ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ ОБРАЗОВАТЕЛЬНОЕ
УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
«РЫБИНСКИЙ ГОСУДАРСТВЕННЫЙ АВИАЦИОННЫЙ ТЕХНИЧЕСКИЙ
УНИВЕРСИТЕТ ИМЕНИ П. А. СОЛОВЬЕВА»
(РГАТУ имени П.А. Соловьева)

ОБРАЗОВАТЕЛЬНАЯ ПРОГРАММА ПОДГОТОВКИ АСПИРАНТОВ

направление подготовки 09.06.01 Информатика и
вычислительная техника

профиль подготовки 05.13.06 Автоматизация и управление
технологическими процессами и производствами (в промышленности)

КОНСПЕКТ ЛЕКЦИЙ

по дисциплине

Организация программного обеспечения АСУ

Разработал: д.т.н. Юдин А. В.

Рыбинск, 2014 г.

Рассматриваются специализированные информационные технологии в сфере автоматизации процессов и производств, технологии программирования, виды и компоненты программного обеспечения, операционные системы, моделирующие системы в АСУ, системы моделирования электрических схем.

Содержание

Лекция 1. Технологии структурного и объективно-ориентированного программирования .4
Лекция 2. Программирование математических структур (матрицы и конечные графы) 11
Лекция 3. Методические и инструментальные средства разработки модульного программного обеспечения АСУ 17
Лекция 4. Компиляция и редактирование связей 24
Лекция 5. Автоматизация разработки программных проектов 26
Лекция 6. Операционные системы. Трансляторы. Эмуляторы 34
Лекция 7. Сравнительный анализ формальных алгоритмических языков программирования 38
Лекция 8. Системы моделирования электрических схем 41
Лекция 9. Математические модели отдельных компонент схемы 46
Лекция 10. Управляющие программы АСУ 50
Лекция 11. Обрабатывающие программы АСУ 51
Лекция 12. Системная диспетчерская программа АСУ 52

Лекция 1. Технологии структурного и объективно-ориентированного программирования.

ИСТОРИЯ СОЗДАНИЯ ОБЪЕКТНО-ОРИЕНТИРОВАННОГО ПРОГРАММИРОВАНИЯ

Практически сразу после появления языков третьего поколения (1967) ведущие специалисты в области программирования выдвинули идею преобразования постулата фон Неймана: "данные и программы неразличимы в памяти машины". Их цель заключалась в максимальном сближении данных и кода программы. Решая поставленную задачу, они столкнулись с задачей, решить которую без декомпозиции оказалось невозможно, а традиционные структурные декомпозиции не сильно упрощали задачу. Усилия многих программистов и системных аналитиков, направленные на формализацию подхода, увенчались успехом.

Были разработаны три основополагающих принципа того, что потом стало называться объектно-ориентированным программированием (ООПр): наследование; инкапсуляция; полиморфизм.

Результатом их первого применения стал язык Симула-1 (Simula-1), в котором был введен новый тип — объект. В описании этого типа одновременно указывались данные (поля) и процедуры, их обрабатывающие — методы. Родственные объекты объединялись в классы, описания которых оформлялись в виде блоков программы. При этом класс можно использовать в качестве префикса к другим классам, которые становятся в этом случае подклассами первого. Впоследствии Симула-1 был обобщен, и появился первый универсальный ООПр — объектно-ориентированный язык программирования — Симула-67 (67 — по году создания).

Как выяснилось, ООПр оказались пригодными не только для моделирования (Simula) и разработки графических приложений-(SmallTalk), но и для создания большинства других приложений, а их приближенность к человеческому мышлению и возможность многократного использования кода сделали их наиболее используемыми в программировании.

Объектно-ориентированный подход помогает справиться с такими сложными проблемами, как уменьшение сложности программного обеспечения; повышение надежности программного обеспечения; обеспечение возможности модификации отдельных компонентов программного обеспечения без изменения остальных его компонентов; обеспечение возможности повторного использования отдельных компонентов программного обеспечения.

ВВЕДЕНИЕ В ОБЪЕКТНО-ОРИЕНТИРОВАННЫЙ ПОДХОД К РАЗРАБОТКЕ ПРОГРАММ

В основу структурного мышления положены структуризация и декомпозиция окружающего мира. Задача любой сложности разбивается на подзадачи, а те в свою очередь разбиваются далее и т. д., пока каждая подзадача не станет простой, соответствующей модулю.

Модуль в понятии структурного программирования — это подпрограмма (функция или процедура), оформленная определенным образом и выполняющая строго одно действие. Методы структурного проектирования используют модули в качестве строительных блоков программы, а структура программы представляется иерархией подчиненности модулей.

Модуль ООПр — файл описаний объектов и действий над ними.

Методы объектно-ориентированного проектирования используют в качестве строительных блоков объекты. Каждая структурная составляющая является самостоятельным объектом, содержащим свои собственные коды и данные. Благодаря этому уменьшена или отсутствует область глобальных данных.

Объектно-ориентированное мышление адекватно способу естественного человеческого мышления, ибо человек мыслит "образами" и "абстракциями". Чтобы проиллюстрировать некоторые из принципов объектно-ориентированного мышления, обратимся к следующему примеру, основанному на аналогии мира объектов реальному миру.

Рассмотрим ситуацию из обыденной жизни. Допустим, вы решили поехать в другой город на поезде. Для этого вы приходите на ближайшую железнодорожную станцию и сообщаете кассиру номер нужного поезда и дату, когда планируете уехать. Теперь можете быть уверены, что ваш запрос будет удовлетворен (при условии, что вы покупаете билет заранее).

Таким образом, для решения своей проблемы вы нашли *объект* "кассир железнодорожной кассы" и передали ему *сообщение,* содержащее запрос. *Обязанностью* объекта "кассир железнодорожной кассы" является удовлетворение запроса.

У кассира имеется некоторый определенный *метод,* или эвроритм, или последовательность операций (процедура), которые используют работники кассы для выполнения вашего запроса. Имеются у кассира и другие методы, например по сдаче денег, — инкассации.

Вам совершенно не обязательно знать не только детально метод, который используется кассиром, но даже весь набор методов работы кассира. Однако если бы вас заинтересовал вопрос как работает кассир, то обнаружили бы, что кассир пошлет свое сообщение автоматизированной системе железнодорожного вокзала. Та, в свою очередь, примет необходимые меры и т. д. Тем самым ваш запрос, в конечном счете, будет удовлетворен через последовательность запросов, пересылаемых от одного объекта к другому.

Таким образом, действие в объектно-ориентированном программировании инициируется посредством передачи сообщений объекту, ответственному за действие. Сообщение содержит запрос на осуществление действия конкретным объектом и сопровождается дополнительными аргументами, необходимыми для его выполнения. Пример аргументов вашего сообщения: дата отъезда, номер поезда, тип вагона. Сообщения кассира: дайте паспорт, заплатите такую-то сумму, получите билет и сдачу.

Кассир, находящийся на рабочем месте, не обязан отвлекаться от работы для пустой болтовни с покупателем билета, например, сообщать ему свой домашний телефон или сумму денег, находящуюся в сейфе кассы. Таким образом, кассир взаимодействует с другими объектами ("покупатель билета", "автоматизированная система", "инкассатор", "бригадир" и т. д.) только по строго регламентированному *интерфейсу.* Интерфейс — это набор форматов допустимых сообщений. Для исключения возможных, но недопустимых сообщений используется механизм *сокрытия информации* (инструкция, запрещающая кассиру болтать впустую на рабочем месте).

Помимо методов, кассир для успешной работы должен располагать наборами чистых бланков билетов, купюрами и монетами наличных денег (хотя бы для сдачи покупателю). Такие наборы хранятся в особых отсеках кассы, особых коробках. Места хранения этих наборов называют *полями объектов.* В программах полям объектов соответствуют переменные, которые могут хранить какие-то значения.

Покупатель билета не может положить деньги непосредственно в отсек кассового аппарата или сейф кассира, а также самостоятельно отсчитать себе сдачу. Таким образом, кассир как бы заключен в оболочку, или капсулу, которая отделяет его и покупателя от лишних взаимодействий. Помещение кассы (капсула) имеет особое устройство, исключающее доступ покупателей билетов к деньгам. Это и есть *инкапсуляция* объектов, позволяющая использовать только допустимый интерфейс — обмен информацией и предметами только посредством допустимых сообщений, а может быть, еще и подаваемых в нужной последовательности. Именно только через вызов сообщениями особых методов осуществляется обмен данных, отделяя покупателей от полей. Благодаря инкапсуляции покупатель может лишь отдавать в качестве оплаты деньги за билет в

форме сообщения с аргументом "сумма". Аналогично, но в обратном направлении кассир возвращает сдачу.

Вы можете передать свое сообщение, например, объекту "свой приятель", и он его, скорее всего, поймет, и как результат — действие будет выполнено (а именно билеты будут куплены). Но если вы попросите о том же объект "продавец магазина", у него может не оказаться подходящего метода для решения поставленной задачи. Если предположить, что объект "продавец магазина" вообще воспримет этот запрос, то он "выдаст" надлежащее сообщение об ошибке. В отличие от программ, люди работают не по алгоритмам, а по эвроритмам. Человек может самостоятельно менять правила методов своей работы. Так, продавец магазина при виде аргумента "очень большая сумма", может закрыть магазин и побежать покупать железнодорожный билет. Напомним, что такие ситуации для программ пока еще невозможны.

Различие между вызовом процедуры и пересылкой сообщения состоит в том, что в последнем случае существует определенный получатель и интерпретация (т. е. выбор подходящего метода, запускаемого в ответ на сообщение), которая может быть различной для разных получателей.

Обычно конкретный объект-получатель неизвестен вплоть до выполнения программы, так что определить, какой метод, какого объекта будет вызван, заранее невозможно (конкретный кассир заранее не знает, кто и когда из конкретных покупателей обратится к нему). В таком случае говорят, что имеет место позднее связывание между сообщением (именем процедуры или функции) и фрагментом кода (методом), исполняемым в ответ на сообщение. Эта ситуация противопоставляется раннему связыванию (на этапе компилирования или компоновки программы) имени с фрагментом кода, что происходит при традиционных вызовах процедур.

Фундаментальной концепцией в объектно-ориентированном программировании является понятие *классов.* Все объекты являются представителями, или экземплярами, классов. Например: у вас наверняка есть примерное представление о реакции кассира на запрос о заказе билетов, поскольку вы имеете общую информацию о людях данной профессии (например, кассире кинотеатра) и ожидаете, что он, будучи представителем данной категории, в общих чертах будет соответствовать шаблону. То же самое можно сказать и о представителях других профессий, что позволяет разделить человеческое общество на определенные категории по профессиональному признаку (на классы). Каждая категория в свою очередь делится на представителей этой категории. Таким образом, человеческое общество представляется в виде иерархической структуры с наследованием свойств классов объектов всех категорий. В корне такой классификации может находиться класс "HomoSapience" или даже класс "млекопитающие" (рис. 8.1).

Метод, активизируемый объектом в ответ на сообщение, определяется классом, к которому принадлежит получатель сообщения. Все объекты одного класса используют одни и те же методы в ответ на одинаковые сообщения.

Классы могут быть организованы в иерархическую структуру с *наследованием* свойств.*Класс-потомок* наследует атрибуты *родительского класса,* расположенного ниже в иерархическом дереве (если дерево иерархии наследования растет вверх). *Абстрактный родительский класс* — это класс, не имеющий экземпляров объектов. Он используется только для порождения потомков. Класс "HomoSapience", скорее всего, будет абстрактным, поскольку для практического применения, например работодателю, экземпляры его объектов не интересны.

Рис.1. Понятие создания объекта как экземпляра класса ПТИЦЫ

Итак, пусть абстрактным родительским классом у работодателя будет класс "трудоспособный человек", который включает методы доступа к внутренним данным, а также поля самих внутренних данных: фамилия; имя; отчество; дата рождения; домашний адрес; домашний телефон; сведения об образовании; сведения о трудовом стаже и т. д. От данного класса могут быть унаследованы классы: "кассир", "водитель автомобиля", "музыкант". Класс "кассир" располагает методами работы: общение с клиентом по правилам, получение денег, выдача денег, общение с инкассатором и т. д. От класса "кассир" могут быть унаследованы классы: "кассир, выдающий зарплату", "кассир железнодорожной кассы". Кассир железнодорожной кассы отличается от кассира, выдающего зарплату, дополнительными знаниями и навыками работы.

От класса "кассир железнодорожной кассы" могут быть получены экземпляры объектов: "кассир кассы № 1", "кассир кассы № 2", "кассир кассы № 3" и т. д.

В помещении большого вокзала можно обнаружить множество одинаково оборудованных объектов — касс. Однако среди касс можно выделить различающиеся кассы: суточные, предварительные, воинские, работающие по бронированию билетов и т. д. Для того чтобы начальнику вокзала поменять один вид кассы на другой, нет необходимости перестраивать помещение кассы и менять оборудование. Ему достаточно заменить в кассе кассира с одними навыками на кассира с другими навыками. Кассир вставляет табличку с новой надписью вида кассы — и все. Заметим, что смена функции касс произошла без остановки работы вокзала. Такая замена становится простой именно потому, что все помещения касс имеют одинаковый интерфейс с кассирами и клиентами. Теперь разные объекты, поддерживающие одинаковые интерфейсы, могут выполнять в ответ на запросы разные операции.

Ассоциация запроса с объектом и одной из его операций во время выполнения называется*динамическим связыванием.* Динамическое связывание позволяет во время выполнения подставить вместо одного объекта другой, если он имеет точно такой же интерфейс. Такая взаимозаменяемость называется *полиморфизмом* и является еще одной фундаментальной особенностью объектно-ориентированных систем (рис. 8.2).

Пусть, согласно произведенной классификации, объекты "скрипач с фамилией Петров" и "водитель автомобиля Сидоров" будут экземплярами разных классов. Для того чтобы получить объект "Иванов, являющийся одновременно скрипачом и водителем", необходим особый класс, который может быть получен из классов "скрипач" и "водитель автомобиля"*множественным наследованием* (рис. 8.3). Теперь работодатель, послав особое *сообщение делегирования,* может поручить (делегировать) объекту "Иванов" выполнять функцию либо водителя, либо скрипача. Объект "Иванов", находящийся за рулем автомобиля, не должен начать играть на скрипке. Для этого должен быть реализован механизм самоделегирования полномочий — объект "Иванов", находясь за рулем, запрещает сам себе игру на скрипке. Таким образом, понятие обязанности или ответственности за выполнение действия является фундаментальным в объектно-ориентированном программировании.

Рис. 8.2. Понятие полиморфизма

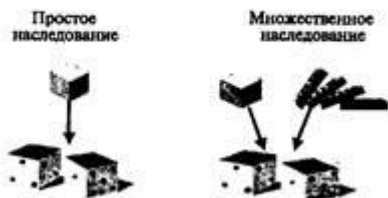


Рис. 8.3. Пример простого и множественного наследования

В системах программирования с отсутствующим множественным наследованием задачи, требующие множественного наследования, всегда могут быть решены композицией (агрегированием) с последующим делегированием полномочий.

Композиция объектов — это реализация составного объекта, состоящего из нескольких совместно работающих объектов и образующих единое целое с новой, более сложной функциональностью.

Агрегированный объект — объект, составленный из подобъектов. Подобъекты называются*частями* агрегата, и агрегат отвечает за них. Например, в системах с множественным наследованием шахматная фигура ферзь может быть унаследована от слона и ладьи. В системах с отсутствующим множественным наследованием можно получить ферзя двумя способами. Согласно первому способу, можно создать класс "любая_фигура" и далее, в периоде выполнения, делегировать полномочия каждому объекту-экземпляру данного класса быть ладьей, слоном, ферзей, пешкой и т. д. По второму способу после получения классов "ладья" и "слон" их можно объединить композицией в класс "ферзь". Теперь объект класса "ферзь" можно использовать как объект "ферзь" или даже как объект "слон", для чего объекту "ферзь" делегируется выполнение полномочий слона. Более того, можно делегировать объекту "ферзь" полномочия стать объектами "король" или даже "пешка"! Для композиции требуется, чтобы объединяемые объекты имели четко определенные интерфейсы. И у наследования, и у композиции есть достоинства и недостатки.

Наследование класса определяется статически на этапе компиляции; его проще использовать, поскольку оно напрямую поддержано языком программирования.

Но у наследования класса есть и минусы. Во-первых, нельзя изменить унаследованную от родителя реализацию во время выполнения программы, поскольку само наследование фиксировано на этапе компиляции. Во-вторых, родительский класс нередко, хотя бы частично, определяет физическое представление своих подклассов. Поскольку подклассу доступны детали реализации родительского класса, то часто говорят, что *наследование нарушает инкапсуляцию.* Реализации подкласса и родительского класса настолько тесно связаны, что любые изменения последней требуют изменять и реализацию подкласса.

Композиция объектов определяется динамически во время выполнения за счет того, что объекты получают ссылки на другие объекты. Композицию можно применить, если объекты соблюдают интерфейсы друг друга. Для этого, в свою очередь, требуется тщательно проектировать интерфейсы, так чтобы один объект можно было использовать вместе с широким спектром других. Но и выигрыш велик, поскольку доступ к объектам

осуществляется только через их интерфейсы, мы не нарушаем инкапсуляцию. Во время выполнения программы любой объект можно заменить другим, лишь бы он имел тот же тип. Более того, поскольку при реализации объекта кодируются прежде всего его интерфейсы, то зависимость от реализации резко снижается.
Композиция объектов влияет на дизайн системы и еще в одном аспекте. Отдавая предпочтение композиции объектов, а не наследованию классов, вы инкапсулируете каждый класс и даете ему возможность выполнять только свою задачу. Классы и их иерархии остаются небольшими, и вероятность их разрастания до неуправляемых размеров невелика. С другой стороны, дизайн, основанный на композиции, будет содержать больше объектов (хотя число классов, возможно, уменьшится), и поведение системы начнет зависеть от их взаимодействия, тогда как при другом подходе оно было бы определено в одном классе.
Это подводит еще к одному правилу объектно-ориентированного проектирования:*предпочитайте композицию наследованию класса.*
В идеале, чтобы добиться повторного использования кода, вообще не следовало бы создавать новые компоненты. Хорошо бы, чтобы можно было получить всю нужную функциональность, просто собирая вместе уже существующие компоненты. На практике, однако, так получается редко, поскольку набор имеющихся компонентов все же недостаточно широк. Повторное использование за счет наследования упрощает создание новых компонентов, которые можно было бы применять со старыми. Поэтому наследование и композиция часто используются вместе.
Тем не менее опыт показывает, что проектировщики злоупотребляют наследованием. Нередко программы могли бы стать проще, если бы их авторы больше полагались на композицию объектов.
С помощью *делегирования* композицию можно сделать столь же мощным инструментом повторного использования, сколь и наследование. При делегировании в процесс обработки запроса вовлечено *два* объекта: получатель поручает выполнение операций другому объекту — *уполномоченному.* Примерно так же подкласс делегирует ответственность своему родительскому классу. Но унаследованная операция всегда может обратиться к объекту-получателю через переменную-член (в C++) или переменную self (в Smalltalk). Чтобы достичь того же эффекта для делегирования, получатель передает указатель на самого себя соответствующему объекту, чтобы при выполнении делегированной операции последний мог обратиться к непосредственному адресату запроса.
Например, вместо того чтобы делать класс Window (окно) подклассом класса Rectangle (прямоугольник) — ведь окно является прямоугольником, — мы можем воспользоваться внутри Window поведением класса Rectangle, поместив в класс Window переменную экземпляра типа Rectangle и делегируя ей операции, специфичные для прямоугольников. Другими словами, окно не *является* прямоугольником, а *содержит* его. Теперь класс Window может явно перенаправлять запросы своему члену Rectangle, а не наследовать его операции.
Главное достоинство делегирования в том, что оно упрощает композицию поведения во время выполнения. При этом способ комбинирования поведения можно изменять. Внутреннюю область окна разрешается сделать круговой во время выполнения простой подставкой вместо экземпляра класса Rectangle экземпляра класса Circle. Предполагается, конечно, что оба эти класса имеют одинаковый тип.
У делегирования есть и недостаток, свойственный и другим подходам, применяемым для повышения гибкости за счет композиции объектов. Заключается он в том, что динамическую, в высокой степени параметризованную программу труднее понять, чем статическую. Есть, конечно, и некоторая потеря машинной производительности, но неэффективность работы проектировщика гораздо более существенна. Делегирование можно считать хорошим выбором только тогда, когда оно позволяет достичь упрощения,

а не усложнения. Нелегко сформулировать правила, ясно говорящие, когда следует пользоваться делегированием, поскольку эффективность его зависит от контекста и личного опыта программиста.

Таким образом, можно выделить следующие фундаментальные характеристики объектно-ориентированного мышления:

Характеристика 1. Любой предмет или явление могут рассматриваться как объект.

Характеристика 2. Объект может размещать в своей памяти (в полях) личную информацию, независимую от других объектов. Рекомендуется использовать инкапсулированный (через особые методы) доступ к информации полей.

Характеристика 3. Объекты могут иметь открытые по интерфейсу методы обработки сообщений. Сами сообщения вызовов методов посылаются другими объектами, но для осуществления разумного интерфейса между объектами некоторые методы могут быть скрыты.

Характеристика 4. Вычисления осуществляются путем взаимодействия (обмена данными) между объектами, при котором один объект требует, чтобы другой объект выполнил некоторое действие (метод). Объекты взаимодействуют, посылая и получая сообщения. Сообщение — это запрос на выполнение действия, дополненный набором аргументов, которые могут понадобиться при выполнении действия. Объект — получатель сообщения — обрабатывает сообщения своими внутренними методами.

Характеристика 5. Каждый объект является представителем класса, который выражает общие свойства объектов данного класса в виде одинаковых списков набора данных (полей) в своей памяти и внутренних методов, обрабатывающих сообщения. В классе методы задают поведение объекта. Тем самым все объекты, которые являются экземплярами одного класса, могут выполнять одни и те же действия.

Характеристика 6. Классы организованы в единую квазидревовидную структуру с общим корнем, которая называется иерархией наследования. Обычно корень иерархии направлен вверх. При множественном наследовании ветви могут срастаться, образуя сеть наследования. Память и поведение, связанные с экземплярами определенного класса, автоматически являются доступными любому классу, расположенному ниже в иерархическом дереве.

Характеристика 7. Благодаря полиморфизму — способности подставлять во время выполнения вместо одного объекта другой, с совместимым интерфейсом, в периоде выполнения одни и те же объекты могут разными методами исполнять одни и те же запросы сообщений.

Характеристика 8. Композиция является предпочтительной альтернативой множественному наследованию и позволяет изменять состав объектов агрегата в процессе выполнения программы.

Характеристика 9. Структура объектно-ориентированной программы на этапе выполнения часто имеет мало общего со структурой ее исходного кода. Последняя фиксируется на этапе компиляции. Ее код состоит из классов, отношения наследования между которыми неизменны. На этапе же выполнения структура программы — быстро изменяющаяся сеть из взаимодействующих объектов. Две эти структуры почти независимы.

Лекция 2. Программирование математических структур (матрицы и конечные графы)

Определение. Если на плоскости задать конечное множество V точек и конечный набор линий X, соединяющих некоторые пары из точек V, то полученная совокупность точек и линий будет называться **графом**.
При этом элементы множества V называются **вершинами** графа, а элементы множества X – **ребрами**.
В множестве V могут встречаться одинаковые элементы, ребра, соединяющие одинаковые элементы называются **петлями**. Одинаковые пары в множестве X называются **кратными** (или параллельными) ребрами. Количество одинаковых пар (v, w) в X называется **кратностью** ребра (v, w).
Множество V и набор X определяют граф с кратными ребрами – **псевдограф**.
$G = (V, X)$
Псевдограф без петель называется **мультиграфом**.
Если в наборе X ни одна пара не встречается более одного раза, то мультиграф называется **графом**.
Если пары в наборе X являются упорядочными, то граф называется ориентированным или **орграфом**.
Графу соответствует геометрическая конфигурация. Вершины обозначаются точками (кружочками), а ребра – линиями, соединяющими соответствующие вершины.
Определение. Если $x = \{v, w\}$ – ребро графа, то вершины v, w называются концами ребра x.
Если $x = (v, w)$ – дуга орграфа, то вершина v – начало, а вершина w – конец дуги x.
Определение. Вершины v, w графа G = (V, X) называются **смежными**, если $\{v,w\}$ÎX. Два ребра называются **смежными**, если они имеют общюю вершину.
Определение. **Степенью** вершины графа называется число ребер, которым эта вершина принадлежит. Вершина называется**изолированной**, если если ее степень равна единице и **висячей**, если ее степень равна нулю.
Определение. Графы $G_1(V_1, X_1)$ и $G_2(V_2, X_2)$ называются **изоморфмными**, если существует взаимно однозначное отображение j: V_1® V_2, сохраняющее смежность.
Определение. **Маршрутом (путем)** для графа G(V, X) называется последовательность $v_1x_1v_2x_2v_3\ldots x_kv_{k+1}$. Маршрут называется**замкнутым**, если его начальная и конечная точки совпадают. Число ребер (дуг) маршрута (пути) графа называется **длиной** маршрута (пути).
Определение. Незамкнутый маршрут (путь) называется **цепью**. Цепь, в которой все вершины попарно различны, называется **простой цепью**.
Определение. Замкнутый маршрут (путь) называется **циклом (контуром)**. Цикл, в котором все вершины попарно различны, называется **простым циклом**.
Матрицы графов.
Пусть D = (V, X) – орграф, где $V = \{v_1, \ldots, v_n\}$, $X = \{x_1, \ldots , x_m\}$.
Определение. **Матрицей смежности** орграфа D называется квадратичная матрица A(D) = $[a_{ij}]$ порядка n, у которой

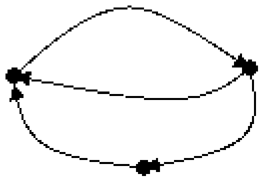
$$a_{ij} = \begin{cases} 1, & если\ (v_i, v_j) \in X \\ 0, & если\ (v_i, v_j) \notin X \end{cases}$$

Определение. Если вершина v является крнцом ребра x, то говорят, что v и x – **инциндентны**.
Определение. **Матрицей инциндентности** оргафа D называется матрица размерности n´m B(D) = $[b_{ij}]$, у которой

$$b_{ij} = \begin{cases} 1, & \text{если вершина } v_i \text{ является концом дуги } x_j \\ -1, & \text{если вершина } v_i \text{ является началом дуги } x_j \\ 0,, & \text{если вершина } v_i \text{ не инцидентна дуге } x_j \end{cases}$$

Пример. Записать матрицы смежности и инцидентности для графа, изображенного на рисунке.

x_1



v_1 x_4 v_2

x_2

x_3

v_3

Составим матрицу смежности:

	v_1	v_2	v_3
v_1	0	1	0
v_2	1	0	1
v_3	1	0	0

Т.е. $A(D) = \begin{pmatrix} 0 & 1 & 0 \\ 1 & 0 & 1 \\ 1 & 0 & 0 \end{pmatrix}$ — матрица смежности.

Матрица инциндентности:

	x_1	x_2	x_3	x_4
v_1	-1	0	1	1
v_2	1	-1	0	-1
v_3	0	1	-1	0

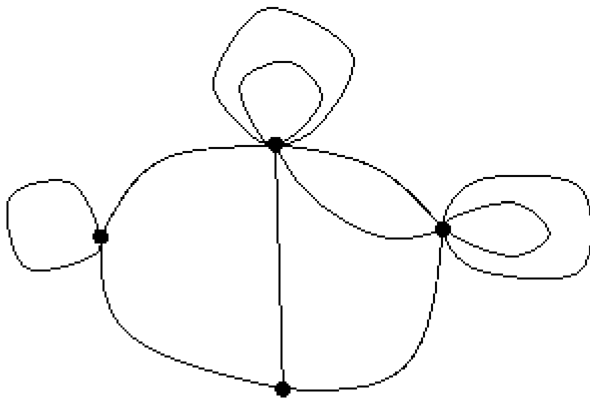
Т.е. $B(D) = \begin{pmatrix} -1 & 0 & 1 & 1 \\ 1 & -1 & 0 & -1 \\ 0 & 1 & -1 & 0 \end{pmatrix}$

Если граф имеет кратные дуги (ребра), то в матрице смежности принимается $a_{ij}=k$, где k – кратность дуги (ребра).

С помощью матриц смежности и инциндентности всегда можно полностью определеить граф и все его компоненты. Такой метод задания графов очень удобен для обработки данных на ЭВМ.

Пример. Задана симметрическая матрица Q неотрицательных чисел. Нарисовать на плоскости граф G(V, X), имеющий заданную матицу Q своей матрицей смежности. Найти матрицу инциндентности R графа G. Нарисованть также орграф $\vec{G}(N, A)$, имеющий матрицу смежности Q, определить его матрицу инциндентности C.

$$Q = \begin{pmatrix} 1 & 1 & 0 & 1 \\ 1 & 2 & 2 & 1 \\ 0 & 2 & 2 & 1 \\ 1 & 1 & 1 & 0 \end{pmatrix}$$



x_4

x_3

v_2

x_2 x_5

x_6

x_1 v_1 v_3 x_7 x_8

x_{10}

x_{11} x_9

v_4

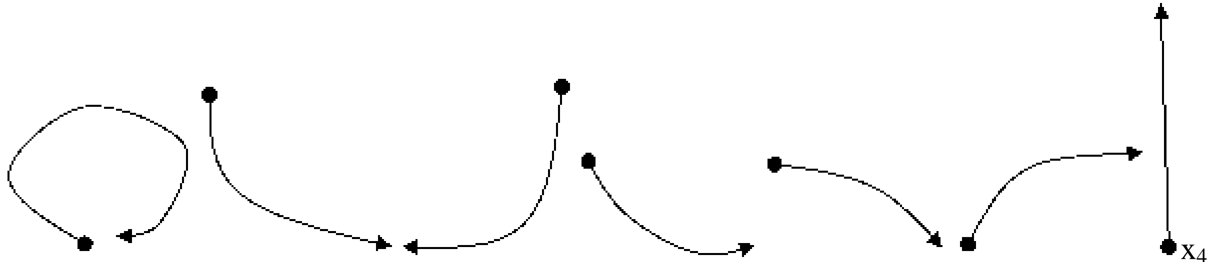
Составим матрицу инциндентности:

	x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8	x_9	x_{10}	x_{11}
v_1	1	1	0	0	0	0	0	0	0	0	1
v_2	0	1	1	1	1	1	0	0	0	1	0
v_3	0	0	0	0	1	1	1	1	1	0	0
v_4	0	0	0	0	0	0	0	0	1	1	1

Итого:

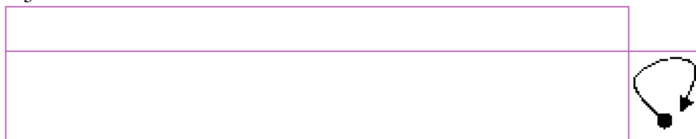
$$R = \begin{pmatrix} 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 \end{pmatrix}$$

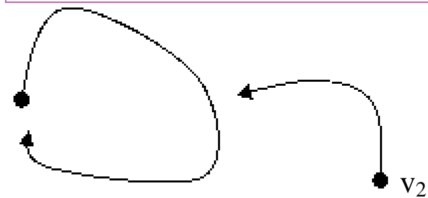
Построим теперь ориентированный граф с заданной матрицей смежности.



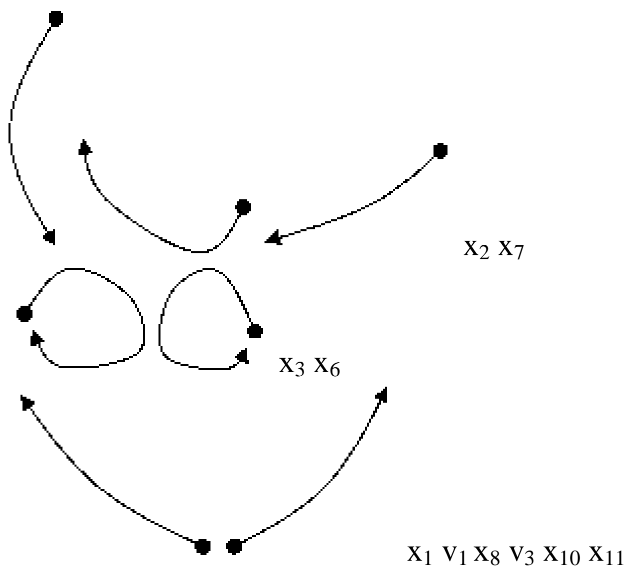
x_4

x_5





v_2



x_2 x_7

x_3 x_6

x_1 v_1 x_8 v_3 x_{10} x_{11}

x_9

x_{17} x_{15} x_{14}

x_{16} x_{13} x_{12}

v_4

Составим матрицу инциндентности для ориетированного графа.

Элемент матрицы равен 1, если точка является концом дуги, -1 – если началом дуги, если дуга является петлей, элемент матрицы запишем как ±1.

$$C = \begin{pmatrix} \pm1 & -1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & 1 \\ 0 & 0 & 0 & \pm1 & \pm1 & -1 & 1 & -1 & 1 & 0 & 0 & 0 & 0 & 1 & -1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & -1 & 1 & -1 & \pm1 & \pm1 & -1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & -1 & -1 & 1 & 1 & -1 \end{pmatrix}$$

Таким образом, операции с графами можно свести к операциям с их матрицами.

Достижимость и связность.

Определение. Вершина *w* графа D (или орграфа) называется **достижимой** из вершины *v*, если либо *w*=*v*, либо существует путь из *v* в*w*(маршрут, соединяющий *v* и *w*).

Определение. Граф (орграф) называется **связным**, если для любых двух его вершин существует маршрут (путь), который их связывает. Орграф называется **односторонне связным**, если если для любых двух его вершин по крайней мере одна достижима из другой.

Определение. Псевдографом D(V, X), **ассоциированным** с ориентированным псевдографом, называется псевдограф G(V, X_0) в котором X_0 получается из X заменой всех упорядоченных пар (*v, w*) на неупорядоченные пары (*v, w*).

Определение. Орграф называется **слабо связным**, если связным является ассоциированный с ним псевдограф.

Эйлеровы и гамильтоновы графы.

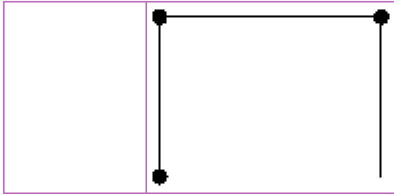
Определение. Цепь (цикл) в псевдографе G называется **эйлеровым**, если она проходит по одному разу через каждое ребро псевдографа G.

Теорема. *Для того, чтобы связный псевдограф G обладал эйлеровым циклом, необходимо и достаточно, чтобы степени его вершин были четными.*

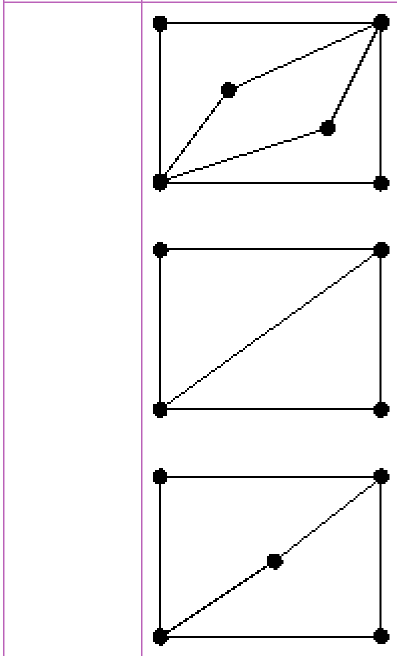
Теорема. *Для того, чтобы связный псевдограф G обладал эйлеровой цепью, необходимо и достаточно, чтобы он имел ровно две вершины нечетной степени.*

Определение. Цикл (цепь) в псевдографе G называется **гамильтоновым**, если он проходит через каждую вершину псевдографа G ровно один раз.

Пример.



- в графе есть и эйлеровый и гамильтонов циклы



- в графе есть эйлеров цикл, но нет гамильтонова
- в графе есть гамильтонов, но нет эйлерова цикла
- в графе нет ни эйлерова, ни гамильтонова цикла

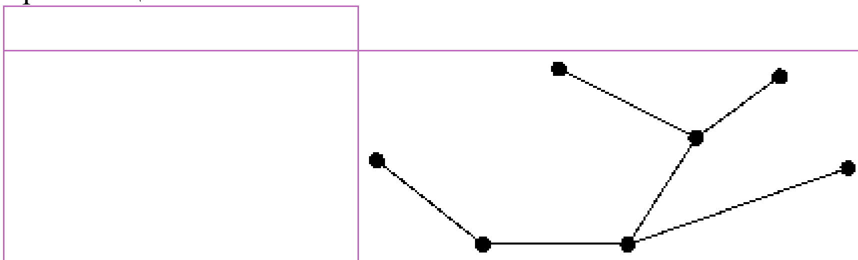
Граф G называется **полным**, если если каждая его вершина смежна со всеми остальными вершинами. В полном графе всегда существуют гамильтоновы цмклы.

Также необходимым условием существования гамильтонова цикла явояется связность графа.

Деревья и циклы.

Определение. Граф G называется **деревом**, если он является связным и не имеет циклов. Граф G, все компоненты связности которого являются деревьями, называется **лесом**.

У графа, который является деревом, число ребер на единицу меньше числа вершин. Дерево не содержит циклов, любые две его вершины можно соеденить единственной простой цепью.



Если у дерева G есть, по крайней мере, одно ребро, то у него обязательно найдется висячая вершина, т.к. в противном случае в графе будет цикл.

Для графов, которые сами по себе не являются деревьями, вводится понятие остовного дерева.

<u>Определение.</u> Остовным деревом связного графа G называется любой его подграф, содержащий все вершины графа G и являющийся деревом.

Пусть G – связный граф. Тогда остовное дерево графа G (если оно существует) должно содержать n(G)-1 ребер.

Таким образом, любое остовное дерево графа G есть результат удаления из графа G ровно m(G) — (n(G) — 1) = m(G) – n(G) + 1 ребер.

Число ν(G) = m(G) – n(G) + 1 называется **цикломатическим числом** связного графа G.

Одной из самых распространенных задач является задача построения остовного дерева минимальной длины графа. Для решения этой задачи применяется следующий алгоритм.

1) Выберем в графе G ребро минимальной длины. Вместе с инциндентными ему вершинами оно образует подграф G_2.

2) Строим граф G_3, добавляя к графу G_2 новое ребро минимальной длины, выбранное среди ребер графа G, каждое из которых инциндентно какой либо вершине графа G_2, и одновременно инциндентно какой – либо вершине графа G, не содержащейся в графе G_2.

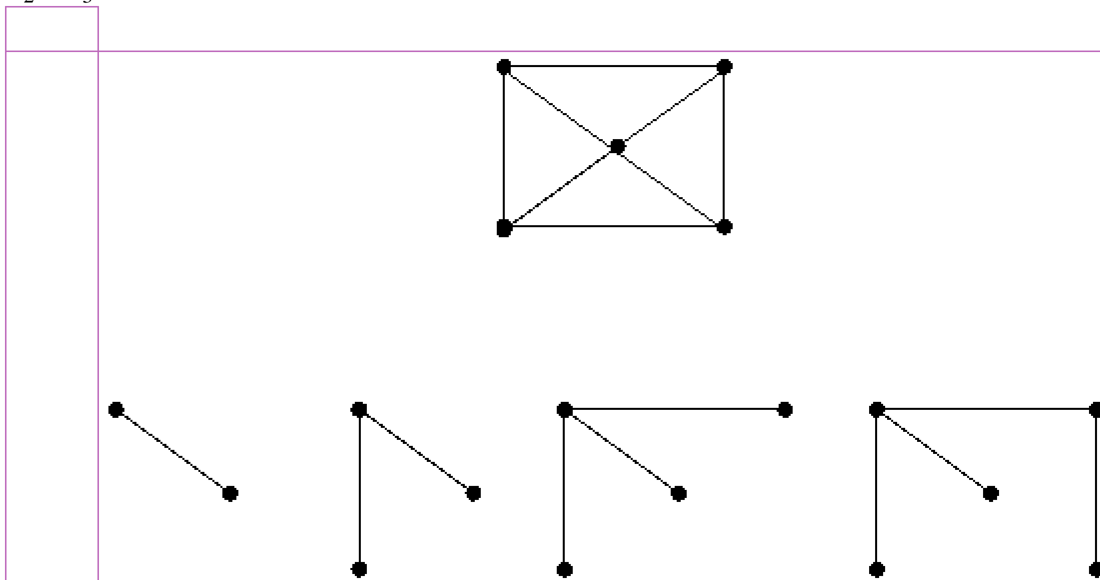
3) Строим графы G_4, G_5, …, G_n, повторяя действия пункта 2 до тех пор, пока не переберем все вершины графа G.

<u>Пример.</u> Определить минимальное остовное дерево нагруженного графа.

Граф называется **нагруженным**, если на множестве его дуг задана некоторая функция, которая называется **весовой** функцией, и определяет длину дуги.

В нашем примере – весовая функция определяет длины дуг числами 1, 2, 3, 4, 5.

v_2 2 v_3



1 4

1 v_5 3

5 3

v_1 4 v_4

2 2

1 1 1 1 1 1 3

G_2 G_3 G_4 G_5

На четвертом шаге алгоритма получили дерево G_5, которое соединяет все вершины исходного графа. Таким образом, дерево G_5 , будет минимальным остовным деревом графа G.

Лекция 3. Методические и инструментальные средства разработки модульного программного обеспечения АСУ

В последние годы во многих отечественных промышленных компаниях все чаще применяются информационные системы (ИС) различного назначения (для управления производством, для электронного документооборота и т.д.). Известно, что магистральным путем развития ИС для решения задач управления современной промышленной компанией является создание комплексной информационной системы управления (часто говорят об интегрированной системе управления или корпоративной информационной системе – КИС). КИС должна обеспечивать поддержку распределенного многоуровневого управления, в том числе крупномасштабной территориально распределенной компанией. Более того, внедренная в компании КИС – это основа для создания и развития единого информационного пространства (ЕИП) компании [1]. Под ЕИП будем понимать совокупность аппаратно-программных средств и всех хранимых данных, функционирующих на основе единых технических, организационных и методических требований и принципов и оперативно обеспечивающих информационными ресурсами специалистов и менеджмент компании независимо от их местонахождения.
В статье рассматривается проблема разработки инструментальных средств для создания ЕИП современных промышленных компаний, имеющих различные ИС и весьма высокий уровень автоматизации производства.
Формирование ЕИП компании. Сегодня для многих отечественных промышленных компаний характерна «лоскутная» автоматизация, т.е. автоматизация отдельных производственных участков и отдельных бизнес-процессов. Именно в таких компаниях важно создавать КИС и реализовывать концепцию ЕИП.
Еще на первом этапе создания ЕИП компании возникает проблема разработки и внедрения во всех цехах и службах единой системы нормативно-справочной информации. Сотрудники компании, работая с разными ИС в рамках ЕИП, должны пользоваться актуальными едиными классификаторами, нормативами и справочниками.
При создании ЕИП необходимо решать задачу организации пользовательского интерфейса. Для сотрудника, работающего в единой информационной среде, важно, чтобы взаимодействие систем было «бесшовным», чтобы пользовательский интерфейс был удобным, чтобы был реализован принцип единой точки доступа в ЕИП. Решение этой задачи может быть обеспечено применением двух методов формирования пользовательского интерфейса. Первый: для пользователя основной для него ИС при работе из нее с данными и (или) функциями другой системы проектируется интерфейс, представляющий собой модификацию интерфейса основной системы. Второй метод: обеспечение вебпортального доступа к данным других внедренных в компании ИС; доступ ведется через интранет-портал компании, при этом легко реализуется принцип единой точки доступа пользователя в ЕИП. К сожалению, интернет-портал не может быть единственным центром доступа ко всем потокам данных и ко всем ИС в рамках ЕИП компании. На практике часто используют комбинированный подход, когда применяются оба метода доступа пользователей в ЕИП компании.
При создании ЕИП компании, имеющей несколько ИС, можно использовать различные способы интеграции ИС: интеграция по данным, интеграция по функциональным сервисам (функциям), интеграция по интерфейсам и комбинированный способ (при объединении двух систем производится интеграция и по данным, и по функциям). Наибольшее распространение получил способ интеграции ИС по данным [2]. Важной проблемой остается разработки инструментальных средств такой интеграции.

Сам процесс интеграции различных ИС по производственным данным также сопряжен с рядом проблем, которые в значительной мере усложняют решение задачи обеспечения сотрудников компании информационными ресурсами.

Во-первых, сложность взаимодействия ИС возникает из-за территориальной распределенности производств и офисов компании, а также из-за разнородности протоколов передачи данных. По мере развития производств, оснащения их разнородными ИС возникает сложная схема сбора и обработки производственных данных.

Во-вторых, в каждой компании формируется свой собственный набор ИС, представляющих собой смесь из:

монолитных и клиент-серверных приложений;

процедурных, объектно-ориентированных и компонентных программных решений;

программ, написанных на различных языках программирования;

ИС, построенных на платформах различных систем управления базами данных (СУБД).

В-третьих, для современной промышленной компании характерны большие объемы технологических данных (данные с датчиков технологического оборудования, результаты химических анализов сырья и продукции, результаты диагностики состояния технологического оборудования и т.п.), что усложняет обмен ими в рамках создаваемого ЕИП компании.

Существуют три базовых метода интеграции ИС по данным: консолидация, федерализация и распространение [2]. При использовании *метода консолидации* данные собираются из нескольких первичных систем и интегрируются в одно постоянное место хранения. Такое место хранения может быть использовано для подготовки отчетности и проведения анализа, как в случае хранилища данных, или как источник данных для других приложений.

Метод федерализации данных обеспечивает единую виртуальную картину нескольких первичных источников данных. Для получения сведений о некотором процессе, обрабатываемом в нескольких оперативных приложениях, процессор федерализации данных извлекает сведения из соответствующих первичных складов данных, интегрирует их таким образом, чтобы они отвечали виртуальной картине и требованиям запроса, и отправляет результаты в то бизнес-приложение, из которого пришел запрос.

Метод распространения данных подразумевает их копирование из одного места в другое. Этот метод обычно используется для операций реального времени и является событийно управляемым.

Все перечисленные методы имеют свои преимущества и недостатки. Каждый из них является наиболее эффективным в определенных условиях. Например, федерализацию данных рекомендуется использовать в тех случаях, когда стоимость реализации метода консолидации данных перевешивает бизнес-преимущества, которые он предоставляет.

Решая проблему создания ЕИП компаний путем интеграции ИС по данным, ИТ-отделы нередко выбирают наиболее очевидный путь, создавая частные интеграционные решения и налаживая тем самым обмен данными между двумя и более конкретными ИС. Нельзя отрицать очевидную эффективность такого подхода благодаря как высокой производительности обмена данными (за счет использования «родных» для ИС механизмов обмена), так и невысокой стоимости такого интеграционного решения. Тем не менее очевидны недостатки этого подхода, связанные прежде всего с отсутствием требуемой гибкости и масштабируемости решений. В результате экономия на разработке может нивелироваться существенными затратами на поддержку работоспособности частных интеграционных решений в условиях постоянно меняющейся информационной среды компании.

Существуют более общие подходы к решению задачи интеграции ИС по данным на уровне компании, например использование *интеграционных платформ*. Сегодня на рынке представлены универсальные интеграционные платформы Microsoft BizTalk [3], IBM

WebSphere [4] и др. Их отличительной особенностью является инвариантность к предметной области, позволяющая строить интеграционные решения для различных бизнес-отраслей. Однако универсальность оборачивается неэффективностью в силу избыточности функционала и сложности архитектуры. Понимая это, многие производители программного обеспечения (ПО) для решения задач интеграции ИС по данным создают отраслевые шаблоны интеграционных решений, основанные на универсальных платформах. Однако такие шаблоны сегодня имеются лишь для ограниченного числа отраслей и классов интегрируемых систем, в основном для непроизводственной сферы и для финансово-экономических ИС.

На наш взгляд, разумной альтернативой этим подходам в случае промышленных компаний является создание *отраслевой* инструментальной системы (совокупности отраслевых систем) для интеграции ИС компаний отрасли (отраслей) по производственным данным. Такая инструментальная система интеграции производственных данных (СИПД) должна сочетать мощь и гибкость универсальных платформ с высокой производительностью и предметной ориентированностью частных интеграционных решений и учитывать специфику той или иной отрасли. Кроме того, СИПД должна быть легко адаптируемой для решения задач интеграции ИС конкретных компаний отрасли.

Принципы построения и обобщенная архитектура инструментальной СИПД. На основе анализа направлений деятельности современных промышленных компаний и используемых в них ИС для решения производственных задач можно сформулировать следующие общие принципы построения инструментальной СИПД для компаний той или иной отрасли.

1. Принцип *унифицированного федеративного доступа* к данным, согласно которому участник (любая ИС), являющийся потребителем или источником данных, должен взаимодействовать с инструментальной СИПД, выступающей в роли сервера федерализации данных. Участник, являющийся потребителем данных, должен взаимодействовать с единым виртуальным источником данных, иначе ему придется взаимодействовать с несколькими источниками данных индивидуально через различные интерфейсы, с использованием разных протоколов. Благодаря федерализации данных участник-потребитель имеет дело с единообразным интерфейсом, и ему нет необходимости знать, где эти данные хранятся и какой язык программирования или интерфейс поддерживается участником-источником. Участнику-потребителю также не обязательно знать о физических условиях хранения данных или используемых сетевых протоколах (сетевая прозрачность).

2. Принцип *интеграции данных* компании на основе интеграционной модели производственных данных (ИМПД). Каждый участник по-своему интерпретирует предметную область (отрасль), в связи с этим задача сопоставления семантически подобных сущностей из схем данных участников при их интеграции оказывается достаточно сложной. Наличие ИМПД, в определенной степени общей для всех участников, значительно облегчает эту задачу. Использование ИМПД предметной области (отрасли) позволяет сделать процесс обмена данными простым и прозрачным: этот подход гарантирует, что все сообщения между интегрируемыми участниками будут правильно поняты и интерпретированы.

3. Принцип *адаптивности инструментальной СИПД*, в соответствии с которым особенности конкретной компании должны легко учитываться при создании конкретной СИПД путем развития, масштабирования и адаптации отраслевой инструментальной СИПД.

4. Принцип *раздельного описания логики* процессов передачи данных и *параметров доступа* к данным конкретных интегрируемых ИС. Согласно ему для каждого участника необходим адаптер, предоставляющий конкретной инструментальной СИПД интерфейс к

его данным в виде, соответствующем ИМПД. Адаптер должен преобразовывать информацию из схемы данных участника в схему данных ИМПД.
5. Принцип *сервисной ориентированности архитектуры* (Service Oriented Architecture, SOA) инструментальной системы. Согласно ему транспортный уровень движения данных интегрируемых участников должен быть организован посредством веб-сервисов, а передача данных – реализована в формате XML. Следование этому принципу позволяет обращаться не только к данным, хранящимся в базах, но и к данным в коммерческих и заказных приложениях, веб-контенте, документах, рисунках и пр. [5].
6. Принцип *информационной безопасности*, согласно которому инструментальная система должна обладать развитыми механизмами защиты передаваемых данных, а также уметь адаптироваться к различным архитектурам безопасности интегрируемых участников и требованиям к информационной безопасности конкретных компаний.
На основе изложенных принципов можно предложить обобщенную архитектуру инструментальной СИПД, схема которой представлена на рисунке 1.

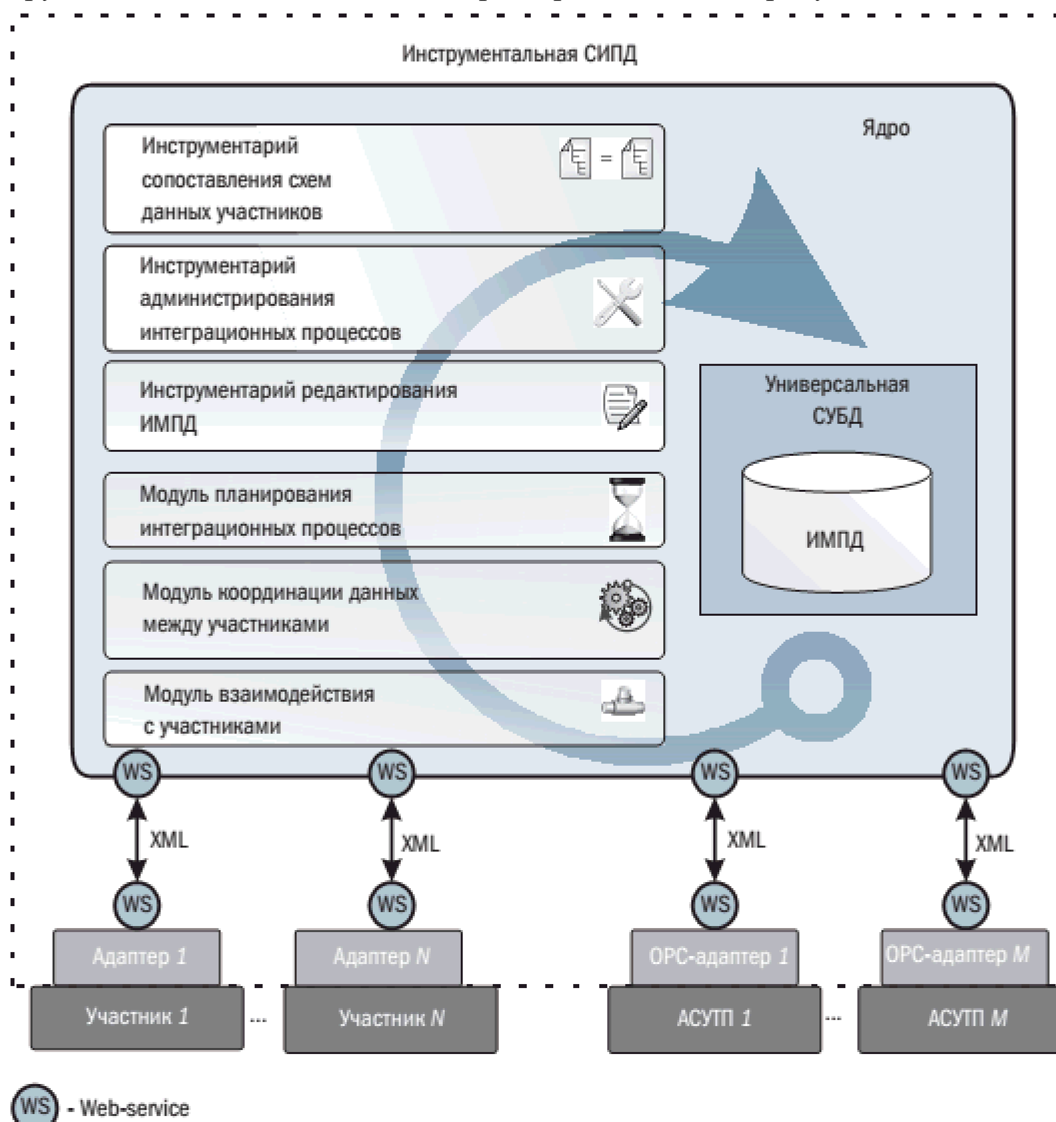


Рис. 1. Схема обобщенной архитектуры инструментальной СИПД
Основными компонентами инструментальной системы являются ядро и адаптеры, обеспечивающие механизм унифицированного взаимодействия интегрируемых участников с ядром СИПД. Согласно принципу 4 для каждого участника интеграции необходим адаптер, предоставляющий СИПД интерфейс к данным этого участника в виде, описанном в ИМПД.

Большим классом производственных данных в промышленной компании являются технологические данные, содержащиеся в автоматизированных системах управления технологическими процессами (АСУ ТП). Данные с АСУ ТП передаются на верхние уровни управления компанией. На рисунке 1 изображено М таких АСУ ТП. Так как технологические данные имеют свои особенности по сравнению с остальными производственными данными, для их сбора и передачи должен быть разработан отдельный OPC-адаптер, желательно универсальный для всех АСУ ТП, поддерживающих стандарт OPC.

Ядро инструментальной СИПД выполняет следующие основные функции:

обеспечивает хранение ИМПД в базе данных под управлением универсальной СУБД и предоставляет средства для ее модификации;

предоставляет инструментарий для создания, настройки, исполнения и контроля всех интеграционных процессов (настройка интеграционного процесса включает в себя сопоставление объектов интеграции и их атрибутов участников на основе ИМПД);

предоставляет возможность создания правил преобразования данных при передаче их в ту или иную ИС, приведения к общим единицам измерения;

обеспечивает обмен данными в разных режимах (по расписанию, по запросу пользователя, по изменению данных).

Процесс реализации на основе отраслевой инструментальной СИПД системы интеграции производственных данных конкретной промышленной компании будем называть *формированием конкретной СИПД,* которое включает:

настройку при помощи инструментальных средств структур данных такой системы, включая наполнение ИМПД классами производственных объектов, регистрацию объектов компании и т.д.;

подключение новых адаптеров, позволяющих задействовать конкретные производственные ИС;

адаптацию, включающую доработку ПО инструментальной СИПД в случае, если настройка только при помощи инструментальных средств системы невозможна.

Интеграционная модель производственных данных. ИМПД является важным элементом инструментальной СИПД. Цель ее создания – отразить структуру объектов конкретной компании, сформировать единый реестр производственных объектов, которые необходимо задействовать в процессе интеграции всех ИС компании по данным. Модель позволяет сделать процесс обмена данными между ИС компании простым и прозрачным и гарантирует, что все сообщения между ИС будут правильно поняты и интерпретированы. Наличие ИМПД значительно сокращает количество адаптеров, интегрирующих ИС, и позволяет вместо связи друг с другом через «дорогие» парные адаптеры обеспечить связь через общую модель данных. В каждой ИС существует своя интерпретация предметной области, в связи с этим задача сопоставления сущностей разных ИС при их интеграции является достаточно сложной.

ИМПД также должна обладать возможностью неограниченного расширения множества сущностей и атрибутов, описывающих предметную область компании. Одним из способов, позволяющих реализовать это требование, является использование предметно-ориентированных метаданных. Метаданные обеспечивают необходимую гибкость ИМПД, позволяющую легко адаптировать отраслевую инструментальную СИПД к потребностям любой компании.

Для создания базовой ИМПД был использован подход, называемый «сущность–атрибут–значение» (англ. EAV, Entity–Attribute–Value model), базирующийся на принципе метаописания сущностей, их атрибутов и значений. Этот подход обеспечивает:

во-первых, гибкость модели. Представим, что необходимо постоянно добавлять новые атрибуты к уже существующим, для этого придется каждый раз изменять структуру базы

данных. В случае EAV-модели схема базы данных при изменении модели не меняется. Это означает, что EAV-модель более динамична по сравнению с традиционной реляционной моделью. Отсутствуют ограничения на количество атрибутов сущностей, число параметров может расти в процессе развития модели без изменения схемы базы данных;

во-вторых, эффективность хранения. EAV-модель содержит набор сущностей (entities) и набор атрибутов, связанных с этими сущностями. Некоторые и даже большинство значений атрибутов могут отсутствовать. Можно представить сущность, содержащую до сотни и даже тысячи атрибутов. Если хранить запись экземпляра (instance) сущности в таблице реляционной базы данных, то таблица должна содержать сотни столбцов, причем значения большинства из них могут отсутствовать, что является неэффективным способом хранения данных.

Программное обеспечение инструментальной СИПД и её апробация. На основе сформулированных выше принципов создания инструментальной СИПД, а также предложенной ее обобщенной архитектуры была разработана структура базового ПО этой системы (рис. 2).

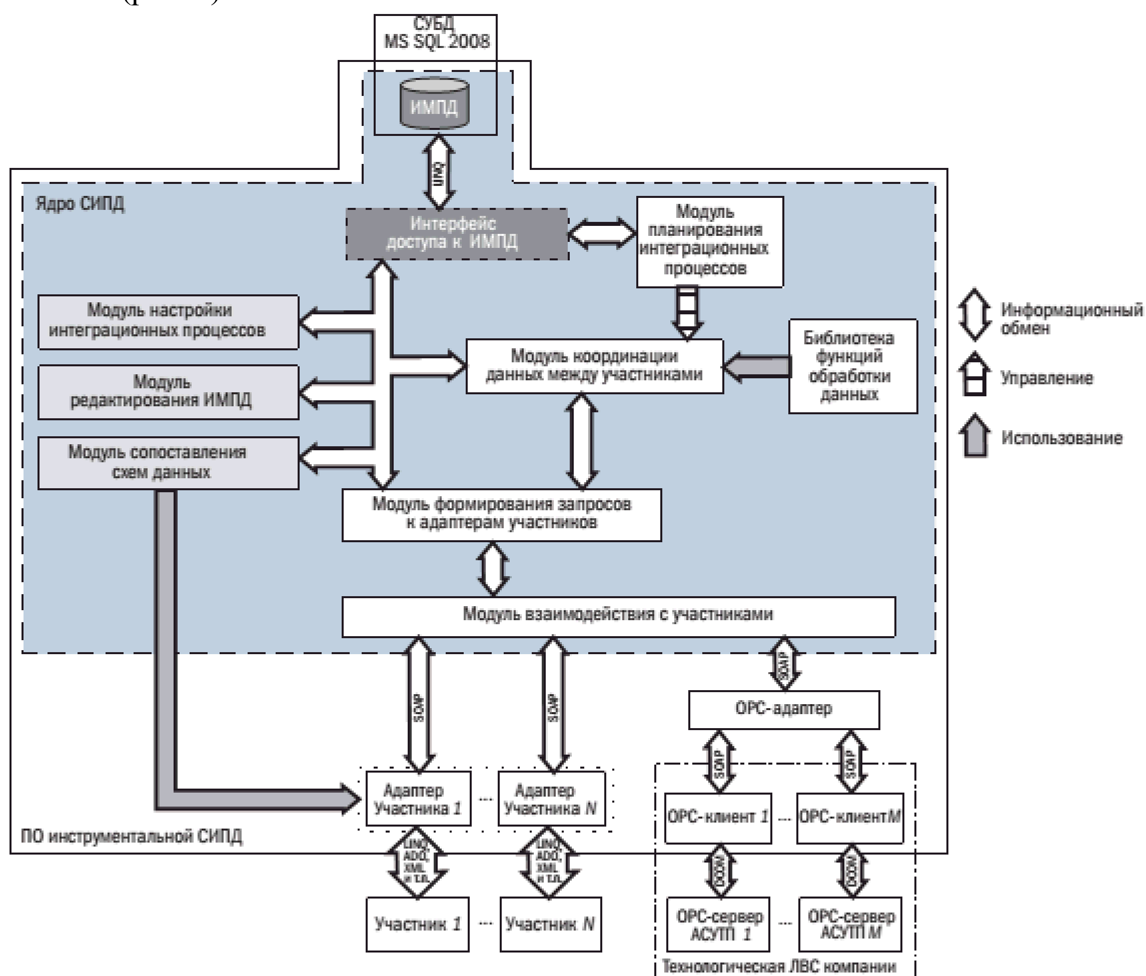


Рис. 2. Схема структуры базового ПО инструментальной СИПД

Интеграцию ИС по технологическим данным, имеющим специфику, следует выделить как отдельную и очень важную задачу. OPC-адаптер, обеспечивающий доставку технологических данных в ядро инструментальной СИПД, должен быть единым для всех АСУ ТП. С этой целью нами использовался стандарт OPC (OLE for Process Control) для сбора и передачи технологических данных. OPC является технологией, обеспечивающей универсальный механизм обмена данными, который устанавливает требования к классам объектов доступа к данным и их специализированным интерфейсам. Во многих случаях в соответствии со стандартами информационной безопасности локальные вычислительные

сети (ЛВС) компании разделены на технологические и офисные. Технологическая сеть объединяет датчики, контроллеры и другое оборудование АСУ ТП, с помощью которого производится сбор технологических данных. Офисная сеть обеспечивает пользователям доступ к данным ИС класса ERP и т.п. Доступ к технологическим данным из офисной ЛВС должен осуществляться через OPC-сервер, расположенный в пределах технологической сети.

С учетом особенностей стандарта OPC и необходимости обеспечения информационной безопасности OPC-серверов технологических сетей нами предложен новый способ сбора и передачи технологических данных [6]. Именно он позволяет использовать единый OPC-адаптер инструментальной СИПД для различных АСУ ТП.

Ключевой особенностью такого способа, обеспечивающей в первую очередь информационную безопасность передаваемых данных, является использование сервисноориентированной и двухзвенной архитектуры разрабатываемого на его основе программного продукта. Д*вухзвенная архитектура* программного средства, названного OPC-клиент, предполагает, что передающая его часть находится в технологической ЛВС, т.е. вместе с OPC-сервером. Сбор технологических данных с OPC-сервера в технологической сети производится по спецификации OPC DA, так как она поддерживается большинством OPC-серверов АСУ ТП, используемых во многих компаниях. Другая часть OPC-клиента (вебсервис, выполняющий функцию приемника данных в XML-формате) и OPC-адаптер, отвечающий за доставку данных в ядро СИПД, находятся в офисной ЛВС. Таким образом, принимающий данные вебсервис представляет собой «пассивный» интерфейс, т.е. все действия инициируются передающей частью, расположенной в технологической сети. Это способствует усилению информационной безопасности, т.к. исключает доступ в технологическую сеть.

На основе базового ПО инструментальной СИПД была разработана отраслевая инструментальная СИПД для компаний нефтегазовой отрасли. ИМПД такой отраслевой системы основывается на стандарте моделей данных PRODML, являющемся наиболее зрелым стандартом для нефтегазовой отрасли [5]. Схема данных в рамках этого стандарта описывает практически все производственные данные, используемые в бизнес-процессах нефте(газо)добывающих компаний. Особенности этой отраслевой СИПД описаны в [6].

Апробация такой инструментальной СИПД осуществлялась путем адаптации ее к специфике компаний ОАО «Востокгазпром» и ОАО «Томскгазпром» и создания с помощью полученных конкретных инструментальных систем ЕИП каждой из компаний. Отметим, что эти компании имеют довольно большое число подлежащих интеграции ИС: ERP – системы MS Axapta 3.0 и MS Ax2009, играющие в компаниях роль КИС, ИС оперативного управления производством добычи и подготовки углеводородного сырья, специализированные ИС обработки и интерпретации геолого-геофизических данных и т.п. Для каждого участника интеграции был разработан адаптер, с помощью которого он подключался к ядру СИПД. Для доставки с промыслов технологических данных с АСУ ТП, использующих SCADA (RS3, ROC, DeltaV, Simatic и т.д.) различных производителей, разработаны OPC-клиенты и единый OPC-адаптер. Благодаря этому ежедневно обеспечивается доставка на верхние уровни управления более 12 тысяч значений технологических параметров каждой компании.

Лекция 4. Компиляция и редактирование связей

Редактор связей[1] выполняет две функции. Во-первых, как можно заключить по его названию, он комбинирует (компонует, редактирует) различные объектные файлы. Вторая его функция — разрешать адреса вызовов и инструкций загрузки, найденных в редактируемых объектных файлах. Чтобы понять принцип работы редактора связей, рассмотрим подробнее процесс раздельной компиляции.

Раздельная компиляция

Раздельная компиляция — это возможность, позволяющая разбить программу на несколько файлов, скомпилировать каждый из этих файлов отдельно, а потом *скомпоновать*[2] их, чтобы в конечном итоге создать *исполняемый файл*[3]. Результатом работы компилятора является объектный файл, а результатом работы редактора связей — исполняемый файл. Редактор связей физически связывает файлы, внесенные в список компоновки, в один программный файл и разрешает внешние ссылки. *Внешняя ссылка* создается каждый раз, когда программа из одного файла ссылается на код из другого файла. Это происходит при вызове функции и при ссылке на глобальную переменную. Например, при компоновке двух приведенных ниже файлов, должна быть разрешена ссылка в файле 2 на идентификаторcount, объявленный в файле 1. Редактор связей сообщает программе из файла 2, где найти count.

```
  Файл 1                  Файл 2
int count;              #include <stdio.h>
void display(void);       extern int count;

int main(void)           void display(void)
{                       {
count = 10;               printf("%d", count);
display();              }
return 0;
}
```

Аналогично, редактор связей укажет файлу 1, где находится функция display(), чтобы можно было ее вызвать.

При генерации объектного кода функции display(), компилятор подставляет в него вместо адреса идентификатора count "заполнитель", т.е. ссылку на внешнее имя, потому что он не располагает информацией о том, где находится count. Нечто подобное происходит при компиляции main(). Адрес функции display() не известен, поэтому вместо него используется "заполнитель", т.е. ссылка на внешнюю программу. При компоновке этих двух файлов содержащиеся в них внешние ссылки заменяются адресами соответствующих элементов. Являются ли эти адреса абсолютными или переместимыми, — зависит от среды[4].

Переместимые коды и абсолютные коды

В результате работы редактора связей для большинства видов вычислительной среды получается *переместимый код*. Так называют объектный код, который может работать в любой свободной области памяти, способной его уместить. В переместимом объектном файле адрес каждой инструкции вызова или загрузки является не фиксированным, а относительным. Таким образом, адреса в переместимом коде отсчитываются от адреса начала программы. При загрузке программы в память для выполнения, загрузчик преобразует относительные адреса в физические адреса, соответствующие адресам ячеек памяти, в которую загружается программа.

В некоторых вычислительных средах, таких как специализированные устройства управления, в которых для всех программ используется одно и то же адресное

пространство, редактор связей подставляет в конечный результат своей работы физические адреса. В этом случае он генерирует *абсолютный код*[5].

Редактирование связей с оверлеями

Хотя в наше время эта возможность применяется редко, следует отметить, что компиляторы С некоторых вычислительных сред в дополнение к обычным компоновщикам предоставляют компоновщики оверлеев. *Компоновщик оверлеев* работает так же как и обычный, но он также может создавать оверлеи[6]. *Оверлей* — это фрагмент объектного[7] кода, который хранится в файле на диске и загружается для работы только по мере необходимости. Место в памяти, которое отводится для загрузки оверлеев, называется *оверлейной областью памяти*. Оверлеи позволяют создавать и запускать программы, которые занимали бы большую область памяти, чем имеющаяся в наличии, потому что в каждый момент времени в памяти находится только та часть программы, которая нужна.

Чтобы понять, как работают оверлеи, представим, что имеется программа, состоящая из семи объектных файлов, которые называются F1, F2, ..., F7. Допустим, имеющейся свободной памяти недостаточно для загрузки программы, скомпонованной обычным образом из всех объектных файлов. Есть возможность скомпоновать только первые пять файлов, а при большем количестве наступит переполнение памяти. Выйти из подобной ситуации можно, дав компоновщику команду создать оверлеи из файлов F5, F6 и F7. При каждом вызове функции, которая содержится в одном из этих, файлов,*администратор оверлейной загрузки* (программа, предоставляемая компоновщиком или редактором связей) находит необходимый файл и помещает его в оверлейную область памяти, создавая условия для работы программы. Коды, которые получились при компиляции файлов F1 — F4, остаются резидентными. Данная схема проиллюстрирована на рис. 12.1.

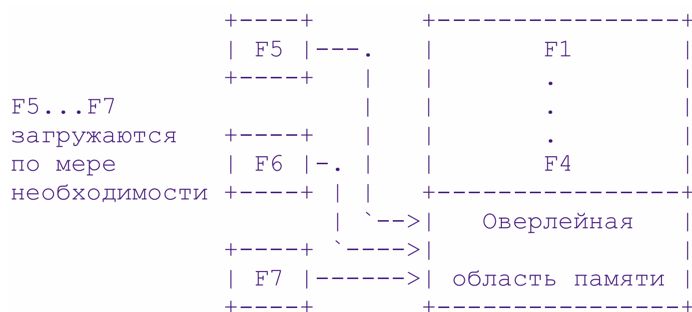


Рис.12.1 Программа с оверлеями загруженными в память.

Как читатель, возможно, уже догадался, принципиальным преимуществом оверлеев является возможность создавать с их помощью очень большие программы. Основной же их недостаток и причина редкого использования состоит в том, что процесс загрузки занимает определенное время и в значительной мере влияет на быстродействие программы. Поэтому при использовании оверлеев следует взаимосвязанные между собой функции группировать вместе, чтобы свести к минимуму число загрузок оверлеев.

Например, если приложение обрабатывает список рассылки, то имеет смысл поместить все подпрограммы сортировки в один оверлей, подпрограммы печати — в другой и т.д.

Как уже было сказано, в современных вычислительных средах оверлеи применяются редко.

Связывание с динамически подсоединяемыми библиотеками (DLL)

Операционная система Windows предоставляет другой вид связывания, так называемое *динамическое связывание*. Динамическое связывание — это процесс, при котором объектный код функции остается в отдельном файле на диске до тех пор, пока не запустится использующая его программа. При запуске такой программы динамически загружаются затребованные, связанные с ней функции. Динамически связанный функции

помещаются в специальный тип библиотек, которые называются *динамически подсоединяемыми библиотеками* (DLL — Dynamic-Link Library).
Основным преимуществом таких библиотек является возможность значительно сократить размер исполняемых программ, потому что отпадает необходимость в том, чтобы каждая программа содержала в себе копию используемых библиотечных функций. Другим положительным моментом является то, что при обновлении функций DLL, использующие их программы автоматически используют и все улучшения новых версий.
Стандартная библиотека С не содержится в динамически подсоединяемой библиотеке, но многие другие типы функций там есть. Например, при написании приложений для Windows, в DLL хранится полный набор функций программного интерфейса приложений (API — Application Program Interface). Нужно отметить, что для программы, написанной на языке С, обычно не имеет значения, хранятся ли библиотечные функции в DLL или в обычном файле библиотек.

---

[1]Иногда называется также *компоновщиком*. Впрочем, компоновщиками обычно называют программы, обладающие несколько меньшими возможностями, чем развитые редакторы связей.
[2]Этот процесс называется *редактированием связей*.
[3]Называется также *загрузочным модулем*.
[4]Редакторы связей обычно располагают широким набором возможностей, и при необходимости пользователь может указать необходимые параметры.
[5]Называется также *программой в абсолютных адресах*.
[6]Создание оверлейных программ — стандартная функция для редакторов связей. Кроме того редакторы связей обычно могут создавать даже загрузочные модули, загружаемые "вразброс", т.е. в несмежные участки памяти. Все это имеет огромное значение для систем, в которых отсутствует виртуальная память. Но в системах с виртуальной памятью эти возможности часто выглядят как ненужные излишества.
[7]Согласно оригиналу. Все же точнее *часть загрузочного модуля*.

Лекция 5. Автоматизация разработки программных проектов

Под средствами проектирования информационных систем (СП ИС) будем понимать комплекс инструментальных средств, обеспечивающих в рамках выбранной методологии проектирования поддержку полного жизненного цикла (ЖЦ) ИС, который включает в себя, как правило, стратегическое планирование, анализ, проектирование, реализацию, внедрение и эксплуатацию. Каждый этап характеризуется определенными задачами и методами их решения, исходными данными, полученными на предыдущем этапе, и результатами. При анализе СП их следует рассматривать не локально, а в комплексе, что позволяет реально охарактеризовать их достоинства, недостатки и место в общем технологическом цикле создания ИС.
В общем случае стратегия выбора СП для конкретного применения зависит от следующих факторов:

характеристик моделируемой предметной области;
целей, потребностей и ограничений будущего проекта ИС, включая квалификацию участвующих в процессе проектирования специалистов;
используемой методологии проектирования.
Тенденции развития современных информационных технологий приводят к постоянному возрастанию сложности ИС, создаваемых в различных областях экономики. Современные сложные ИС и проекты, обеспечивающие их создание, характеризуются, как правило,

следующими особенностями:

сложность предметной области (достаточно большое количество функций, объектов, атрибутов и сложные взаимосвязи между ними), требующая тщательного моделирования и анализа данных и процессов;
наличие совокупности тесно взаимодействующих компонентов - подсистем, имеющих свои локальные задачи и цели функционирования;
иерархическую структуру взаимосвязей компонентов, обеспечивающую устойчивость функционирования системы;
иерархическую совокупность критериев качества функционирования компонентов и ИС в целом, обеспечивающих достижение главной цели - создания и последующего применения системы;
отсутствие прямых аналогов, ограничивающее возможность использования каких-либо типовых проектных решений и прикладных систем;
необходимость достаточно длительного сосуществования старых приложений и вновь разрабатываемых БД и приложений;
наличие потребности как в традиционных приложениях, связанных с обработкой транзакций и решением регламентных задач, так и в приложениях аналитической обработки (поддержки принятия решений), использующих нерегламентированные запросы к данным большого объема;
поддержка одновременной работы достаточно большого количества локальных сетей, связываемых в глобальную сеть масштаба предприятия, и территориально удаленных пользователей;
функционирование в неоднородной операционной среде на нескольких вычислительных платформах;
разобщенность и разнородность отдельных микроколлективов разработчиков по уровню квалификации и сложившимся традициям использования тех или иных инструментальных средств;
существенная временная протяженность проекта, обусловленная, с одной стороны, ограниченными возможностями коллектива разработчиков, и, с другой стороны, масштабами организации-заказчика и различной степенью готовности отдельных ее подразделений к внедрению ИС.
Методология проектирования определяется как совокупность трех составляющих:

пошаговой процедуры, определяющей последовательность технологических операций проектирования;
критериев и правил, используемых для оценки результатов выполнения технологических операций;
нотаций (графических и текстовых средств), используемых для описания проектируемой системы.
На выбор СП могут существенно повлиять следующие особенности методологии проектирования:

ориентация на создание уникального или типового проекта;
итерационный характер процесса проектирования;
возможность декомпозиции проекта на составные части, разрабатываемые группами исполнителей ограниченной численности с последующей интеграцией составных частей;
жесткая дисциплина проектирования и разработки при их коллективном характере;
необходимость отчуждения проекта от разработчиков и его последующего централизованного сопровождения.
Критерии выбора

Традиционно при обсуждении проблемы выбора СП (в особенности CASE-средств) большое внимание уделялось особенностям реализации той или иной методологии анализа предметной области (E-R, IDEF0, IDEF1X, Gane/Sarson, Yordon, Barker и др.). Безусловно, богатство изобразительных и описательных средств дает возможность на этапах стратегического планирования и анализа построить наиболее полную и адекватную модель предметной области. С другой стороны, если говорить о конечных результатах - базах данных и приложениях, то обнаруживается, что часть описаний в них практически не отражается, оставаясь чисто декларативной (на выходе мы в любом случае получим описание БД в табличном представлении с минимальным набором ограничений целостности и исполнимый код приложений, большую часть которых составляют экранные формы, не выводимые непосредственно из моделей предметной области). Опытные аналитики и проектировщики всегда с большими или меньшими трудозатратами придут к нужному конечному результату независимо от того, какая конкретно методология или ее разновидность реализована в данном инструменте. Это, конечно, не означает, что методология не важна, напротив, отсутствие или неполнота описательных средств могут с самого начала значительно затруднить работу над проектом. Однако, зачастую на первом плане оказываются другие критерии, невыполнение которых может породить гораздо большие трудности.

Может создаться впечатление, что если можно сформировать необходимую аппаратную платформу из компонентов различных фирм-производителей, то так же просто можно выбрать и скомплексировать разные инструментальные средства, каждое из которых является одним из мировых лидеров в своем классе. Однако в случае инструментальных средств в настоящее время, в отличие от оборудования, отсутствуют международные стандарты на основные свойства конечных продуктов (программ, баз данных и их сопряжение). Однако, поскольку составные части проекта должны быть интегрированы в единый продукт, следовательно, имеет смысл рассматривать не любые, а только сопряженные инструментальные средства, которые в принципе могут быть ориентированы - даже внутри одного класса - на разные методологии; при этом необходимо отбирать в состав комплекса СП средства, поддерживающие по крайней мере близкие методологии, если не одну и ту же.

Исходя из перечисленных выше соображений, примем в качестве основных критериев выбора СП следующие критерии:

Поддержка полного жизненного цикла ИС с обеспечением эволюционности ее развития.

Полный жизненный цикл ИС должен поддерживаться "сквозной" технологической цепочкой средств разработчика, обеспечивающей решение следующих задач:

обследование и получения формализованных знаний о предметной области (последовательный и логически связный переход от формализованного описания предметной области к ее моделям);

декомпозиция проекта на составные части и интеграция составных частей;

проектирование моделей приложений (логики приложений и пользовательских интерфейсов);

прототипирование приложений;

проектирование баз данных;

коллективная, территориально распределенная разработка приложений с использованием различных инструментальных средств (включая их интеграцию, тестирование и отладку);

разработка распределенных баз данных (с выбором оптимальных вариантов распределения);

разработка проектной документации с учетом требований проектных стандартов;

адаптация к различным системно-техническим платформам и СУБД;

тестирование и испытания;

сопровождение, внесение изменений и управление версиями и конфигурацией ИС;

интеграция с существующими разработками (включая реинжиниринг приложений, конвертирование БД);
администрирование ИС (оптимизация эксплуатационных характеристик);
управление разработкой и сопровождением ИС (планирование, координация и контроль за ресурсами и ходом выполнения работ);
прогнозирование и оценка трудоемкости, сроков и стоимости разработки.
Для существующих ИС должен обеспечиваться плавный переход из старой среды эксплуатации в новую с минимальными переделками и поддержкой эксплуатируемых баз данных и приложений, внедренных до начала работ по созданию новой системы.
Обеспечение целостности проекта и контроля за его состоянием. Данное требование означает наличие единой технологической среды создания, сопровождения и развития ИС, а также целостность базы проектных данных (репозитория). Единая технологическая среда должна обеспечиваться за счет использования единственной CASE-системы для поддержки моделей ИС, а также за счет наличия программно-технологических интерфейсов между отдельными инструментальными средствами, сертифицированных и поддерживаемых фирмами-разработчиками соответствующих средств. В частности, интерфейс между CASE-системой и средствами разработки приложений должен выполнять две основные функции: а) непосредственный переход в рамках единой среды от описания логики приложения, реализованного CASE-системой, к разработке пользовательского интерфейса (экранных форм); б) перенос описания БД из репозитория CASE-системы в репозиторий средства разработки приложений и обратно. Вся информация о проекте должна автоматически помещаться в базу проектных данных, при этом должны поддерживаться согласованность, непротиворечивость, полнота и минимальная избыточность проекта, а также корректность операций его редактирования. Это может быть достигнуто при условии исключения или существенного ограничения возможности актуализации репозитория различными средствами. Должны также обеспечиваться возможности для централизованного сбора, хранения и распределения информации между различными этапами проекта, группами разработчиков и выполняемыми операциями. Поддержка базы проектных данных может быть реализована собственными средствами СП или средствами целевой СУБД (второй вариант предпочтительнее, поскольку упрощается технология ведения репозитория).
Невыполнение требования целостности в условиях разобщенности разработчиков и временной протяженности крупного проекта может означать утрату контроля за его состоянием.
Независимость от программно-аппаратной платформы и СУБД. Требование определяется неоднородностью среды функционирования ИС. Такая независимость может иметь две составляющих: независимость среды разработки и независимость среды эксплуатации приложений. Она обеспечивается за счет наличия совместимых версий СП для различных платформ и драйверов соответствующих сетевых протоколов, менеджеров транзакций и СУБД.
Один из дополнительных факторов, который при этом следует учитывать - это способ взаимодействия с СУБД (прямой или через ODBC), поскольку использование ODBC может заметно ухудшить производительность и надежность интерфейса.
Поддержка одновременной работы групп разработчиков. Развитые СП должны обладать возможностями разделения полномочий персонала разработчиков и объединения отдельных работ в общий проект. Должна обеспечиваться одновременная работа проектировщиков БД и разработчиков приложений (разработчики приложений в такой ситуации могут начинать работу с базой данных, не дожидаясь полного завершения ее проектирования CASE-средствами). При этом все группы специалистов должны быть обеспечены адекватным инструментарием, а внесение изменений в проект различными разработчиками должно быть согласованным и

корректным. Каждый разработчик должен иметь возможность работы со своим личным репозиторием, являющимся фрагментом или копией общего репозитория. Должны обеспечиваться содержательная интеграция всех изменений, вносимых разработчиками, в общем репозитории, одновременная доступность для разработчика общего и личного репозиториев и простота переноса объектов между ними.
Помимо перечисленных основных критериев, предварительный анализ при выборе СП должен учитывать следующие аспекты:
Возможность разработки приложений "клиент-сервер" требуемой конфигурации.
Подразумевается сочетание наличия развитой графической среды разработки приложений (многооконность, разнообразие стандартных графических объектов, разнообразие используемых шрифтов и т.д.) с возможностью декомпозиции (partitioning) приложения на "клиентскую" часть, реализующую пользовательский экранный интерфейс и "серверную" часть. При этом должна обеспечиваться возможность перемещения отдельных компонентов приложения между "клиентом" и "сервером" на наиболее подходящую платформу, обеспечивающую максимальную эффективность функционирования приложения в целом, а также возможность использования сервера приложений (менеджера транзакций).
Открытая архитектура и возможности экспорта/импорта.
Открытая и общедоступная информация об используемых форматах данных и прикладных программных интерфейсах должна позволять интегрировать инструментальные средства третьих фирм и относительно безболезненно переходить от одной системы к другой. Возможности экспорта/импорта означают, что спецификации, полученные на этапах анализа, проектирования и реализации для одной ИС, могут быть использованы для проектирования другой ИС. Повторное проектирование и реализация могут быть обеспечены при помощи средств экспорта/импорта спецификаций в различные СП.
Качество технической поддержки в России, стоимость приобретения и поддержки, опыт успешного использования.
Имеется в виду наличие квалифицированных дистрибьюторов и консультантов, быстрота обслуживания пользователей, высокое качество технической поддержки и обучения продукту и методологии его применения для больших коллективов разработчиков (наличие сведений о практике использования системы, качество документации, укомплектованность примерами и обучающими курсами, наличие прототипных проектов). Затраты на обучение новым технологиям значительны, однако потери от использования современных сложных технологий необученными специалистами могут оказаться значительно выше.
Кроме того, фирмы-поставщики инструментальных средств должны быть устойчивыми, так как технология выбирается не на один год, а также должны обеспечивать хорошую поддержку на территории России (горячая линия, консультации, обучение, консалтинг), возможно, через дистрибьюторов.
Что касается стоимости, следует учитывать возможность получения бесплатной пробной лицензии (trial license), стоимость лицензии на одно рабочее место СП, скидки, предоставляемые фирмой в случае приобретения большого количества лицензий, необходимость приобретения run-time версий для эксплуатации приложений и т.д. В то же время стоимость продукта должна рассматриваться не сама по себе, а с учетом ее соответствия возможностям продукта.
Простота использования.
Учитываются следующие характеристики:
доступность пользовательского интерфейса;
время, необходимое для обучения;
простота инсталляции;
качество документации.

Обеспечение качества проектной документации. Это требование относится к возможностям СП анализировать и проверять описания и документацию на полноту и непротиворечивость, а также на соответствие принятым в данной методологии стандартам и правилам (включая ГОСТ, ЕСПД). В результате анализа должна формироваться информация, указывающая на имеющиеся противоречия или неполноту в проектной документации. Должна быть также обеспечена возможность создавать новые формы документов, определяемые пользователями.

Использование общепринятых, стандартных нотаций и соглашений. Для того, чтобы проект мог выполняться разными коллективами разработчиков, необходимо использование стандартных методов моделирования и стандартных нотаций, которые должны быть оформлены в виде нормативов до начала процесса проектирования. Несоблюдение данного требования ставит разработчиков в зависимость от фирмы-производителя данного средства, делает затруднительным формальный контроль корректности используемых нотаций и снижает возможности привлечения дополнительных коллективов разработчиков, поскольку число специалистов, знакомых с данным методом (нотацией) может быть ограниченным.

В идеальном случае (что, конечно же, далеко не всегда возможно) окончательный выбор может быть произведен по результатам тестирования в соответствии с заданным планом, которое должно включать имитацию проектирования реальной БД и разработки приложений и состоять из следующих шагов:

установка и конфигурирование (ясность и точность инструкций по установке, наличие подсказок в процессе установки, возможность установки по выбору и задания многопользовательской конфигурации);

разработка концептуальной схемы БД (понятность и простота построения, модификации и документирования различных элементов диаграмм "сущность-связь", отображение ограничений ссылочной целостности и бизнес- правил, управление режимом отображения);

формирование отчета о концептуальной схеме (список сущностей с определениями и атрибутами, включая указание ключей, список атрибутов, сгруппированных по сущностям, список связей между сущностями, возможность форматирования отчета, составления отчета по выделенной части схемы, передачи отчета, например, в другие приложения (текстовые процессоры));

разработка графической схемы БД для конкретной СУБД с учетом специфичных для нее структур данных и ограничений (выбор целевой СУБД и реализация элементов схемы - ввод и модификация имен таблиц и столбцов, определение типов данных, доменов, индексов, значений по умолчанию и неопределенных значений, порядка индексирования, а также задание ограничений ссылочной целостности и дополнительных бизнес-правил, характеризующих предметную область, управление триггерами и хранимыми процедурами);

формирование отчета о схеме БД (печать диаграммы схемы, списка таблиц с соответствующими столбцами, первичными ключами, индексами и т.д., возможность форматирования отчета, составления отчета по выделенной части схемы, передачи отчета в другие приложения);

генерация схемы БД (трансформация схемы БД в файл DDL в текстовом формате или непосредственный интерфейс с целевой СУБД);

разработка простейшего приложения (описание экранных форм, программирование или описание логики приложения и интерфейса с БД, загрузка БД тестовыми данными и тестирование приложения);

сопровождение схем БД (внесение изменений - создание новых сущностей и атрибутов, изменение схемы БД, повторная генерация схемы, управление версиями, обеспечение сохранности данных, синхронизация концептуальной схемы и самой БД);

обратное проектирование - реинжиниринг (полное и точное восстановление исходной концептуальной схемы по файлам DDL или непосредственно из словаря целевой СУБД).
В результате выполненного анализа может оказаться, что ни одно доступное средство не удовлетворяет в нужной мере всем основным критериям и не покрывает все потребности проекта. В этом случае может применяться набор средств, позволяющий построить на их базе единую технологическую среду.

Анализ средств проектирования информационных систем

Современные СП могут быть разделены на две большие категории. Первую составляют CASE- системы (как независимые (upper CASE), так и интегрированные с СУБД), обеспечивающие проектирование БД и приложений в комплексе с интегрированными средствами разработки приложений "клиент-сервер" (например, Westmount I-CASE+Uniface, Designer/2000+Developer/2000). Их основное достоинство заключается в том, что они позволяют разрабатывать всю ИС целиком (функциональные спецификации, логику процессов, интерфейс с пользователем и базу данных), оставаясь в одной технологической среде. Инструменты этой категории, как правило, обладают существенной сложностью, широкой сферой применения и высокой гибкостью. Вторую категорию составляют собственно средства проектирования БД, реализующие ту или иную методологию, как правило, "сущность-связь" ("entity-relationship") и рассматриваемые в комплексе со средствами разработки приложений. К средствам этой категории можно отнести такие, как SILVERRUN+JAM, ERwin/ERX+PowerBuilder и др. Помимо указанных категорий, СП можно классифицировать по следующим признакам:
степени интегрированности: (отдельные локальные средства, набор частично интегрированных средств, охватывающих большинство этапов жизненного цикла ИС и полностью интегрированные средства, связанные общей базой проектных данных - репозиторием);
применяемым методологиям и моделям систем и БД;
степени интегрированности с СУБД;
степени открытости;
доступным платформам.
В разряд СП попадают как относительно дешевые системы для персональных компьютеров (ПК) с весьма ограниченными возможностями, так и дорогостоящие системы для неоднородных вычислительных платформ и операционных сред. Так, современный рынок программных средств насчитывает около 300 различных CASE-систем, наиболее мощные из которых так или иначе используются практически всеми ведущими западными фирмами.
Применение СП требует от потенциальных пользователей специальной подготовки и обучения. Опыт показывает, что внедрение СП осуществляется медленно, однако по мере приобретения практических навыков и общей культуры проектирования эффективность применения этих средств резко возрастает, причем наибольшая потребность в использовании СП испытывается на начальных этапах разработки, а именно на этапах анализа и спецификации требований. Это объясняется тем, что цена ошибок, допущенных на начальных этапах, на несколько порядков превышает цену ошибок, выявленных на более поздних этапах разработки.
На сегодняшний день Российский рынок программного обеспечения располагает следующими наиболее развитыми СП:
Westmount I-CASE;
Uniface;
Designer/2000+Developer/2000 (ORACLE);
SILVERRUN+JAM;
ERwin/ERX+PowerBuilder.
Приведенный список не претендует на полноту. Кроме того, на рынке постоянно появляются как новые (для отечественных пользователей) системы, так и новые версии и

модификации перечисленных систем (например, CASE/4/0, System Architect и т.д.). Некоторое представление о возможностях наиболее развитых СП может дать краткая характеристика следующих программных продуктов:

Westmount I-CASE 3.2 (CADRE Technologies Inc.)

Westmount I-CASE представляет собой интегрированный программный продукт, обеспечивающий выполнение следующих функций:

графическое проектирование архитектуры системы (проектирование состава и связи вычислительных средств, распределения задач системы между вычислительными средствами, моделирование отношений типа "клиент- сервер", анализ использования мониторов транзакций и особенностей функционирования систем в реальном времени);

проектирование диаграмм потоков данных, "сущность-связь", структур данных, структурных схем программ и последовательностей экранных форм;

генерация кода программ на 4GL целевой СУБД с полным обеспечением программной среды и генерация SQL-кода для создания таблиц БД, индексов, ограничений целостности и хранимых процедур;

программирование на языке C со встроенным SQL;

управление версиями и конфигурацией проекта;

генерация проектной документации по стандартным и индивидуальным шаблонам;

экспорт и импорт данных проекта в формате CDIF.

Westmount I-CASE можно использовать в конфигурации "клиент-сервер", при этом база проектных данных может располагаться на сервере, а рабочие места разработчиков могут быть клиентами.

Westmount I-CASE функционирует на всех основных UNIX-платформах и VMS. В качестве целевой СУБД могут использоваться ORACLE, Informix, Sybase и Ingres.

В качестве отдельного продукта поставляется интерфейс Westmount-Uniface Bridge, обеспечивающий совместное использование двух систем в рамках единой технологической среды проектирования (при этом схемы БД, структурные схемы программ и последовательности экранных форм непосредственно в режиме on-line, без создания каких-либо файлов экспорта- импорта, переносятся в репозиторий Uniface, и, наоборот, прикладные модели, сформированные средствами Uniface, могут быть перенесены в репозиторий Westmount I-CASE. Возможные рассогласования между репозиториями двух систем устраняются с помощью специальной утилиты).

В рамках версии Westmount I-CASE 4.0 предполагается обеспечить возможность функционирования клиентской части в среде Windows 95, а серверной - в среде Windows NT.

Uniface (Compuware)

Uniface 6.1 представляет собой среду разработки крупномасштабных приложений "клиент-сервер" и имеет следующую компонентную архитектуру:

Application Objects Repository (репозиторий объектов приложений) содержит метаданные, автоматически используемые всеми остальными компонентами на протяжении жизненного цикла ИС.

Application Model Manager поддерживает прикладные модели, каждая из которых представляет собой подмножество общей схемы БД с точки зрения данного приложения.

Rapid Application Builder - средство быстрого создания экранных форм и отчетов на базе объектов прикладной модели. Оно включает графический редактор форм, средства прототипирования, отладки, тестирования и документирования. Реализован интерфейс с разнообразными типами оконных элементов управления (Open Widget Interface) для существующих графических систем - MS Windows (включая VBX), Motif, OS/2.

Developer Services (службы разработчика) - используются для поддержки крупных проектов и реализуют контроль версий, права доступа, глобальные модификации и т.д. Это обеспечивает разработчиков средствами параллельного проекти-рования, входного и

выходного контроля, поиска, просмотра, поддержки и выдачи отчетов по данным системы контроля версий.
Deployment Manager (управление распространением приложений) - средства, позволяющие подготовить созданное приложение для распространения, установить и сопровождать его (при этом платформа пользователя может отличаться от платформы разработчика). В их состав входят сетевые драйверы и драйверы СУБД, сервер приложений (полисервер), средства распространения приложений и управления базами данных. Uniface поддерживает интерфейс практически со всеми известными программно-аппаратными платформами, СУБД, CASE-средствами, сетевыми протоколами и менеджерами транзакций.
Personal Series (персональные средства) - используются для создания сложных запросов и отчетов в графической форме, а также для переноса данных в такие системы, как WinWord и Excel.

Лекция 6. Операционные системы. Трансляторы. Эмуляторы

Транслятор (англ. translator - переводчик) - это программа-переводчик. Она преобразует программу, написанную на одном из языков программирования, в бинарный файл программы, состоящей из машинных команд, либо непосредственно выполняет действия программы.
Трансляторы реализуются в виде компиляторов, интерпретаторов, препроцессоров и эмуляторов. С точки зрения выполнения работы компилятор и интерпретатор существенно различаются.
Компилятор (англ. compiler - составитель, собиратель) - читает всю программу целиком, делает ее перевод и создает законченный вариант программы на машинном языке, то есть бинарный файл, содержащий перечень машинных команд. Бинарный файл может быть исполняемым, библиотечным, объектным), он выполняется.операционной системой без участия компилятора.
Интерпретатор (англ. interpreter - истолкователь, переводчик) - переводит программу построчно (по одному оператору) в машинный код (команды процессора, ОС, иной среды), выполняет переведенный оператор (строку программы), а затем переходит к следующей строке программного текста. Интерпретатор не формирует исполняемых файлов, он сам выполняет все действия, записанные в тексте исходной программы.
После того, как программа откомпилирована, ни сама исходная программа, ни компилятор более не нужны. В то же время программа, обрабатываемая интерпретатором, должна заново переводиться на машинный язык при каждом очередном запуске программы.
Откомпилированные программы работают быстрее, но интерпретируемые проще исправлять и изменять.
Каждый конкретный язык ориентирован либо на компиляцию, либо на интерпретацию — в зависимости от того, для каких целей он создавался. Например, Паскаль обычно используется для решения довольно сложных задач, в которых важна скорость работы программ. Поэтому данный язык обычно реализуется с помощью компилятора.
С другой стороны, Бейсик создавался как язык для начинающих программистов, для которых построчное выполнение программы имеет неоспоримые преимущества.
Иногда для одного языка имеется и компилятор, и интерпретатор. В этом случае для разработки и тестирования программы можно воспользоваться интерпретатором, а затем откомпилировать отлаженную программу, чтобы повысить скорость ее выполнения.

Препроцессор - это транслятор с одного языка программирования в другой без создания исполняемого файла или выполнения программы.

Препроцессоры удобны для расширения возможностей языка и удобства программирования путем использования на этапе написания программы более удобного для человека диалекта языка программирования и ее перевода препроцессором на текст стандартного языка программирования, который можно откомпилировать стандартным компилятором.

Эмулятор - функционирующее в некоторой целевой операционной системе и аппаратной платформе программное и/или аппаратное средство, предназначенное для исполнения программ, изготовленных в другой операционной системе или работающих на отличном от целевого аппаратном обеспечении, но позволяющее осуществлять те же самые операции в целевой среде, что и в имитируемой системе.

К эмулирующим языкам относятся такие системы, как Java, .Net, Mono, в которых на этапе создания программы производится ее компиляция в специальный байт-код и получение бинарного файла, пригодного для исполнения в любой операционной и аппаратной среде,а исполнение полученного байт-кода производится на целевой машине с помощью простого и быстрого интерпретатора (виртуальной машины).

Реассемблер, дизассемблер - программное средство, предназначенное для расшифровки бинарного кода с представлением его в виде текста ассемблера или текста иного языка программирования, позволяющее проанализировать алгоритм исходной программы и использовать полученный текст для необходимой модификации программы, к примеру поменять адреса внешних устройств, обращения к системным и сетевым ресурсам, выявить скрытые функции бинарного кода (к примеру, компьютерного вируса или иной зловредной программы: трояна, червя, кейлоггера и пр.).

Существует большое количество ОС. Можно выполнить обширную классификацию их по самым различным критериям. Поэтому при изучении операционных систем выделим только те функции, которые присущи всем ОС как классу продуктов.

Можно встретить различные определения операционной системы, но смысл их одинаковый.

Операционная система – комплекс взаимосвязанных программ, который действует как интерфейс между приложениями и пользователями с одной стороны, и аппаратурой компьютера – с другой стороны.

Отсюда две группы функций ОС, определяющих ее двухстороннее назначение:

предоставление пользователю или программисту вместо реальной аппаратуры компьютера расширенной виртуальной машины, с которой удобнее работать и которую легче программировать. Программные модули ОС, формирующие человеко-машинный интерфейс, предназначены для повышения эффективности работы человека, которая достигается максимальным использованием всех его органов чувств при работе с компьютером;

повышение эффективности использования компьютера путем рационального управления его ресурсами.

Первая группа функций операционной системы направлена на взаимодействие с пользователем ОС. При этом следует различать интерфейс прикладного программиста, создающий операционную среду, и пользовательский, человеко-машинный интерфейс.

Вторая группа функций ОС направлена на взаимодействие с аппаратурой компьютера. Рациональное управления ресурсами компьютера повышает эффективность его использования.

Операционная система избавляет программистов, пишущих приложения, от необходимости напрямую работать с аппаратурой компьютера: системой команд процессора, секторами и дорожками диска, физическими адресами памяти и т. п.

Программисту ОС дает так называемую операционную среду – набор системных функций (сервисов), который вместе с правилами их использования создает интерфейс прикладного программирования (Application Program Interface, API) этой ОС.
Приложение выполняет обращение к функциям API с помощью системных запросов. Операционная система выполняет функции API, запуская специальные системные программные модули, входящие в ее состав. В высокоуровневых языках программирования обращение к системным функциям зачастую скрыто определенными синтаксическими конструкциями языка. Поэтому прикладной программист напрямую может к ним и не обращаться.
У разных операционных систем свой API. Очевидно, что программа, созданная в некоторой операционной системе с одним API, не будет работать в операционной системе с другим API. Пытаясь преодолеть это ограничение, в ОС стали применять поддержку нескольких API. Таким образом, в общем случае операционная система может поддерживать несколько операционных сред.
Другое направление преодоления этого ограничения – попытка стандартизации функции API. Примером может служить известный и, пожалуй, единственный стандарт **POSIX** (Portable Operating System Interface for Computer Environments) – независимый от платформы системный интерфейс для компьютерного окружения. В этом стандарте перечислен большой набор функций, их параметров и возвращаемых значений. Стандарт предназначен для открытых систем и поэтому базируется на UNIX-системах, но допускает реализацию и в других операционных системах.
Частным случаем попытки стандартизации API является внутренний корпоративный стандарт компании Microsoft, известный как WinAPI, который ориентирован на работу в графической среде. С точки зрения WinAPI базовой задачей является окно.
Операционная система предоставляет пользователю набор команд, отражающий функциональные возможности ОС. Для удобства работы пользователю за компьютером современные операционные системы имеют пользовательский интерфейс. Раньше для этих целей использовались операционные оболочки – специальные прикладные программы, формирующие удобный пользовательский интерфейс и принимающие от пользователя управляющие команды. Понятие операционной оболочки на сегодняшний день уже устарело, т. к. формирование удобного пользовательского интерфейса стало обязательной задачей самой операционной системы.
Часто приводят следующую классификацию различных интерфейсов общения человека и компьютера:
Командный интерфейс. Он называется так потому, что в этом виде интерфейса человек подает "команды" компьютеру, а компьютер их выполняет и выдает результат человеку. Командный интерфейс реализован в виде пакетной технологии и технологии командной строки. Простейший программный модуль, входящий в состав операционной системы, который отвечает за чтение отдельных команд пользователя или последовательности команд, поступающих из командного файла, называют командным интерпретатором (например, в Windows-2000 – это программа **cmd**, а в Linux – **shell**).
WIMP-интерфейс (Window – окно, Image – образ, Menu – меню, Pointer – указатель). Характерной особенностью этого вида интерфейса является то, что диалог с пользователем ведется не с помощью команд, а с помощью графических образов – меню, окон, других элементов. Хотя и в этом интерфейсе подаются команды машине, но это делается опосредственно, через графические образы. Основными понятиями графического интерфейса являются понятия рабочей области, окна и иконки, или значка. Любой объект в графическом интерфейсе, так или иначе, связан с этими понятиями.
Следует отметить, что WIMP требует для своей реализации цветной дисплей с высоким разрешением и манипулятор. Также программы, ориентированные на этот вид интерфейса, предъявляют повышенные требования к производительности компьютера, объему его памяти, пропускной способности шины и т. п. Однако этот вид интерфейса

наиболее прост в усвоении и интуитивно понятен. Поэтому сейчас WIMP-интерфейс стал стандартом де-факто. Набор системных функций API, предоставляющих графические возможности прикладным программам, обозначают как GUI (Graphical User Interface – графический интерфейс пользователя).

Ярким примером программ с графическим интерфейсом является операционная система Microsoft Windows.

SILK-интерфейс (Speech – речь, Image – образ, Language – язык, Knowlege – знание). Этот вид интерфейса наиболее приближен к обычной, человеческой форме общения. В рамках этого интерфейса идет обычный "разговор" человека и компьютера. При этом компьютер находит для себя команды, анализируя человеческую речь и находя в ней ключевые фразы. Результат выполнения команд он также преобразует в понятную человеку форму. Это очень перспективное направление хотя бы по указанной выше причине: вводить информацию с голоса – самый быстрый и удобный способ. Но его практические реализации пока не стали доминирующими – все-таки качество распознавания устной речи пока далеко от идеала.

Компания IBM поведала об успешном завершении разработки системы распознавания голосовых команд. Новинка ViaVoice IBM 4.4 основана на семантической интерпретации и так называемом языковом моделировании. В отличие от большинства систем голосового управления, лишь сравнивающих ту или иную команду с записанным в их памяти образцом, новая система позволяет управлять каким-либо оборудованием, отдавая команды в свободной форме, не требуя от пользователя запоминать их четкую формулировку.

Первейшей областью применения таких систем станут автомобили – пользователи смогут отдавать команды автомагнитоле и другой встроенной электронике, а также сервисным системам самого автомобиля. Дальнейшее совершенствование человеко-машинного интерфейса направлено в сторону повышения комфортности работы пользователя с использованием достижений в области мультимедиа, гипермедиа, систем распознавания речи, сенсорных технологий и т. п.

В частности, в конце 90-х гг. XX в. возникла так называемая биометрическая технология ("**мимический интерфейс**"). В этой технологии для управления компьютером используется выражение лица человека, направление его взгляда, размер зрачка и другие признаки. Для идентификации пользователя используется рисунок радужной оболочки его глаз, отпечатки пальцев и другая уникальная информация. Изображения считываются с цифровой видеокамеры, а затем с помощью специальных программ распознавания образов из этого изображения выделяются команды. Эта технология, по-видимому, займет свое место в программных продуктах и приложениях, где важно точно идентифицировать пользователя компьютера.

В 2006 г. Microsoft представила рабочий вариант одной из программ системы оптимизации информационной загруженности человека. Система претендует на роль основы принципиально нового человеко-машинного интерфейса, который придет на смену концепции Windows. Система комбинирует функции секретаря и регулировщика трафика. Она решает, какую именно информацию из входящего потока уместно предоставить пользователю в данный момент. При этом она контролирует, чем занят в данный момент пользователь – набирает текст на клавиатуре, говорит по телефону или общается лично с кем-либо в офисе, ест или спит. Поведение пользователя вне компьютера отслеживается с помощью видеокамеры и микрофона. Система непрерывно следит за человеком, анализируя его действия и зону внимания.

Вместо графического интерфейса GUI основная роль ложится на "интерфейс внимания" – Attentional User Interface (AUI). Он активно фильтрует и распределяет информационные потоки – электронную и голосовую почту, интернет-новости, сообщения сетевых пейджеров и др. Этот процесс происходит на основе "приоритетов срочности", определяемых и задаваемых системой по 100-балльной шкале. Таким образом,

устраняется опасность информационной перегрузки человека и повышается эффективность его работы.
Предполагается, что вся система (AUI) не будет простым приложением к персональному компьютеру. Он должен обслуживать человека везде, где бы тот ни находился, общаясь с ним через мобильный телефон, карманный компьютер, пейджер и любые другие доступные средства связи. Все это вписывается в проводимую Microsoft генеральную доктрину.NET.
Руководство Microsoft считает, что оконно-мышиную концепцию интерфейса, которая не менялась с середины 1980-х, сменит интеллектуальная технология, подобная представленной.

Лекция 7. Сравнительный анализ формальных алгоритмических языков программирования

Разные типы процессоров имеют разные наборы команд. Если язык программирования ориентирован на конкретный тип процессора и учитывает его особенности, то он называется языком программирования низкого уровня. В данном случае «низкий уровень» не значит «плохой». Имеется в виду, что операторы языка близки к машинному коду и ориентированы на конкретные команды процессора.

Языком самого низкого уровня является язык ассемблера, который просто представляет каждую команду машинного кода, но не в виде чисел, а с помощью символьных условных обозначений, называемых мнемониками. Однозначное преобразование одной машинной инструкции в одну команду ассемблера называется транслитерацией. Так как наборы инструкций для каждого модели процессора отличаются, конкретной компьютерной архитектуре соответствует свой язык ассемблера, и написанная на нем программа может быть использована только в этой среде.

С помощью языков низкого уровня создаются очень эффективные и компактные программы, так как разработчик получает доступ ко всем возможностям процессора. С другой стороны, при этом требуется очень хорошо понимать устройство компьютера, затрудняется отладка больших приложений, а результирующая программа не может быть перенесена на компьютер с другим типом процессора. Подобные языки обычно применяют для написания небольших системных приложений, драйверов устройств, модулей стыковки с нестандартным оборудованием, когда важнейшими требованиями становятся компактность, быстродействие и возможность прямого доступа к аппаратным ресурсам. В некоторых областях, например в машинной графике, на языке ассемблера пишутся библиотеки, эффективно реализующие требующие интенсивных вычислений алгоритмы обработки изображений.

Языки программирования высокого уровня значительно ближе и понятнее человеку, нежели компьютеру. Особенности конкретных компьютерных архитектур в них не учитываются, поэтому создаваемые программы на уровне исходных текстов легко переносимы на другие платформы, для которых создан транслятор этого языка. Разрабатывать программы на языках высокого уровня с помощью понятных и мощных команд значительно проще, а ошибок при создании программ допускается гораздо меньше.

Языки программирования принято делить на пять поколений. В первое поколение входят языки, созданные в начале 50-х годов, когда первые компьютеры только появились на свет. Это был первый язык ассемблера, созданный по принципу «одна инструкция - одна строка».

Расцвет второго поколения языков программирования пришелся на конец 50-х - начало 60-х годов. Тогда был разработан символический ассемблер, в котором появилось понятие переменной. Он стал первым полноценным языком программирования. Благодаря его возникновению заметно возросли скорость разработки и надежность программ.

Появление третьего поколения языков программирования принято относить к 60-м годам. В это время родились универсальные языки высокого уровня, с их помощью удается решать задачи из любых областей. Такие качества новых языков, как относительная простота, независимость от конкретного компьютера и возможность использования мощных синтаксических конструкций, позволили резко повысить производительность труда программистов. Понятная большинству пользователей структура этих языков привлекла к написанию небольших программ (как правило, инженерного или экономического характера) значительное число специалистов из некомпьютерных областей. Подавляющее большинство языков этого поколения успешно применяется и сегодня.

С начала 70-х годов по настоящее время продолжается период языков четвертого поколения. Эти языки предназначены для реализации крупных проектов, повышения их надежности и скорости создания. Они обычно ориентированы на специализированные области применения, где хороших результатов можно добиться, используя не универсальные, а проблемно-ориентированные языки, оперирующие конкретными понятиями узкой предметной области. Как правило, в эти языки встраиваются мощные операторы, позволяющие одной строкой описать такую функциональность, для реализации которой на языках младших поколений потребовались бы тысячи строк исходного кода.

Рождение языков пятого поколения произошло в середине 90-х годов. К ним относятся также системы автоматического создания прикладных программ с помощью визуальных средств разработки, без знания программирования. Главная идея, которая закладывается в эти языки, - возможность автоматического формирования результирующего текста на универсальных языках программирования (который потом требуется откомпилировать). Инструкции же вводятся в компьютер в максимально наглядном виде с помощью методов, наиболее удобных для человека, не знакомого с программированием.

Обзор языков программирования высокого уровня

FORTRAN (Фортран). Это первый компилируемый язык, созданный Джимом Бэкусом в 50-е годы. Программисты, разрабатывавшие программы исключительно на ассемблере, выражали серьезное сомнение в возможности появления высокопроизводительного языка высокого уровня, поэтому основным критерием при разработке компиляторов Фортрана являлась эффективность исполняемого кода. Хотя в Фортране впервые был реализован ряд важнейших понятий программирования, удобство создания программ было принесено в жертву возможности получения эффективного машинного кода. Однако для этого языка было создано огромное количество библиотек, начиная от статистических комплексов и кончая пакетами управления спутниками, поэтому Фортран продолжает активно использоваться во многих организациях, а сейчас ведутся работы над очередным стандартом Фортрана F2k, который появится в 2000 году. Имеется стандартная версия Фортрана HPF(High Performance Fortran) для параллельных суперкомпьютеров со множеством процессоров.

COBOL (Кобол). Это компилируемый язык для применения в экономической области и решения бизнес - задач, разработанный в начале 60-х годов. Он отличается большой « многословностью » - его операторы иногда выглядят как обычные английские фразы. В Коболе были реализованы очень мощные средства работы с большими объемами данных, хранящимися на различных внешних носителях. На этом языке создано очень

много приложений, которые активно эксплуатируются и сегодня. Достаточно сказать, что наибольшую зарплату в США получают программисты на Коболе.

Algol (Алгол). Компилируемый язык, созданный в 1960 году. Он был призван заменить Фортран, но из-за более сложной структуры не получил широкого распространения. В 1968 году была создана версия Алгол 68, по своим возможностям и сегодня опережающая многие языки программирования, однако из-за отсутствия достаточно эффективных компьютеров для нее не удалось своевременно создать хорошие компиляторы.

Paskal (Паскаль). Язык Паскаль, созданный в конце 70-х годов основоположником множества идей современного программирования Никлаусом Виртом, во многом напоминает Алгол, но в нем ужесточен ряд требований к структуре программы и имеются возможности, позволяющие успешно применять его при создании крупных проектов.

Basik (Бейсик). Для этого языка имеются и компиляторы, и интерпретаторы, а по популярности он занимает первое место в мире. Он создавался в 60-х годах в качестве учебного языка и очень прост в изучении.

С (Си). Данный язык был создан в лаборатории Bell и первоначально не рассматривался как массовый. Он планировался для замены ассемблера, чтобы иметь возможность создавать столь же эффективные и компактные программы и в то же время не зависеть от конкретного типа процессора.

Си во многом похож на Паскаль и имеет дополнительные средства для прямой работы с памятью (указатели). На этом языке в 70-е годы написано множество прикладных и системных программ и ряд известных операционных систем (11пгх).

С++ (Си ++). Си ++ - это объектно-ориентированное расширение языка Си, созданное Бьярном Страуструпом в 1980 году. Множество новых мощных возможностей, позволивших резко повысить производительность программистов, наложилось на унаследованную от языка Си определенную низкоуровневость, в результате чего создание сложных и надежных программ потребовало от разработчиков высокого уровня профессиональной подготовки.

Java (Джава, Ява). Этот язык был создан компанией 5ип в начале 90-х годов на основе Си ++. Он призван упростить разработку приложений на основе Си ++ путем исключения из него всех низкоуровневых возможностей. Но главная особенность этого языка - компиляция не в машинный код, а в платформно-независимый байт-код (каждая команда занимает один байт). Этот байт-код может выполняться с помощью интерпретатора – виртуальной java- машины JVM(Java Virtual Machine), версии которой созданы сегодня для любых платформ. Благодаря наличию множества Java-машин программы на Java можно переносить не только на уровне исходных текстов, но и на уровне двоичного байт-кода, поэтому по популярности язык Ява сегодня занимает второе место в мире после Бейсика.

Особое внимание в развитии этого языка уделяется двум направлениям: поддержке всевозможных мобильных устройств и микрокомпьютеров, встраиваемых в бытовую технику (технология Jini) и созданию платформно-независимых программных модулей, способных работать на серверах в глобальных и локальных сетях с различными операционными системами (технология Java Beans). Пока основной недостаток этого языка - невысокое быстродействие, так как язык Ява интерпретируемый.

C# (Си Шарп). В конце 90-х годов в компании Microsoft, руководством Андерса Хейльсберга был разработан язык C#. В нем воплотились лучшие идеи Си и Си ++, а также достоинства Java. Правда, C#, как и другие технологии Microsoft, ориентирован на платформу Windows. Однако формально он не отличается от прочих универсальных языков, а корпорация даже планирует его стандартизацию. Язык C# предназначен для быстрой разработки. NET-приложений, и его реализация в системе Microsoft Visual Studio. NET содержит множество особенностей, привязывающих C# к внутренней архитектуре Windows и платформы .NET.

Лекция 8. Системы моделирования электрических схем

В процессе разработки какого – либо устройства электронной техники наряду с задачей синтеза принципиальной схемы устройства не менее актуальна задача анализа его электрических характеристик. Существенную помощь в проведении такого рода анализа могут оказать специализированные САПР. Все схемотехнические САПР можно разделить по принципу их организации на две большие группы: модульные и интегрированные. Для модульного построения САПР характерно наличие специализированных программных модулей, выполняющих некоторые функции, например создание принципиальной схемы, ее анализ и вывод результатов анализа в виде графиков. Такая организация САПР удобна для работы больших коллективов разработчиков. К таким пакетам относятся системы PSPICE и более современная версия – MicroSim.

Рассматриваемый в настоящем пособии пакет семейства Micro-Cap относится к интегрированным САПР, совмещая все основные функции в единой программной оболочке, и обладает следующими функциональными возможностями:

производиться расчет временных процессов в цепях, расчет цепей по постоянному и переменному току, анализ спектрального состава полученных сигналов.

реализован учет случайного разброса параметров компонентов по методу Монте-Карло при проведении всех видов анализа характеристик цепей;

возможен многовариантный анализ.

В пакете используются унифицированные математические модели полупроводниковых компонентов (диоды, биполярные, полевые и МОП - транзисторы, операционные усилители) с широко распространенной программой PSPICE.

В состав библиотеки компонентов включены модели пользователей, в частности, электрические макромодели цифровых интегральных схем;

Кроме того, существуют дополнительные сервисные возможности:

1) введен графический редактор изображений компонентов, с помощью которого можно удовлетворить требованиям ГОСТ;

2) предусмотрена возможность преобразования задание на моделирования в формат входного языка программы PSPICE.

Создание электрической принципиальной схемы

Создание схемы производиться в следующей последовательности:

1.Создается новый файл с расширением cir или загружается для редактирования уже существующий с помощью последовательности команд меню:

File (работа с файлами):

1: Create new circuit (создание новой схемы);

2: Load circuit (загрузка схемы);

2. Добавляются при необходимости новые компоненты.

Для этого в окне инструментов выбирается позиция ADD и COMP, а в окне компонентов выбирается нужный компонент (список приведен в приложении). Для фиксации выбранного компонента на рабочем поле достаточно щелкнуть левой кнопкой мыши «мыши» на нужном месте схемы. После этого автоматически будет предложено выбрать из списка тип активного компонента или обозначение (Label) пассивного компонента. Этот параметр является обязательным и при отказе от его ввода компонент автоматически удаляется. В случае если нет подходящих обозначений, то можно выбрать в списке позицию Own (OtherwiseName) и ввести собственное обозначение.

Внимание. Поскольку при анализе схемы программное обеспечение использует метод узловых потенциалов [1], то **обязательно** требуется заземление одного из узлов схемы. Для этого в схему вводиться компонент **GROUND**.

3. Редактируется положение введенных компонентов с помощью команд MOVE и COMP. При этом могут быть использованы следующие клавиши управления:

End - перемещение курсора к концу строки,

Home - перемещение курсора к началу строки,

Ctrl/End - удаление части строки от текущего положения курсора до конца строки,

Ctrl/Ins - копирование выделенного текста в буфер,

Shift/Ins - перемещение текста из буфера на экран, начиная с текущего положения курсора,

Ctrl/Y - удаление строки,

Shift/Backspace - отмена последнего изменения схемы или текста,

Tab - перевод курсора в виде мерцающей линии на следующую позицию.

Space - поворот выделенного объекта на 90 градусов.

4. Вводятся линии электрических соединений, при выборе команд ADD и LINE. Рисование линии начинается в точке, указанной при нажатии левой кнопки мыши, а заканчивается в точке, указанной при нажатии правой кнопки.

Удаление объектов производится по команде Zap, для чего курсор подводится к удаляемому объекту и нажимается левая клавиша "мыши". Перемещение объектов производится по команде Move, причем связи между элементами схемы при этом разрываются.

5. При необходимости производится переопределение обозначений компонентов с помощью команд DEF и COMP.

6. При необходимости производится сопоставление обозначений компонентов и их параметров с помощью команд INFO и COMP. В простейшем случае указывается номинал пассивного компонента. При определении параметров активных компонентов запрашивается достаточно большое количество параметров, и этот диалог будет рассмотрен позже.

7. Сохраняется текущая схема с помощью команд:

File (работа с файлами):

4: Save circuit (сохранение схемы);

Анализ принципиальных схем в пакете Micro-Cap.

В более новой версии пакета Micro-Cap V под Windows введены некоторые изменения, в частности, объединены в единое окно запроса запросы в режиме Limits и Monitor. Теперь достаточно вызвать режим Limits для задания всех параметров. Кроме того, введена позиция P, определяющая в каком окне будет выводиться тот или иной график. В предыдущей версии разделение по окнам происходило поочередно из расчета два графика на окно вывода. Несколько изменился формат задания номеров узлов в позиции Waveform. Теперь здесь задается тип графика, а затем в круглых скобках – номер узла или его название. К дополнительным удобствам также можно отнести возможность автоматического масштабирования графиков.

Создание электрической принципиальной схемы

При создании принципиальной схемы используется набор инструментов, расположенных в левой части первой строки панели управления, см. рис. 3.

Перед началом работы может быть установлен режим нанесения координатной сетки и курсора в виде перекрестия.

Для добавления компонента в схему нужно выполнить следующие действия: выбрать на панели управления режим добавления компонентов, а затем в верхнем меню Component выбрать название нужного компонента. После этого достаточно «щелкнуть» указателем «мыши» в нужном месте и компонент будет установлен. При установке простых компонентов, таких как резисторы, конденсаторы, сразу же будет запрошено значение основного параметра Value. Для более сложных компонентов, параметры которых определены их названием, значение основных параметров может быть

отредактировано при нажатии кнопки I в панели инструментов и выборе интересующего компонента.

Соединение компонентов линиями электрической связи осуществляется в режиме разводки проводников (ортогональном или диагональном). В этом режиме рисование соединительной линии начинается при нажатии левой кнопки «мыши» и заканчивается при ее отпускании.

Перемещение и вращение компонентов и соединительных линий производиться в режиме выбора объекта. В этом режиме выбранный объект выделяется цветом и при нажатой левой кнопки «мыши» перемещается вслед за курсором. Кроме того, нажатие правой кнопки мыши при удержании левой приводит к повороту выбранного объекта на 90 градусов.

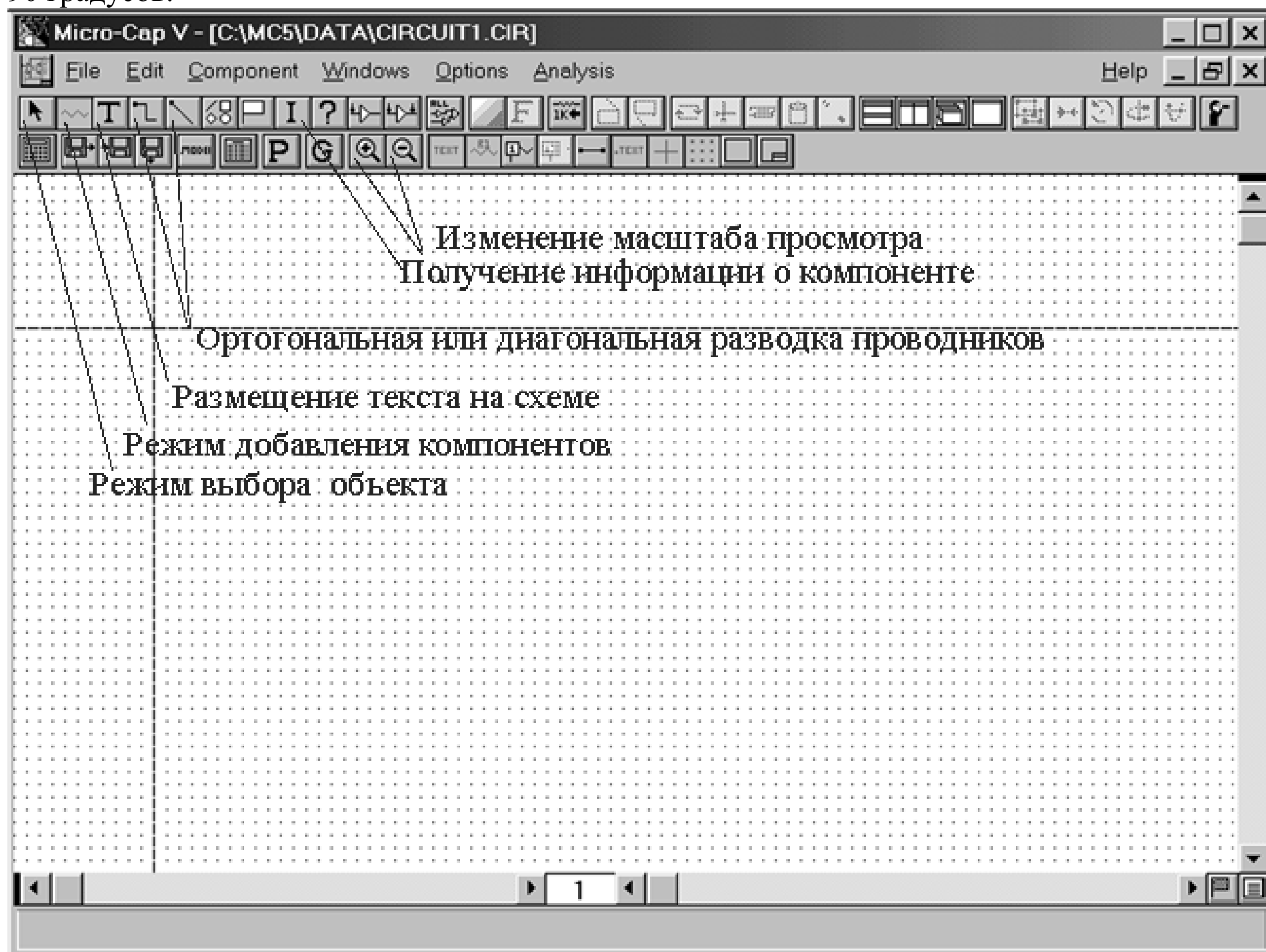


Рис. 1. Рабочее пространство пакета Micro-Cap V.

Расчет цепи

Для расчета созданной принципиальной схемы вызывается режим ***Run*** через одноименную позицию меню. В этом режиме доступны следующие подрежимы:

1: Transient (переходные процессы)

2: AC (характеристики по переменному току)

3: DC (режим по постоянному току)

4: Fourier (спектральный анализ по Фурье)

5: Quit analysis, F3 (выход из режима)

В каждом из этих подрежимов имеется список дополнительных возможностей (их перечень зависит от выбранного подрежима).

Расчет цепи во временной области (Transient)

Это наиболее часто используемый режим расчета. Он служит для анализа процессов протекающих в электрических цепях во временной области, то есть осью абцисс на анализируемых графиках является время. В качестве оси ординат в данном

режиме могут выступать ток, напряжение или активная мощность на заданном компоненте схемы.

Для удобства работы можно установить режим просмотра номеров узлов на схеме через позицию меню: ***View (изображение схемы):***7: Show nod number (вывод на экран номеров узлов). Информация о номерах узлов потребуется в дальнейшем для задания режима просмотра результатов расчета. Для удобства работы узлам можно присвоить имена. Для этого после выбора позиций ADD и TEXT щелкнуть указателем мыши по нужному узлу схемы (рядом с номером) и ввести текстовое обозначение узла.

Внимание. Каждому соединению компонентов на схеме должен соответствовать только один номер узла. Наличие нескольких номеров говорит об отсутствии электрического соединения между этими компонентами (иногда визуально не заметному). В этом случае следует откорректировать электрическую цепь с помощью команд ADD LINE или MOVE LINE.

После запуска режима Transient выдается запрос на задание временных пределов расчета – режим Limits(быстрый вызов режима Transient по клавише F9). Здесь нужно определить следующие параметры:

Simulation time - конечное, начальное время интегрирование и максимальный шаг; формат Tmax[/Tmin/Timestep]

Display time - интервал времени, отображаемый на графиках; формат Dmax[/Dmin[/Printstep]]

Maximum change, % - максимальное относительное изменение графика переменных между двумя точками (обычно 0,5-5 %)

Convergence criteria - максимальная допустимая относительная ошибка на каждом шаге интегрирования (обычно 10E-3- 10E-6)

Temperature - диапазон изменения температуры; формат High[/Low/Step]

Minimum timestep - минимально допустимый шаг интегрирования (обычно 10E-12-10E-15)

Maximum voltage error - максимально допустимая абсолютная ошибка расчета напряжений

Maximum current error - Максимально допустимая абсолютная ошибка расчета тока.

Далее системе следует указать, для каких узлов будет производиться вывод графиков. Это делается с помощью опции Monitor (быстрый вызов из режима Transient по клавише F12). В позиции Type указывается тип сигнала: V – напряжение, I – ток, P - мощность, E - энергия или U - пользовательская функция. В зависимости от содержания этой позиции и позиции Waveform будут построены графики в соответствии с таблицей 1.

Таблица 1

Задание типа выводимого графика

Type	Wafeform	Тип графика
V	УзелA	Напряжение
I	УзелA /УзелB	Ток
V	УзелA /УзелB	Разность напряжений
P	УзелA /УзелB/УзелC	Ток AB*Напряжение BC
E	УзелA /УзелB/УзелC	Интеграл (Ток AB*Напряжение BC) dt
Любой тип	Имя узла, например IN	График для именованного узла
U	Любая вычисляемая функция от времени (T), например:IN*sin(T)	Функция пользователя

Позиция Scale Rang определяет диапазон отображения графиков по оси ординат. Обязательно должен быть задан верхний предел, при необходимости через "/" задается нижний предел изображения.

Позиция Plot позволяет разрешить или запретить вывод на экран того или иного графика, определенного в соответствии с таблицей 1.

Позиция User определяет вывод данного графика в служебный файл с расширением tsa для повторного использования результатов расчета. Для этого необходимо, чтобы в Transient options был установлен режим расчета с сохранением результатов Save.

Позиция Out направляет результаты расчета на принтер или в файл. При этом в Transient options должен быть установлен флажок Numeric output в состояние Enable. Устройство, на которое направляется вывод –файл, экран или принтер - определяется установкой опций в **Options:**6:Output.

Позиция Format определяет формат вывода числовых данных. Здесь указывается число цифр до запятой, а затем через точку – число цифр после запятой.

После установки всех параметров производится запуск программы расчета цепи командой Transient\Run (быстрый вызов F2).

Расчет цепи во временной области (Transient)

В этом режиме в качестве выражения для X Expression задается величина времени. Задание диапазона расчета и наблюдаемых сигналов производиться в одном совмещенном окне рис.4.

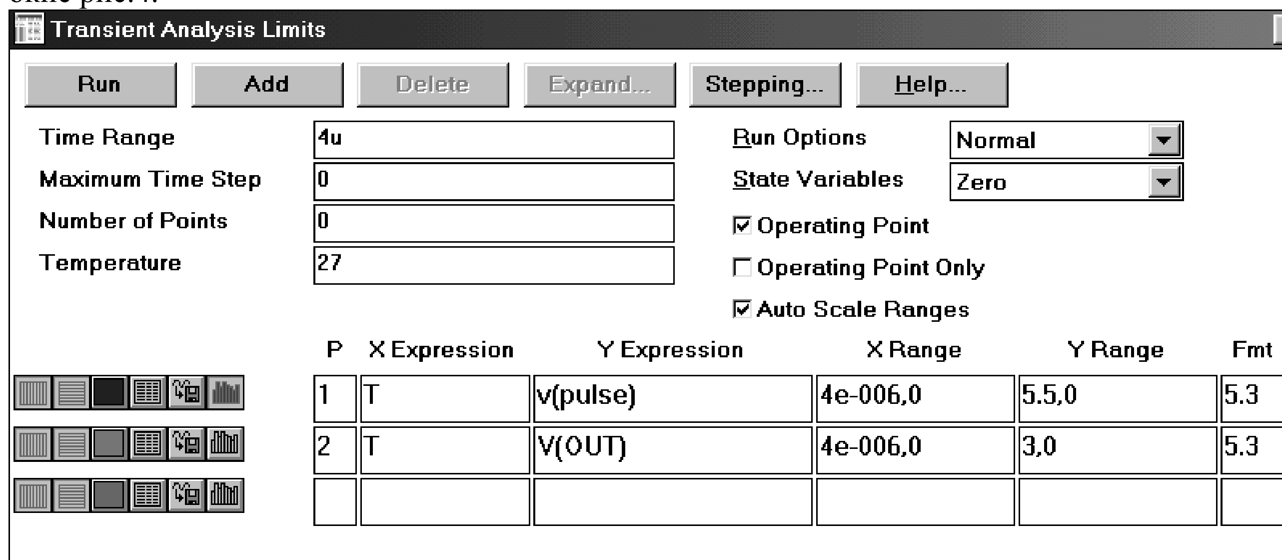


Рис. 2. Задание параметров расчета переходного процесса

Максимальное время расчета задается в графе TimeRange, а интервал времени отображаемого на графике может быть задан произвольным в позиции X Range или равным максимальному времени расчета при установке флага Auto Scale Ranges. Установка этого флага также определяет автоматическое масштабирование графиков и по координате Y, то есть значение Y Range.

Цифра от 1 до 9 в позиции P определяет, в какой системе координат будет выводиться график. Если в каких либо строках стоят одинаковые цифры в этих позициях, то эти графики будут выводиться в одной системе координат. Отсутствие цифры в этой позиции приводит к подавлению вывода данного графика. Общее число описаний выводимых на экран величин может изменяться с помощью кнопок Add и Delete. Первые две графические кнопки в левой части рисунка определяют вид шкалы (линейная или логарифмическая) для оси X и Y соответственно.

В качестве примера рассмотрим процессы, происходящие в интегрирующей цепочке рис.5.

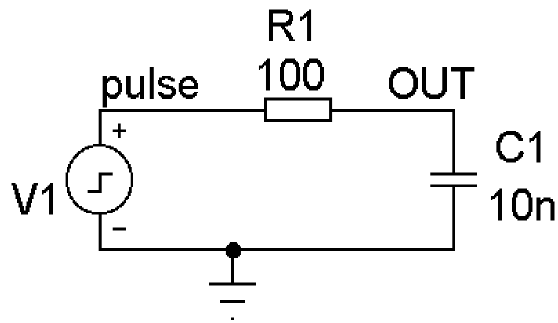


Рис. 3. Схема для анализа переходного процесса

Результат расчета такой цепи приведен на рис.6. Здесь согласно запросу рис выводиться в различных системах координат графики входного напряжения и напряжения на конденсаторе. Масштабирование графиков произведено автоматически.

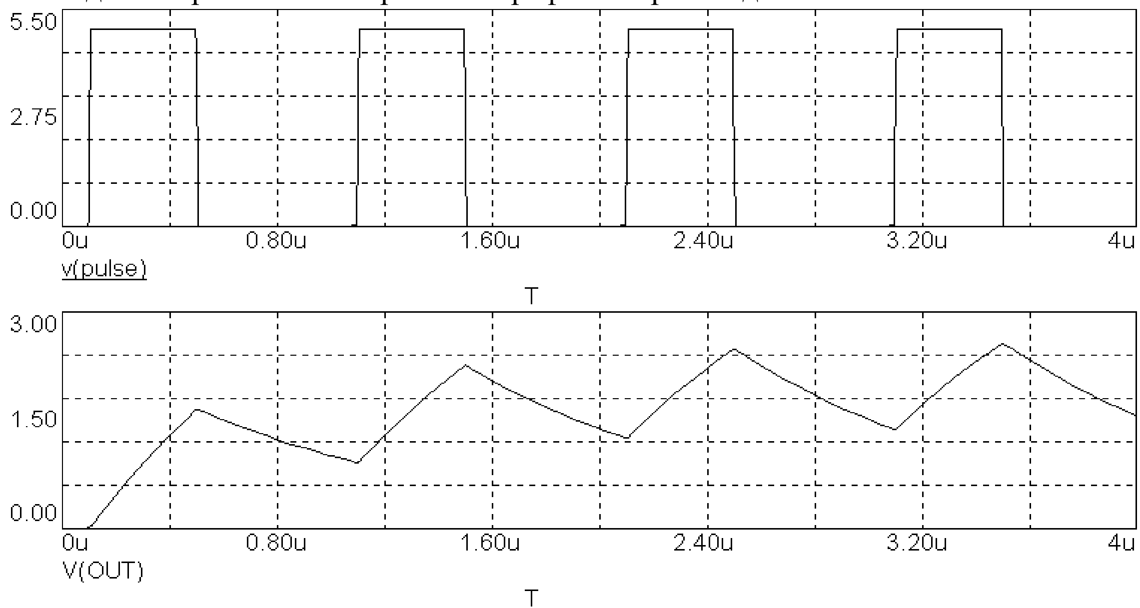


Рис. 4. Результат расчета переходного процесса

Лекция 9. Математические модели отдельных компонент схемы

Структура модели компонента

Каждый сложный компонент характеризуется некоторым набором информации заключенном в различных служебных библиотеках.

Библиотека схемных образов

Данная библиотека содержит графические формы, используемые при построении схемы. Каждая форма составлена из различных графических примитивов – линий, прямоугольников и т.д. Информация о графических формах хранится в файле Shape.MC9. Этот файл должен находиться в том же самом каталоге, что и MC9.EXE. Схемный образ компонента может быть изменен с помощью редактора Shape editor.

Библиотека компонентов

Хранит информацию об имени каждого компонента, имени схемного образа из библиотеки Shape, имени используемой модели Definition, а также о размещении и типе входных и выходных контактов описанных в модели, размещении текстового обозначения компонента. Все компоненты, от резисторов до макрокоманд и подсхем при создании схемы выбираются из библиотеки компонентов. Вся перечисленная информация

храниться в файле COMP.MC5. Он также должен находиться в том же самом каталоге, что и MC5.EXE. Данная библиотека может быть отредактирована с помощью редактора Component Editor.

Библиотека организована в четыре основных группы:

Аналоговые примитивы

Библиотека аналоговых компонентов

Цифровые примитивы

Библиотека цифровых компонентов

Компоненты в аналоговых примитивах и цифровых примитивах требуют, чтобы пользователь выбрал модельное имя, или для простых компонентов подобно резистору, значение.

Компоненты в библиотеках аналоговых и цифровых компонентов не требуют выбора модельного имени, поскольку имя модели совпадает с именем компонента. Список доступных библиотек храниться в файле NOM.LIB.

Библиотека моделей

Хранит информацию об электрических параметрах моделируемого компонента для любого типа. Модели могут быть представлены в одной из четырех форм: модельные таблицы параметров, определители моделей, подсхемы или макросы.

Формат описания модели компонента.

Micro-Cap обрабатывает два базисных формата: Schematics - схемные образы и SPICE text files – текстовые файлы SPICE.

Схемные образы состоят из рисунков и текста. Рисунки обеспечивают систему информацией о составе и взаимосвязях компонентов, а текст содержит информацию необходимую для моделирования

В этом режиме рабочее пространство содержит область рисования (рисунка) и текстовую область.

Область рисунка содержит одну или большее количество страниц аналоговых и цифровых компонентов, взаимосвязанных проводами. Они могут также содержать графические объекты или текст, которые не содержат никакую электрическую информацию, и текст, содержащий модельную информацию.

Текстовые области содержат только текст. Они обычно используются для описания моделей, которые могут быть слишком большие для описания дисплея в области рисования.

Дисплей может переключаться между текстовой областью и областью рисования по нажатию клавиш CTRL + G или нажимая на значок переключателя расположенного в нижнем правом углу рабочего пространства.

Текстовые файлы SPICE – стандартное текстовое описание схемы в SPICE формате.

Текстовые файлы вводятся и редактируются в любом текстовом редакторе. Для этих целей может использоваться встроенный редактор, который активизируется при создании нового описания командой New/Spice или при редактировании существующего описания по команде Options/Mode/Info (или значок I на панели управления). При описании модели в Spice формате вся информация должна располагаться в одну строку. Если это оказывается не удобным можно переносить запись на следующую строку, начиная ее со знака «+».

При изменении описания модели содержимое файла контролируется только в момент расчета схемы использующей этот компонент.

Формы представления модельной информации.

Модельные таблицы параметров (Model parameter lists)

Они представляют собой табличное описание модельных параметров компонентов. Списки сохранены в двоичных файлах с расширением 'LBR'. Эти файлы могут просматриваться и редактироваться только редактором моделей, вызываемого командой

Mode/Info. В такой форме храниться, главным образом, информация для компонентов из аналоговой библиотеки. Исключение составляет - раздел Vendor, который содержит нетипичные модели микросхем, поставляемые различными фирмами, описанные в текстовых файлах, использующих расширение LIB.

Определители моделей (Model statements)

Определители содержат информацию о название (имени) компонента, его типе, а также модельные параметры. Они сохранены в текстовых файлах, использующих расширение 'LIB', и используются, прежде всего, как часть описания модели в форме подсхемы (SUBCKT), а также для описания аналоговых компонентов.

Формат описания:

.MODEL <Модельное имя> [AKO:<имя модели прототипа>] <тип модели> ([<Название параметра>=<значение>] [LOT=<tol1>[%]] [DEV=tol2[%]]).

Пример описания:

.MODEL 2N2222 NPN(BF=150 IS=1E-14)

.model M1 NMOS(KP=2.5E-5 VTO=1.345 CGDO=14.67p+ CGBO=4.566p)

.MODEL DK D(IS=1E-18)

В этом формате достаточно широко используется оператор AKO, с помощью которого можно клонировать новые модели на основе существующих прототипов. Все параметры клонированного компонента идентичны параметрам прототипа, кроме переопределенных при создании новой модели. Это свойство особенно удобно при описании транзисторов, отличающихся несколькими параметрами, например транзистор с буквой A в конце обозначения можно описать на основе базовой модели с тем же названием. Причем новая модель будет отличаться только коэффициентом усиления: .model 2N2222A AKO:2N2222 NPN (BF=55).

Допустимое отклонение того или иного параметра определяется размещением ключевого слова после этого параметра. Причем может быть задан допуск для всех компонентов в партии (LOT) и разброс параметров в зависимости от номера компонента в пределах партии (DEV).

Например, при описании следующей модели: .MODEL 2N2241 NPN (BF=100 LOT=20% DEV=1%) в наихудшем случае любой транзистор может иметь величину коэффициента усиления в диапазоне от 80 до 120:

BF = 80 = 100 - .□□□ □ □

BF = 120 = 100 + .2□ 100.

В то же время для каждого вновь устанавливаемого в схему такого транзистора этот диапазон также может изменяться на 1%:

79 = 80 - .01□ 100

81 = 80 + .01□ 100.

Подсхемы (SUBCKT)

Содержат текстовое описание, эквивалентное электрической схеме компонента. Подсхемы хранятся в текстовых файлах, использующих расширение, 'LIB'. Вся цифровая библиотека выполнена с помощью подсхем. Описание вызываются для редактирования по команде Options/Mode/Info или Open/Library. По второй команде доступна для редактирования вся библиотека (например, ТТЛ микросхем).

Чаще всего подсхемы описываются на основании цифровых примитивов. Так, например ТТЛ микросхема, содержащая 4 элемента 2И-НЕ, может быть описана с помощью модели цифрового примитива И-НЕ(nand):

*Заголовок

* DIGITAL LIBRARY 7400-

* Quad 2-Input Nand Gates

.SUBCKT 7400 1A 1B 1Y

+optional: DPWR=$G_DPWR DGND=$G_DGND

+params: MNTYMXDLY=0 IO_LEVEL=0

```
*Описание цифрового примитива.
U1 nand(2) DPWR DGND
+1A 1B 1Y
+DLY_00 IO_STD MNTYMXDLY={MNTYMXDLY} IO_LEVEL={IO_LEVEL}
*Описание нестандартной модели.
.model DLY_00 ugate (tplhTY=11ns tplhMX=22ns tphlTY=7ns tphlMX=15ns)
.ENDS  7400
```

В данном примере описания можно выделить три раздела: заголовок, описание примитива и описание нестандартной временной модели. В заголовке после служебного слова SUBCKT указывается уникальное имя, создаваемой модели, и в качестве параметров передаются значения задержки распространения сигнала и уровень работы интерфейса ввода-вывода.

Цифровой примитив описывается в соответствии со своим форматом. В данном примере при описании примитива использована стандартная модель интерфейса ввода-вывода IO_STD и не стандартная временная модель DLY_00, параметры которой описываются в третьем разделе.

Для того чтобы описать пользовательскую модель в формате SUBCKT необходимо выполнить следующие действия:

Создать схемный образ создаваемого компонента с помощью встроенного редактора Shape editor.

Указать атрибуты создаваемого компонента с помощью встроенного редактора Component editor.

Для этого рекомендуется командой Add grup создать новую группу компонентов, например User. Указать имя создаваемого компонента, имя схемного образа и тип модели (в данном случае SUBCKT). Также здесь следует указать расположение и наименование входных и выходных узлов на изображении схемного образа и их тип (цифровые или аналоговые).

Создать текстовое описание модели в формате SUBCKT. Для этого рекомендуется создать командой New\Spice/Text новую библиотеку с расширением lib, например “User.lib”. С помощью встроенного или внешнего текстового редактора в этом файле необходимо описать модель. В большинстве случаев описание модели состоит из 3 разделов: заголовка, описания примитива и описания нестандартных моделей, используемых при описании примитива.

Указать имя созданной библиотеки в файле nom.lib в формате <.lib “имя библиотеки”>. После этого созданная модель будет помещена в индекс существующих компонентов и доступна для использования. Рекомендуется указывать имя пользовательской библиотеки в первой строке. В этом случае при индексации в случае совпадения имен моделей пользовательская модель будет обладать более высоким приоритетом.

Макросы

Макросы представляют собой части схем, оформленные в виде функционально законченных блоков. Они сохраняются в схематических файлах, с расширением 'CIR'. В этом формате выполнены некоторые части аналоговых компонентов. Использование макросов удобно в том случае, если в разрабатываемой схеме многократно повторяется какой либо функционально законченный блок. В этом случае имеет смысл оформить его в виде макроса и использовать наравне с уже существующими компонентами.

Чтобы создать макрос необходимо выполнить следующие действия:

1.Создать схему. Разместить на схеме имена узлов, которые в последствии должны быть доступны в создаваемом макросе в качестве внешних выводов. Сохранить схему на диск, используя в качестве имени файла имя создаваемого макроса.

2.Ввести новый компонент-макрос в библиотеку компонентов следующим образом:

Вызвать встроенный редактор командой Windows/Component Editor.

Ввести имя файла, созданного в пункте 1, в поле Name.

В позиции Shape выбрать схемный образ, соответствующий макросу. Схемный образ при этом должен быть заранее создан с помощью встроенного графического редактора, вызываемого командой Windows/Shape Editor.

Выбрать в поле Defenition значение macro, соответствующее описанию модели компонента в форме макроса.

Разместить на изображении макроса выводы, для подключения его в электрическую цепь. Имена и тип выводов (цифровой или аналоговый) должны совпадать с именем и типом узлов в схеме созданной в пункте 1.

Для того чтобы создать макрос, реализующий функции цифрового одновибратора [2] необходимо создать схему, приведенную на рис.18. В составе данного макроса используется цифровая линия задержки, которая определяет длительность выходного импульса. Начало формирования импульса определяется приходом импульса запуска на вход Cloc.

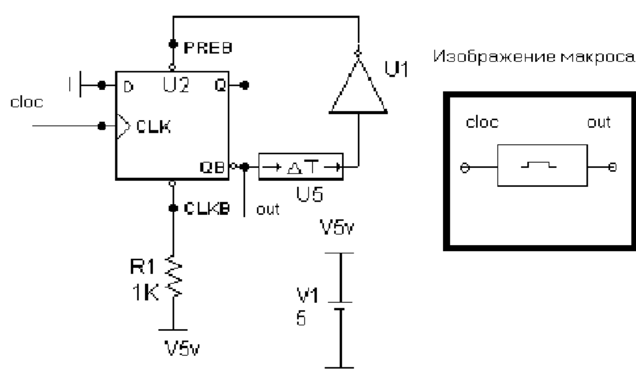


Рис. 5. Исходная схема и изображение макроса

На схеме обозначить входной узел Cloc и выходной out. Сохраним схему в файле под произвольным уникальным именем, например Dmono. Создать схемный образ макроса (на рисунке приведен в правой части) и сохранить его под тем же именем. Следует заметить, что в Component Editor **не нужно** указывать имена входных и выходных выводов. Имена выводов и их тип следует указать при описание компонента в редакторе Component Editor. В данном случае оба вывода являются цифровыми. После описания макроса он становится доступен для использования наравне с другими компонентами.

Лекция 10. Управляющие программы АСУ

Существуют четыре уровня автоматизации при программировании обработки для станков с ЧПУ: 1 уровень – подготовка управляющих программ вручную с применением настольных или карманных калькуляторов и устройств подготовки данных на перфоленте. 2 уровень – использование ЭВМ для обработки некоторых задач, в основном, расчетно-вычислительного характера. 3 уровень – обработка на ЭВМ отдельных переходов. 4 уровень – разработка с помощью ЭВМ оперативного ТП и всех этапов ТП. 2 и 4 уровень автоматизации и подготовки управляющих программ соответствует методика, реализуемая системой автоматического программирования, использующая определенный программно-ориентированный язык программирования. Для второго уровня автоматизации характерно решение следующих задач: 1) расчет координат опорных точек, 2) преобразование систем координат, 3) формирование элементарных перемещений, 4) определение технологических команд, 5) кодирование

управляющей программы, 6) запись управляющей программы на носитель, 7) контроль программоносителя, 8) контроль траектории инструмента, 9) редактирование управляющей программы.
Методика подготовки управляющей программы, включая ее редактирование, зависит от типа устройства ЧПУ станка, условий производства, организационных принципов работы станка с ЧПУ. Обработка деталей заданной конфигурации на станке с ЧПУ обеспечивается перемещением инструмента по траектории, необходимой для получения заданного контура детали. Контур детали можно представить в общем случае состоящим из отдельных отрезков прямых, дуг окружностей и кривых высших порядков. Отдельные геометрические элементы соединяются между собой пересечением или касанием. Точки конца одного геометрического элемента и начало другого называются узловыми, или опорными точками. Информация о перемещении инструмента задается в программе для устройства ЧПУ в виде координат опорных точек. Эти координаты могут быть заданы в абсолютной системе координат, связанной с нулевой точкой станка, или в виде приращений координат конечных точек геометрических элементов контура относительно начальных.
Программа обработки детали описывает траекторию движения опорной точки инструмента, называемой центром инструмента. Траектория центра инструмента эквидистантна контуру обрабатываемой детали. В программе обработки детали целесообразно задавать координаты перемещения центра инструмента, т.е. необходимо уметь рассчитывать эквидистантный контур. Расчет координат опорных точек эквидистантного контура производится на основании координат опорных точек контура детали, которые определяются из чертежа детали. Координаты опорных точек эквидистантной траектории инструмента наиболее просто представить как геометрическое место точек, равноудаленных от контура детали на расстояние, равное радиусу инструмента, строится справа или слева от элементов этого контура в зависимости от элементов этого контура в зависимости от расположения инструмента относительно обрабатываемого контура.
Методика соединения элементов эквидистанты выбирают в зависимости от угла, образованного соседними элементами контура, если смотреть со стороны инструмента при обходе этого контура. Этот угол для пары отрезков измеряют непосредственно между ними. Если же элементом контура является дуга окружности, то угол измеряют относительно касательной к этой дуге в общей точке рассматриваемой пары элементов контура детали.

Лекция 11. Обрабатывающие программы АСУ

Обрабатывающие программы включают в себя систему автоматизации программирования и обслуживающие программы.

Функции системы автоматизации программирования следующие: запись программ на входных языках программирования; трансляции программ на внутренний язык ЭВМ; объединение (сборка) нужных конфигураций (сегментов) из стандартных подпрограмм; отладка программ на уровне входных языков; корректировка программ на уровне входных языков.

Основными задачами обслуживающих программ являются следующие: запись программ в библиотеку; исключение программ из библиотеки; перезапись программ с одного магнитного носителя на другой, печать и вывод программ на перфоносители; вызов нужных программ в процессе работы в оперативную память и настройка ее по месту размещения.

Основными компонентами МО АСУ являются системная диспетчерская программа и библиотека стандартных подпрограмм и типовых программ, предназначенных для обработки производственно-экономической информации.

Системная диспетчерская программа обеспечивает функционирование АСУП в режиме, определенном производственно-хозяйственной или административной деятельностью.

Библиотека стандартных подпрограмм, имеющаяся в МО ЭВМ, является переходной ступенью к разработке системной библиотеки, ориентированной на процессы обработки информации в АСУ. *Системная библиотека* должна содержать:

программы ввода и преобразования в машинную форму документов и других письменных источников исходных данных;

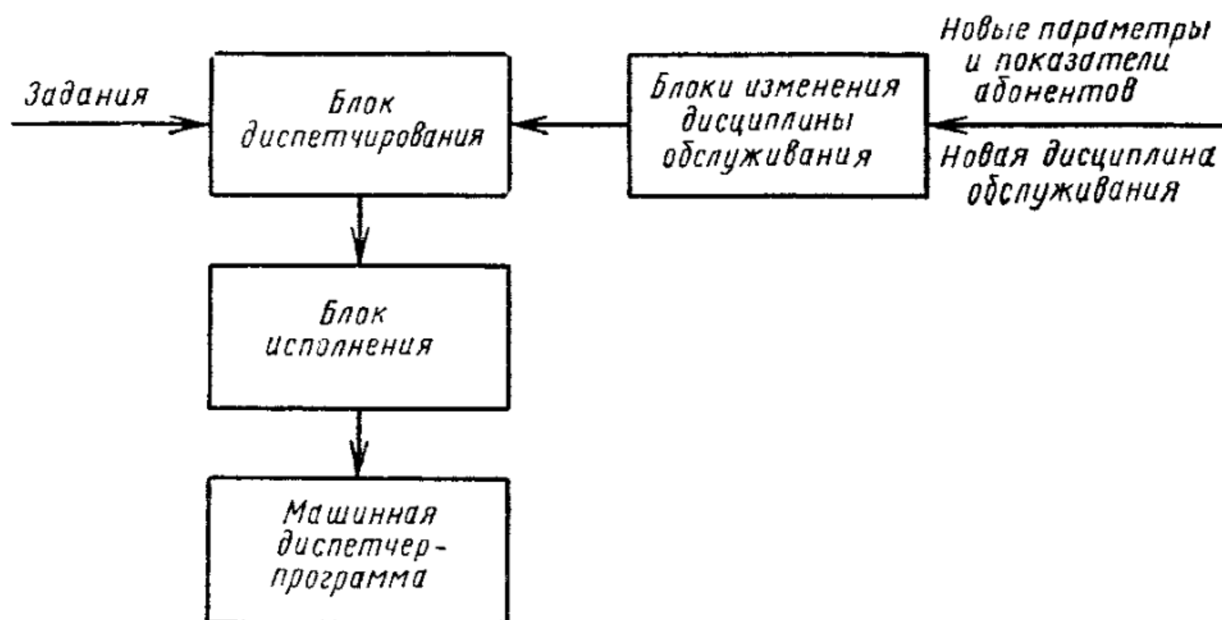
программы для организации машинных массивов, характеризуемых как большими объемами, так и сложностью их структуры, для эффективного поиска и извлечения требуемых данных из массивов;

программы для преобразования данных в наиболее приемлемую для человека форму (в виде графиков, схем, изображений) и вывода их на внешние устройства.

Лекция 12. Системная диспетчерская программа АСУ

Системная диспетчерская программа, как и машинная, должна располагаться в оперативной памяти ЭВМ. Однако ограниченные возможности последней (по объему) не позволяют постоянно хранить в ней как всю машинную, так и общесистемную диспетчерскую программу. Поэтому в оперативной памяти постоянно содержится лишь часть диспетчерской программы, называемой резидентом.

Системная диспетчерская программа обеспечивает функционирование АСУП в режиме, определенном производственно-хозяйственной или административной деятельностью.



Системная диспетчерская программа предназначается для организации потоков данных и обработки их в последовательности и ритме, определяемым производственно-хозяйственной деятельностью объекта управления. Под процессом обработки данных

следует понимать всю совокупность операций, выполняемых с данными в системе: обработку, сбор, передачу, хранение, распределение и обмен данными с внешним окружением.

Разрабатывается системная диспетчерская программа на основе информационной модели и установленного режима функционирования конкретной системы. Задача эта чрезвычайно сложна и в значительной степени зависит от возможностей операционной системы конкретной ЭВМ. Комплекс системных диспетчерских программ (операционная система АСУ) предназначен для обеспечения функционирования АСУ в режиме, соответствующем ритму производственно-хозяйственной деятельности системы. В соответствии с этим основное назначение системных диспетчерских программ состоит в регулярном контроле календарных графиков решения задач АСУ, учете поступающих от абонентов сообщений и дополнительных заданий и в обеспечении их эффективной реализации.

Основным критерием качества функционирования *системной диспетчерской программы* является суммарное время запаздывания решения задач АСУ с учетом их приоритета.

Основными компонентами МО АСУ являются *системная диспетчерская программа* и библиотека стандартных подпрограмм и типовых программ, предназначенных для обработки производственно-экономической информации. [

После приема сообщений и заявок на решение задач, поступающих от абонента по каналам связи, *системная диспетчерская программа* размещает поступившую информацию в памяти системы, заносит сведения о ней в таблицу очередности и извещает абонента о приеме или невозможности приема информации. При этом структура содержательной части поступающей информации должна полностью соответствовать структуре информационного обеспечения; АСУ.

Оно содержит программы типовых процессов обработки данных в АСУ (ввода, контроля, сортировки, корректировки, дублирования, поиска и вывода информации), программы решения конкретных задач АСУП и*системную диспетчерскую программу*.

Оно предоставляет пользователю спектр приемов и процедур для программирования задач и работы на ЭВМ применительно к АСУ и, таким образом, представляет собой операционную систему. Основными частями ОСМО являются *системная диспетчерская программа* и библиотека стандартных подпрограмм и типовых программ, предназначенных для обработки производственно-экономической информации. Инструментом программирования ОСМО является машинное МО, в первую очередь это относится к системе автоматизации программирования и обслуживающим программам. Библиотека стандартных подпрограмм, имеющаяся в машинном МО, является переходной ступенью к развитию и разработке системной библиотеки, ориентированной на процессы обработки информации в АСУ.

Комплекс системных диспетчерских программ (операционная система АСУ) предназначен для обеспечения функционирования АСУ в режиме, соответствующем ритму производственно-хозяйственной деятельности системы. В соответствии с этим основное назначение *системных диспетчерских программ* состоит в регулярном контроле календарных графиков решения задач АСУ, учете поступающих от абонентов сообщений и дополнительных заданий и в обеспечении их эффективной реализации.

Условия расчета внутреннего приоритета задаются и реализуются *системной диспетчерской программой*.

Параметры заказа–это элементы, внешние по отношению к программе. Они включаются в состав программы перед пуском, задаются программе *системной диспетчерской программой*, результатом предыдущего расчета или просто оператором. Исходя из параметров заказа программа считывает нужные информационные таблицы, описания массивов, осуществляет доступ к массивам и решает задачу.

Системная диспетчер-программа обеспечивает функционирование системы в ритме, определяемом производственно-хозяйственной деятельностью, по расписанию и заданиям, поступающим от абонентов. Периодически с заданным интервалом (несколько минут) просматривается имеющееся расписание решения задач, которые и запускаются в работу, прерывая решение задач, не предъявляющих особых требований к времени решения. В то же время *системная диспетчерская программа* принимает задания от абонентов.

Приоритет решения задачи может быть внешним и внутренним. Внешний приоритет присваивается задачам и сообщениям еще до их поступления в ЭВМ. Он зависит от важности задачи для производственно-хозяйственной деятельности системы и от того, кем является пользователь задачи. Исходя из этого каждая задача, обслуживаемая *системной диспетчерской программой*, должна С0 пр1овождаться сведениями об ее внешнем приоритете, а также о требуемом объеме оперативной памяти и средней длительности обработки.