Оглавление

1.Технология структурного программирования. Преимущества и недостатки структурного программирования.

Структурное программирование - разбивка по действиям от сложного к простому. В основе структурного программирования лежит алгоритмическая декомпозиция – разбиение задачи на подзадачи по ДЕЙСТВИЮ, отвечая на вопросы «что нужно делать».

Три основные части технологии:

- Нисходящая разработка;
- Сквозной структурный контроль;
- Использование базовых логических структур.

Этапы (нисходящая разработка):

- Анализ (ТЗ, возможность формализации;
- Проектирование (разработка алгоритмов);
- Кодирование;
- Тестирование.
- Сопровождение
- Модификация

  От проектирования до модификации – нисходящий по дход в структурном программировании.

Преимущества:

- Легко распределять работу между программистами;

- Естественные контрольные точки;

- Легко выявлять ошибки;

- Легко поддается тестированию (комплексное тестирование);

- Раннее начало процесса кодирования;

- Снижается вероятность допустить логическую ошибку;

- Возможен контакт с заказчиком на ранних стадиях, управление сроками;

- Упрощенное чтение кода;

Недостатки:

- Отсутствие гибкости системы. После некоторого количества модификаций происходит смещение уровней абстракции, нарушается структура, что приводит к потере надежности (сопровождение затруднено и много стоит);

- Сложно изменить формы данных и структур;

- Сложно сопровождать программный продукт;

2. Структурное программирование: нисходящая разработка, сквозной структурный контроль. Использование базовых логических структур.

Три основные части технологии:

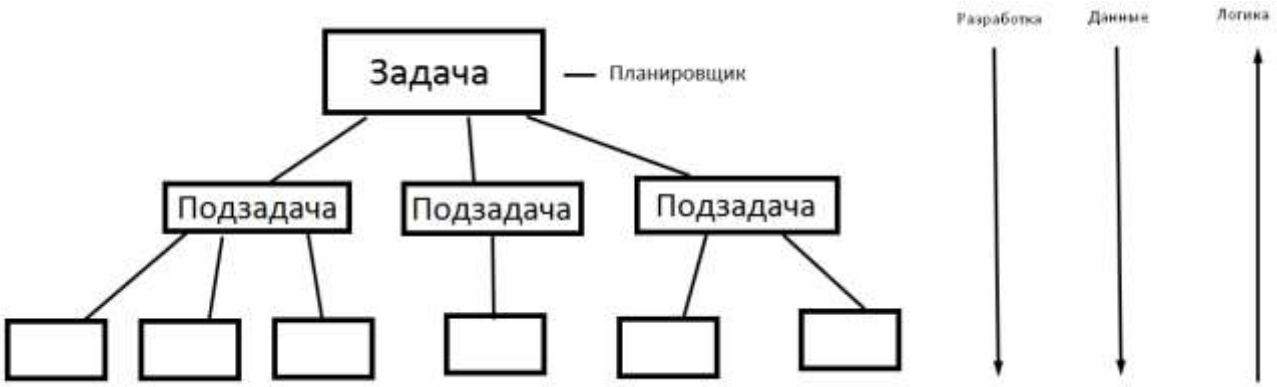
- Нисходящая разработка;
- Сквозной структурный контроль;
- Использование базовых логических структур.

Нисходящая разработка:

Используется на этапах проектирования, кодирования и тестирования.

Разбивка на подзадачи, которые детерминизированы.

Разработка: сверху-вниз, логика: снизу-вверх, данные: сверху-вниз



Принципы:

- Тесты составляются до написания кода;

- Подзадачи - модули

- Не должно быть участков кода, выполняющих одинаковые действия (отсутствие дублирования);

- Результат только вверх *// ???*

- Обработка всех ошибок;

- Без функций с сюрпризами;

- Глубина вложенности ограничивается цифрой 3;

- Когда разбиваем программу на подзадачи, выделяем не больше 7 подзадач;

- Комментарии - перед функциями;

- Логика – на самом верху;

- Данные передаются и возвращаются явно;

- Уровни абстракции: файл -> функция -> блок.

Три подхода программирования модулей:

- Иерархический (по уровню абстракции);

- Операционный (в порядке вызова модулей);

- Смешанный.

Использование базовых логических структур:

Алгоритм любой сложности можно реализовать с использованием трёх базовых логических структур: условие (развилка), следование, цикл (повторение). Ввела IBM *(и)* Майер. GO TO НЕЛЬЗЯ!

Развилка:

if (между двумя), switch (множественный выбор)

Виды циклов:

- Цикл с постусловием do … while();
- Цикл с предусловием while();
- Цикл со счетчиком for ();
- Безусловный цикл loop(цикл с выходом из тела цикла);

// А теперь я про циклы

Повторение:

a) Цикл «до» («until») – сделали – проверяем
b) Цикл «пока» («while»)
c) Цикл «пока» с переменной («for», «для»)
d) Безусловный цикл («loop»)

Замечание: выход из цикла должен быть 1!

Сквозной структурный контроль:

Размер рабочий группы желательно не должен превышать 7; однако группой необходимо руководить, уровень руководителя должен быть выше, чем у подчиненных. Однако руководитель не вовлечен в написание кода. Контроль осуществляют сами программисты. Производится формализация задачи и устраивается «контрольная сессия» с коллегами. Задачи контрольных сессий: выявление недостатков (особенно на ранних стадиях), мозговой штурм.

3. Технология ООП: преимущества и недостатки.

Идеи ООП (Хоар, 1966, “Совместное использование кода”):

1. Инкапсуляция (объединение данных и действий над ними, или как по лекции Маслова, для каждого типа данных – свои функции-действия);
2. Наследование (модификация развития программы за счет надстроек; вместо изменения написаного кода – делаем над ним надстройки);
3. Организация взаимодействия между объектами; перенесение взаимодействия объектов из физического мира в программирование.

Два вида взаимодействия:

- Акцессорное – вступление в контакт, получение информации от объектов (синхронное взаимодействие);
- Событийное взаимодействие – взаимодействие, связанное с изменением состояния объекта (асинхронное взаимодействие);

Преимущества ООП:

- Возможность легкой модификации (при грамотном анализе и проектировании);

- Возможность отката при наличии версий;

- Более легкая расширяемость;

Недостатки ООП:

- Требуется другая квалификация;

- Резко увеличивается время на анализ и проектирование систем;

- Увеличение времени выполнения;

- Размер кода увеличивается;

- Неэффективно с точки зрения памяти (мертвый код - тот, который не используется);

- Сложность распределения работ на начальном этапе;

- Себестоимость больше.

4. Этапы разработки ПО с использованием объектно-ориентированного подхода.

Этапы:

- Анализ (Построение модели будущей программы);
- Проектирование (Перенос документов анализа в документы написания кода)
- Эволюция (Этап объединяет кодирование и тестирование. Позволяет при этом вернуться к этапу анализа или проектирования. Изменение должно сводиться только к добавлению класса или изменению его реализации);
- Модификация (после получения готового продукта).

Не путать эволюцию и модификацию: модификация – программный продукт готов, эволюция – добавление нового функционала.

// Дальше по Маслову

Преимущества эволюции:

a) Обратная связь с пользователем
b) Различные версии структур системы (плавный переход от старой системы к новой)
c) Меньше вероятности отмены проекта

Изменения в процессе эволюции (по возрастанию сложности)

1) Добавление класса
2) Изменение реализации класса
3) Изменение представления класса
4) Реорганизация структуры класса
5) Изменение интерфейса

5. Понятия ООП: инкапсуляция, наследования, полиморфизм. Объекты, классы, домены, отношения между ними.

Основные принципы ООП:

- Инкапсуляция - свойство системы, позволяющее объединить данные и методы, работающие с ними, в классе;

- Наследование – позволяет создать новый класс на основе уже существующего, частично или полностью заимствуя его функциональность;

- Полиморфизм – использование объектов с одинаковым интерфейсом без информации о типе и внутренней структуре объекта.

Объект – конкретная реализация абстрактного типа, обладающая следующими характеристиками: состояние, поведение и индивидуальность.

Состояние – один из возможных вариантов формы объекта.

Поведение – описание объекта в терминах изменения его состояния во время жизни или под воздействием других объектов (на его состояние могут влиять внутренние данные).

Индивидуальность – сущность объекта, отличающая его от других объектов.

Модель состояний Мура состоит:

1. Из множества состояний: каждое состояние представляет стадию в жизненном цикле объекта.

2. Из множества событий: каждое событие означает инцидент, указывающий на эволюционирование.

3. Из правил перехода: правило определяет, какое новое состояние достигается объектом под воздействием события.

4.Из действий: операции, которые должны быть выполнены, чтобы объект перешел в какое-то состояние.

Категории объектов:

1.Реальные объекты – абстракция фактического существующего объекта реального мира.

2.Роли – абстракции цели или назначения человека, части оборудования или организации.

3.Инциденты – абстракция чего-то происшедшего или случившегося (наводнение, скачёк напряжения, выборы).

4.Взаимодействия – объекты получаемые из отношений между другими объектами (перекресток, договор, взятка).

5.Спецификации – используется для представления правил, критериев качества, стандартов (правила дорожного движения, распорядок дня).

Отношения между объектами:

Отношения использования (*старшинства*) - каждый объект включается в отношения. Может играть 3 роли:
1. Активный объект– объект может воздействовать на другие объекты, но сам не поддается воздействию (воздействующий).
2. Пассивный объект – объект может только подвергаться управлению, но не выступает в роли воздействующего (исполнитель).
3. Посредники – такой объект может выступать в роли воздействующего, так и в роли исполнителя (создаются для помощи воздействующим). Чем больше посредников тем легче модифицировать программу.

Отношения *включения* – один объект включает другие объекты.

Класс – такая абстракция множества предметов реального мира, что все предметы этого множества(объекты) имеют одни и те же характеристики, все экземпляры подчинены и согласованы с одним и тем же поведением.

Отношения между классами:
1.Наследование – на основе одного класса, мы строим новый класс, путем добавления новых характеристик и методов.

2.Использование – один класс вызывает методы другого класса.

3 Представление (наполнение) – это когда один класс содержит другие классы.

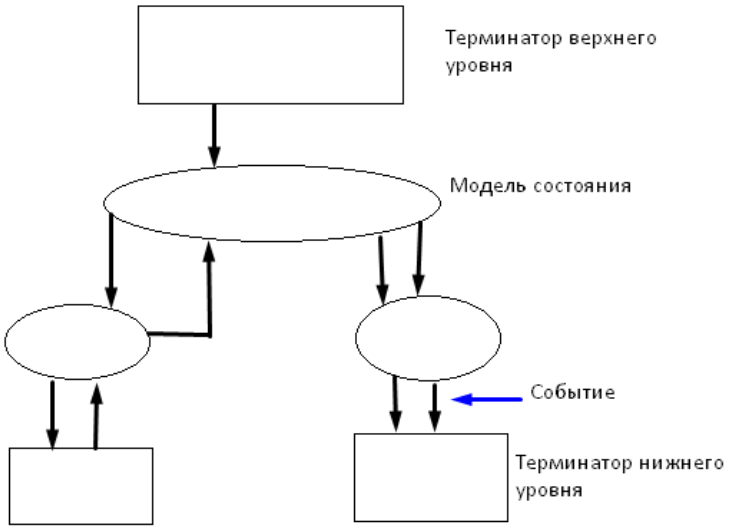
4. Метакласс – класс, существующий для создания других классов.

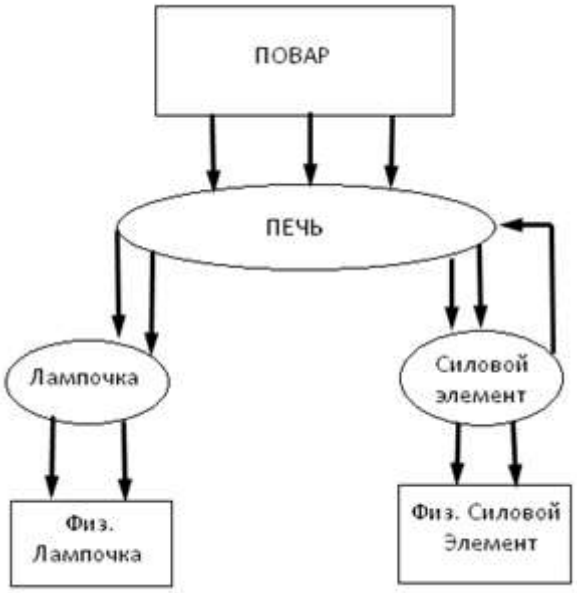
Домен – отдельный, реальный, гипотетически и абстрактный мир, населенный отчетливым набором объектов, которые ведут себя в соответствии с предусмотренным доменом правилами. Каждый домен образует отдельное и связное единое целое.

6. Объектно-ориентированный анализ. Динамика систем, схемы взаимодействия, каналы управления, имитирование.

Модель взаимодействия объектов (МВО) – графическое представление взаимодействия между моделями состояний и внешними сущностями. (Строится для каждой подсистемы или домена).

Модель состояния – овал. Внешние сущности (терминаторы) – прямоугольник. События – стрелки.





Типы событий:

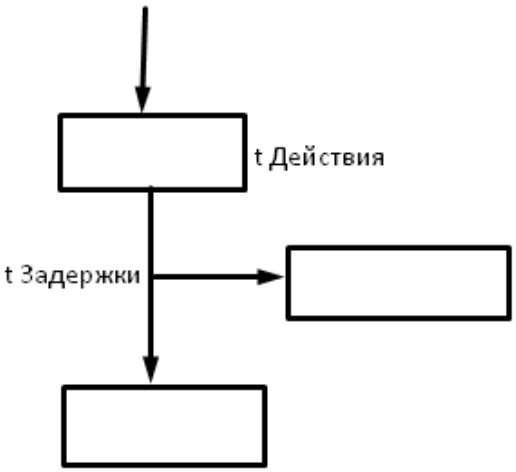
1. Внешние события (приходят от терминатора):
   a. Незапрашиваемые события (не являются результатом действий подсистемы)
   b. Запрашиваемые
2. Внутренние события (порождаются какой-либо моделью состояний подсистемы)

Канал управления – последовательность событий и действий, происходящих в ответ на поступление некоторого незапрашиваемого события. Если возникло событие к терминатору, влекущее за собой новые события от терминатора, то они тоже включаются в канал управления.

Процесс имитирования: задается некоторое начальное состояние. Генерируем некоторое внешнее событие и смотрим за изменением состояния всех объектов в системе.

Время имитирования:

1. Время выполнения действия;
2. Время задержки – время, в течении которого объект должен находится в определенном состоянии (резкий переход из состояния в состояние невозможен).



Этапы имитирования:

1. Установка начального состояния;
2. Прием незапрашиваемого события и выполнение канала управления;
3. Оценка конечного результата.

7. Рабочие продукты объектно-ориентированного анализа и проектирования.

Анализ – построение модели системы.
Действия при анализе и результаты (рабочие продукты):

1. Разбиваем задачу на домены:
   a. Схема доменов;
   b. Проектная матрица.
2. Разбиваем домены на подсистемы:
   a. Модель взаимодействия подсистемы;
   b. Модель связей подсистемы;
   c. Модель доступа к подсистемам.
3. Для каждой подсистемы получаем:
   a. Информационная модель (получаем описание классов и их атрибутов, а также описание их связей);
   b. Модель взаимодействия объектов (получаем список событий в подсистеме);
   c. Модель доступа к объектам (получаем таблицу процессов состояний).
4. Для каждого объекта получаем *модель переходов состояний*;
5. Для каждого состояния каждой модели состояния строим *диаграмму потоков данных действий;*
6. Для каждого процесса получаем *описание процесса.*

Не путать домен (мир) с сервисом (функционал)!

На основе полученных в ООА документов мы приходим к проектированию.
Четыре основных рабочих продукта:

1. Диаграмма класса (проектируется вокруг объекта класса и класса).
2. Схема структуры класса (для внутренней структуры класса).
3. Диаграмма зависимостей – схема использования.
4. Диаграмма наследований – схема наследования классов.

8. ООА. Концепция информационного моделирования. Понятие классов, атрибутов и связей. Формализация связей.

Концепция:

- Выделение физических объектов
- Короткое описание классов (чтобы установить, является ли объект экземпляром класса)
- Выделение характеристик объектов и идентификаторов (множество из одного или нескольких атрибутов, однозначно определяющих экземпляр класса)
- Графическое обозначение класса на информационной модели:

<номер><имя><ключевой литерал (краткое имя)>
Атрибуты *<> - идентифицирующие (выделяем привилегированный идентификатор) V V } - неидентифицирующие V

- Строим таблицу атрибутов: для каждого объекта должен быть определен однозначно атрибут

(<имя>(<иден.>,<список атрибутов>) - текстовое описание)

Типы атрибутов:

- Описательные – факты, внутренне присущие каждому объекту (если меняется, то только какой-то аспект, сам объект остается прежним)
- Указывающие (идентифицирующие) – использующиеся как идентификатор или часть идентификатора (если меняется, то объект сменил имя, но сам остался)
- Вспомогательные – выделяемые из состояния объекта или его отношений, связей с другими объектами (при изменении соответственно меняются связи или состояние объекта)

Если описательный, то описание – информационная строка, которая показывает реальную характеристику, как определяется характеристика и почему она уместна для данного объекта.

Если указывающий, то описание – форма указания, кто назначает атрибуты использующиеся в идентификаторе.

Если вспомогательный, то описание – какое отношение или состояние сохраняются.

Правила:

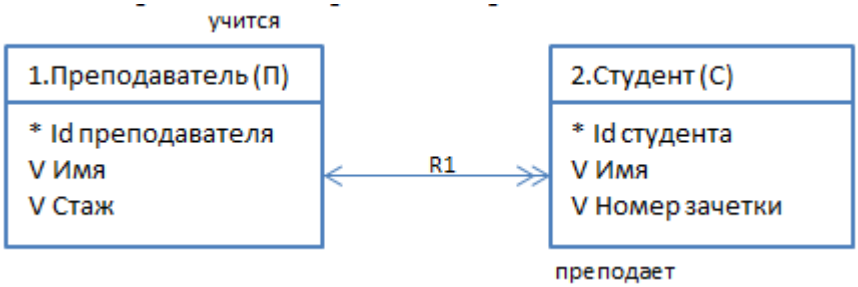
1) Каждый объект выделенного класса имеет в каждый момент времени одно единственное значение.
2) Атрибут не должен содержать внутренней структуры (она нам не важна, мы рассматриваем атрибут как объект)
3) Когда объект имеет составной идентификатор, каждый атрибут, являющийся частью этого идентификатора, представляет характеристику всего объекта, а не части объекта и, тем более, не чего-то другого.
4) Каждый атрибут, не являющийся частью идентификатора, представляет характеристику объекта, а не другого атрибута или чего-то другого.

Связи:

Связь – абстракция набора отношений, которые возникают между объектами в реальном мире.

Задаем каждую связь из перспективы участвующих объектов.

Каждой связи присваивается уникальный идентификатор, состоящий из буквы и номера



Типы:

а) один к одному

один ко многим

многие ко многим

б) безусловная (всегда держится)

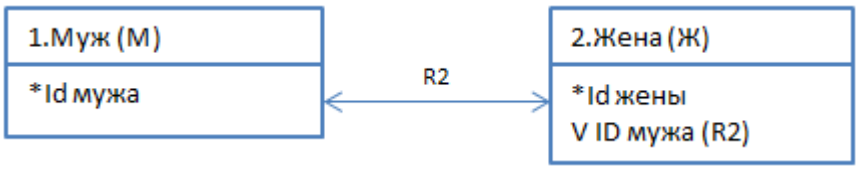
условная (один из объектов может не участвовать в связи)

биусловная (возможно неучастие с обеих сторон)

Иногда с какой-то стороны объект МОЖЕТ не участвовать в связи, тогда связь – условная (преподаватель не руководит студентами – условная связь со стороны преподавателя, т.к. у студента должен быть руководитель, а преподаватель не обязан руководить). Если с обоих сторон объекты МОГУТ не участвовать в связи, то связь – биусловная (УчебныйКурс – Студент; курс не читается в этом семестре, или студент не выбрал этот курс). Итого – 3 безусловных (ОКО, ОКМ, МКМ), 4 условных ОуКМ, ОКМу, МуКМ, МкМу), 3 биусловных(ОукОу, ОуКМу, МуКМу).

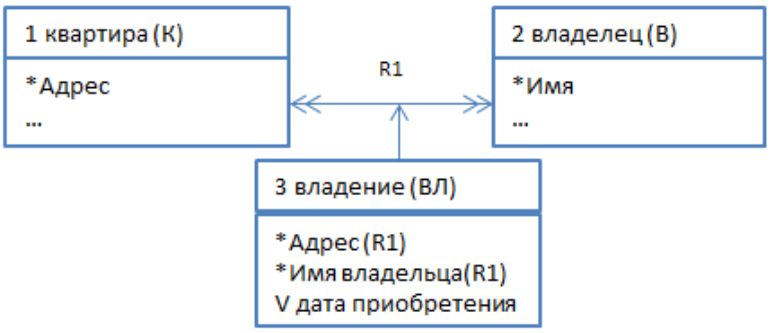
Формализация связей:

1) Безусловная один к одному: добавляем атрибут связи в любой из объектов (тот, который более осведомлен о системе)



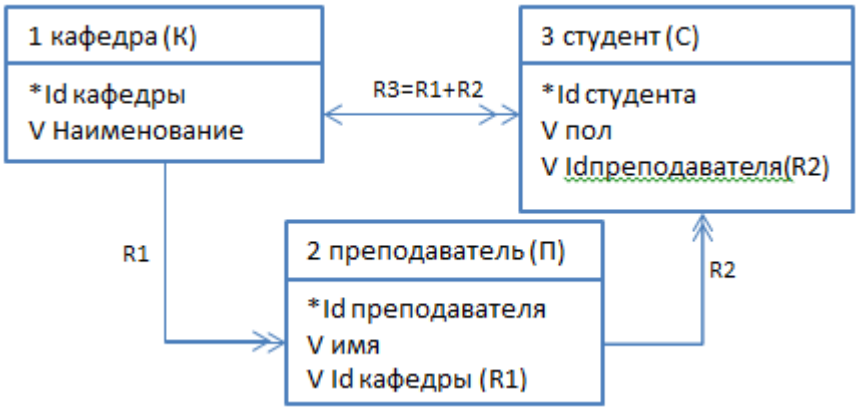
2) Безусловная один ко многим: формализуется со стороны многих (добавляем им атрибут связи)

3)Многие ко многим: формализуется через ассоциативный объект:



4) Если связь условна, биусловна или имеет динамическое поведение, то она формализуется ассоциативным объектом

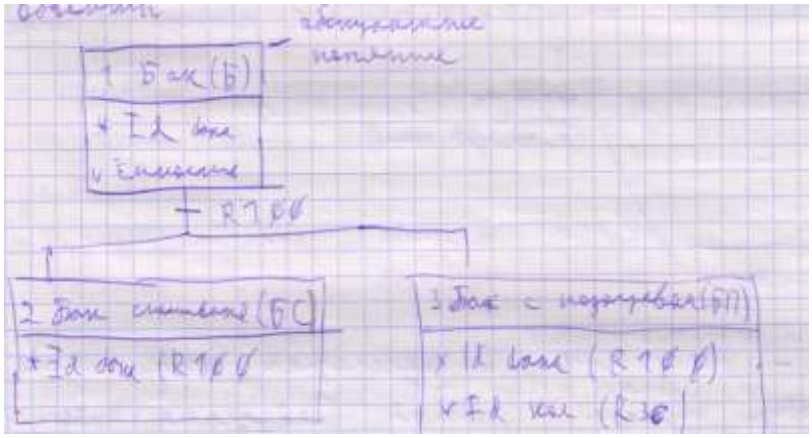
Некоторые связи – композиция других связей:



Подтипы, супертипы:

В супертипе – общие атрибуты для разных объектов.

Супертип – всегда абстрактное понятие.



При информационном моделировании получаем:

а) информационную модель

б) описание объектов и их атрибутов

в) описание связей и их формализацию

9. ООА. Динамическое поведение объектов, понятия состояний, событий, действий состояний, жизненный цикл.

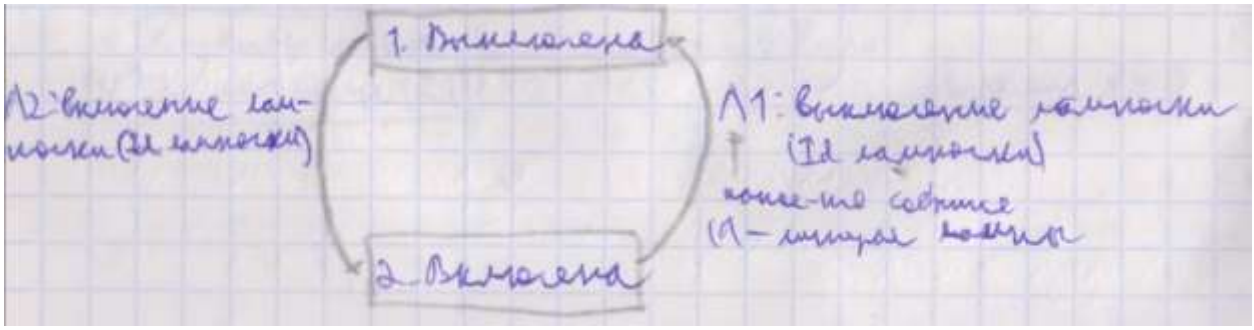
Жизненные циклы объектов:

- В каждый момент объект находится в какой-то одной стадии
- Переход из одной стадии в другую происходит скачкообразно и является реакцией на какой-то инцидент
- Переходы возможны не из всех состояний

Модель состояний Мура:

1) Множество состояний
2) Множество событий (инцидентов)
3) Правила перехода
4) Действие (для каждого состояний)

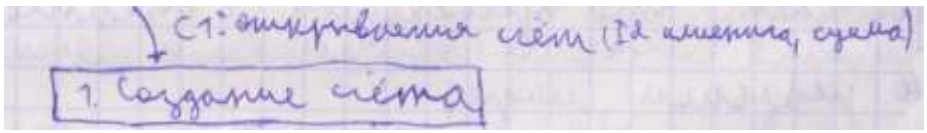
Диаграмма переходов состояний (ДПС):



Состояние – положение объекта, в котором применяется особый набор правил и линий поведения, предписаний физических законов (для состояния ставятся в соответствие уникальные в рамках данной модели состояний им и номер)

Виды состояний:

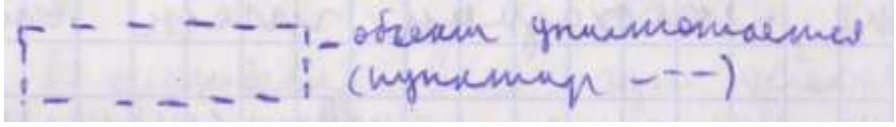
1) Состояние создания (переход в состояния создания – не из состояния)



2) Заключительные состояния

   а) состояния, из которых объект больше не переходит

   б) состояния, в которых объект уничтожается



3) текущее состояние (состояние, котором объект может находиться, кроме 1) и 2))

Если для объекта выделяется модель состояний, то на информационной модели для класса этого объекта нужно добавить атрибут «статус», хранящий состояние

Событие – это абстракция инцидента или сигнала в реальном мире, сообщающего о перемещении чего-либо в новое состояние.

Характеристики событий:

a) Значение – короткая фраза, которая сообщает, что происходит с объектом в реальном мире.

b) Предназначение – модель состояний, принимающая событие (единственная)

c) Метка события – ключевой литерал + номер (уникальные), внешние события помечаются буквой *E*.

d) Данные – необходимы, чтобы выполнилось действие, которое соответствует состоянию. Могут быть идентифицирующие и вспомогательные (дополнительные)

e) События должны переносить данные

События, переводящие объект из одного состояния в другое (не создающие объект) должны нести идентификатор объекта.

Правила связи состояний и событий:

1) Все события, которые вызывают переход в одно и то же состояние, должны нести одни и те же данные.

2) Если это не состояние создания, то событие должно переносить идентификатор объекта.

3) Событие, переводящее в состояние создания, не несет идентификатор объектов.

Можно добавлять состояния, соответствующие переходным процессам.

Действие – деятельность или операция, которая выполняется при достижении объектом состояния. Каждому состоянию ставится в соответствие одно действие. Действие должно выполнятся любым объектом одинаково.

Действие может:

1) Выполнять любые вычисления

2) Порождать любые события для текущего объекта

3) Порождать события для чего-либо вне области анализа

4) Порождать события для классов этого домена

5) Читать, записывать атрибуты собственного класса и других классов

Требования на действия:

1) Действие не должно оставлять данные объекта противоречивыми

2) Действие должно менять статус или атрибут состояния (если не переводит в то же состояние)

Если псевдокод действия маленький, то его можно написать под состоянием.

Действие – событие – время:

1) Только одно действие конечного автомата (КА) может выполняться в конкретный момент

2) Действия различных КА могут выполняться параллельно

3) События никогда не теряются

4) Если событие порождено для объекта, который в данный момент выполняет действие, то данное событие не будет принято, пока действие не закончится.

5) Не все события обрабатываются, событие может быть проигнорировано.

Таблица переходов состояний (ТПС):

Матрица: строки – состояние, столбцы – события

События / Состояния			

Должны быть все состояния и события, и все клетки должны быть заполнены.

Варианты заполнения:

- номер нового состояния

- игнорирование (-)

- событие не может произойти (x)

Формы жизненных циклов:

- циркуляционная (циклические)

- «рождение – смерть»

Для объектов, которые осведомлены о системе – циклический.

Для объектов с ЖЦ в одном классе посмотреть, отличаются ли ЖЦ и если да, то разделить на разные классы.

При сравнивании ЖЦ выделить общую и разную части.

Миграция объекта между подклассами (????)

Когда выделяем ЖЦ:

1. Создание или/и уничтожение во время выполнения
2. Миграция между подклассами (например, наполнение атрибутов)
3. Объект производится или возникает поэтапно
4. Объект – задача или запрос
5. Формализация динамических связей (ассоциативные объекты)
6. Объект – пассивный или спецификация

Выявление и идентификация отказов в ООМ(модели)

- Гарантируется ли исполнение события

- Для отказа добавляем событие, контролирующее отказ.

Продукты анализа событий

1) Диаграммы переходов состояний (ДПС)
2) Таблица переходов состояний (ТПС)
3) Алгоритмы действий

4) Список


1 Установка
4 Сброс
2 Отсчёт времени
3 Подача сигнала
T2
T6
1 Таймер (T)

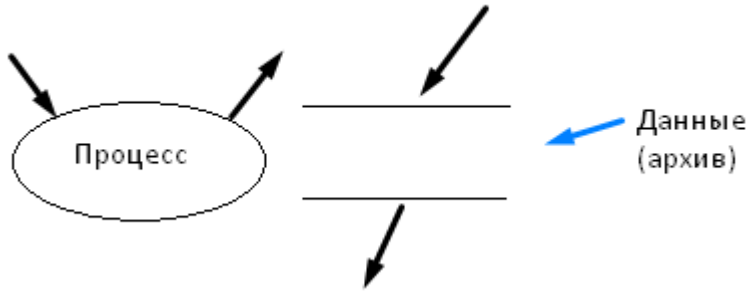
10. ООА. Диаграмма потоков данных действия. Понятие процесс и потоков управления. Модель доступа к объектам.

Диаграмма потоков данных действия (ДПДД) – графическое представление модулей процесса в пределах действия и взаимодействие между ними. Для каждого действия каждого состояния каждой модели состояния.

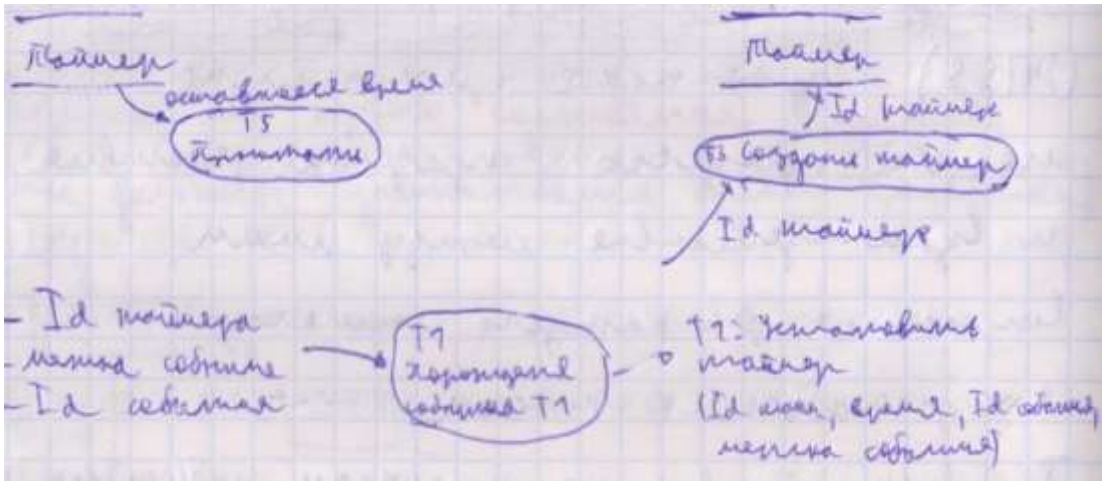
Строится для каждого состояния каждого объекта класса.

Разбиваем действия на процессы, которые могут происходить:

- Процесс проверки
- Процесс преобразования
- Аксессоры (процесс для получения данных из одного архива данных)
    - Создание
    - Чтение
    - Запись
    - Уничтожение
- Генераторы событий (создает одно событие как вывод)



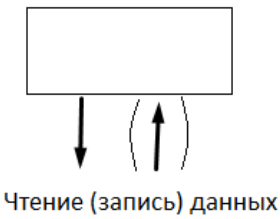
Пример:

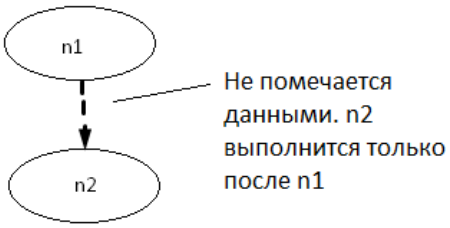


Процесс не может выполняться, пока все вводы не доступны.

Выводы процесса доступны тогда, когда процесс завершит выполнение.

Данные объектов, событий, архива и терминаторов всегда доступны.


Чтение (запись) данных из внешних сущностей



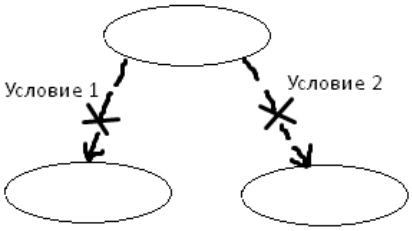
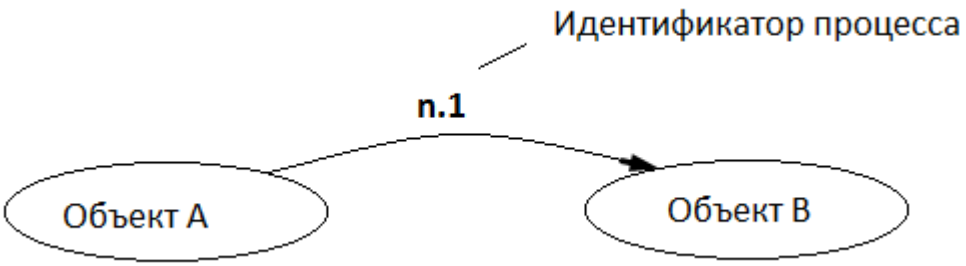


Таблица процессов действий состояний:

Id процесса	Тип	Название	Где используются	
			Модель состояний	Действия

Модель доступа к объектам:



11. ООА. Модели доменного уровня, понятие мостов, клиентов, серверов.

Домен – отдельный реальный или гипотетический населенный отчетливым набором объектов, которые ведут себя в соответствии с присущими домену правилам и линиями поведения.

Типы доменов:

a) Прикладные (предметная область системы с точки зрения пользвователя)
b) Сервисные (функционал для поддержки прикладного домена)
c) Архитектурные (обеспечивают единые механизмы управления данными и всей программой как единым целым)
d) Реализационные (библиотечный класс – взаимодействие по сети, протоколы взаимодействия по сети)

Сервис и домен – разные понятия!

Мост – связь между доменами, когда один домен использует механизмы и возможности другого.

Клиент – использующий возможности домен.

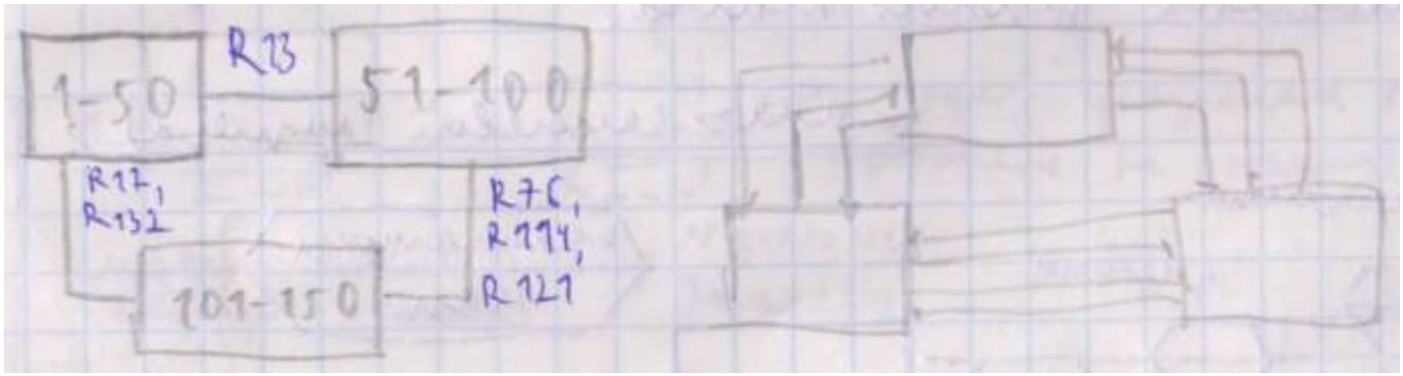
Сервер – домен, предоставляющий свои возможности для использования.

Клиент рассматривает мост как набор предложений, которые будут представлены другим доменом. Сервер подходит к мосту как к набору требований для выполнения.

Схема доменов

Домены и мосты между ними. В прямоугольниках – домены, стрелки – мосты.

Сервисные домены и домены к задаче – снизу:



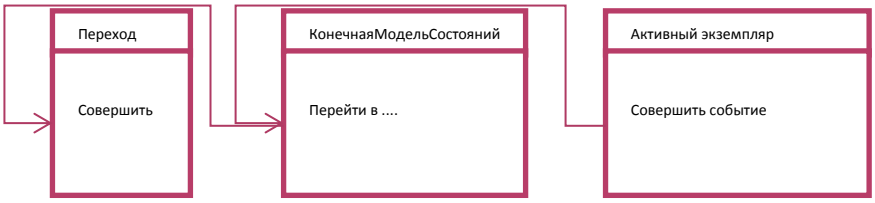
Схемы для подсистем:

- Модель связи подсистем (по информационной модели)
- Модель взаимодействия подсистем (по МВО)
- Модель доступа подсистем (по МДО)(стрелки - аксессоры)

12. Объектно-ориентированное проектирование. Принцип проектирования. Архитектурный домен. Шаблоны для создания прикладных классов.

Архитектурный домен обеспечивает единые механизмы управления данными и всей программой как единым целым. Архитектурный домен можно реализовать как паттерн КМС.

Создаются несколько классов для архитектурного домена, задача которых – задать правила перехода из состояния в состояние при возникновении событий. Классы, выделяемые для архитектурного домена:



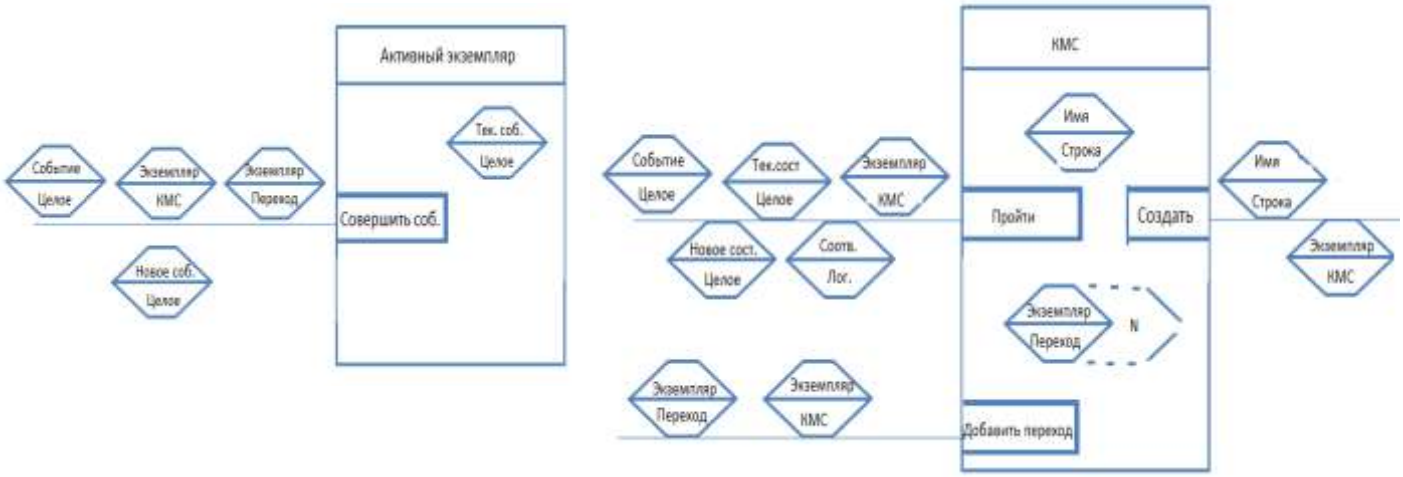
Паттерн КМС берет на себя функцию контроля. То есть уже нет необходимости проверять возможность перехода в данное состояние, как в четвертой лабе.

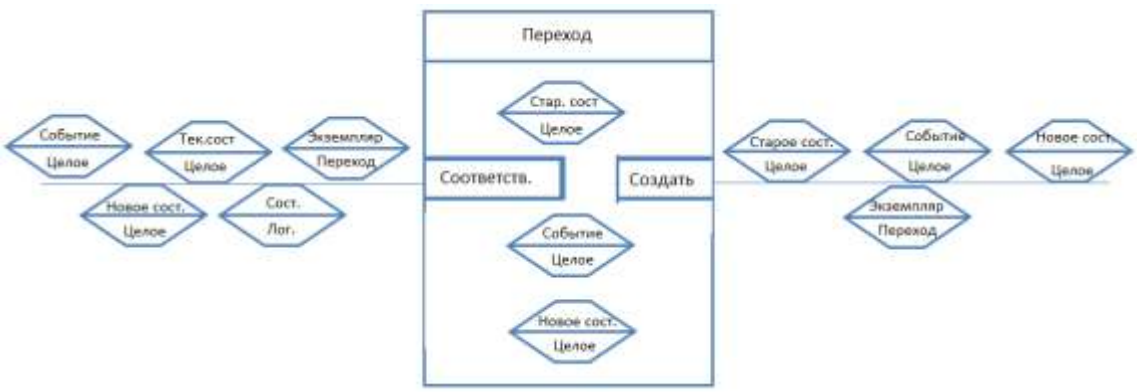
КМС делается шаблонным классом – позволяет сделать систему гибкой.
Все активные классы будут производными от активного экземпляра.

Выделяют два типа объектов:

1. Пассивные: конструкторы, аксессоры (объектов,класса);
2. Активные: обработчики событий, конструкторы, аксессоры, инициализатор кмс. (Для активных выделяют жизненные циклы).

Выделяются также объекты, осуществляющие только связь между другими – определители: конструкторы, обработчики, инициализатор КМС.(Определяют модель состояний связей объектов)



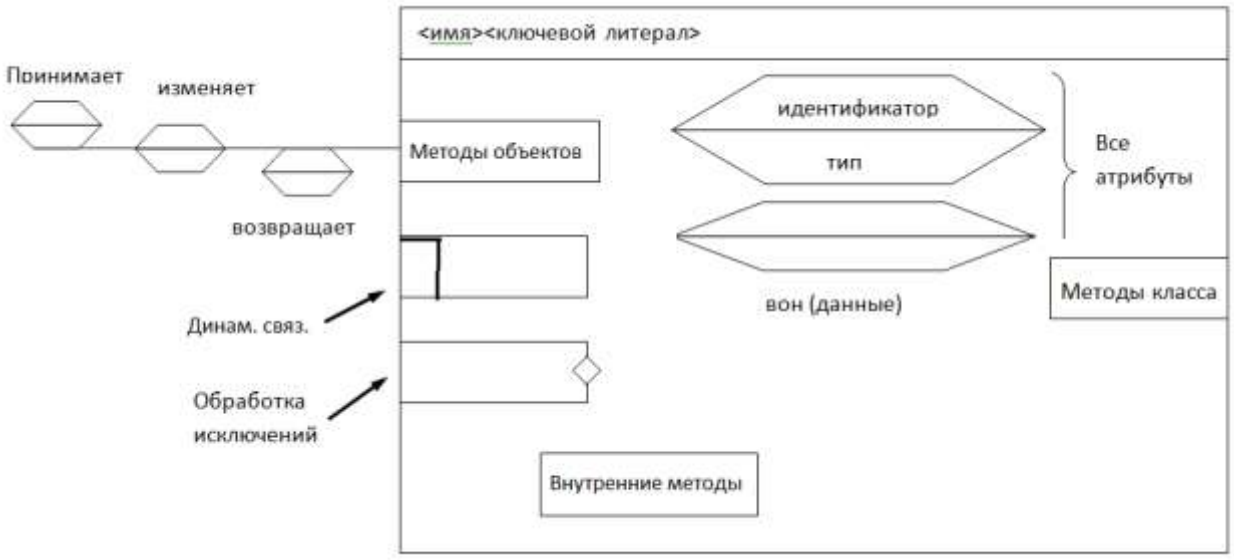


13. Объектно-ориентированное проектирование. Диаграмма класса, схема структуры класса, диаграмма зависимости, диаграмма наследования.

Нотация OODL (object oriented design language):

- Диаграмма класса (Буга-Бахра)
- Схема структуры класса (внутренняя структура кода операций класса)
- Диаграмма зависимостей (взаимодействие клиент-сервер, механизм использования/выполнения и дружественных связей)
- Диаграмма наследования (показать наследование)

Диаграмма класса:



Описывает данные объекта и класса, внешнее представление данного класса.

Схема структуры класса:

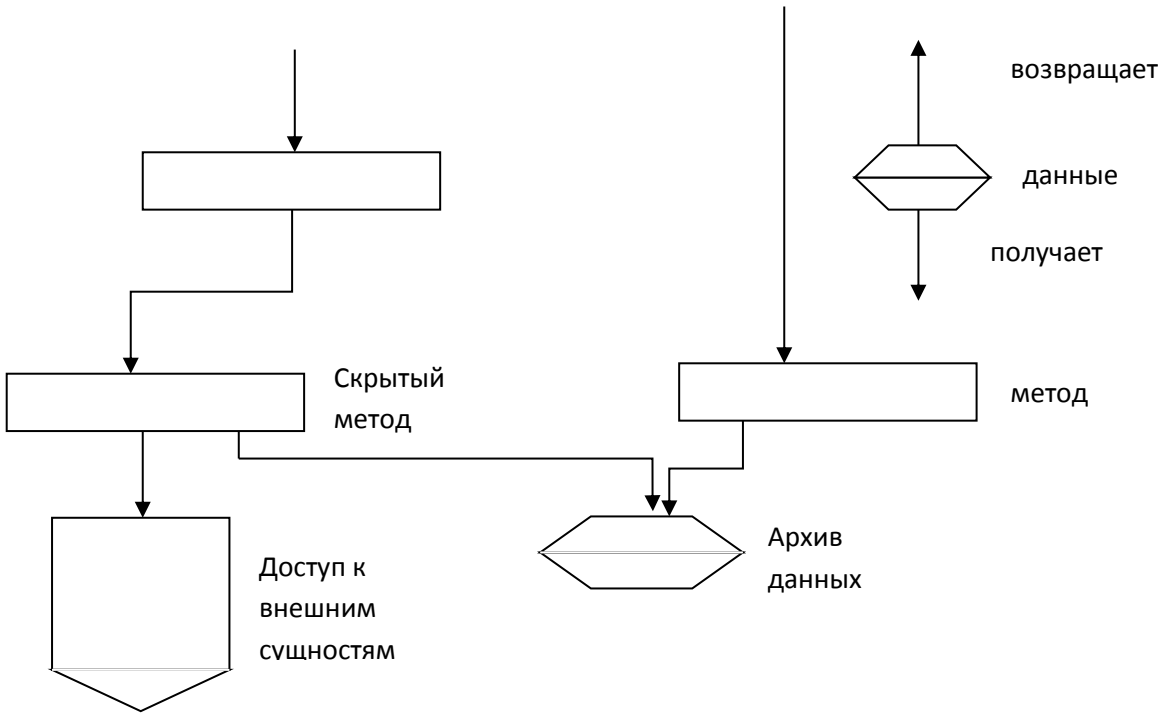


Диаграмма зависимостей:

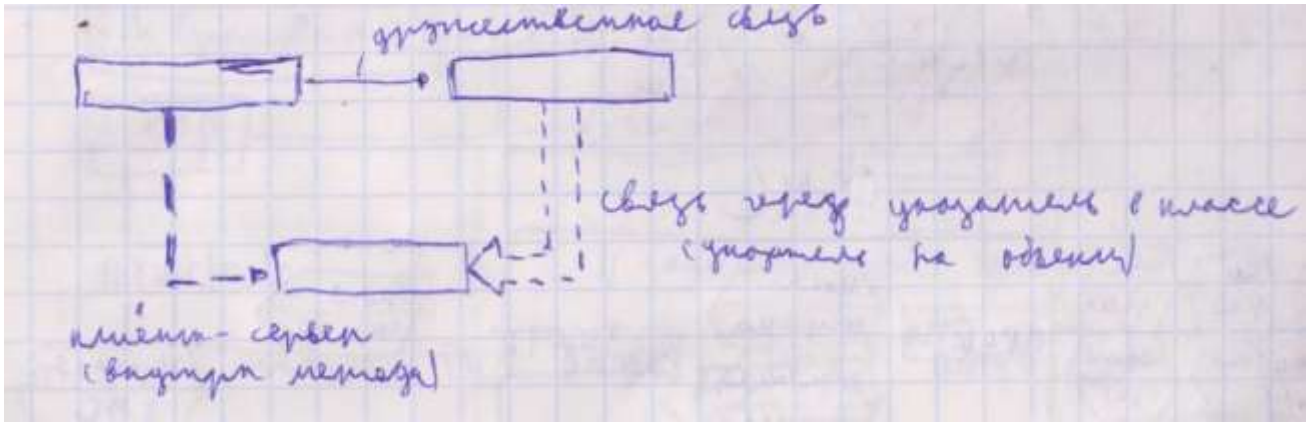
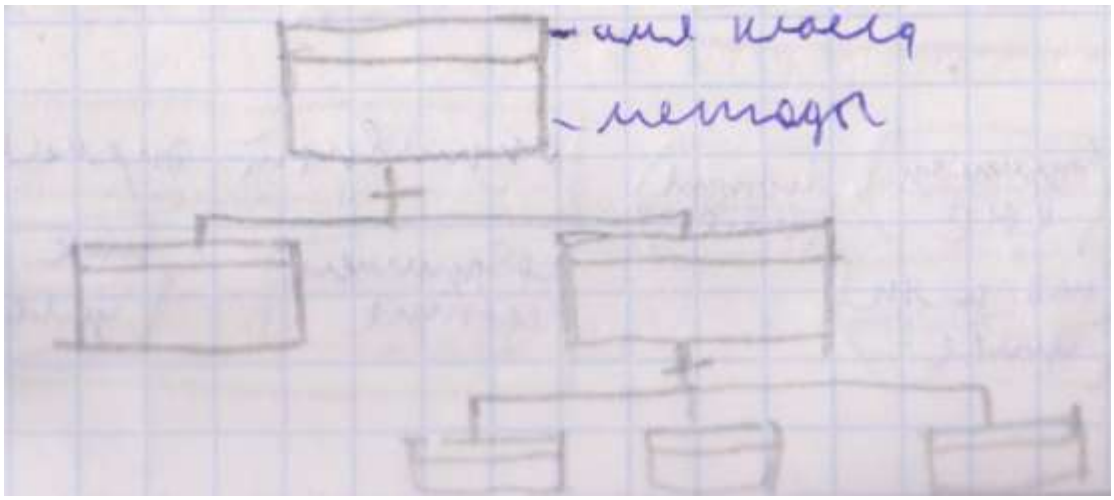


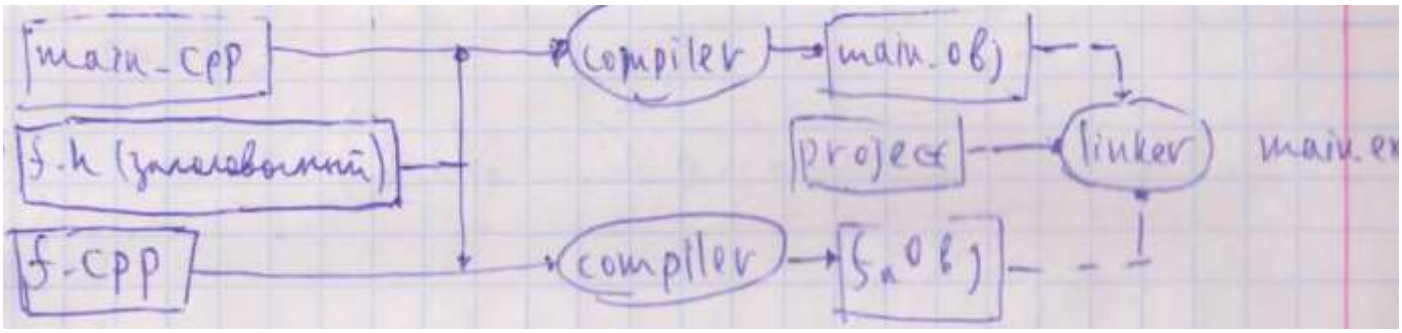
Диаграмма наследования:



1. Структура программы на языках C, C++.

Главная функция – main. Должна обязательно быть, не может вызывать саму себя.

Файлы компилируются раздельно. Чтобы нормально собрать программу, создают заголовочные фалы, где содержатся константы и объявления функций.




```
#include “f.h”

int main(int argc, char* argv[])

{

   return 0;

}
```

Для избегания многократного объявления:

```
#pragma once или

#ifndef G_H

#define G_H

…

#endif
```

Отличия C++ от C:

- Классы и шаблоны
- Перегрузка функций
- Операторы new и delete
- Обработка исключений через throw/catch

2. Классы и объекты, ограничение доступа.

Класс – описывается в заголовочных файлах.

Механизм описания класса.

- struct
- union – не может быть базовым классом, и не может быть производным.
- По умолчанию в структуре и объединении члены классов открыты, в классе закрыты, чем достигается принцип инкапсуляции (нет доступа к данным извне)
- Class

Пример:

```
class <имя класса> [:<список базовых классов>]
{
private: //доступ к данным есть только внутри самого класса
        int a;
protected: //доступ есть внутри класса и во всех его наследниках
        int b;
public: //доступны для внешнего кода
        int f();
}; //класс заканчивается ; как и структура
//Классы описываются в заголовочных файлах .h. Методы же определяются в .cpp
```

Уровни доступа:
1. private
2. protected
3. public

Функции класса будем называть методами.
Protected – члены к которым имеют доступ только методы класса и ПРОИЗВОДНЫЕ от него классы
Располагать члены в порядке private, protected, public.
Если создаём библиотеки, то описываем в обратном порядке.
Схем наследования тоже 3:
1. private
2. protected
3. public

Пример наследования:

```
class A
{
private:
        int a;
protected:
        int b;
public:
        int f();
};

class B: private(или protected, или public) A
{
private:
        int c;
protected:
        int d
public:
        int g ();
};
```

Рассмотрим каждое наследование по отдельности:
private – полное сохранение интерфейса базового класса. (рисунок 1)
protected – (рисунок 2)
public – (рисунок 2)

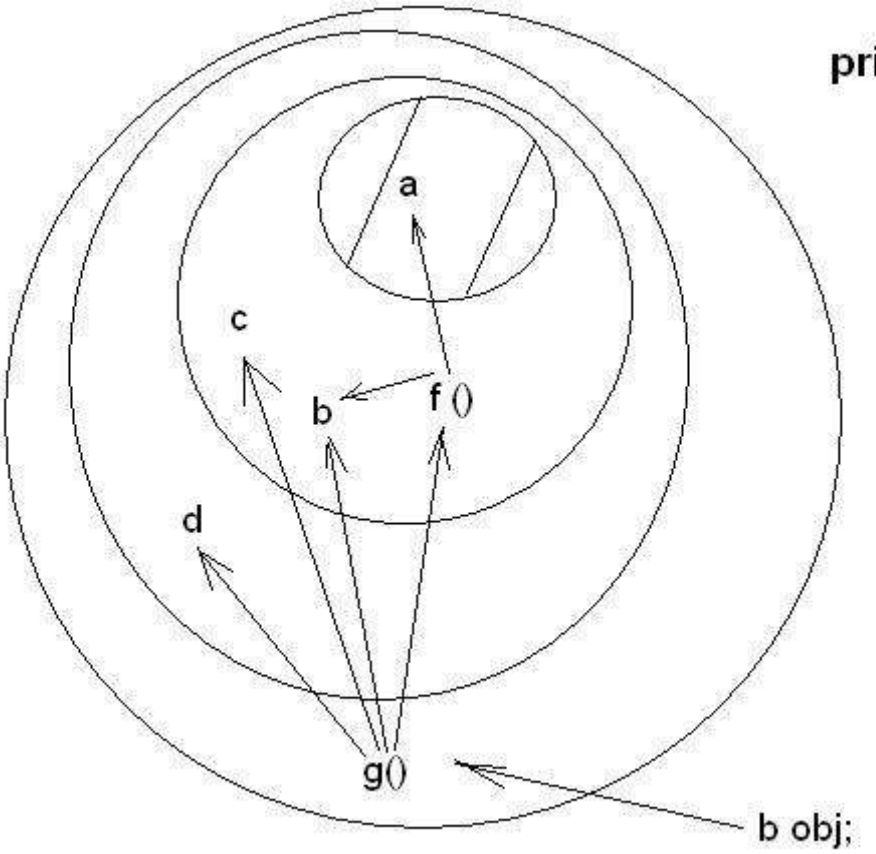


Рис 1

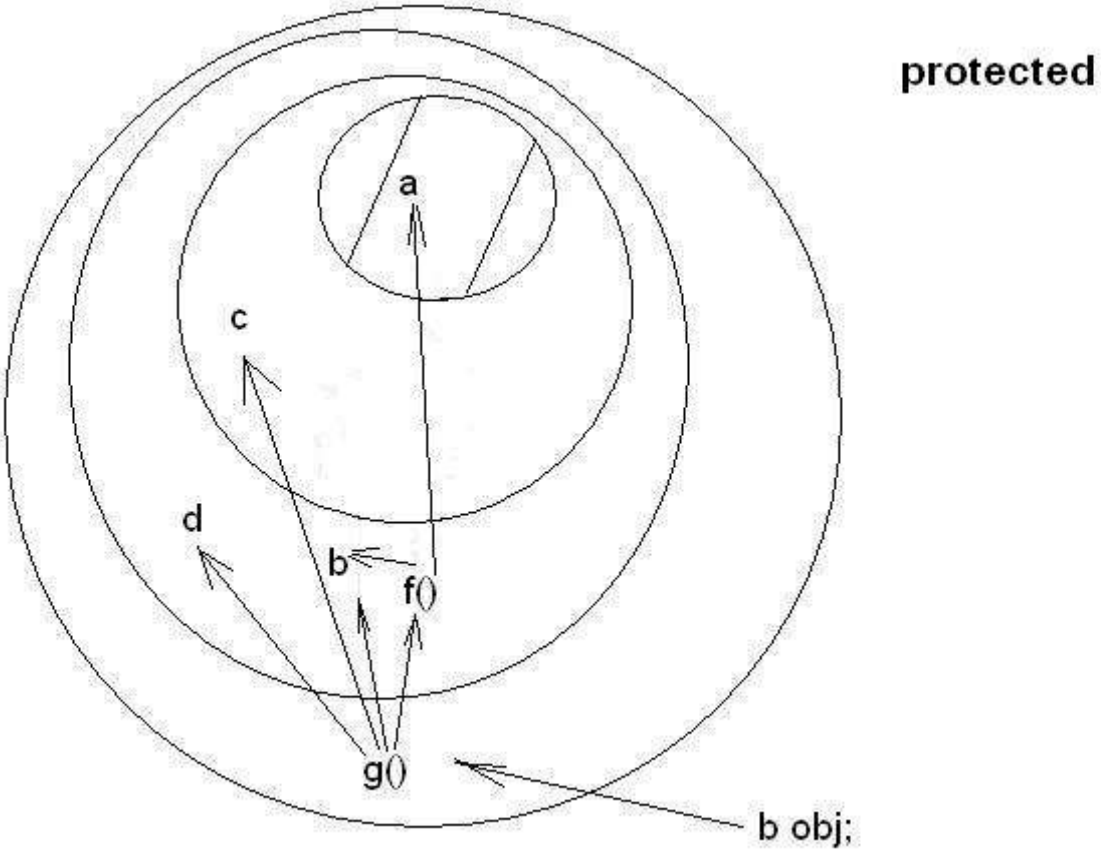


Рис 2

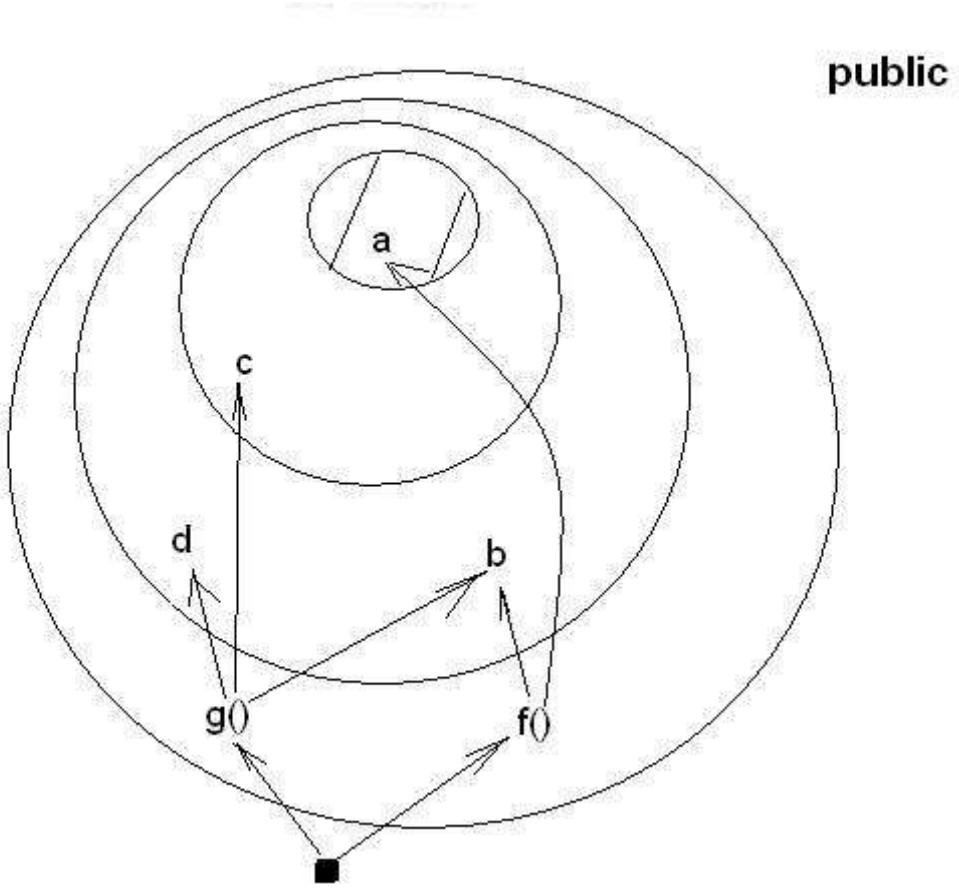


Рис 3.

Статистические члены класса:

- Как данные так и методы
- Особенности статистических данных

```
Class A
{
Private:
        Int a;
        Static int b;
Public:
        Int f();
        Static int g(A* obj)
}
```

- Не член объекта, а член класса (общий для всех экземпляров)
- Int A::b = 0 инициализация статического члена
- Не применим указатель this
- Вызывается для класса
- Можно вызвать без создания объекта (A::g(pobj))

Константные члены

```
Class B
{
Private:
        Const int a;
        Const static int b  = 2;
Public:
        Int f(); //1
        Int f() const; // 2
}
```

- Не меняются во время жизни объекта
- Можно инициализировать только в конструкторе
- Можно создать константный объект с константными методами
- Для константного объекта возможен вызов только константных методов

```
B obj;
Const B cobj;
```

```
Obj.f() // 1
Cobj.f() //2
```

Ссылки

```
int i;
int& ai = I;
ai = 1; // i = 1
```

Пример:

```
void swap(double& T1, double& T2)
{
        double tmp = T1;
        T1 = T2;
        T2 = tmp;
}
swap(A[i], A[j]);
```

Пример:

```
(f(A)*f(x) > 0? A:B) = x;

double&  g(double& A, double& B, double x)
{
    return f(A)*f(x) >0? A:B;
}

g(A,B,x) = x;
```

- Несколько имен для одного объекта
- Используется для изменения и передачи больших значений

3. Создание и уничтожение объектов.

Выделили метод, который называли конструктор.

Это метод вызываемый при инициализации объекта. У него отсутствует тип возврата.

Конструктор можно перегружать. Конструктор не наследуется.

Когда вызывается конструктор:

1) При определении для статических и внутренних объектов. Выполняется до функции *main ().*
2) *При определении локальных объектов.*
3) При выполнении оператора new
4) Для временных объектов.

Если не определить ни один конструктор, то будет вызываться конструктор по умолчанию.

Конструктор копирования принимает ссылку на константный объект.0

Конструктор копирования вызывается:

- При инициализации одного объекта другим.
- При передаче по значению параметров
- При возврате по значению.

Пример:

```
class Complex
{
private:
   double re, im;
public:
   Complex ();                        //1
   Complex (double r);                //2
   Complex (double r, double i);      //3
   Complex (Complex &c);              //4
};
Complex a(); // функция, возвращающая объект
Complex b; //1
Complex c1 (1.); //2
Complex c2 = 2.; //2
Complex d1 (3. ,4.); //3
d2 = Complex (5., 6.); //3
e1 (d1), e2 = d2 //4;
```

Конструктор не может быть volatile, static, const, virtual.

В C++ реализуется неявный вызов деструктора. Этот метод не принимает параметров. Нет типа возврата. Деструктор имеет такое же имя что и конструктор, но начинается со знака ~.

Деструкторы вызываются в обратном порядке.

Для локальных статических объектов вызывается деструктор до уничтожения глобальных статических объектов, но после выполнении программы.

Временные объекты уничтожаются, когда в них отпадает надобность.

Деструктор не перегружается.

Деструктор не может быть const, volatile, static, но может быть virtual.

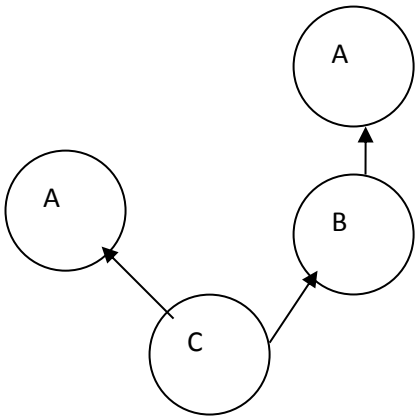
4. Наследование, построение иерархии, множественное наследование и неоднозначности в нём.

Расширение и выделение общей части из разных классов.
Причины выделения общей части:

1. Общая схема использования.
2. Сходство между наборами операций.
3. Сходство реализации.

II. Расщепление классов

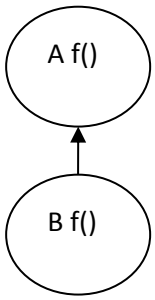
a) Два подмножества операций в классе используются в разной манере.
b) Методы класса имеют несвязную реализацию.
c) Класс оперирует очевидным образом в 2-х несвязных обсуждениях проекта.



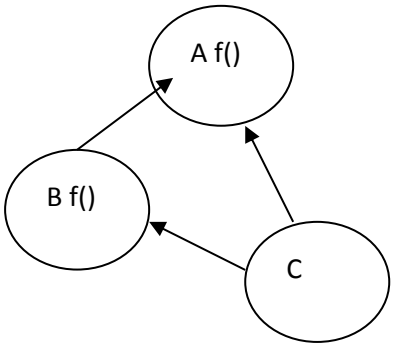
Плюсы множественного:

- Гибкая схема
- Уменьшение иерархии наследования

Минус множественного наследования : неоднозначности.
Прямая база – непосредственная база класса. Прямая база может входить только один раз
Косвенная база – А для С. Может быть несколько раз.
Тут А для С и косвенная и прямая база.



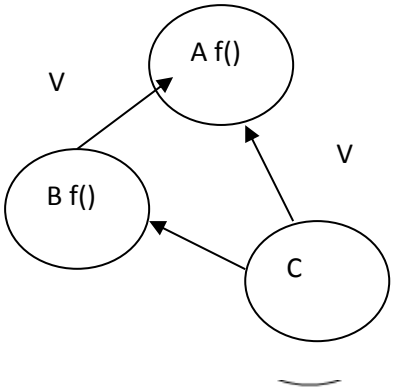
Доминирование:
Метод f () в производном классе B доминирует над методом в доминантном классе A.
Не путать доминирование с virtual, потому что при наличии virtual используется таблица виртуальных функций.

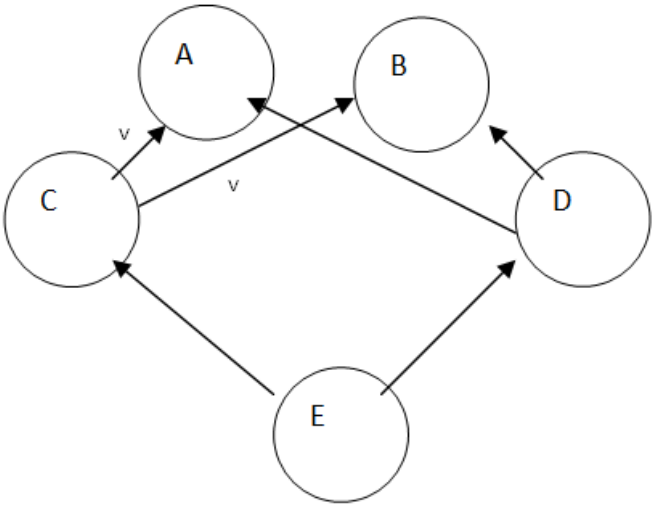


ОШИБКА!

(Неоднозначность)

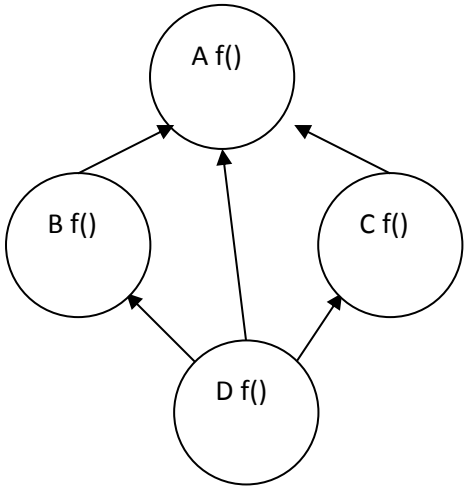
Виртуальное наследование.






```
class A {};
class B {};
class C:virtual public A,
      virtual public B{};
class D:virtual public A,
      virtual public B{};
class E:public C, public D{};
```

A для C, B для C, C, D, E




```
void D::f()
{
      A::_f();
      B::_f();
      C::_f();
      _f();
```

Пример:

```
class A
{
public:
	int a;
	int (*b)();
	int f();
	int f(int);
	int g();
};

class B
{
private:
	int a;
	int b;
public:
	int f();
	int g;
	int h();
	int h(int);
};

class C:public A, public B{};

void f (C *pc)
{
	pc->a = 1;		//Error!	Проверка на неоднозначность происходит
	pc->b ();		//Error!	до проверки на степень доступа.
	pc->f ();		//Error!
	pc->f (1);		//Error!
	pc->g = 1;		//Error!
	pc->h(); pc->h(1); // OK!
}
```

Решение: переопределить неоднозначные методы в классе C.

5. Полиморфизм, понятие абстрактного класса. Дружественные связи.

Полиморфизм – использование объектов с одинаковым интерфейсом без информации о типе и внутренней структуре объекта.(“Один интерфейс, множество реализаций.”)

Виртуальные методы – это методы, которые предполагаемо будут переопределены в производных классах. Ключевое слово –virtual. Гарантируется, что будет выбрана верная функция.

Виртуальный метод может быть дружественным (friend) и inline.

Пример:

```
class A {
public:
    virtual void f();
};
class B : public A {
public:
    B() { f(); }
    void g() { f(); }
};
class C: public B{
public:
    C(){}
    void f();
};

    C c; //A::f()
    c.g(); //C::f()
```

Чисто виртуальная функция не имеет тела. Класс с хотя бы одной чисто виртуальной функцией называется абстрактным.

```
class A {
public:
    virtual void f() = 0;
};
Класс, который не реализует число виртуальный метод, так же называется абстрактным.
class B : public A {};
```

Дружественный класс получает доступ ко всем объектам данного класса(*объекты плохо звучит)*, но не наоборот. Дружественным может быть и метод.

Дружба не наследуется и не транзитивна.

```
class B;
class A
{
    friend class B;
    friend void B::f(A&); //  только метод имеет доступ
};
```

Дополнительная информация:

- В том случае если базовый и производный классы имеют общий открытый интерфейс, говорят, что производный класс представляет собой подкласс базового.
- Отношение между классом и подклассом, позволяющее указателю или ссылке на базовый класс без вмешательства программиста адресовать объект производного класса, возникает в C++ благодаря поддержке полиморфизма.
- Полиморфизм позволяет предложить такую реализацию ядра объектно-ориентированного приложения, которая не будет зависеть от конкретных используемых подклассов.
- Динамическая идентификация типов времени выполнения (англ. Real-Time Type Identification) обеспечивает специальную поддержку полиморфизма и позволяет программе узнать реальный

производный тип объекта, адресуемого по ссылке или по указателю на базовый класс. Поддержка RTTI в C++ реализована двумя операциями:

1. операция **dynamic_cast** поддерживает преобразование типов времени выполнения;
2. операция **typeid** идентифицирует реальный тип выражения.

- Операции RTTI — это события времени выполнения для классов с виртуальными функциями и события времени компиляции для остальных типов. Исследование RTTI-информации полезно, в частности, при решении задач системного программирования.

dynamic_cast:

- Встроенная унарная операция dynamic_cast языка C++ позволяет:
    1. **безопасно трансформировать указатель на базовый** класс в указатель на производный класс (с возвратом нулевого указателя при невозможности выполнения трансформации);

    2. **преобразовывать леводопустимые значения**, ссылающиеся на базовый класс, в ссылки на производный класс (с возбуждением исключения bad_cast при ошибке).

       //Леводопустимое выражение – это выражение, адрес которого мы можем получить.(но лучше так Тассову не говорить ☺)
- Единственным операндом dynamic_cast должен являться тип класса, в котором имеется хотя бы один виртуальный метод

Пример для указателей:

```
Alpha *alpha = new Beta;
if(Beta *beta = dynamic_cast<Beta*>(alpha)) {
// успешно…
}
else {
// неуспешно…
}
```

Пример для ссылок:

```
#include <typeinfo> // для std::bad_cast
void foo(Alpha &alpha) {
// …
try {
Beta &beta = dynamic_cast<Beta&>(alpha))
}
catch(std::bad_cast) { /* … */ }
}
```

typeid:

Встроенная унарная операция typeid:

1. позволяет установить фактический тип выражения-операнда;
2. может использоваться с выражениями и именами любых типов (включая выражения встроенных типов и константы).

Если операнд typeid принадлежит типу класса с одной и более виртуальными функциями (не указателю на него!), результат typeid может не совпадать с типом самого выражения.

Операция typeid имеет тип (возвращает значение типа) type_info и требует подключения заголовочного файла <typeinfo>.

Реализация класса type_info зависит от компилятора, но в общем и целом позволяет получить результат в виде неизменяемой C-строки (const char*), присваивать объекты type_info друг другу (operator=), а также сравнивать их на равенство и неравенство (operator==, operator!=).

6. Перегрузка операторов

Существует два основных способа перегрузки операторов: глобальные функции, дружественные для класса, или подставляемые функции самого класса.

```
class Integer
{
    //унарный +
    friend const Integer& operator+(const Integer& i);
    bool operator==(const Integer& left, const Integer& right) {
        return left.value == right.value;
}
```

- Нельзя перегрузить операторы “.”, “::”(оператор разрешения области видимости),” , “?:”(тернарный оператор),sizeof, typeid, “.*” (выбор члена через указатель на член)
- При перегрузке не меняется сущность, приоритет и порядок выполнения.
- При перегрузке не меняется смысл*(желательно не должен меняться)*
    - Операторы = , (), [], ->, ->* перегружаются только как члены класса.
    - Оператор = не наследуется и возвращает ссылку или void
    - Перегружать () можно только один раз
    - [] - ассоциативные массивы(выбор по критерию)
    - ->, ->* возвращает указатель или ссылку на объект

```
class A{
public:
    void f(){}
};
class B{
public:
    A* operator ->(){
}
};
B obj;
obj->f();
obj.operator ->()->f(); //то же самое, что и выше
```

- <знак>= - лучше как член класса
- Унарные – лучше как член класса
- Бинарные: если объект – член класса, если создание – выполняется перегрузка
- ++a; // a.operator++() как унарный
  a++; // a.operator++(0) как бинарный

```
complex complex:: operator +(const complex &c1, const complex &c2)
{
    complex c(c1.ze + c2.ze, c1.im + c2.im);
    return c;
}
```

c1 = c2 + c3;

- Перенос и копирование

```
class A
{
public:
    A(const A&);                // Копирование
    A(A&&);                     // Перенос
    A& operator =(const A&);    // Копирование
    A& operator =(A&&);
private:
    int *buf;
};
A& A::operator =(const A& obj)
{
    delete buf;
    buf = new int[sizeof(obj)];
    copy(buff, obj.buf);
    return *this;
}
A& A::operator =(A&& obj)
{
    delete buf;
```

```
    buf = obj.buf;
    obj.buf = nullptr;
    return *this;
}
```

- new, delete – только как члены класса

```
class A
{
public:
void* operator new(size_t size);                    // new A
void* operator new(size_t size, void *p);           // new(buff) A
void* operator new [](size_t size);                 // new A[n]
void operator delete(void *p);                      // delete pobj
void operator delete[](void *p);                    // delete []p
};
```

7. Шаблоны классов

Шаблонные классы позволяют получать классы, отличающиеся только типом в отдельных местах

```
template <typename T>
class A
{
    T* u;
public:
    void f();
};
```

У класса могут быть шаблонные методы

```
template <typename T>
void A(t)::f()
{

}
```

Так же шаблоны могут иметь специализацию для отдельного типа

```
template<>
class A<float>
{
}
```

Частичная специализация:

template<typename T1, typename T2>
class A{…}

template<typename T>
class A<T,T>{…}

template<typename T>
class A<T,int>{…}

template<typename T1, typename T2>
class A<T1*,T2*>{…}

A<int,float>a1;

A<float,float>a2;

A<float,int>a3;

A<int*,float*>a4;

A<int,int>a5; // ошибка

A<int*,float*>a6; // ошибка

Так же можно указаывать значения шаблонных параметров по умолчанию(в конце списка параметров)

template <typename T=int>…

Параметры-значения: только константные внешние объекты.

template<char *const name>
class A { }

extern char const st[] = “name”
A<st>obj;

8. Обработка ошибок

Выбрасываем ошибку

```
if(...) throw <имя>();
```

Ловим ошибку

```
try
{
...
}
catch (<имя> & идентификатор)
{
 ...
}
```

В качестве выбрасываемого объекта передаем объекта класса, отвечающий за определенный тип ошибки, но унаследованный от базового класса ошибки(как во второй л\р).

Как делать нельзя: (страшный код)

```
try
{
  A* pobj;
  pobj->f();
  delete pobj;
}
```

Дополнительная информация:

- Естественный порядок функционирование программ нарушают возникающие нештатные ситуации, в большинстве случаев связанные с ошибками времени выполнения (иногда — с необходимостью внезапно переключить контекст приложения).
  В языке C++ такие нештатные ситуации называются **исключительными (иначе говоря — исключениями).** Например:
  1. нехватка оперативной памяти;
  2. попытка доступа к элементу коллекции по некорректному индексу;
  3. попытка недопустимого преобразования динамических типов и пр.

  Архитектурной особенностью механизма обработки исключительных ситуаций в языке C++ является принципиальная **независимость** (несвязность) фрагментов программы, где исключение возбуждается и где оно обрабатывается. Обработка исключительных ситуаций носит невозвратный характер.

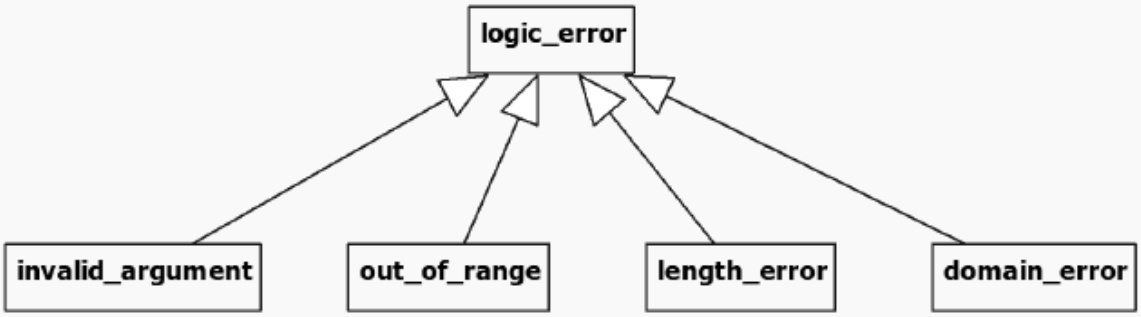
- Носителями информации об аномальной ситуации (исключении) в C++ являются объекты заранее выбранных на эту роль типов (пользовательских или — Sic! — базовых, например, char*). Такие объекты называются **объектами-исключениями**.
  Жизненный цикл объектов-исключений начинается с возбуждения исключительной ситуации посредством оператора throw:

```
throw "Illegal cast"; // char *
throw IllegalCast(); // class IllegalCast
enum EPrgStatus {OK, BADINDEX, ILLEGALCAST};
      throw ILLEGALCAST; // enum EPrgStatus
```

- Исключение, для обработки которого не найден catch-блок, инициирует запуск функции terminate(), передающей управление функции abort(), которая аварийно завершает программу.
- Стандартная библиотека языка C++ содержит собственную иерархию классов исключений, являющихся прямыми или косвенными потомками базового класса exception. Потомки класса exception условно представляют две категории ошибок: логические ошибки и ошибки времени исполнения.

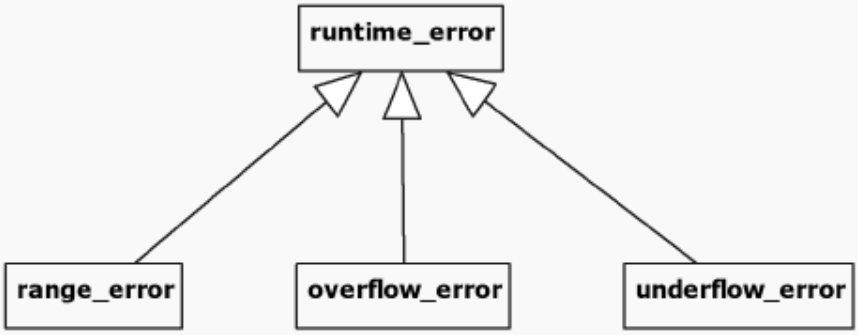
В число классов категории «логические ошибки» входят **базовый** промежуточный класс std::logic_error, а также производные от него **специализированные** классы:

- std::invalid_argument — ошибка «неверный аргумент»;
- std::out_of_range — ошибка «вне диапазона»;
- std::length_error — ошибка «неверная длина»;
- std::domain_error — ошибка «вне допустимой области».



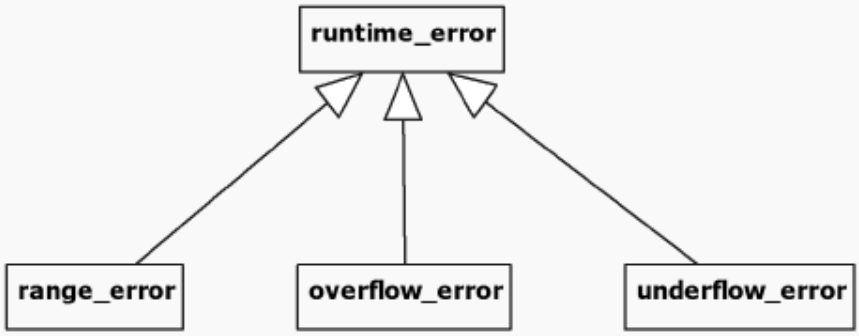
В число классов категории «ошибки времени исполнения» входят **базовый** промежуточный класс std::runtime_error, а также производные от него **специализированные** классы:

- std::range_error — ошибка диапазона;
- std::overflow_error — переполнение;
- std::underflow_error — потеря значимости.

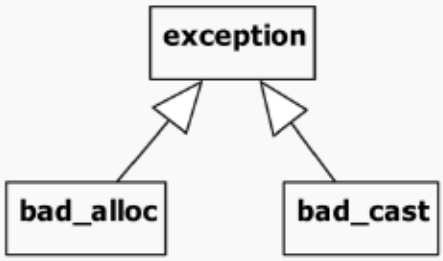


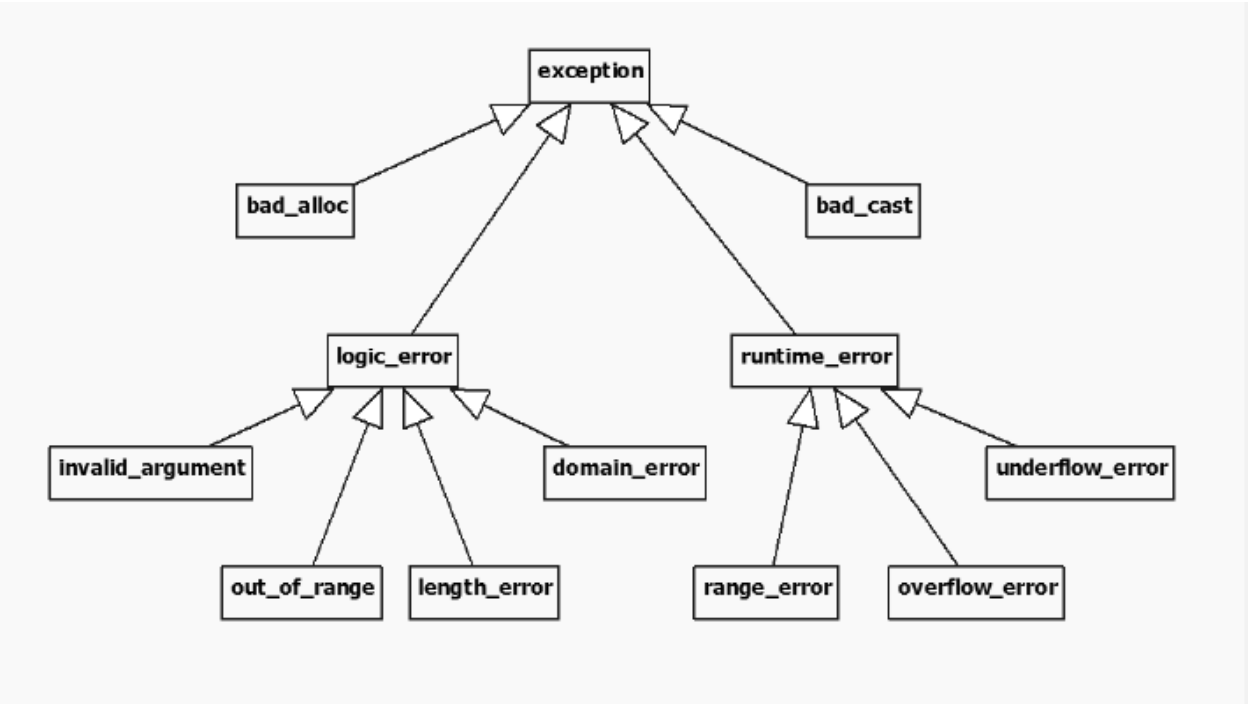
В число классов категории «ошибки времени исполнения» входят **базовый** промежуточный класс std::runtime_error, а также производные от него **специализированные** классы:

- std::range_error — ошибка диапазона;
- std::overflow_error — переполнение;
- std::underflow_error — потеря значимости.



Также производными от std::exception являются классы std::bad_alloc и std::bad_cast, сигнализирующие об ошибках при выделении динамической памяти и неудаче при выполнении «ссылочного» варианта операции dynamic_cast, соответственно.




exception
bad_alloc
bad_cast
logic_error
runtime_error
invalid_argument
domain_error
underflow_error
out_of_range
length_error
range_error
overflow_error