве систем могут выступать не только материальные объекты, но и процессы, явления и понятия.

Исходная система располагается на нулевом уровне. После её расчленения получаются подсистемы первого уровня. Расчленение этих подсистем или некоторых из них приводит к появлению подсистем второго уровня и т. д.

Упрощённое графическое представление декомпозированной системы называется её иерархической структурой.

Иерархическая структура может быть изображена в виде ветвящейся блок-схемы, наподобие представленной на рис. 1.



Здесь на нулевом уровне располагается исходная система С1, на следующих уровнях — её подсистемы (число уровней и количество подсистем, показанных на рисунке, выбрано произвольно). С целью получения более полного представления о системе и её связях в структуру включают надсистему и составляющие её части (системы нулевого уровня, например, вторая система С2).

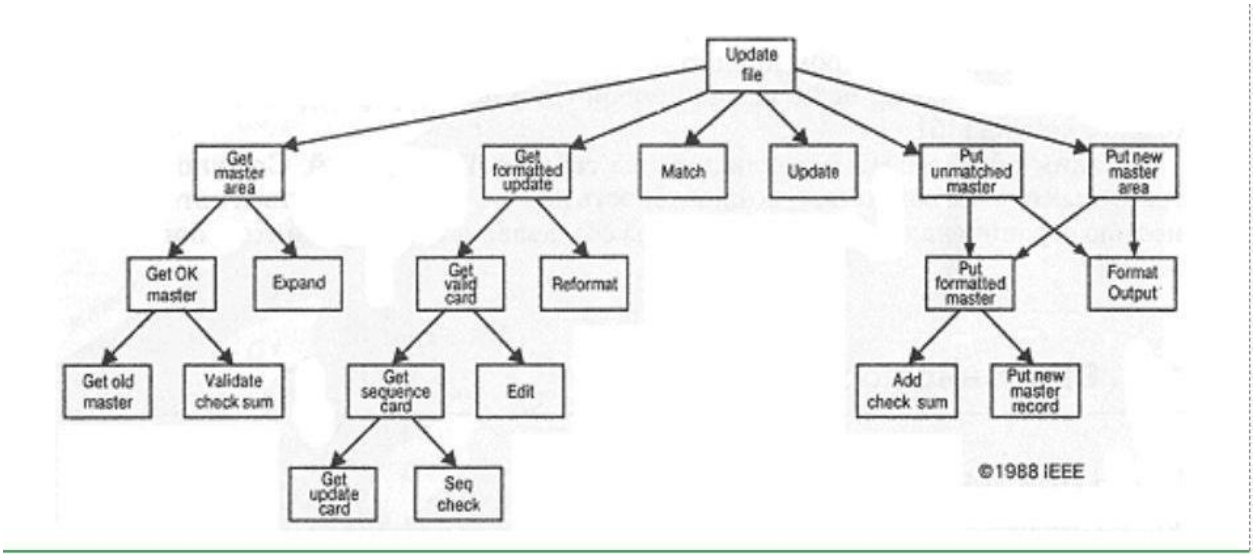
Итог. Роль декомпозиции.

При проектировании сложной программной системы необходимо разделять ее на все меньшие и меньшие подсистемы, каждую из которых можно совершенствовать независимо.

Декомпозиция вызвана сложностью программирования системы, поскольку именно эта сложность вынуждает делить пространство состояний системы.

8. Что такое алгоритмическая декомпозиция.

Алгоритмическая декомпозиция - это обычное разделение алгоритмов, где каждый модуль системы выполняет один из этапов общего процесса.



Известно, что при проектировании сложной системы ее необходимо представить совокупностью небольших подсистем, каждая из которых может разрабатываться независимо от других. Всякое систематическое решение задачи, какой бы сложной или простой она не была, основано на принципе разбиения на уровни абстракции.

Мы привыкли связывать такое разбиение с алгоритмической декомпозицией, в ходе которой осуществляется структурное проектирование по методу "сверху-вниз" и алгоритм задачи разбивается на отдельные модули.

Каждый модуль системы реализует один из алгоритмов общей задачи. Принципы алгоритмической декомпозиции поддерживаются языком Турбо-Паскаль, являющимся хорошим инструментом структурного программирования

9.Что такое объектно-ориентированная декомпозиция

- Это декомпозиция основана на объектах, а не на алгоритмах в которой мир представлен совокупностью автономных действующих лиц, которые взаимодействуют друг с другом, чтобы обеспечить поведение системы,
соответствующее более высокому уровню. Get formatted update (Получить изменения в отформатированном виде) больше не присутствует в качестве независимого алгоритма; это действие существует теперь как операция над объектом File of Updates (Файл изменений). Эта операция создает другой объект — Update to Card (Изменения в карте). Таким образом, каждый объект обладает своим собственным поведением, и каждый из них моделирует некоторый объект реального мира. С этой точки зрения объект является вполне осязаемой вещью, которая демонстрирует вполне определенное поведение. Объекты что-то делают, и мы можем, послав им сообщение, попросить их выполнить то-то и то- то.

10.Сравните алгоритмическую и объектно-ориентированную декомпозиции.

Большинство из нас формально обучено структурному проектированию «сверху вниз», и мы воспринимаем декомпозицию как обычное разделение алгоритмов, где каждый модуль системы выполняет один из этапов общего процесса – это особенность алгоритмической декомпозиции. В тоже время разделение системы, где каждый объект обладает своим собственным поведением, и каждый из них моделирует некоторый объект реального мира.

Поэтому объектно-ориентированнаядекомпозиция считается "более продвинутой" (при разработке сложных систем) в отличии от Алгоритмической - при которой просто происходит разбиение функционала - ни по отношению к сущности (типа -набор методов в данном классе реализует функционал описываемого объекта), а просто "сверху вниз" - то есть об общего к частному.

11.Как используются абстракции и иерархии

Абстракции - это модель некоего объекта или явления реального мира, откидывающая незначительные детали, не играющие существенной роли в данном приближении. Применяется для управления сложностью проектируемой системы при декомпозиции, когда система представляется в виде иерархии уровней абстракции.

Иерархии – используются в сложных системах для организации внутри системы иерархий классов и объектов структур сложной системы, что в свою очередь облегчает наше понимание этой системы. Объектная структура важна, так как она иллюстрирует схему взаимодействия объектов друг с другом, которое осуществляется с помощью механизмов взаимодействия. Структура классов не менее важна: она определяет общность структур и поведения внутри системы.

12.Иерархии классов

Иерархия классов- означает классификацию объектных типов, рассматривая объекты как реализацию классов (класс похож на заготовку, а

объект — это то, что строится на основе этой заготовки) и связывая различные классы отношениями наподобие «наследует», «расширяет», «является его абстракцией», «определение интерфейса». Индивидуальные объекты называются **экземплярами класса**,а **класс в ООП**— это шаблон по которому строятся объекты.

13.Принципы проектирования сложных систем

1. «удовлетворяет заданным (возможно, неформальным) функциональным спецификациям;
2. согласована с ограничениями, накладываемыми оборудованием;
3. удовлетворяет явным и неявным требованиям по эксплуатационным качествам и ресурсопотреблению;
4. удовлетворяет явным и неявным критериям дизайна продукта;
5. удовлетворяет требованиям к самому процессу разработки, таким, например, как продолжительность и стоимость, а также привлечение дополнительных инструментальных средств» По предположению Страуструпа: «Цель проектирования — выявление ясной и относительно простой внутренней структуры, иногда называемой архитектурой... Проект есть окончательный продукт процесса проектирования» [41]. Проектирование подразумевает учет противоречивых требований. Его продуктами являются модели, позволяющие нам понять структуру будущей системы, сбалансировать требования и наметить схему реализации.

14.Объектно-ориентированные модели.

Наилучший способ разделить сложную систему на подсистемы – это создавать такие модели, которые фокусируют внимание на объектах, найденных в самой предметной области, и образуют то, что мы назвали объектно-ориентированной декомпозицией Объектно-ориентированный анализ и проектирование — это метод, логически приводящий нас к объектно-ориентированной декомпозиции. Применяя объектно-ориентированное проектирование, мы создаем гибкие программы, написанные экономными средствами. При разумном разделении пространства состояний мы добиваемся большей уверенности в правильности нашей программы. В итоге, мы уменьшаем риск при разработке сложных программных систем.

Так как построение моделей крайне важно при проектировании сложных систем, объектно-ориентированное проектирование предлагает богатый выбор моделей: динамическая модель, статистическая модель, логическая модель, физическая модель. Объектно-ориентированные модели проектирования отражают иерархию и классов, и объектов системы. Эти модели покрывают весь спектр важнейших конструкторских решений, которые необходимо рассматривать при разработке сложной системы, и таким образом вдохновляют нас на создание проектов, обладающих всеми пятью атрибутами хорошо организованных сложных систем.

15.Эволюция объектной модели

Вегнер сгруппировал некоторые из наиболее известных языков высокого уровня в четыре поколения в зависимости от того, какие языковые конструкции впервые в них появились:

Языки первого поколения ориентировались на научно-инженерные применения, и словарь этой предметной области был почти исключительно математическим. Такие языки, как FORTRAN I, были созданы для упрощения программирования математических формул, чтобы освободить программиста от трудностей ассемблера и машинного кода. Первое поколение языков высокого уровня было шагом, приближающим программирование к предметной области и удаляющим от конкретной машины.

Во втором поколении языков основной тенденцией стало развитие алгоритмических абстракций. В это время мощность компьютеров быстро росла, а компьютерная индустрия позволила расширить области их применения, особенно в бизнесе. Главной задачей стало инструктировать машину, что делать: сначала прочти эти анкеты сотрудников, затем

отсортируй их и выведи результаты на печать. Это было еще одним шагом к предметной области и от конкретной машины. В конце 60-х годов с появлением транзисторов, а затем интегральных схем, стоимость компьютеров резко снизилась, а их производительность росла почти экспоненциально. Появилась возможность решать все более сложные задачи, но это требовало умения обрабатывать самые разнообразные типы данных. Такие языки как ALGOL-68 и затем Pascal стали поддерживать абстракцию данных. Программисты смогли описывать свои собственные типы данных. Это стало еще одним шагом к предметной области и от привязки к конкретной машине. 70-е годы знаменовались безумным всплеском активности: было создано около двух тысяч различных языков и их диалектов. Неадекватность более ранних языков написанию крупных программных систем стала очевидной, поэтому новые языки имели механизмы, устраняющие это ограничение. Лишь немногие из этих языков смогли выжить (попробуйте найти свежий учебник по языкам Fred, Chaos, Tranquil), однако многие их принципы нашли отражение в новых версиях более ранних языков. Таким образом, мы получили языки Smalltalk (новаторски переработанное наследие Simula), Ada (наследник ALGOL-68 и Pascal с элементами Simula, Alphard и CLU), CLOS (объединивший Lisp, LOOPS и Flavors), C++ (возникший от брака C и Simula) и Eiffel (произошел от Simula и Ada). Наибольший интерес для дальнейшего изложения представляет класс языков, называемых объектными и объектно-ориентированными, которые в наибольшей степени отвечают задаче объектно-ориентированной декомпозиции программного обеспечения.

16.Основные положения объектной модели.

Методы структурного проектирования помогают упростить процесс разработки сложных систем за счет использования алгоритмов как готовых строительных блоков. Аналогично, методы объектно- ориентированного проектирования созданы, чтобы помочь разработчикам применять мощные выразительные средства объектного и объектно-ориентированного программирования, использующего в качестве блоков классы и объекты. Объектно-ориентированный анализ и проектирование отражают эволюционное, а не революционное развитие проектирования; новая методология не порывает с прежними методами, а строится с учетом предшествующего опыта. Алгоритмическая декомпозиция помогает только до определенного предела, и обращение к объектно-ориентированной декомпозиции необходимо. Более того, при попытках использовать такие

языки, как C++ или Ada, в качестве традиционных, алгоритмически ориентированных, мы не только теряем их внутренний потенциал — скорее всего результат будет даже хуже, чем при использовании обычных языков C и Pascal

17.Объектно-ориентированное программирование.

Объектно-ориентированное программирование – это программирование сфокусированное на данных, причем данные и поведение неразрывно связанные. Вместо данные и поведение, представляют собой класс. А объекты являются экземплярами класса. Например, многочлен имеет область значений, и она может изменятся такими операциями как сложение и умножение многочленов.

Основные принципы структурирования в случае ООП связаны с различными аспектами базового понимания предметной задачи, которое требуется для оптимального управления соответствующей моделью:

Абстрагирование для выделения в моделируемом предмете важного для решения конкретной задачи по предмету, в конечном счете — контекстное понимание предмета, формализуемое в виде класса;

Икапсуляция для быстрой и безопасной организации собственно иерархической управляемости: чтобы было достаточно простой команды «что делать», без одновременного уточнения как именно делать, так как это уже другой уровень управления;

Наследование для быстрой и безопасной организации родственных понятий: чтобы было достаточно на каждом иерархическом шаге учитывать только изменения, не дублируя все остальное, учтенное на предыдущих шагах; полиморфизм для определения точки, в которой единое управление лучше распараллелить или наоборот - собрать воедино.

18.Объектно-ориентированный анализ.

Объектно-ориентированный анализ- это методология, при которой требования к системе воспринимаются с точки зрения классов и объектов, выявленных в предметной области.

Объектно-ориентированное проектирование- это методология проектирования, соединяющая в себе процесс объектной декомпозиции и приемы представления логической и физической, а также статической и динамической моделей проектируемой системы.

В основе Объектно-ориентированное проектирование лежит представление о том, что программную систему необходимо проектировать как совокупность взаимодействующих друг с другом объектов, рассматривая каждый объект

как экземпляр определенного класса, причем классы образуют иерархию. Объектно-ориентированный подход отражает топологию языков высокого уровня.

Границы между стадиями анализа и проектирования размыты, но решаемые ими задачи определяются достаточно четко. В процессе анализа мы моделируем проблему, обнаруживая классы и объекты, которые составляют словарь проблемной области. При объектно-ориентированном проектировании мы изобретаем абстракции и механизмы, обеспечивающие поведение, требуемое моделью.

19.Абстрагирование.

Абстракция — в объектно-ориентированном программировании это придание объекту характеристик, которые отличают его от всех других объектов, четко определяя его концептуальные границы. Основная идея состоит в том, чтобы отделить способ использования составных объектов данных от деталей их реализации в виде более простых объектов, подобно тому, как функциональная абстракция разделяет способ использования функции и деталей её реализации в терминах более примитивных функций, таким образом, данные обрабатываются функцией высокого уровня с помощью вызова функций низкого уровня.

Абстрагирование означает выделение значимой информации и исключение из рассмотрения незначимой. В ООП рассматривают лишь абстракцию данных (нередко называя её просто «абстракцией»), подразумевая набор значимых характеристик объекта, доступный остальной программе. абстрагирование для выделения в моделируемом предмете важного для решения конкретной задачи по предмету, в конечном счете - контекстное понимание предмета, формализуемое в виде класса; Седвиц и Старк выделили дальнейший спектр абстракций:

- Абстракция сущности — Объект представляет собой пригодную модель некой сущности в предметной области
- Абстракция поведения — Объект состоит из обобщенного множества операций
- Абстракция виртуальной машины — Объект группирует операции, которые совместно применяются больше высоким уровнем управления, либо сами применяют определенный комплект операций больше низкого яруса
- Произвольная абстракция — Объект включает в себя комплект операций, не имеющих друг с ином ничего всеобщего

20.Инкапсуляция.

Инкапсуляция — свойство языка программирования, позволяющее пользователю не задумываться о сложности реализации используемого программного компонента (что у него внутри?), а взаимодействовать с ним посредством предоставляемого интерфейса (публичных методов и членов), а также объединить и защитить жизненно важные для компонента данные.

« Инкапсуляция – это процесс отделения друг от друга элементов объекта, определяющих его устройство и поведение; инкапсуляция служит для того, чтобы изолировать контрактные обязательства абстракции от их реализации »

При этом пользователю предоставляется только спецификация (интерфейс) объекта. Пользователь может взаимодействовать с объектом только через этот интерфейс. Реализуется с помощью ключевого слова: public. Пользователь не может использовать закрытые данные и методы. Реализуется с помощью ключевых слов: private, protected, internal. Инкапсуляция - один из четырёх важнейших механизмов объектно-ориентированного программирования (наряду с абстракцией, полиморфизмом и наследованием). Сокрытие реализации целесообразно применять в следующих случаях: предельная локализация изменений при необходимости таких изменений, прогнозируемость изменений (какие изменения в коде надо сделать для заданного изменения функциональности)
● прогнозируемость последствий изменений.

21.Модульность.

Модульность – система состоит из независимых компонент – модулей, для которых определены программные интерфейсы их взаимодействие, а сами они построены по принципу «черного ящика» - их внутреннее содержание скрыто от внешнего пользователя;

Модульное программирование – это организация программы как совокупности небольших независимых блоков, называемых модулями, структура и поведение которых подчиняются определённым 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.

Принцип модульности является средством упрощения задачи проектирования ПС и распределения процесса разработки ПС между группами разработчиков. При разбиении ПС 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. 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.

22.Иерархия.

Иерархия– одни из компонент системы могут быть использованы как составные части для построения более сложных компонент. В иерархических системах используется также принцип рекурсии – в компоненту могут входить в качестве составных частей компоненты такого же типа (а в нее – аналогичные, и т.д. до бесконечности);

Иерархия классовв информатике означает классификацию объектных типов, рассматривая объекты как реализацию классов (класс похож на заготовку, а объект — это то, что строится на основе этой заготовки) и связывая различные классы отношениями наподобие «наследует», «расширяет», «является его абстракцией», «определение интерфейса

Иерархия алгоритмов– библиотека функций. Функция в программе является также и элементом иерархии, поскольку последовательность вызовов функций сверху-вниз обычно соответствует иерархии решаемых в системе задач. Набор функция одного уровня, предназначенный для решения одного типа задач или работающих с данными одного типа, объединяются в библиотеки.

Иерархия данных– производный тип данных. По аналогии с алгоритмической компонентой программы данные также допускают

иерархическое описание (5.4), оно касается форм представления – типов данных. Производные типы данных конструируются из уже известных. Иерархия типов данных существует в программе параллельно с иерархией функций, их обрабатывающих.

В ***ООП используются два вида иерархии***.

1. Иерархия «целое/часть» – показывает, что некоторые абстракции включены в рассматриваемую абстракцию, как ее части, например, лампа состоит из цоколя, нити накаливания и колбы. Этот вариант иерархии используется в процессе разбиения системы на разных этапах проектирования (на логическом уровне – при декомпозиции предметной области на объекты, на физическом уровне - при декомпозиции системы на модули и при выделении отдельных процессов в мультипроцессной системе).

2. Иерархия «общее/частное» – показывает, что некоторая абстракция является частным случаем другой абстракции, например, «обеденный стол – конкретный вид стола», а «столы – конкретный вид мебели». Используется при разработке структуры классов, когда сложные классы строятся на базе более простых путем добавления к ним новых характеристик и, возможно, уточнения имеющихся.

23.Контроль типов.

Тип - это точная характеристика структуры и поведения, присущих некоторой совокупности объектов.

Контроль типов - это правила использования объектов, не допускающие или ограничивающие взаимную замену объектов разных классов. Контроль типов заставляет проектировщиков выражать свои абстракции так, чтобы язык программирования, используемый для реализации системы, поддерживал принятые проектные решения. Конкретный язык программирования может иметь сильный или слабый контроль типов, и даже совсем не иметь такого свойства, оставаясь объектно-ориентированным.

Основным понятием контроля типов является соответствие. Рассмотрим, например, абстракцию физической единицы измерения. Если разделить расстояние на время, мы получим число, означающее скорость, а не вес. Аналогично, деление единицы силы на температуру бессмысленно, а деление единицы силы на массу — нет. Эти примеры относятся к строгому контролю типов, поскольку правила исследуемой предметной области четко определены и регламентируют допустимые сочетания абстракций. Достоинства строгого контроля типов: позволяет использовать язык

программирования для поддержки определенных проектных решений и позволяет справиться с нарастающей сложностью систем. Недостатки: на практике она порождает семантические зависимости, при которых небольшое изменение интерфейса в базовом классе вынуждает повторную компиляцию всех его подклассов.

Сильный и слабый контроль типов, с одной стороны, и статический и динамический контроль типов, с другой стороны, это разные вещи. Строгий контроль типов означает соответствие типов, а статический контроль типов (иначе называемый статическим, или ранним связыванием, определяет момент времени на этапе компиляции, в который фиксируется тип переменных или выражений. Динамический контроль типов (называемый также поздним связыванием) означает, что типы переменных и выражений до момента выполнения программы остаются неизвестными.

24.Параллелизм.

Параллелизм - это свойство, отличающее активные объекты от пассивных. Наличие в системе нескольких потоков управления одновременно. Объект может быть активен, т. е. может порождать отдельный поток управления. Различные объекты могут быть активны одновременно.

В информатике параллелизм - это свойство систем, при котором несколько вычислений выполняются одновременно, и при этом, возможно, взаимодействуют друг с другом. Вычисления могут выполняться на нескольких ядрах одного чипа с вытесняющим разделением времени потоков на одном процессоре, либо выполняться на физически отдельных процессорах. Для выполнения параллельных вычислений разработаны ряд математических моделей, в том числе сети Петри, исчисление процессов, модели параллельных случайных доступов к вычислениям и модели акторов.

Есть системы, например реального времени, что должны обрабатывать много событий одновременно. В других случаях потребность в вычислительной мощности превышает ресурсы одного процессора. В каждой из таких ситуаций естественно использовать несколько компьютеров для решения задачи или задействовать многозадачность на многопроцессорном компьютере. Процесс (потек управление) - это фундаментальная единица действия в системе. Каждая программа имеет, по крайней мере, один поток управления, параллельная система имеет много таких потоков: век одних недолгий, а другие живут в течение всего сеанса работы системы. Реальная параллельность достигается лишь на многопроцессорных системах, а

системы с одним процессором имитируют параллельность за счет алгоритмов деления времени.

Каждый объект (полученный из абстракции реального мира) может являть собой отдельный поток управления (абстракцию процесса). Такой объект называется активным. Для таких систем мир может быть представлен, как совокупность взаимодействующих объектов, часть из которых является активной и выступает в роли независимых вычислительных центров.

Как только в систему введен параллелизм, сразу возникает вопрос о том, как синхронизировать отношения активных объектов друг с другом, а также с другими объектами, которые действуют последовательно. Например, если два объекта посылают сообщение третьему, должен быть какой-то механизм, который гарантирует, что объект, на который направлено действие, не разрушится во время одновременной попытки двух активных объектов изменить его состояние. В этом вопросе соединяются абстракция, инкапсуляция и параллелизм. В параллельных системах недостаточно определить поведение объекта, нужно еще принять меры, которые гарантируют, что он не будет изодран на части несколькими независимыми процессами. Все это привело к созданию таких механизмов как семафоры и критические секции.

25.Преимущества объектной модели.

Объектная модель принципиально отличается от моделей, которые связаны с более традиционными методами структурного анализа. Она не требует отказа от всех ранее найденных и испытанных временем методов и приемов, но вносит некоторые новые элементы, которые добавляются к предшествующему опыту, имея ряд существенных удобств, которые другими моделями не предусматривались. Вважно, что объектный подход позволяет создавать системы, которые удовлетворяют пяти признакам хорошо структурированных сложных систем, при этом еще имея пять преимуществ. Во-первых, объектная модель позволяет в полной мере использовать выразительные возможности объектных и объектно-ориентированных языков программирования. Опыт показал, что при использовании таких языков, как Smalltalk, Object Pascal, C++, CLOS

и Ada вне объектной модели, их наиболее сильные стороны либо игнорируются, либо применяются неправильно. Во-вторых, использование объектного подхода существенно повышает уровень унификации разработки и пригодность для повторного использования не только

программ, но и проектов, что в конце концов ведет к созданию среды разработки . Объектно-ориентированные системы часто получаются более компактными, чем их не объектно-ориентированные эквиваленты. А это означает не только уменьшение объема кода

программ, но и удешевление проекта за счет использования предыдущих разработок, что дает выигрыш в стоимости и времени. В-третьих, использование объектной модели приводит к построению систем на основе стабильных

промежуточных описаний, что упрощает процесс внесения изменений. Это дает системе возможность развиваться постепенно и не приводит к полной ее переработке даже в случае

существенных изменений исходных требований. В-четвертых, объектная модель уменьшает риск разработки сложных систем, прежде всего потому, что процесс интеграции растягивается на все

время разработки, а не превращается в единовременное событие. Объектный подход состоит из ряда хорошо продуманных этапов

проектирования, что также уменьшает степень риска и повышает уверенность в правильности принимаемых решений.

26.Природа объектов.

Термин объект в программном обеспечении впервые был введен в языке Simula и применялся для моделирования реальности. Объектами реального мира не исчерпываются типы объектов, интересные при проектировании программных систем. Другие важные типы объектов вводятся на этапе проектирования, и их взаимодействие друг с другом служит механизмом отображения поведения более высокого уровня. Существуют такие объекты, для которых определены явные концептуальные границы, но сами объекты представляют собой неосязаемые события или процессы. Рассмотрим систему пространственного проектирования CAD/CAM. Два тела, например, сфера и куб, имеют как правило нерегулярное пересечение. Хотя эта линия

пересечения не существует отдельно от сферы и куба, она все же является самостоятельным объектом с четко определенными концептуальными границами. Объект- это нечто, имеющее четко определенные границы, но этого недостаточно, чтобы отделить один объект от другого или дать оценку качества абстракции.

К числу свойств объекта относятся присущие ему
или приобретаемые им характеристики, черты, качества или способности, делающие данный объект самим собой. Например, для лифта характерным является то, что он сконструирован для поездок вверх и вниз, а не горизонтально. Перечень свойств объекта является, как правило, статическим, поскольку эти свойства составляют неизменяемую основу объекта. Все свойства имеют некоторые значения. Эти значения могут быть простыми количественными характеристиками, а могут ссылаться на другой объект.

Поведение объекта- это его наблюдаемая и проверяемая извне деятельность. Операцией называется определенное воздействие одного объекта на другой с целью вызвать соответствующую реакцию. Совокупность всех методов и свободных процедур, относящихся к конкретному объекту, образует протокол этого объекта. Протокол, таким образом, определяет поведение объекта, охватывающее все его статические и динамические аспекты.

Началом времени существования любого объекта является момент его создания (отведение участка памяти), а окончанием - возвращение отведенного участка памяти системе. Объекты создаются явно или неявно. Есть два способа создать их явно. Во-первых, это можно сделать при объявлении: тогда объект размещается в стеке. Во-вторых, можно разместить объект, то есть выделить ему память из "кучи". В C++ в любом случае при этом вызывается конструктор, который выделяет известное ему количество правильно инициализированной памяти под объект. Часто объекты создаются неявно. Так, передача параметра по значению в C++ создает в стеке временную копию объекта. Более того, создание объектов транзитивно: создание объекта тянет за собой создание других объектов, входящих в него. При явном или неявном уничтожении объекта
К C++ вызывается соответствующий деструктор. Его задача не только освободить память, но и решить,

что делать с другими ресурсами, например, с открытыми файлами.

27.Отношения между объектами.

Сами по себе объекты не представляют никакого интереса: только в процессе взаимодействия объектов реализуется система. Отношения двух любых объектов основываются на предположениях, которыми один обладает относительно другого: об операциях, которые можно выполнять, и об ожидаемом поведении. Особый интерес для объектно- ориентированного анализа и проектирования представляют два типа иерархических соотношений объектов: • связи • агрегация.

Связи

Объект сотрудничает с другими объектами через связи, соединяющие его с ними. Другими словами, связь- это специфическое сопоставление, через которое клиент запрашивает услугу у объекта-сервера или через которое один объект находит путь к другому.

Связь между объектами и передача сообщений обычно односторонняя. Участвуя в связи, объект может выполнять одну из следующих трех ролей:

• Актер. Объект может воздействовать на другие объекты, но сам никогда не подвергается воздействию других объектов; в определенном смысле это соответствует понятию активный объект

• Сервер. Объект может только подвергаться воздействию со стороны других объектов, но он никогда не выступает в роли воздействующего объекта

• Агент. Такой объект может выступать как в активной, так и в пассивной роли; как правило, объект-агент создается для выполнения операций в интересах какого-либо объекта-актера или агента.

Когда один объект посылает по связи сообщение другому, связанному с ним, они, как говорят, синхронизируются. В строго последовательном приложении синхронизация объектов и состоит в запуске метода.Однако в многопоточной системе объекты требуют более изощренной схемы

передачи сообщений, чтобы разрешить проблемы взаимного исключения, типичные для параллельных систем. Активные объекты выполняются как потоки, поэтому присутствие других активных объектов на них обычно не влияет. Если же активный объект имеет связь с пассивным, возможны следующие три подхода к синхронизации: • Последовательный- семантика пассивного объекта обеспечивается в присутствии только одного активного процесса.
• Защищенный- семантика пассивного объекта обеспечивается в присутствии многих потоков

управления, но активные клиенты должны договориться и обеспечить взаимное исключение.
• Синхронный- семантика пассивного объекта обеспечивается в присутствии многих потоков

управления; взаимное исключение обеспечивает сервер.

Агрегация

В то время, как связи обозначают равноправные или "клиент-серверные" отношения между объектами, агрегация описывает отношения целого и части, приводящие к соответствующей иерархии объектов, причем, идя от целого (агрегата), мы можем придти к его частям (атрибутам). В этом смысле агрегация - специализированный частный случай ассоциации. Агрегация может означать физическое вхождение одного объекта в другой, но не обязательно.

Выбирая одно из двух - связь или агрегацию - надо иметь в виду следующее. Агрегация иногда предпочтительнее, поскольку позволяет скрыть части в целом. Иногда наоборот предпочтительнее связи, поскольку они слабее и менее ограничительны. Принимая решение, надо взвесить все. Объект, являющийся атрибутом другого объекта (агрегата), имеет связь со своим агрегатом. Через эту связь агрегат может посылать ему сообщения.

28.Природа класса.

Понятия класса и объекта настолько тесно связаны, что невозможно говорить об объекте безотносительно к его классу. Однако существует важное различие этих двух понятий. В то время как объект обозначает конкретную сущность,

определенную во времени и в пространстве, класс определяет лишь абстракцию существенного в объекте.

Класс- это некое множество объектов, имеющих общую структуру и общее поведение. Важно отметить, что классы, необходимы, но не достаточны для декомпозиции сложных систем. Некоторые абстракции так сложны, что не могут быть выражены в терминах простого описания класса.

По своей природе,класс -это генеральный контракт между абстракцией и всеми ее клиентами. Выразителем обязательств класса служит его интерфейс, причем в языках с сильной типизацией потенциальные нарушения контракта можно обнаружить уже на стадии компиляции. Главное в интерфейсе - объявление операций, поддерживаемых экземплярами класса. К нему можно добавить объявления других классов, переменных, констант и исключительных ситуаций, уточняющих абстракцию, которую класс должен выражать. Мы можем разделить интерфейс класса

на три части: • открытую (public)- видимую всем клиентам; • защищенную (protected)- видимую самому классу, его подклассам и друзьям (friends); • закрытую (private)- видимую только самому классу и его друзьям.

Состояние объекта задается в его классе через определения констант или переменных, помещаемые в его защищенной или закрытой части. Тем самым они инкапсулированы, и их изменения не влияют на клиентов.

29.Отношения между классами.

Теория ООП выделяет три основных отношения между классами:

1. Ассоциация
2. Агрегация и композиция
3. Обобщение/Расширение (наследование)

Ассоциацияозначает, что объекты двух классов могут ссылаться один на другой, иметь некоторую связь между друг другом. пример — Преподаватель и Студент — т.е. какой-то Студент учится у какого-то Преподавателя. Ассоциация и есть описание

связи между двумя объектами. Студент учится
у Преподавателя. Идея достаточно простая —
два объекта могут быть связаны между собой и это надо как-то описать.

Агрегация и композиция– частные случаи ассоциации. Это более конкретизированные отношения между объектами.

Агрегация — отношение когда один объект является частью другого. Например Студент входит в Группу любителей физики.

Композиция — еще более «жесткое отношение, когда объект не только является частью другого объекта, но и вообще не может принадлежат еще кому-то. Например Машина и Двигатель. Хотя двигатель может быть и без машины, но он вряд ли сможет быть в двух или трех машинах одновременно. В отличии от студента, который может входить и в другие группы тоже. Такие описания всегда несколько условны, но тем не менее.

Наследованиеклассов позволяет создавать
производные классы (классы наследники), взяв
за основу все методы и элементы базового
класса (класса родителя). Таким образом
экономится масса времени на написание и
отладку кода новой программы. Объекты
производного
класса свободно могут использовать всё, что
создано и отлажено в базовом классе.

30.Взаимосвязь классов и объектов

Классы и объекты - тесно связанные понятия. Каждый объект - экземпляр какого-либо класса; класс может порождать любое число объектов. Любой созданный объект относится
к строго фиксированному классу. Объекты в процессе выполнения программы создаются и уничтожаются.

На этапе анализа и ранних стадиях проектирования решаются две основные задачи:

• Выявление классов и объектов, составляющих словарь предметной области.

• Построение структур, обеспечивающих взаимодействие объектов, при котором выполняются требования задачи.

В первом случае говорят о ключевых
абстракциях задачи (совокупность классов и объектов), во втором - о механизмах реализации (совокупность структур). На ранних стадиях внимание проектировщика сосредоточивается на внешних проявлениях ключевых абстракций и механизмов. Такой подход создает логический каркас системы: структуры классов и объектов. На последующих фазах проекта, включая реализацию, внимание переключается на внутреннее поведение ключевых абстракций и
механизмов, а также их физическое представление. Принимаемые в процессе проектирования решения задают архитектуру
системы: и архитектуру процессов, и архитектуру модулей.

31.Основные абстракции и механизмы.

Ключевые абстракции отражают словарь предметной области; их находят либо в ней самой, либо изобретают в процессе проектирования. Механизмы обозначают стратегические проектные решения относительно совместной деятельности объектов многих различных типов.

Абстракция – совокупность существенных характеристик некоторого объекта, которые отличают его от всех других видов объектов и, таким образом, четко определяют особенности данного объекта с точки зрения дальнейшего рассмотрения и анализа.
Современный уровень абстракции предполагает объединение всех свойств абстракции (как касающихся состояния анализируемого объекта, так и определяющих его поведение) в единую программную единицу некий абстрактный тип (класс).

Ключевая абстракция- это класс или объект,
который входит в словарь проблемной области.

Главная ценность ключевых абстракций - они определяют границы нашей проблемы: выделяют то, что входит в нашу систему и устраняют лишнее. Определение ключевых абстракций включает в себя два процесса: открытие и изобретение. Наиболее мощный

способ выделения ключевых абстракций - сводить задачу к уже известным классам и объектам.

Классы и объекты должны быть на надлежащем уровне абстракции: не слишком высоко и не слишком низко последовательных приближений. Очень важно отразить в обозначении классов и объектов сущность описываемых ими предметов.

Мы предлагаем следующие правила:

•Объекты следует называть существительными:
theSensor или shape.

•Классы следует называть обобщенными существительными:
Sensors, Shapes.

•Операции-модификаторы следует называть активными глаголами:
Draw, moveLeft.

•У операций-селекторов в имя должен включаться запрос или форма глагола "to be":
extentOf, isOpen.

Механизмыпредставляют шаблоны поведения. Ключевые абстракции определяют словарь проблемной области, механизмы определяют суть проекта. Механизмы, представляют собой стратегические решения в проектировании

Механизмы - суть средства, с помощью которых объекты взаимодействуют друг с другом для достижения необходимого поведения более высокого уровня кивая друг друга.

32.Язык UML.

UML (Unified Modeling Language) предназначен для упрощения взаимодействия участников проекта, сокращения времени на объяснение и усвоение информации, облегчения документирования. UML - графическая нотация, предназначенная для описания и моделирования процессов, протекающих в ходе разработки

Словарь UML состоит из следующих элементов:

- Сущности - это абстракции, которые являются основными элементами моделей;
- Отношения - это связи между сущностями;

- Диаграммы - это отражение взаимодействия сущностей и отношений.

1. Сущности.

Сущности в UML могут быть 4 типов:

- Структурныесущности являются именами существительными модели, ее статические части. Это классы, компоненты, интерфейсы и т.д., которые соответствуют физическим элементам системы.
- Поведенческиесущности являются глаголами модели, описывают ее поведение во времени и пространстве. Их существует всего 2 типа: взаимодействие (обмен сообщениями) и автомат (последовательность состояний).
- Аннотационныесущности являются пояснительными частями модели.Это примечания и комментарии к элементам системы.
- Группирующиесущности - организующие части модели. Они организуют элементы системы в группы.

2. Отношения.

Существует 4 типа отношений в UML:

- Зависимость- это отношение между двумя сущностями, при котором изменение одной (независимой) сущности приводит к изменению второй сущности. Графически это изображается пунктирной стрелкой.
- Обобщение- это отношение “специализации-обобщения”, где специализированный объект может быть подставлен вместо обобщенного.
- Ассоциация- отношение, описывающее семантическую связь между объектами. Графически изображается в виде сплошной стрелки, которая может содержать кратность или имена ролей. (Агрегация - разновидность ассоциации и отражает отношение части к целому).
- Композиция- разновидность агрегации, где взаимосвязь части с целым еще более сильная.
- Реализация- отношение между классификаторами, при котором один

определяет обязательство, а второй осуществляет его выполнение.

3. Диаграммы.

в UML существует 10 типов диаграмм:

- Объектов (object);
- Классов (class);
- Взаимодействия (interaction):
- Вариантов использования (use-case);
- Последовательности (sequence);
- Состояний (statechart);
- Кооперaций (collaboration);
- Компонентов (component);
- Развертывания (размещения) (deployment);
- Деятельностей (activity).

32 ЯЗЫК UML

UML – **это унифицированный графический язык моделирования для описания, визуализации, проектирования и документирования ОО систем**. UML

призван поддерживать процесс моделирования ПС на основе ОО подхода, организовывать взаимосвязь концептуальных и программных понятий, отражать проблемы масштабирования сложных систем. Модели на UML **используются на всех этапах жизненного цикла ПС, начиная с бизнес-анализа и заканчивая сопровождением системы.**

Отличительной его особенностью является то, что словарь языка образуют графические элементы.

Каждому графическому символу соответствует конкретная семантика, поэтому модель, созданная одним разработчиком, может однозначно быть понята другим, а также программным средством,

интерпретирующим UML. **Отсюда, в частности, следует, что модель ПС, представленная на** UML,
может автоматически быть переведена на ОО язык программирования (такой, как Java, C++, VisualBasic)

Модель представляется в виде сущностей и отношений между ними, которые показываются на диаграммах. (Сущности– это абстракции, являющиеся основными элементами моделей.

- **Имеется четыре типа сущностей:**
- **структурные (класс, интерфейс, компонент, вариант использования, кооперация, узел)**
- **поведенческие (взаимодействие, состояние)**
- **группирующие (пакеты)**
- **аннотационные (комментарии). Каждый вид сущностей имеет свое графическое представление.**

Отношенияпоказывают различные связи между сущностями. В UML **определены следующие типы отношений:**

- ***Зависимость*показывает такую связь между двумя сущностями, когда изменение одной из них – независимой – может повлиять на семантику другой – зависимой. Зависимость изображается пунктирной стрелкой,**
  направленной от зависимой сущности к независимой.
- ***Ассоциация*– это структурное отношение, показывающее, что объекты одной сущности связаны с объектами другой. Графически ассоциация показывается в виде линии, соединяющей связываемые сущности.**

  Ассоциации служат для осуществления навигации между объектами. Например, ассоциация между классами «Заказ» и «Товар»

  может быть использована для нахождения всех товаров, указанных в конкретном заказе – с одной стороны, или для нахождения всех заказов в которых есть данный товар, – с другой. Понятно, что в соответствующих программах должен быть реализован механизм, обеспечивающий такую навигацию.

  Если требуется навигация только в одном направлении, оно показывается стрелкой на конце ассоциации. Частным случаем ассоциации является агрегирование –

отношение вида «целое» – «часть». Графически оно выделяется с помощью ромбика на конце около сущности-целого.

- ***Обобщение*– это отношение между сущностью-родителем и сущностью-потомком. По существу, это отношение отражает свойство наследования для классов и объектов. Обобщение показывается в виде линии, заканчивающейся треугольничком направленным к родительской сущности. Потомок наследует структуру (атрибуты) и поведение (методы) родителя, но в то же время он может иметь новые элементы структуры и новые методы. UML допускает множественное наследование, когда сущность связана более чем с одной родительской сущностью.**
- ***Реализация*– отношение между сущностью,**

  определяющей спецификацию поведения (интерфейс) с сущностью, определяющей реализацию этого поведения (класс, компонент). Это отношение обычно используется при моделировании компонент и будет подробнее описано в последующих статьях.

Диаграммы.В UML **предусмотрены следующие диаграммы**:

- **Диаграммы, описывающие поведение системы**:
  - **Диаграммы состояний** (State diagrams),
  - **Диаграммы деятельностей** (Activity diagrams),
  - **Диаграммы объектов** (Object diagrams),
  - **Диаграммы последовательностей** (Sequence diagrams),
  - **Диаграммы взаимодействия** (Collaboration diagrams);

- **Диаграммы, описывающие физическую реализацию системы**:
  - **Диаграммы компонент** (Component diagrams);
  - **Диаграммы развертывания** (Deployment diagrams).

33 ДИАГРАММЫ ПАКЕТОВ

Диаграммой пакетов является диаграмма, содержащая пакеты классов и зависимости между ними.

Зависимость между двумя элементами имеет место
в том случае, если изменения в определении одного элемента могут повлечь за собой изменение в другом. Что касается классов, то причины зависимостей могут быть самыми разными: один класс посылает сообщение другому; один класс включает часть данных другого класса… Если класс меняет свой интерфейс, то любое сообщение, которое он посылает, может стать неправильным.

Зависимость между двумя пакетами существует в том случае, если имеется какая-либо зависимость между любыми двумя классами в пакетах.

Пакеты являются жизненно необходимым средством для больших проектов. Их следует использовать в тех случаях, когда диаграмма классов, охватывающая всю систему в целом и размещенная на единственном листе бумаги формата А4, становится трудночитаемой.

Пакеты не дают ответа на вопрос, каким образом можно уменьшить количество зависимостей в разрабатываемой системе, однако они помогают выделить эти зависимости. Сведение к минимуму количества зависимостей позволяет снизить связанность компонентов системы. Но эвристический подход к этому процессу далек от идеала.

Пакеты особенно полезны при тестировании.
Каждый пакет при тестировании может содержать один или несколько тестовых к

34 ДИАГРАММЫ КОМПОНЕНТОВ

Компоненты на диаграмме компонентов представляют собой физические модули программного кода. Обычно они в точности соответствуют пакетам на диаграмме пакетов; таким образом, диаграмма компонентов отражает выполнение каждого пакета в системе.

Зависимости между компонентами должны совпадать с зависимостями между пакетами. Эти зависимости показывают, каким образом одни

компоненты взаимодействуют с другими. Направление данной зависимости показывает уровень осведомленности о коммуникации.

Диаграмма компонентов описывает особенности физического представления системы, позволяет определить архитектуру разрабатываемой системы, установив зависимости между программными компонентами, в роли которых может выступать исходный, бинарный и исполняемый код.
Во многих средах разработки модуль или компонент соответствует файлу.
Пунктирные стрелки, соединяющие модули, показывают отношения взаимозависимости, аналогичные тем,
которые имеют место при компиляции исходных текстов программ.
Основными графическими элементами диаграммы компонентов являются компоненты, интерфейсы и зависимости между ними.

В разработке диаграмм компонентов участвуют
как системные аналитики и архитекторы, так и программисты.
Диаграмма компонентов обеспечивает согласованный переход от логического представления к конкретной
реализации проекта в форме программного кода.
Одни компоненты могут существовать только на
этапе компиляции программного кода, другие – на этапе его исполнения.
Диаграмма компонентов отражает общие зависимости между компонентами, рассматривая последние в качестве отношений между ними.

Диаграмма компонентов разрабатывается для следующих целей:

1. Визуализации обшей структуры исходного кода программной системы.
2. Спецификации исполнимого варианта программной системы.
3. Обеспечения многократного использования отдельных фрагментов
4. программного кода.
5. Представления концептуальной и физической схем баз данных.

35 ДИАГРАММЫ РАЗВЕРТЫВАНИЯ

Диаграммы развертывания представляют физическое расположение системы, показывая, на каком физическом оборудовании запускается та или иная составляющая программного обеспечения.

Главными элементами диаграммы являются узлы, связанные информационными путями. Узел (node) – это то, что может содержать программное обеспечение. Узлы бывают двух типов. Устройство (device) – это физическое оборудование: компьютер или устройство, связанное с системой. Среда выполнения (execution environment) – это программное обеспечение, которое само может включать другое программное обеспечение, например операционную систему или процесс-контейнер.

Узлы могут содержать артефакты (artifacts), которые являются физическим олицетворением программного обеспечения; обычно это файлы. Такими файлами могут быть исполняемые файлы (такие как файлы *.ёхе*, двоичные файлы, файлы DLL, файлы JAR, сборки или сценарии) или файлы данных, конфигурационные файлы, HTML-документы и т. д.

Перечень артефактов внутри узла указывает на то, что на данном узле артефакт разворачивается в запускаемую систему.

Информационные пути между узлами представляют обмен информацией в системе. Можно сопровождать эти пути информацией об используемых информационных протоколах.

36 ДИАГРАММЫ ПРЕЦЕДЕНТОВ ИСПОЛЬЗОВАНИЯ

Суть данной диаграммы состоит в следующем:

проектируемая система представляется в виде множества актеров, взаимодействующих с системой с помощью так называемых вариантов использования. При этом *актером*(действующим лицом, актантом, актором) называется любой объект, субъект или система, взаимодействующая с моделируемой системой извне. В свою очередь *вариант использования*– это спецификация сервисов (функций), которые система предоставляет актеру. Другими словами, каждый вариант использования определяет некоторый набор действий, совершаемых системой при взаимодействии с актером. При этом в модели никак не отражается то, каким образом будет реализован этот набор действий.

Согласно UML *актера* графически можно отобразить тремя способами:

«проволочный человечек»

класс с текстовым стереотипом «actor»

произвольная иконка

Вариант использования, который инициализируется по запросу пользователя,
представляет собой законченную последовательность действий. Это означает, что после того, как система закончит обработку запроса актера, она должна возвратиться в состояние, в котором готова к выполнению следующих запросов.

Варианты использования могут включать в себя описание особенностей способов реализации сервиса и различных исключительных ситуаций, таких как корректная обработка ошибок системы.

Связи между актерами и вариантами отображаются с использованием *отношений* четырех видов:

- ассоциаций; (служит для обозначения взаимодействия актера с вариантом использования.)

- обобщения; (служит для указания того факта, что некоторая сущность А может быть обобщена до сущности В. В этом случае сущность А будет являться специализацией сущности В. На диаграмме данный вид отношения можно отображать только между однотипными сущностями)

- включения; (указывает, что некоторое заданное поведение одного варианта использования *обязательно* включается в качестве составного компонента в последовательность поведения другого варианта использования.)

- расширения. (определяет *потенциальную возможность* включения поведения одного варианта использования в состав другого.)

Ввиду того, что допускаемая скорость в кривых участках пути зависит в том числе и от возвышения наружного рельса, перед определением допускаемых скоростей может потребоваться определение и установление новых возвышений, которые в свою очередь зависят от структуры пропускаемого поездопотока.

Варианты использования, которые расширяют базовый, подключаются к нему (активируются при его выполнении) через так называемые точки расширения. Каждая точка расширения маркируется меткой и условием

активации. Обычно перечень точек расширения указывается в базовом варианте использования ниже горизонтальной линии

37 ДИАГРАММЫ ДЕЯТЕЛЬНОСТИ

В UMLаналогом блок-схем являются диаграммы деятельности (активности), схожие с ними по своей семантике и выразительным средствам (набору элементов).

Каждая диаграмма деятельности акцентирует внимание на последовательности выполнения определенных действий, которые в совокупности приводят к получению желаемого результата. Они могут быть построены для отдельного варианта использования, кооперации, метода и т. д.

Диаграммы деятельности являются разновидностью диаграмм автоматов, но если на второй основное внимание уделяется статическим состояниям, то на первой – действиям.

Каждая диаграмма деятельности акцентирует внимание на последовательности выполнения определенных действий, которые в совокупности приводят к получению желаемого результата. Они могут быть построены для отдельного варианта использования, кооперации, метода и т. д.

Диаграммы деятельности являются разновидностью диаграмм автоматов, но если на второй основное внимание уделяется статическим состояниям, то на первой – действиям.

*Основными элементами*диаграммы являются:

- исполняемые узлы; (К *исполняемым узлам*относятся *действия*и *деятельности*. Обычное использование исполняемых узлов заключается в моделировании одного шага выполнения алгоритма (процедуры) или потока управления)

- объекты; (К *объектам*относятся непосредственно объекты в традиционном понимании UML, отправка сигнала, прием сигнала и событие времени)

- переходы; (как и на диаграмме автоматов, отображается ассоциацией. На диаграммах деятельности различают следующие виды переходов: поток управления, объектный поток, поток прерывания или исключения)

- управляющие узлы; (на диаграмме деятельности соответствуют псевдосостояния на диаграмм е автоматов.)

- коннекторы; (выступают в качестве соединителей, применяемых на блок-схемах. Они используются для прерывания потока в одной части диаграммы и продолжении в другой, если диаграмма занимает несколько листов или отображение потока перенасыщает диаграмму.)

-группирующие элементы. (К *группирующим элементам*относятся разделы деятельности и прерываемые регионы. *Разделы деятельности*обычно используют для моделирования бизнес-процессов или совместной работы нескольких сущностей (актеров, объектов, компонентов, узлов и т.д.). В этом случае диаграмма делится на разделы (области) вертикальными или горизонтальными линиями, в заголовке которых указываются имена сущностей, ответственных за выполнение действий внутри соответствующего раздела. *Прерываемый регион* группирует действия, обычная последовательность выполнения которых может прервана в результате наступления нестандартной ситуации (например, при оформлении кредита клиент от него отказывается). Он отображается четырехугольником со скругленными углами и штриховым контуром.)

38 ДИАГРАММЫ КЛАССОВ

В UML диаграмма классовявляется типом диаграммы статической структуры.Она описывает структуру

системы, показывая её классы, их атрибуты и опера торы, а также взаимосвязи этих классов.

Взаимосвязь — это особый тип логических отношен ий между сущностями, показанных на диаграммах к лассови объектов. В UML представлены следующие ви ды отношений:

*Ассоциация*показывает, что объекты одной сущн ости (класса) связаны с объектами другой сущност и.
Существует пять различных типов ассоциации. Наи более распространёнными являются двунаправле нная иоднонаправленная.

Двойные ассоциации (с двумя концами) представ ляются линией, соединяющей два классовых блока.
Ассоциации более высокой степени имеют более дв ух концов и представляются линиями, один конец к оторыхидет к классовому блоку, а другой к общему ромбику. В представлении однонаправленной ассо циациидобавляется стрелка, указывающая на напр авление ассоциации.

Агрегация— это разновидность ассоциации при отношении между целым и его частями. Как тип ассо циацииагрегация может быть именованной.
Одно от ношение агрегации не может включать более двух классов (контейнер и содержимое).

Агрегация встречается, когда один класс являе тся коллекцией или контейнером других. Причём п оумолчанию, агрегацией называют *агрегацию по сс*
ылке,то есть когда время существования содержа
щихся
классов не зависит от времени существования сод
ержащего их класса. Если контейнер будет уничтожен, то его содержимое — нет.

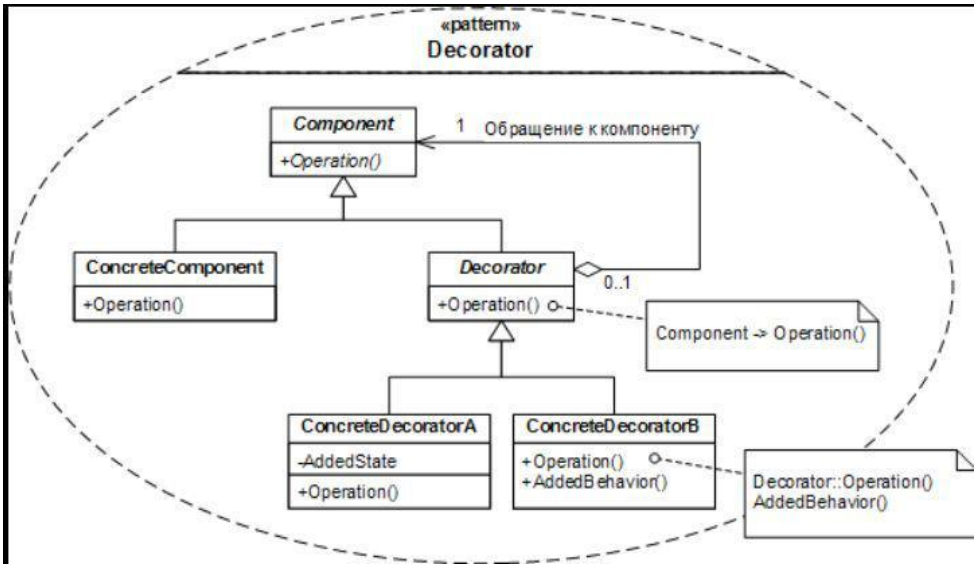
Композиция— более строгий вариант агрегации.
Известна также как агрегация по значению.
*Композиция*имеет жёсткую зависимость времени с уществования экземпляров класса контейнера иэ кземпляров содержащихся классов. Если контейне
р будет уничтожен, то всё его содержимое будет та кжеуничтожено.

41.Диаграммы композитных структур.

Диагра́мма компози́тной/составно́й структу́ры, Composite structure diagram — статическая структурная диаграмма, демонстрирует внутреннюю структуру классов и, по возможности, взаимодействие элементов (частей) внутренней структуры класса.

Подвидом диаграмм композитной структуры являются диаграммы кооперации (Collaboration diagram, введены в UML 2.0), которые показывают роли и взаимодействие классов в рамках кооперации. Кооперации удобны при моделировании шаблонов проектирования.

Диаграммы композитной структуры могут использоваться совместно с диаграммами классов.



42.Диаграммы конечных автоматов.

Диаграммы автоматов (англ. state machine) используются для описания поведения, реализуемого в рамках варианта использования, или поведения экземпляра сущности (класса, объекта, компонента, узла или системы в целом) [26].
Поведение моделируется через описание возможных состояний экземпляра сущности и переходов между ними на протяжении его жизненного цикла, начиная от создания и заканчивая уничтожением. Диаграмма автоматов представляет собой связный ориентированный граф, вершинами которого являются состояния, а дуги служат для обозначения переходов из состояния в состояние.

1. Под состоянием (англ. state) понимается ситуация в ходе жизни экземпляра сущности, когда эта ситуация удовлетворяет некоторому условию, экземпляр выполняет некоторые операции или ждет наступления некоторого события. Например, для объекта его состояние может быть задано в виде набора конкретных значений атрибутов, при этом изменение этих значений будет приводить к изменению состояния моделируемого объекта.

В UML различают два вида операций: действие и деятельность. Действие (англ. action) – это атомарная операция, выполнение которой не может быть прервано, приводящая к смене состояния или возвращающая значение. Примерами действий служат операции создания или уничтожения объекта, расчет факториала и т. д. Деятельность (англ. activity) – это составная (неатомарная) операция, реализуемая экземпляром в конкретном состоянии, выполнение которой может быть прервано. В частности, под деятельностью можно понимать процедуры расчета допускаемых скоростей или шифрования данных.

Событие (англ. event) – это спецификация существенного факта, который может произойти в конкретный момент времени. События могут быть внутренними или внешними. Внешние события передаются между системой и актерами (например, нажатие кнопки или посылка сигнала от датчика передвижений). Внутренние события передаются между объектами внутри системы. В UML можно моделировать следующие виды событий:

- посылка сообщения (англ. message):
- вызов (англ. call);
- сигнал (англ. signal);
- любое сообщение (англ. any receive);
- событие времени (англ. time);
- изменение состояния (англ. change).

Вызов – спецификация факта посылки синхронного сообщения между объектами, предписывающего выполнение операции (действия или деятельности) объектом, которому посылается сообщение. Синхронность означает, что после посылки вызова объект-отправитель передает управление объекту-получателю и после выполнения

последним операции получает управление обратно. Например, закрасить фигуру красным фоном fill(red) или рассчитать допускаемые скорости calculateVdop().

Сигнал – спецификация факта посылки асинхронного сообщения между объектами. Исключения, которые поддерживаются в большинстве современных языков программирования, являются наиболее распространенным видом внутренних сигналов.

Событие времени – спецификация факта, обозначающего наступление конкретного момента времени (англ. absolute time) или истечение определенного промежутка времени (англ. relative time). В UML данный факт обозначается с помощью ключевых слов «at» (например, at 9:00:00) и «after» (например, after 2 seconds).

Изменение состояния – спецификация логического условия, соответствующего изменению состояния экземпляра сущности. В UML оно обозначается с помощью ключевого слова «when» (например, when A < B) или сторожевого условия (например, [A < B]).

2. Переход (англ. transition) – отношение между двумя состояниями, показывающее возможный путь изменения состояния экземпляра сущности.
Считается, что в состоянии экземпляр сущности находится продолжительное время, а переход выполняется мгновенно.

Переход отображается в виде однонаправленной ассоциации между двумя состояниями. При смене состояний говорят, что переход срабатывает. До срабатывания перехода экземпляр сущности находится в состоянии, называемом исходным, а после его срабатывания – в целевом.

Различают два вида переходов: нетриггерный и триггерный. Переход первого вида, называемый также переходом по завершении, срабатывает неявно, когда все основные операции (с метками entry, do и exit) в исходном состоянии успешно завершают свою работу. Данный вид перехода обозначается стрелкой без надписи. Для наступления триггерного перехода необходимо наступление некоторого события, которое записывается над стрелкой. В общем случае над стрелкой может быть записана строка текста вида «событие [сторожевое условие] / действие». Указываемое действие представляет собой атомарную операцию, выполняемую сразу после срабатывания

соответствующего перехода и до начала каких бы то ни было операций в целевом состоянии. Разрешается указывать не одно, а несколько обособленных действий, отделенных друг от друга точкой с запятой. Обязательное требование – все действия в списке должны четко различаться между собой и следовать в порядке их записи.

43.Диаграммы синхронизации.

Диаграмма синхронизации (timing diagram) представляет собой особую форму диаграммы последовательности, на которой особое внимание уделяется изменению состояний 1 различных экземпляров классификаторов и их временной синхронизации 2.

Описание диаграммы последовательности

Диаграмма последовательностей является наиболее распространенным видом диаграммы взаимодействия, которая фокусируется на обмене сообщениями между несколькими линиями жизнеобеспечения.

На диаграмме последовательности изображаются только те объекты, которые непосредственно участвуют во взаимодействии. Ключевым моментом для диаграмм последовательности является динамика взаимодействия объектов во времени.

Диаграмма последовательности является одной из разновидности диаграмм взаимодействия и предназначена для моделирования взаимодействия объектов Системы во времени, а также обмена сообщениями между ними.

Одним из основных принципов ООП является способ информационного обмена между элементами Системы, выражающийся в отправке и получении сообщений друг от друга. Таким образом, основные понятия диаграммы последовательности связаны с понятием Объект и Сообщение.

На диаграмме последовательности, каждый участник представлен вместе со своей линией жизни (lifeline), это вертикальная линия под объектом, вертикально упорядочивающая сообщения на странице. Важно: все сообщения на диаграмме следует читать сверху вниз. Каждая линия жизни

имеет полосу активности (прямоугольники), которая показывает интервал активности каждого участника при взаимодействии.

Участники диаграммы именуются следующим образом: имя : Класс, где и имя, и класс являются не обязательными, но если используется класс, то присутствие двоеточия обязательно.

Сообщения можно разделить на 2 вида: синхронные (synchronous message) – требующие возврата ответа и асинхронные (asynchronous message) – ответ не тебуется и вызывающий объект может продолжать работу. На диаграмме синхронные вызовы обозначаются закрашенными стрелочками. Асинхронные – незакрашенными или половинными стрелочками.

У первого сообщения, в нашей диаграмме, нет участника, пославшего его, поскольку оно приходит от неизвестного источника. Такое сообщение называется найденным сообщением (found message).

44.Диаграммы объектов.

Диаграмма объектов (object diagram) – это снимок объектов системы в какой-то момент времени. Поскольку она показывает экземпляры, а не классы, то диаграмму объектов часто называют диаграммой экземпляров.

Диаграмму объектов можно использовать для отображения одного из вариантов конфигурации объектов. (На рис. 6.1 показано множество классов, а на рис. 6.2 представлено множество связанных объектов.) Последний вариант очень полезен, когда допустимые связи между объектами могут быть сложными.

Можно определить, что элементы, показанные на рис. 6.2, являются экземплярами, поскольку их имена подчеркнуты. Каждое имя представляется в виде: имя экземпляра : имя класса. Обе части имени не являются обязательными, поэтому имена John, : Person и aPerson являются допустимыми. Если указано только имя класса, то необходимо поставить двоеточие. Можно также задать значения и атрибуты, как показано на рис. 6.2.

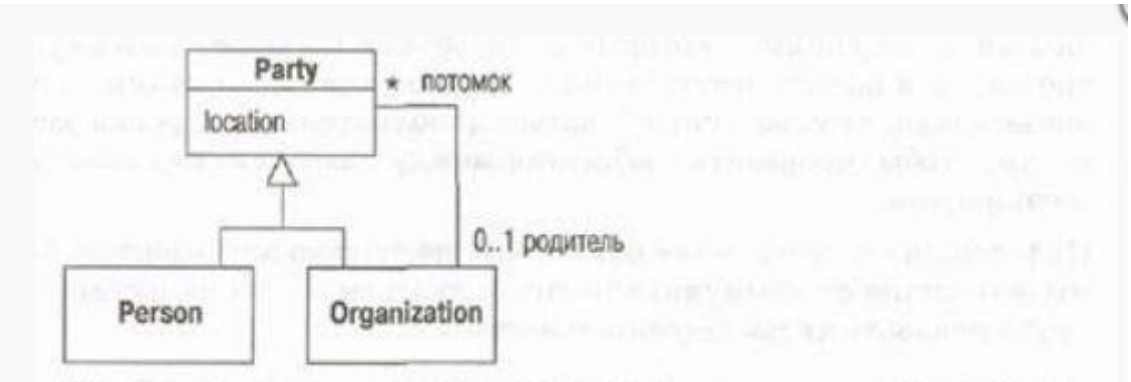


Рис. 6.1. Диаграмма классов, показывающая структуру класса Party (вечеринка)

Строго говоря, элементы диаграммы объектов – это спецификации экземпляров, а не сами экземпляры. Причина в том, что разрешается оставлять обязательные атрибуты пустыми или показывать спецификации экземпляров абстрактных классов. Можно рассматривать спецификации экземпляров (instance specifications) как частично определенные экземпляры.

С другой стороны, диаграмму объектов можно считать коммуникационной диаграммой (стр. 152) без сообщений.

Когда применяются диаграммы объектов

Диаграммы объектов удобны для показа примеров связанных друг с другом объектов. Во многих ситуациях точную структуру можно определить с помощью диаграммы классов, но при этом структура остается трудной для понимания. В таких случаях пара примеров диаграммы объектов может прояснить ситуацию

45.Диаграммы коммуникаций.

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

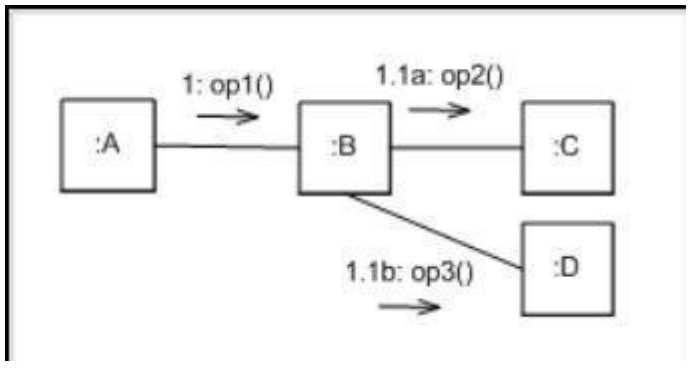
В UML есть четыре типа диаграмм взаимодействия:

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

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

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

Диаграмма коммуникации показывает во многом ту же информацию, что и диаграмма последовательности, но из-за другого способа представления информации какие-то вещи на одной диаграмме видеть проще, чем на другой. Диаграмма коммуникаций нагляднее показывает, с какими элементами взаимодействует каждый элемент, а диаграмма последовательности яснее показывает в каком порядке происходят взаимодействия.



46.Микро и макро процессы.

Микропроцесс проектирования

Микропроцесс состоит из следующих видов деятельности:

1. Выявление классов и объектов на данном уровне абстракции.
2. Выяснение семантики этих классов и объектов.
3. Выявление связей между этими классами и объектами.

4. Спецификация интерфейса и реализация этих классов и объектов.

Макропроцесс проектирования

Является контролирующим по отношению к микропроцессу.

Включает следующие действия:

1. Выявление сущности и требований к программному продукту (концептуализация).

2. Разработка модели требуемого поведения системы (анализ)

3. Создание архитектуры для реализации (проектирование).

4. Итеративное выполнение реализации (эволюция).

5. Управление эволюцией продукта в ходе эксплуатации (сопровождение).

47.Практическое использование.

Проблемы, которые решает объектно-ориентированное проектирование

Рассмотрим ряд типичных проблем больших проектов, которые объектно-ориентированный подход существенно упрощает.

Большая и сложная задача. Это естественно, иначе проект не был бы большим. Когда мы разрабатываем программную систему, мы разрабатываем программную модель предметной области - части реального мира. Самая большая сложность состоит в семантическом разрыве между реальностью и программой. Объектная модель уменьшает этот разрыв. Реальный мир естественно описывать как набор взаимодействующих объектов. Такое описание, с одной стороны, понятно конечному потребителю и эксперту, с другой - легко ложится на объектную модель, а следовательно, его легко реализовать.

Постоянные изменения задания. Чаще всего изменяются не сами объекты, участвующие в системе, а протоколы их взаимодействия. В этом случае основа системы остается без изменения и реализация изменений оказывается существенно проще. Если же изменения касаются какого-либо объекта, они будут локальны для данного класса.

Слишком большая длительность разработки. Объектно-ориентированный подход стимулирует многократное использование программного обеспечения. Количество существенно различных объектов не так уж и велико. Иногда можно применить имеющиеся классы в готовом виде или

объекты в качестве компонентов других объектов. Механизм наследования классов позволяет использовать имеющиеся классы в качестве базовых, из которых выводятся новые, специализированные на конкретное применение. Кроме того, широкое распространение объектно-ориентированного подхода привело к огромному числу предлагаемых готовых библиотек классов, как универсального характера, так и ориентированных на различные сферы бизнеса.

Сложность сопровождения. Использование готовых, многократно проверенных классов уменьшает количество ошибок. Структура программ, основанная на объектах, отображающих объекты реального мира, уменьшает вероятность появления побочных неожиданных эффектов от изменений. Также здесь можно повторить аргументы о сравнительной простоте изменения системы.