ООП

"Programming today is a race between software engineers striving to build bigger and better idiot-proof programs, and the Universe trying to produce bigger and better idiots. So far, the Universe is winning."

Rich Cook

Принципы ОО представления программных систем

Рассмотрение сложной системы - применение декомпозиции - разбиения на составляющие элементы.

Схемы декомпозиции: алгоритмическая и ОО.

В основе алгоритмической - разбиение по действиям - алгоритмам.

ОО - разбиение по объектам реального (или виртуального) мира. Эти объекты - более «крупные» элементы, каждый из них несет в себе и описания действий, и описания данных.

ОО анализ и проектирование отличаются от структурного подхода: другой процесс декомпозиции, другая архитектура программного продукта.
Структурное проектирование основано на структурном программировании, в основе ОО проектирования - методология ОО программирования.

ОО технология основывается на *объектной модели.*
Основные принципы: абстрагирование, инкапсуляция, модульность, иерархичность, типизация, параллелизм и сохраняемость.

Каждый принцип сам по себе не нов, но в объектной модели они впервые применены в совокупности.

Термин «ОО программирование" означает разное.

Ренч [1981]: *"В 1980-х годах ОО программирование будет занимать такое же место, какое занимало структурное программирование в 1970-х, но всем будет нравиться. Каждая фирма будет рекламировать свой продукт как созданный по этой технологии. Все программисты будут писать в этом стиле, причем все по-разному. Все менеджеры будут рассуждать о нем. И никто не будет знать, что же это такое"*

Данные предсказания продолжают сбываться.

Эволюция объектной модели

Тенденции в проектировании

Поколения языков программирования:

- смещение акцентов от программирования отдельных деталей к программированию более крупных компонент;
- развитие и совершенствование языков программирования высокого уровня.

Тенденция перехода от языков, указывающих компьютеру, что делать (императивные языки), к языкам, описывающим ключевые абстракции проблемной области (декларативные языки).

Поколения ЯП высокого уровня в зависимости от языковых конструкций, которые впервые в них появились (Вегнер):

- ***Первое поколение (1954-1958)***

FORTRAN I, ALGOL-58, Flowmatic, IPL V - Математические формулы

- ***Второе поколение (1959-1961)***

FORTRAN II - Подпрограммы, раздельная компиляция

ALGOL-60 - Блочная структура, типы данных

COBOL - Описание данных, работа с файлами

Lisp - Обработка списков, указатели, сборка мусора

- ***Третье поколение(1962-1970)***

PL/I - FORTRAN+ALGOL+COBOL

ALGOL-68 - Более строгий приемник ALGOL-60

Pascal - Более простой приемник ALGOL-60

Simula - Классы, абстрактные данные

- ***Потерянное поколение (1970-1980)***

Много языков созданных, но мало выживших

Основные положения объектной модели Йонесава и Токоро: «*термин "объект" появился практически независимо в различных областях, связанных с компьютерами, и почти одновременно в начале 70-х годов для обозначения того, что может иметь различные проявления, оставаясь целостным. Чтобы уменьшить сложность программных систем, объектами назывались компоненты системы или фрагменты представляемых знании*».

ОО подход был связан с событиями [Леви]:

- прогресс в области архитектуры ЭВМ;
- развитие ЯП: Simula, Smalltalk, CLU, Ada;
- развитие методологии программирования, включая принципы модульности и скрытия данных.

На становление объектного подхода оказали влияние:

- развитие теории баз данных;
- исследования в области искусственного интеллекта;
- достижения философии и теории познания.

Понятие "объект" впервые было использовано при конструировании компьютеров с descriptor-based и capability-based архитектурами.

В этих работах - попытки отойти от архитектуры фон Неймана и преодолеть барьер между высоким уровнем программной абстракции и низким уровнем ЭВМ.

По мнению сторонников этих подходов, были созданы более качественные средства, обеспечивающие: лучшее выявление ошибок, большую эффективность реализации программ, сокращение набора инструкций, упрощение компиляции, снижение объема требуемой памяти.

Ряд компьютеров имеет ОО архитектуру: Burroughs 5000, Plessey 250, Cambridge CAP, SWARD, Intel 432, Caltech's COM, IBM System/38, Rational R1000, BiiN 40 и 60.

С ОО архитектурой связаны ОО ОС.

Дейкстра, работая над мультипрограммной системой THE, впервые ввел понятие машины с уровнями состояния в качестве средства построения системы.

Первые ОО ОС:

Plessey/System 250 (для мультипроцессора Plessey 250),

Hydra (для CMU C.mmp),

CALTSS (для CDC 6400),

CAP (для компьютера Cambridge CAP),

UCLA Secure UNIX (для PDP 11/45 и 11/70),

StarOS (для CMU Cm*),

Medusa (также для CMU Cm*) и iMAX (для Intel 432).

Следующее поколение ОС - Microsoft Cairo и Taligent Pink тоже объектно-ориентированные

Вклад в объектный подход внесен объектными и ОО ЯП.

Впервые понятия классов и объектов введены в языке Simula 67.

Система Flex и диалекты Smalltalk-72, -74, -76, -80, взяв за основу методы Simula, довели их до логического завершения, выполняя все действия на основе классов.

В 1970-х гг. - ЯП, реализующих идею абстракции данных: Alphard, CLU, Euclid, Gypsy, Mesa и Modula.

Методы, используемые в языках Simula и Smalltalk, были использованы в традиционных ЯП высокого уровня.

Внесение ОО подхода в С привело к возникновению языков C++ и Objective C.

На основе ЯП Pascal возникли Object Pascal, Eiffel и Ada.

Появились диалекты LISP: Flavors, LOOPS и CLOS (Common LISP Object System), с возможностями языков Simula и Smalltalk.

Дейкстра указал на необходимость построения систем в виде структурированных абстракций.

Парнас ввел идею скрытия информации.

В 70-х гг. Лисков и Жиль, Гуттаг и Шоу разработали механизмы абстрактных типов данных.

Хоар дополнил эти подходы теорией типов и подклассов.

На объектный подход оказали влияние технологии построения БД, благодаря "сущность-отношение" (ER, entity-relationship). В моделях ER (Чен), моделирование - в терминах сущностей, их атрибутов и взаимоотношений.

В понимание ОО абстракций - вклад разработчики способов представления данных в области ИИ:

в 1975 г. Мински - теория фреймов для представления реальных объектов в системах распознавания образов и естественных языков. Фреймы стали архитектурной основой в интеллектуальных системах.

Объектный подход известен издавна:

Греки - идея о том, что мир можно рассматривать в терминах как объектов, так и событий.

В XVII веке Декарт: люди имеют ОО взгляд на мир.

В XX веке эту тему развивала Рэнд в своей философии объективистской эпистемологии.

Мински предложил модель человеческого мышления, в которой разум человека рассматривается как общность различно мыслящих агентов. Он доказывает, что только совместное действие таких агентов приводит к осмысленному поведению человека.

Объектно-ориентированное программирование

(OOP , object-oriented programming) –

методология программирования, основанная на представлении программы в виде совокупности объектов, каждый из которых является экземпляром определенного класса, а классы образуют иерархию наследования.

3 части:

1) ООР использует в качестве базовых элементов *объекты,* а не алгоритмы (иерархия "быть частью»);
2) каждый объект является *экземпляром* какого-либо определенного *класса;*
3) классы организованы *иерархически* (иерархия «is a».

Программа будет ОО только при соблюдении всех трех требований.

Программирование, не основанное на иерархических отношениях, не относится к ООР, а называется *программированием на основе абстрактных типов данных.*

Не все ЯП объектно-ориентированные.

Страуструп: «*если термин ОО язык вообще что-либо означает, то он должен означать язык, имеющий средства хорошей поддержки ОО стиля программирования... Обеспечение такого стиля означает, что в языке удобно пользоваться этим стилем. Если написание программ в стиле ООР требует специальных усилий или оно невозможно совсем, то этот язык не отвечает требованиям ООР*».

Теоретически возможна имитация ОО программирования на обычных языках (Pascal, COBOL или ассемблер), но это затруднительно.

Карделли и Вегнер:

«*ЯП является ОО тогда и только тогда, когда выполняются условия:*

- *Поддерживаются объекты, то есть абстракции данных, имеющие интерфейс в виде именованных операций и собственные данные, с ограничением доступа к ним.*
- *Объекты относятся к соответствующим типам (классам).*
- *Типы (классы) могут наследовать атрибуты супертипов (суперклассов)*".

Поддержка наследования в таких ЯП означает возможность установления отношения "is-a" («это есть»), например: красная роза - это цветок, а цветок - это растение.

Языки, не имеющие таких механизмов, нельзя отнести к ОО. Такие ЯП – *объектные (*Карделли и Вегнер).

ОО ЯП: Smalltalk, Object Pascal, C++ и CLOS;

Ada - объектный язык.

Так как объекты и классы являются элементами обеих групп языков, желательно использовать и в тех, и в других методы ОО проектирования.

Объектно-ориентированное проектирование

(OOD, object-oriented design) –

методология проектирования, соединяющая процесс объектной декомпозиции и приемы представления логической и физической, статической и динамической моделей проектируемой системы

2 части: OOD

1) основывается на ОО декомпозиции;

2) использует многообразие приемов представления моделей, отражающих логическую (классы и объекты) и физическую (модули и процессы) структуру системы, а также ее статические и динамические аспекты.

Программирование подразумевает правильное и эффективное использование механизмов конкретных ЯП.

Проектирование основное внимание уделяет правильному и эффективному структурированию сложных систем.

Объектно-ориентированный анализ

(*OOA, object-oriented analysis*) –

методология, при которой требования к системе воспринимаются с т.зр. классов и объектов, выявленных в предметной области.

На объектную модель повлияла более ранняя модель ЖЦ ПО. Традиционная техника структурного анализа (Де Марк, Иордан, Гейн и Сарсон, с уточнениями для режимов реального времени - Варда и Меллора, Хотли и Пирбхая) основана на потоках данных в системе.

ОО анализ направлен на создание моделей реальной действительности на основе ОО мировоззрения.

Как соотносятся OOA, OOD и OOP?

На результатах OOA формируются модели, на которых основывается OOD;

OOD создает фундамент для окончательной реализации системы с использованием методологии OOP.

Составные части объектного подхода

Парадигмы программирования

Дженкинс и Глазго: *«в большинстве своем программисты используют в работе один ЯП и следуют одному стилю. Они программируют в парадигме, навязанной используемым ими языком. Часто они оставляют в стороне альтернативные подходы к цели, а следовательно, им трудно увидеть преимущества стиля, более соответствующего решаемой задаче».*

Бобров и Стефик определили понятие стиля программирования: *«Это способ построения программ, основанный на определенных принципах программирования, и выбор подходящего языка, который делает понятными программы, написанные в этом стиле».*

5 основных разновидностей стилей программирования с присущими им видами абстракций:

· процедурно-ориентированный	алгоритмы
· объектно-ориентированный	классы и объекты
· логико-ориентированный	цели, в терминах исчисления предикатов
· ориентированный на правила	правила «если-то»
· ориентированный на ограничения	инвариантные соотношения

Невозможно признать какой-либо стиль программирования лучшим во всех областях практического применения.

Для проектирования БЗ более пригоден стиль, ориентированный на правила, для вычислительных задач - процедурно-ориентированный.

ОО стиль наиболее приемлем для широкого круга приложений.

Каждый стиль программирования имеет свою концептуальную базу.

Для ОО стиля концептуальная база - это *объектная модель.*

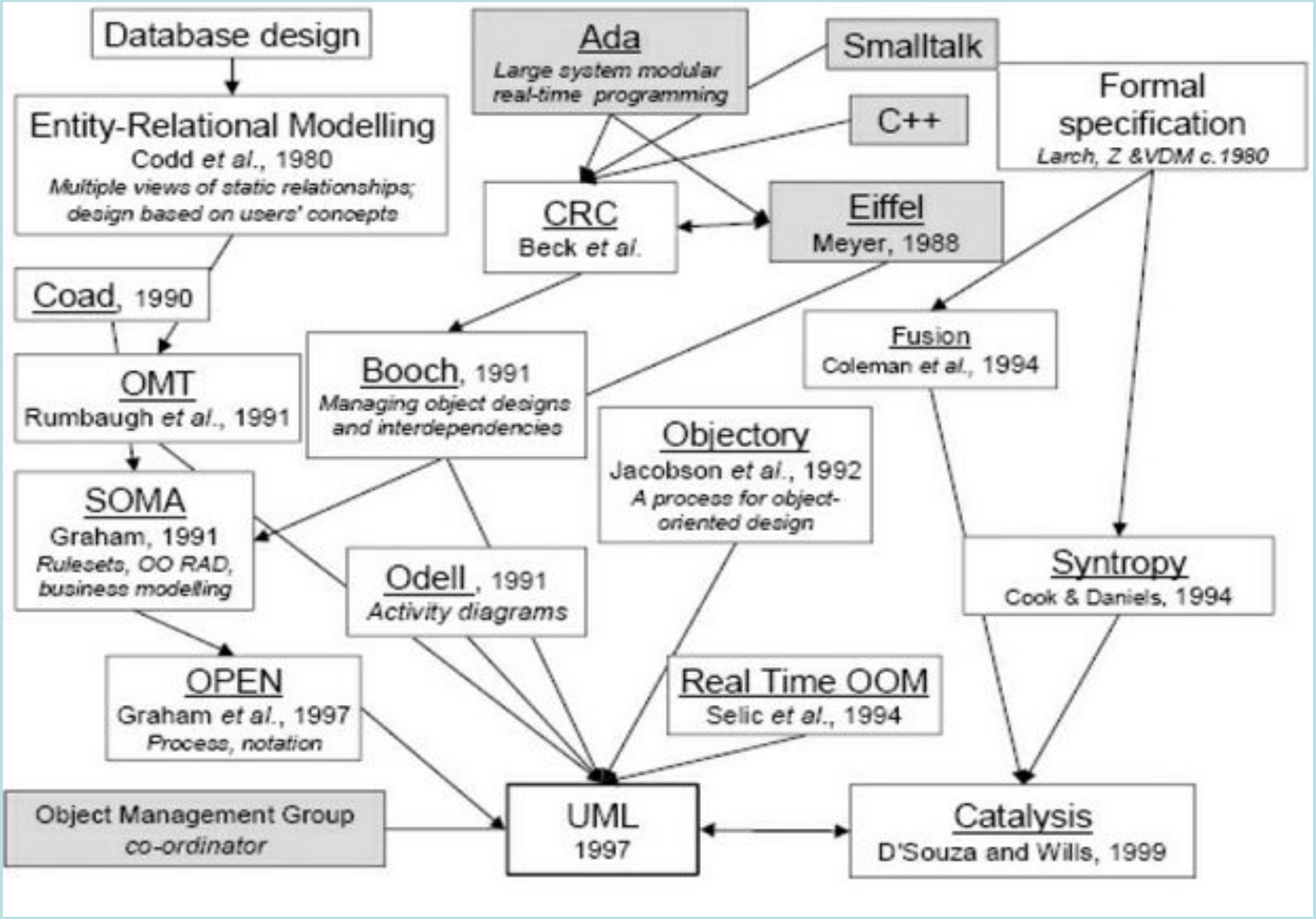
В 80-е годы - множество различных методологий моделирования.

Каждая имела свои достоинства и недостатки, свою нотацию.

То смутное время получило название "войны методов".

Проблема: разные люди использовали разные нотации. Один и тот же символ означал в разных нотациях абсолютно разные вещи.

UML - черты нотаций Гради Буча (Grady Booch), Джима Румбаха (Jim Rumbaugh), Айвара Якобсона (Ivar Jacobson) и других.



Часть методов, которые существовали в то время и повлияли на UML

Появление ООП требовало удобного инструмента для моделирования, единой нотации для описания сложных программных систем.

Три специалиста, три автора наиболее популярных методов решили объединить свои разработки.

В 1991-м каждый из них написал книгу, в которой изложил свой метод ООАП. Каждая методология была по-своему хороша, но имела и недостатки.

Метод **Буча -** хорош в проектировании, но слабоват в анализе.

OMT **Румбаха -** отличное средство анализа, но плох в проектировании.

Objectory **Якобсона -** хорош с точки зрения *user experience*, на который ни метод Буча, ни *OMT* не обращали особого внимания.

Основная идея Objectory - анализ должен начинаться с прецедентов, а не с диаграммы классов, которые должны быть производными от них.

В 1994 г. существовало 72 метода или частные методики. Многие из них «перекрывались» - использовали похожие идеи, нотации.

Чувствовалась острая потребность, "социальный заказ" - закончить "войну методов" и объединить в одном унифицированном средстве все лучшее, что было создано в области моделирования.

Сейчас The UML живет и развивается. Имеем десятки CASE-средств, поддерживающих UML.

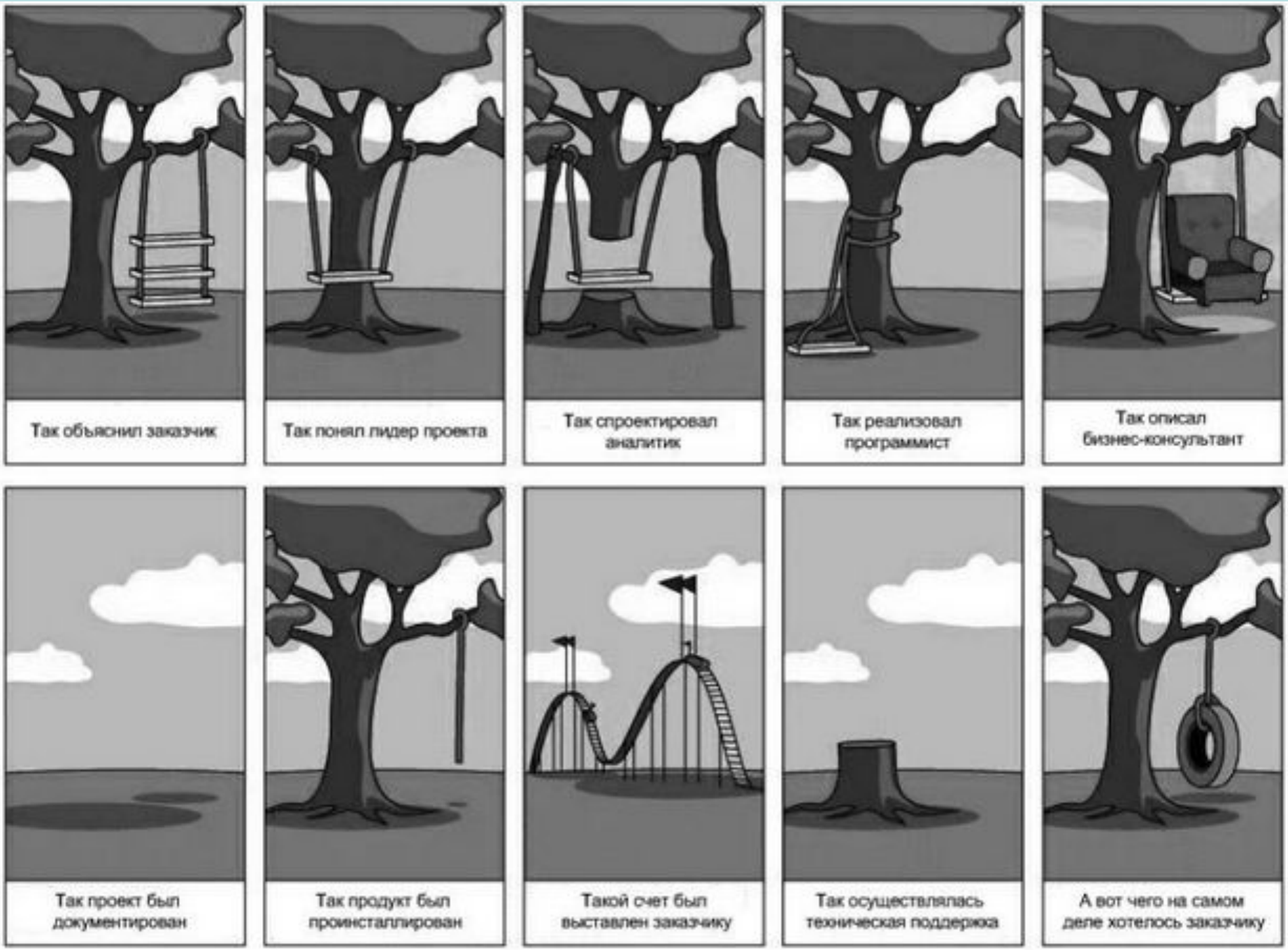
Rational не владеет UML, но продолжает работать над ним. UML принадлежит OMG, а сама Rational является одним из подразделений IBM и фигурирует во всех документах как IBM Rational.

UML получил множество пакетов расширений – ***профайлов***, позволяющих использовать его для моделирования систем из специфических предметных областей.

Известная картинка - типичный процесс создания продукта или "решения" (поскольку продукт решает проблему заказчика).

Здесь видны все проблемы программной инженерии, проблемы с коммуникацией и пониманием, вызванные отсутствием четкой спецификации создаваемого продукта.

UML - графический язык моделирования общего назначения (применяют для проектирования чего угодно - от простых качелей до сложного аппаратно-программного комплекса или космического корабля), предназначенный для **спецификации**, **визуализации**, **проектирования** и **документирования** всех артефактов, создаваемых в ходе разработки.


Так объяснил заказчик
Так понял лидер проекта
Так спроектировал аналитик
Так реализовал программист
Так описал бизнес-консультант
Так проект был документирован
Так продукт был проинсталлирован
Такой счет был выставлен заказчику
Так осуществлялась техническая поддержка
А вот чего на самом деле хотелось заказчику

UML в первую очередь - это спецификации.

Спецификация - подробное описание системы, полностью определяет ее цель и функциональные возможности.

Заказчик и разработчик имеют обычно разное понимание смысла этого понятия. Есть еще аналитики, менеджеры, бизнес-консультанты... Каждый из них называет спецификации по-своему: постановка задачи, требования пользователя, техническое задание, функциональная спецификация, архитектура системы...

Они - специалисты в разных предметных областях, говорят каждый на своем языке и часто не понимают друг друга.

Поэтому и возникает проблема, представленная на рисунке, ее может решить только наличие единого унифицированного средства создания спецификаций, простого и понятного для всех заинтересованных лиц.

Объектная модель:

4 главных элемента (*без любого из них модель не будет ОО*) :

- абстрагирование;
- инкапсуляция;
- модульность;
- иерархия.

3 дополнительных (*полезны, но не обязательны*):

- типизация;
- параллелизм;
- сохраняемость.

Абстрагирование - один из основных методов, для решения сложных задач.

Хоар: *«Абстрагирование проявляется в нахождении сходств между определенными объектами, ситуациями или процессами реального мира, и в принятии решений на основе этих сходств, отвлекаясь на время от имеющихся различий».*

Шоу: *«Упрощенное описание или изложение системы, при котором одни свойства и детали выделяются, а другие опускаются. Хорошей является такая абстракция, которая подчеркивает детали, существенные для рассмотрения и использования, и опускает те, которые на данный момент несущественны».*

Берзинс, Грей и Науман: *«Идея квалифицировалась как абстракция только, если она может быть изложена, понята и проанализирована независимо от механизма, который будет в дальнейшем принят для ее реализации».*

Абстракция выделяет существенные характеристики объекта, отличающие его от всех других видов объектов, и четко определяет его концептуальные границы с точки зрения наблюдателя

Абстрагирование концентрирует внимание на внешних особенностях О, позволяет отделить существенные особенности поведения от несущественных.

Такое разделение смысла и реализации - *барьер абстракции*, основывается на принципе минимизации связей: интерфейс О содержит только существенные аспекты поведения и ничего больше.

Дополнительный принцип - ***принцип наименьшего удивления:*** абстракция должна охватывать все поведение объекта, но не больше и не меньше, и не привносить сюрпризов или побочных эффектов, лежащих вне ее сферы применимости.

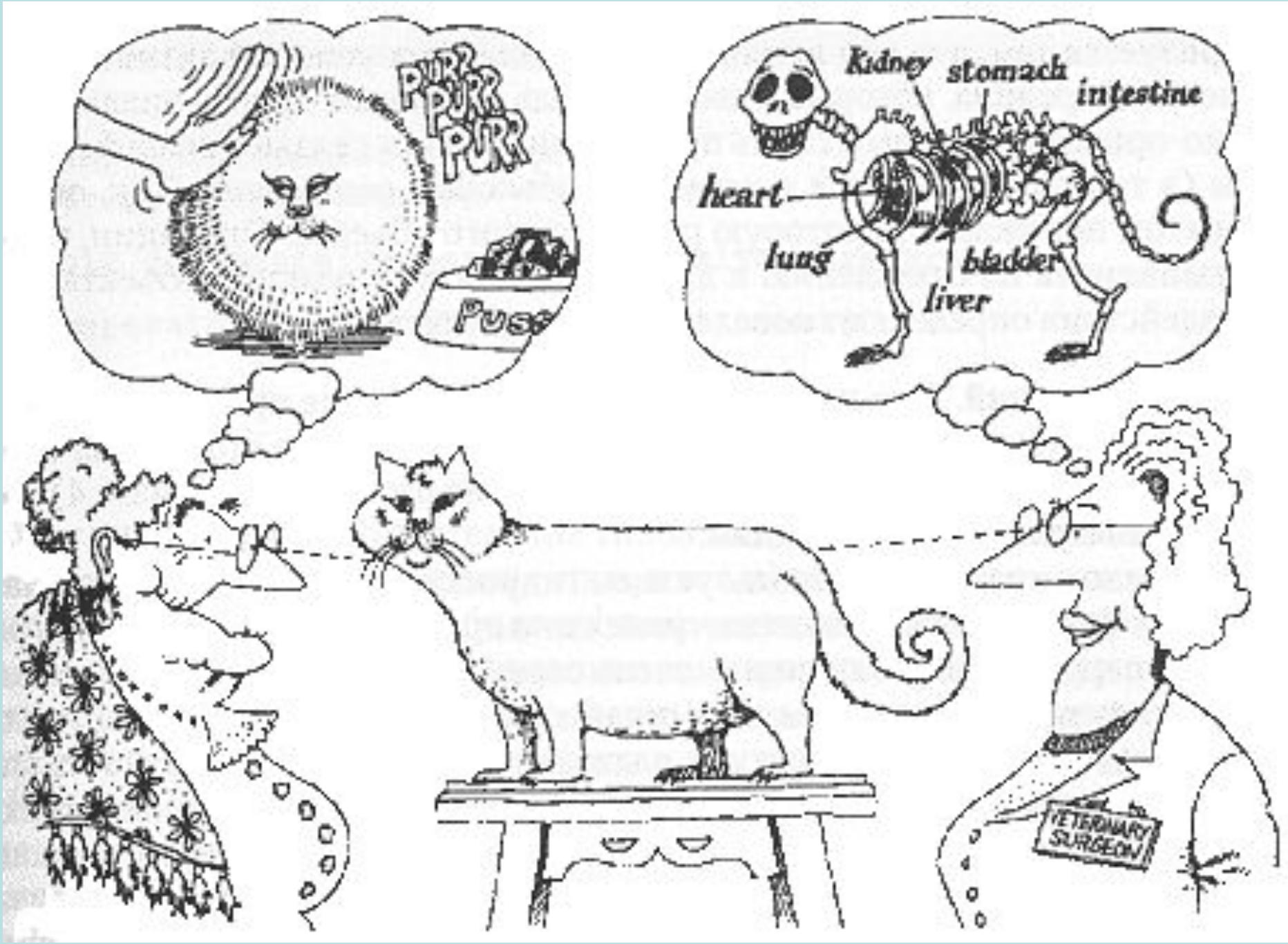
Абстрагирование сводится к формированию абстракций. Каждая абстракция фиксирует основные характеристики объекта, которые отличают его от других видов объектов. Абстракция концентрирует внимание на внешнем представлении объекта, позволяет отделить основное в поведении объекта от его реализации. Абстракцию удобно строить путем выделения обязанностей объекта.

Пример: физический объект - датчик скорости, устанавливаемый на борту летательного аппарата (ЛА). Создадим его абстракцию: сформулируем обязанности датчика:

- знать проекцию скорости ЛА в заданном направлении;
- показывать текущую скорость;
- подвергаться настройке.

Аппарат абстракции — удобный инструмент для борьбы со сложностью реальных систем. Создавая понятие в интересах задачи, мы отвлекаемся (абстрагируемся) от несущественных характеристик конкретных объектов, определяя только существенные характеристики.

Пример: в абстракции «часы» выделяем характеристику «показывать время», отвлекаясь от характеристик конкретных часов: форма, цвет, материал, цена, изготовитель.


PURR PURR PURR
Kidney
stomach
intestine
heart
lung
bladder
liver
VETERINARY SURGEON

Выбор правильного набора абстракций для заданной предметной области - главная задача OOD.

«*Существует целый спектр абстракций, начиная с объектов, которые почти точно соответствуют реалиям предметной области, и кончая объектами, не имеющими право на существование*» [Сейдвиц, Старк].

Абстракции от наиболее полезных к наименее полезным:

· Абстракция сущности - Объект представляет собой полезную модель некой сущности в предметной области

· Абстракция поведения - Объект состоит из обобщенного множества операций

· Абстракция виртуальной машины - Объект группирует операции, которые либо вместе используются более высоким уровнем управления, либо сами используют некоторый набор операций более низкого уровня

· Произвольная абстракция - Объект включает в себя набор операций, не имеющих друг с другом ничего общего

Клиент - объект, использующий ресурсы другого объекта (*сервера*).

Поведение О характеризуют услугами, которые он оказывает другим объектам, и операциями, которые он выполняет над другими объектами.

Такой подход концентрирует внимание на внешних проявлениях объекта и приводит к идее - *контрактной модели* программирования [Мейер]: *внешнее проявление О рассматривается с т.зр. его контракта с другими О, в соответствии с этим должно быть выполнено и его внутреннее устройство.*

Контракт фиксирует обязательства О-сервера перед О-клиентом.

Контракт определяет *ответственность* объекта - поведение, за которое он отвечает.

Каждая операция, предусмотренная контрактом, однозначно определяется ее формальными параметрами и типом возвращаемого значения.

Полный набор операций, которые клиент может осуществлять над другим объектом, вместе с правильным порядком, в котором эти операции вызываются - *протокол.*

Протокол отражает все возможные способы, которыми объект может действовать или подвергаться воздействию.

Центральная идея абстракции - понятие инварианта.

Инвариант - это логическое условие, значение которого (истина или ложь) должно сохраняться.

Для каждой операции объекта можно задать *предусловия* (инварианты предполагаемые операцией) и *постусловия* (инварианты, которым удовлетворяет операция).

Изменение инварианта нарушает контракт, связанный с абстракцией:

- если нарушено предусловие, то клиент не соблюдает свои обязательства и сервер не может выполнить свою задачу правильно;
- если нарушено постусловие, то свои обязательства нарушил сервер, и клиент не может более ему доверять.

В случае нарушения условия возбуждается **исключительная ситуация**

Некоторые ЯП имеют средства для работы с исключительными ситуациями:

объекты могут возбуждать исключения, чтобы запретить дальнейшую обработку и предупредить о проблеме другие объекты, которые могут принять на себя перехват исключения и справиться с проблемой.

Понятия *операция, метод и функция-член* происходят от различных традиций программирования (Ada, Smalltalk и C++ соответственно) и фактически обозначают одно и то.

Абстракции обладают статическими и динамическими свойствами. Например, файл как объект имеет объем памяти, имя и содержание - статические свойства. Конкретные значения каждого свойства динамичны и изменяются в процессе использования объекта.

Процедурный стиль программирования: действия, изменяющие динамические характеристики объектов, составляют суть программы. Любые события связаны с вызовом подпрограмм и с выполнением операторов.

Стиль программирования, ориентированный на правила: под влиянием определенных условий активизируются определенные правила, которые в свою очередь вызывают другие правила, и т.д.

Объектно-ориентированный стиль: связан с воздействием на объекты (в терминах Smalltalk с *передачей объектам сообщений*).

Операция над объектом порождает некоторую реакцию этого объекта. Операции, которые можно выполнить по отношению к данному объекту, и реакция объекта на внешние воздействия определяют **поведение** объекта.

Примеры абстракций.

В тепличном хозяйстве, использующем гидропонику, растения выращиваются на питательном растворе без песка, гравия или другой почвы.

Управление режимом работы парниковой установки зависит от вида выращиваемых культур и от стадии выращивания.

Нужно контролировать факторы: температуру, влажность, освещение, кислотность (показатель pH), концентрацию питательных веществ.

В больших хозяйствах используют автоматические системы, которые контролируют и регулируют эти факторы.

Цель автоматизации - при минимальном вмешательстве человека добиться соблюдения режима выращивания.

Ключевая абстракция - датчик.

Датчики: температуры воды и воздуха, влажности, pH, освещения и концентрации питательных веществ.

Датчик температуры - это объект, способный измерять температуру там, где он расположен.

Температура – это числовой параметр с ограниченным диапазоном значений и с определенной точностью, означающий число градусов по Фаренгейту, Цельсию или Кельвину.

Местоположение датчика - это место в теплице, температуру в котором нам необходимо знать.

Для датчика температуры существенно не столько само местоположение, сколько факт, что данный датчик расположен именно в данном месте, это отличает его от других датчиков.

Каковы обязанности датчика температуры? Датчик должен знать температуру в своем местонахождении и сообщать ее по запросу.

Какие действия может выполнять по отношению к датчику клиент? Клиент может калибровать датчик и получать от него значение текущей температуры.

Инкапсуляция

Абстракция объекта всегда предшествует его реализации. После того, как решение о реализации принято, оно должно трактоваться как секрет абстракции, скрытый от большинства клиентов.

Ингалс: "*Никакая часть сложной системы не должна зависеть от внутреннего устройства какой-либо другой части" .*

Абстракция "*помогает людям думать о том, что они делают", инкапсуляция "позволяет легко перестраивать программы".*

Инкапсуляция и абстракция - взаимодополняющие понятия:

абстракция выделяет внешнее поведение объекта,

инкапсуляция содержит и скрывает реализацию, которая обеспечивает это поведение.

Инкапсуляция достигается с помощью информационной закрытости. Обычно скрываются структура объектов и реализация их методов.

Инкапсуляция является процессом разделения элементов абстракции на секции с различной видимостью. Инкапсуляция служит для отделения интерфейса абстракции от ее реализации.

Пример: физический объект регулятор скорости. Обязанности регулятора:

включаться; выключаться; увеличивать скорость;

уменьшать скорость; отображать свое состояние.

Инкапсуляция определяет четкие границы между абстракциями:

1) структура растения: чтобы понять на верхнем уровне действие фотосинтеза, допустимо игнорировать подробности: функции корней растения, химию клеточных стенок.
2) при проектировании БД пишут программы, которые не зависят от физического представления данных; сосредотачиваются на схеме, отражающей логическое строение данных.

В примерах объекты защищены от деталей реализации объектов более низкого уровня.

Дисков: *"абстракция работает только вместе с инкапсуляцией".*

Это означает наличие двух частей в классе: интерфейса и реализации.

Интерфейс отражает внешнее поведение О, описывая абстракцию поведения всех объектов данного класса.

Внутренняя *реализация* описывает представление этой абстракции и механизмы достижения желаемого поведения объекта.

Принцип разделения интерфейса и реализации:

в интерфейсной части собрано все, что касается взаимодействия данного О с другими О;

реализация скрывает от других О все детали, не имеющие отношения к процессу взаимодействия объектов.

Бритон и Парнас назвали такие детали "тайнами абстракции".

Инкапсуляция скрывает детали реализации объекта.

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.

Модульность

Майерс: *"Разделение программы на модули до некоторой степени позволяет уменьшить ее сложность... Однако гораздо важнее тот факт, что внутри модульной программы создаются множества хорошо определенных и документированных интерфейсов. Эти интерфейсы неоценимы для исчерпывающего понимания программы в целом".*

В некоторых ЯП (Smalltalk) модулей нет, классы составляют единственную физическую основу декомпозиции. В других ЯП (Object Pascal, C++, Ada, CLOS) модуль - это самостоятельная языковая конструкция.

В этих языках классы и объекты составляют логическую структуру системы, они помещаются в *модули,* образующие физическую структуру системы.

Это свойство становится особенно полезным, когда система состоит из многих сотен классов.

Лисков: "*модульность - это разделение программы на фрагменты, которые компилируются по отдельности, но могут устанавливать связи с другими модулями".*

Парнас: *"Связи между модулями - это их представления друг о друге".*

В ЯП, поддерживающих принцип модульности как самостоятельную концепцию, интерфейс модуля отделен от его реализации.

Модульность и инкапсуляция связаны.

В разных ЯП модульность поддерживается по-разному.

В C++ модули - раздельно компилируемые файлы. Помещение интерфейсной части модулей в отдельные файлы .h. Реализация модуля - в файлах с расширением .cpp. Связь между файлами объявляется директивой #include. Такой подход строится на соглашении и не является строгим требованием самого языка.

В ЯП Object Pascal принцип модульности формализован строже. Определен особый синтаксис для интерфейсной части и реализации модуля (unit).

Язык Ada: модуль (package) также имеет две части - спецификацию и тело. Допускается раздельное определение связей с модулями для спецификации и тела пакета.

Правильное разделение программы на модули - сложная задача как выбор правильного набора абстракций.

Зельковиц: *"поскольку в начале работы над проектом решения могут быть неясными, декомпозиция на модули может вызвать затруднения. Для хорошо известных приложений (например, создание компиляторов) этот процесс можно стандартизовать, но для новых задач (военные системы или управление космическими аппаратами) задача может быть очень трудной".*

Модули выполняют роль физических контейнеров, в которые помещаются определения классов и объектов при логическом проектировании системы.

Такая же ситуация - у проектировщиков бортовых компьютеров. Логика электронного оборудования может быть построена на основе элементарных схем типа НЕ, И-НЕ, ИЛИ-НЕ, но можно объединить такие схемы в стандартные интегральные схемы (модули), например, серий 7400, 7402 или 7404.

Для небольших задач допустимо описание всех классов и объектов в одном модуле.

Для большинства программ (кроме тривиальных) лучше сгруппировать в отдельный модуль логически связанные классы и объекты, оставив открытыми те элементы, которые совершенно необходимо видеть другим модулям.

Такой способ разбиения на модули хорош, но его можно довести до абсурда. При внесении в проект изменений потребуется модифицировать и перекомпилировать сотни модулей.

Скрытие информации имеет и обратную сторону. Деление программы на модули бессистемным образом иногда гораздо хуже, чем отсутствие модульности вообще.

В структурном проектировании модульность - это искусство раскладывать п/п по кучкам так, чтобы в одну кучку попадали п/п, использующие друг друга или изменяемые вместе.

В ОО необходимо физически разделить классы и объекты, составляющие логическую структуру проекта.

Парнас: *"конечной целью декомпозиции программы на модули является снижение затрат на программирование за счет независимой разработки и тестирования. Структура модуля должна быть достаточно простой для восприятия; реализация каждого модуля не должна зависеть от реализации других модулей; должны быть приняты меры для облегчения процесса внесения изменений там, где они наиболее вероятны"*

На практике перекомпиляция тела модуля не является трудоемкой операцией: заново компилируется только данный модуль, и программа перекомпонуется.

Перекомпиляция *интерфейсной* части модуля более трудоемка.

В строго типизированных ЯП надо перекомпилировать интерфейс и тело самого измененного модуля, затем все модули, связанные с данным, модули, связанные с ними, и т.д. по цепочке.

Интерфейсная часть модулей должна быть возможно более узкой (в пределах обеспечения необходимых связей). Необходимо скрыть все, что возможно, в реализации модуля. Постепенный перенос описаний из реализации в интерфейсную часть менее опасен, чем "вычищение" избыточного интерфейсного кода.

Программист должен находить баланс между двумя противоположными тенденциями: стремлением скрыть информацию и необходимостью обеспечения видимости тех или иных абстракций в нескольких модулях.

Парнас, Клеменс и Вейс правило: *"Особенности системы, подверженные изменениям, следует скрывать в отдельных модулях; в качестве межмодульных можно использовать только те элементы, вероятность изменения которых мала. Все структуры данных должны быть обособлены в модуле; доступ к ним будет возможен для всех процедур этого модуля и закрыт для всех других. Доступ к данным из модуля должен осуществляться только через процедуры данного модуля" .*

Следует стремиться построить модули так, чтобы объединить логически связанные абстракции и минимизировать взаимные связи между модулями.

Модульность *- это свойство системы, которая была разложена на внутренне связные, но слабо связанные между собой модули.*

Принципы абстрагирования, инкапсуляции и модульности - взаимодополняющие.

Объект логически определяет границы определенной абстракции, а инкапсуляция и модульность делают их физически незыблемыми.

При разделении системы на модули 2 правила:

1) поскольку модули служат в качестве элементарных и неделимых блоков программы, которые могут использоваться в системе повторно, распределение классов и объектов по модулям должно учитывать это.
2) многие компиляторы создают отдельный сегмент кода для каждого модуля. Поэтому могут появиться ограничения на размер модуля. Динамика вызовов подпрограмм и расположение описаний внутри модулей может влиять на локальность ссылок и на управление страницами виртуальной памяти. При плохом разбиении процедур по модулям учащаются взаимные вызовы между сегментами, что приводит к потере эффективности кэш-памяти и частой смене страниц.

Иерархическая организация

3 механизма для борьбы со сложностью:

- Абстракция (упрощает представление физического объекта). Но число абстракций в системе намного превышает наши умственные возможности.

- Инкапсуляция позволяет устранить это препятствие, убрав из поля зрения внутреннее содержание абстракций (закрывает детали внутреннего представления абстракций).

- Модульность также упрощает задачу, объединяя логически связанные абстракции в группы.

Дополнение к ним - ***иерархическая организация*** - формирование из абстракций иерархической структуры. Упрощаются понимание проблем заказчика и их реализация - сложная система становится обозримой человеком.

Иерархия *- это упорядочение абстракций, расположение их по уровням.*

Основные виды иерархических структур:

- структура из классов («is a»-иерархия);
- структура из объектов («part of»-иерархия).

«is a» - строится с помощью наследования. Наследование определяет отношение между классами, где класс разделяет структуру или поведение, определенные в одном другом (единичное наследование) или в нескольких других (множественное наследование) классах.

«part of» - иерархическая структура базируется на отношении агрегации.

Агрегация не является понятием, уникальным для ОО систем: любой ЯП, разрешающий структуры типа «запись», поддерживает агрегацию.

Агрегация полезна в сочетании с наследованием:

- агрегация обеспечивает физическую группировку логически связанной структуры;
- наследование позволяет легко и многократно использовать эти общие группы в других абстракциях.

Сравним элементы иерархий наследования и агрегации с точки зрения уровня сложности.

При наследовании нижний элемент иерархии (подкласс) имеет больший уровень сложности (большие возможности),

при агрегации - наоборот (агрегат обладает большими возможностями, чем его элементы).

Принцип наследования делает проект менее громоздким и более выразительным.

Кокс: *"В отсутствие наследования каждый класс становится самостоятельным блоком и должен разрабатываться "с нуля". Классы лишаются общности, поскольку каждый программист реализует их по-своему. Стройность системы достигается тогда только за счет дисциплинированности программистов. Наследование позволяет вводить в обращение новые программы, как мы обучаем новичков новым понятиям - сравнивая новое с чем-то уже известным".*

Принципы абстрагирования, инкапсуляции и иерархии находятся между собой в конфликте.

Данфорт и Томлинсон: *"Абстрагирование данных создает непрозрачный барьер, скрывающий состояние и функции объекта; принцип наследования требует открыть доступ и к состоянию, и к функциям объекта для производных объектов".*

Для любого класса существуют два вида клиентов: объекты, которые манипулируют с экземплярами данного класса, и подклассы-наследники.

Лисков: «*существуют три способа нарушения инкапсуляции через наследование: "подкласс может получить доступ к переменным экземпляра своего суперкласса, вызвать закрытую функцию, обратиться напрямую к суперклассу своего суперкласса»*.

ЯП по-разному находят компромисс между наследованием и инкапсуляцией; наиболее гибкий - C++. Интерфейс класса разделен на 3 части: закрытую (private), защищенную (protected) и открытую (public).

Типизация

Понятие *типа* взято из теории абстрактных типов данных.

Дойч: *«тип - точная характеристика свойств, включая структуру и поведение, относящуюся к некоторой совокупности объектов".*

Термины *тип* и *класс* взаимозаменяемы

(Тип и класс не вполне одно и то же; в некоторых языках их различают)

Типизация - это способ защититься от использования объектов одного класса вместо другого, или по крайней мере управлять таким использованием.

Вегнер: «*такой способ контроля существенен для программирования "в большом"* ».

В понятии типизации центральное место - идея согласования типов занимает.

Пример: физические единицы измерения. Деля расстояние на время, мы ожидаем получить скорость, а не вес. В умножении температуры на силу смысла нет, а в умножении расстояния на силу - есть.

Это примеры сильной типизации, когда прикладная область накладывает правила и ограничения на использование и сочетание абстракций.

Примеры сильной и слабой типизации

Конкретный ЯП может иметь сильный или слабый механизм типизации или не иметь никакого, оставаясь ОО.

В Eiffel соблюдение правил использования типов строго контролируется - операция не может быть применена к объекту, если она не зарегистрирована в его классе или суперклассе.

В Smalltalk типов нет: во время исполнения любое сообщение можно послать любому объекту, и если класс объекта (или его надкласс) не понимает сообщение, то генерируется сообщение об ошибке.

C++ тяготеет к сильной типизации, но в этом языке правила типизации можно игнорировать или подавить полностью.

Статическое и динамическое связывание.

Сильная (строгая) и статическая типизация - разные вещи.

Строгая типизация следит за соответствием типов, а статическая (*статическое, раннее связывание)* определяет время, когда имена связываются с типами.

Статическая связь - типы всех переменных и выражений известны во время компиляции; *динамическое связывание* (*позднее)* - типы неизвестны до момента выполнения программы.

Типизация и связывание - независимы, в ЯП может быть:

типизация - сильная, связывание - статическое (Ada),

типизация - сильная, связывание - динамическое (C++, Object Pascal),

или и типов нет, и связывание динамическое (Smalltalk).

Язык CLOS - промежуточное положение между C++ и Smalltalk

Параллелизм

ООР основано на абстракции, инкапсуляции и наследовании; параллелизм главное внимание уделяет абстрагированию и синхронизации процессов.

Параллелизм - это свойство, отличающее активные объекты от пассивных.

Параллелизм позволяет различным объектам действовать одновременно.

Сохраняемость

Любой программный объект существует в памяти и живет во времени.

Аткинсон: «есть непрерывное множество продолжительности существования объектов: существуют О, которые присутствуют лишь во время вычисления выражения, но есть такие, как БД, которые существуют независимо от программы».

Объектная модель отличается от моделей, связаных с методами структурного анализа, проектирования и программирования.

Объектная модель не требует отказа от ранее найденных и испытанных методов и приемов.

Она вносит новые элементы, которые добавляются к предшествующему опыту.

Объектный подход обеспечивает ряд удобств, которые другими моделями не предусматривались.

Объектный подход позволяет создавать системы, удовлетворяющие признакам хорошо структурированных сложных систем.

Преимущества объектной модели:

1) Позволяет использовать возможности ОО ЯП.
2) Повышается уровень унификации разработки, повторное использование программ и проектов. Уменьшение объема кода программ, удешевление проекта, выигрыш во времени.
3) Построение систем на основе стабильных промежуточных описаний, это упрощает процесс внесения изменений.
4) Уменьшается риск разработки сложных систем. Объектный подход состоит из ряда хорошо продуманных этапов проектирования.
5) Ориентирована на человеческое восприятие мира, *"многие люди, не имеющие понятия о том, как работает компьютер, находят вполне естественным ОО подход к системам"* .

Выводы:

- Развитие программной индустрии привело к созданию методов OOA, OOD, OOP, служщие для программирования "в большом".

- В программировании существует несколько парадигм, ориентированных на процедуры, объекты, логику, правила и ограничения.

- Абстракция определяет существенные характеристики О, которые отличают его от других видов О, абстракция четко очерчивает концептуальную границу О с т.зр. наблюдателя.

- Инкапсуляция - процесс разделения устройства и поведения О; служит, чтобы изолировать контрактные обязательства абстракции от их реализации.

- Модульность - это состояние системы, разложенной на внутренне связные и слабо связанные между собой модули.
- Иерархия - это ранжирование или упорядочение абстракций.
- Типизация - это способ защититься от использования объектов одного класса вместо другого, или по крайней мере способ управлять такой подменой.
- Параллелизм - это свойство, отличающее активные объекты от пассивных.
- Сохраняемость - способность объекта существовать во времени и (или) в пространстве.