Министерство образования и науки Российской Федерации

Государственное образовательное учреждение
высшего профессионального образования
«Санкт-Петербургский государственный технологический институт
(технический университет)»

Кафедра систем автоматизированного проектирования и управления

О.В. Проститенко, А.Ю. Рогов

ЛИНГВИСТИЧЕСКОЕ И ПРОГРАММНОЕ ОБЕСПЕЧЕНИЕ АВТОМАТИЗИРОВАННЫХ СИСТЕМ

Учебное пособие
для студентов заочной формы обучения
направления подготовки «Информатика и вычислительная техника»

Санкт-Петербург
2011

УДК 681.3.06

Проститенко О.В. Лингвистическое и программное обеспечение автоматизированных систем: учебное пособие / О.В. Проститенко, А.Ю. Рогов – СПб.: СПбГТИ(ТУ), 2011. – 68 с.

Учебное пособие посвящено изучению разделов специальной учебной дисциплины «Лингвистическое и программное обеспечение автоматизированных систем». В пособии разобраны основные вопросы лингвистического и программного обеспечения автоматизированных систем. Рассмотрены принципы, применяемые при разработке интерфейсов с использованием библиотеки MFC. Понятия и методы этой дисциплины широко используются в различных разделах учебных дисциплин, изучаемых на кафедре САПРиУ. Пособие разработано с использованием материалов сайта www.tenisheff.ru.

Учебное пособие предназначено для студентов 4 курса заочной формы обучения направления подготовки 230100 «Информатика и вычислительная техника» и соответствует рабочей программе дисциплины «Лингвистическое и программное обеспечение автоматизированных систем».

Рис. ___, библиогр. 12 назв.

Рецензенты:

Утверждены на заседании учебно-методической комиссии факультета Информационных технологий и управления _____2011 протокол № ___.

Рекомендовано к изданию РИСо СПбГТИ(ТУ)

СОДЕРЖАНИЕ

1 ЛИНГВИСТИЧЕСКОЕ ОБЕСПЕЧЕНИЕ АВТОМАТИЗИРОВАННЫХ СИСТЕМ

Лингвистическое обеспечение автоматизированных систем (АС) представляет собой совокупность языков, необходимых для автоматизации проектирования. Языки и форматы данных служат для представления информации о проектируемых изделиях на различных этапах процесса проектирования.

Под языком понимается формализованный способ задания информации. Языки могут быть как алгоритмическими, так и декларативными или даже просто свободными. Примерами языков являются такие распространенные языки программирования, как C++, C#, Java, Basic. В то же время языками могут быть табличное описание объекта проектирования или форма диалогового ввода информации.

1.1 Языки проектирования автоматизированных систем

Лингвистическое обеспечение АС можно условно разделить на пять групп языков:

- языки пользователя;
- языки пакетов прикладных программ;
- языки внутреннего представления данных;
- языки машинного архива;
- языки разработчиков АС.

Языки пользователя разделяются на входные и выходные. Входные языки обеспечивают возможность описания формализованных заданий на проектирование, оперативного ввода задания на проектирование, формирования информационной базы типовых элементов проектирования.

В системах автоматизированного проектирования пользовательские языки обычно разрабатывают с учетом специфики предметной области. Это означает, что структура языка, типы данных, операторы и функции соответствуют терминологии предметной области с тем, чтобы упростить описание объекта проектирования. Такие языки называются *проблемно-ориентированными* (domain-specific language — DSL).

Пакеты прикладных программ, решающие конкретные задачи, возникающие в ходе проектирования, представляют собственные языки, на которых должны описываться входные задания. Это могут быть программы расчета прочностных характеристик конструкций, решения систем уравнений или многопараметрического поиска. Каждый из таких пакетов определяет собственный входной язык. При интеграции пакета в систему создается набор конверторов, обеспечивающих преобразование информации системы автоматизированного проектирования в формат, требуемый языком пакета прикладных программ. Такой подход называется «обертыванием» (wrapping): входные и выходные языковые конверторы обеспечивают однообразие интерфейсов при подключении пакетов прикладных программ — разработчикам АС не требуется знания языков представления данных каждого прикладного пакета.

Языки внутреннего представления данных обеспечивают представление информационной модели объекта проектирования в оперативной базе данных САПР. Это языки, с помощью которых представляется рабочая информация об объекте: рабочие массивы, реляционные таблицы, файлы.

Языки машинного архива служат для представления графической и текстовой информации по спроектированному объекту Они обеспечивают единую форму представления документации в архиве, необходимую для ее повторного использования при проектировании и выпуске внешней документации.

Языки разработчиков АС служат для описания структуры САПР, формата баз данных, средств диалогового взаимодействия с системой, средств диагностики, мониторинга пакетов прикладных программ и подсистем САПР. Эти языки служат для формирования САПР как программно-аппаратной системы.

В реальности на каждом этапе, когда САПР имеет место с данными — их загрузкой, обработкой, преобразованием — используется тот или иной вид языкового процессора.

Ввиду большой потребности в языковых процессорах разных видов и особых требований к их надежности и достоверности выдаваемых ими результатов необходимо формализовать их разработку с тем, чтобы обеспечить легкость создания новых языковых процессоров и гарантировать, что все преобразования информации выполняются корректно.

Последнее требование особенно актуально, поскольку использование различных языков описания данных создает огромное количество вариаций, что может привести к ошибкам преобразования в самый неожиданный момент. При этом незаметная для пользователя порча данных при загрузке или преобразовании является самым опасным видом ошибки, поскольку обнаружить ее достаточно сложно, а пользовательские данные и результаты проектирования могут быть испорчены. Использование некорректных результатов может представлять еще большую угрозу в реальном производстве.

Формализация разработки позволит значительно увеличить надежность создаваемых языковых процессоров. Поэтому большое внимание уделяется формальному описанию языков и их распознавания с тем, чтобы можно было математически доказать правильность разрабатываемого программного обеспечения.

1.2 Трансляция языков проектирования

Трансляцией называется перевод программы из одного представления в другое, в частности перевод с одного языка программирования на другой язык.

Трансляцию производит *языковый процессор,* называемый *транслятором.* Транслятор создает функционально-эквивалентное представление программы на другом языке. *Выходным языком* может быть любой язык, в том числе машинный, который выполняется на данном вычислительном устройстве. Допустимо преобразование в промежуточную форму или выходной пользовательский язык, воспринимаемый человеком как результат работы системы.

Отдельным видом языкового процессора является *интерпретатор.* Вместо перевода на другой язык он обеспечивает выполнение программы по мере распознавания конструкций входного языка.

Трансляторы могут использоваться как *конверторы* программ при взаимодействии модулей САПР, имеющих различные стандарты описания входной и выходной информации.

Если транслятор используется для перевода программы в исполняемый код, возможны несколько вариантов представления программы:

- код в виде команд целевого вычислительного устройства;
- код в виде команд виртуальной машины (псевдокод, или байт-код);
- последовательность вызова функций среды исполнения (шитый код).

Транслятор языка высокого уровня называется компилятором. Достоинствами компиляторов являются высокое быстродействие результирующего кода и компактность скомпилированной программы.

Интерпретаторы обладают рядом достоинств, обеспечивающих их широкое распространение при реализации входных пользовательских языков:

- пошаговое выполнение программы с возможностью редактирования текста программы без перезапуска и перекомпиляции приложения;
- быстрый запуск приложения без предварительной компиляции;
- простота разработки интерпретатора.

Основным недостатком интерпретатора является низкая скорость работы программ. В среднем приложение работает в 10-100 раз медленнее, чем скомпилированное в машинный код.

1.3 Формальные языки и грамматики

Основой построения трансляторов является механизм формальных языков и грамматик, описывающих языки.

Теория формальных грамматик оперирует следующими понятиями:

- *Символ* — допустимый в языке элементарный неделимый знак (буква, цифра, служебный символ).
- *Алфавит* — множество символов, допустимых в языке.
- *Лексемы* или *слова* — элементарные неделимые конструкции языка, состоящие из символов и имеющие определенный смысл.
- *Лексика языка* — словарный состав языка.
- *Синтаксис языка* — правила, по которым из слов строятся *фразы* или *предложения.*
- *Семантика языка* — описание смысла предложений — значений слов и их связей.
- *Прагматизм* — воздействие, которое оказывает программа на транслятор.
- *Семиотика языка* — объединение синтаксиса, семантики и прагматизма.

Формальные грамматики рассматривают только множество текстов, которые могут быть сформированы согласно правилам данного языка, не рассматривая их смысл. Это необходимо для решения двух задач: формального доказательства однозначности и корректности рассматриваемого языка и возможности формального описания распознающих и генерирующих процедур.

Формальные грамматики дают ответы на такие вопросы, как «Распознаваем ли данный язык?», «Является ли язык легко распознаваемым?», «Принадлежит ли данное слово или цепочка слов данному языку?».

Для решения поставленных задач создается конечный набор правил, называемых грамматикой языка и порождающие цепочки, которые принадлежат разрабатываемому языку. Эти правила называются *порождающей процедурой,* или *порождающей грамматикой,* и используются для описания синтаксиса языка программирования. Затем с помощью распознающего устройства (автомата, построенного на их основе), производится распознавание предложений языка.

1.4 Порождающая грамматика

До определения формальных грамматик введем необходимую базовую терминологию.

- *Терминальным символом,* или *терминалом,* называется конечный неделимый символ в рассматриваемой грамматике.
- *Нетерминальным символом,* или *нетерминалом,* называется составной символ грамматики. Обычно такой символ получается за счет выполнения правила грамматики над терминалами и нетерминалами.
- *Правило грамматики* — правило, формирующее новый нетерминал на основании терминалов и нетерминалов. Правило определяет новое понятие грамматики.

Рассмотрим простое выражение «2*3». В данном случае терминалами являются символы '2', '3' и '*'. Поскольку на этапе синтаксического анализа значения чисел нам не важны, мы можем объединить символы '2' и '3' в понятие «число». Для того чтобы объяснить смысл этого выражения, вводится понятие «произведение».

Таким образом, формируется следующий набор правил грамматики:

1. <Число> есть один из символов '2' или '3'.
2. <Произведение> есть запись <Число>*<Число>.

Указанные правила сформировали новое понятие в описываемом языке, которого до этого не было. Язык состоял только из отдельных символов '2', '3' и '*', взаимное сочетание которых не давало никакого дополнительного смысла, как по-прежнему не имеет смысла запись «23*». Если мы и вкладывали подсознательно какой-то смысл в выражение «2*3», то только потому, что указанные правила грамматики нам знакомы из школьного курса математики. Ведь без указанных правил эта строка ничем не отличается от строки «абв», которую мы можем назвать простой последовательностью букв. Хотя без правила грамматики отличие между этими строками состоит только в написании символов.

Сформированное правило вносит новый смысл — теперь запись, в которой две цифры разделены символом '*', называется произведением. Произведение в данном случае является нетерминалом, поскольку его можно разделить на составляющие его символы. Оно не является неделимой или «заключительной» (терминальной) конструкцией языка, хотя и имеет свой собственный смысл как конструкция.

Символы '2', '3' и '*' являются терминалами, поскольку дальнейшее их разделение на составляющие части невозможно.

Далее для определения смысла выражения «1+2*3» нам необходимо ввести понятие (или правило грамматики) <Сумма>, которое позволит определять сумму чисел.

Сформулируем сказанное более строго в виде математического определения.

Формальной порождающей грамматикой называется четверка G = <T,N,P,S>,

- где T — конечное непустое множество терминальных символов, являющихся словарем грамматики G;
- N — конечное непустое множество нетерминальных символов, являющихся вспомогательным словарем грамматики G;
- P — конечное множество правил грамматики, также называемое схемой грамматики;
- S — главный нетерминальный символ (цель) грамматики G, также называется начальным символом, или аксиомой грамматики.

Правила грамматики являются выражениями вида $\alpha \rightarrow \beta$, называемыми *правилами подстановки,* или *продукциями.* Цепочка символов α содержит хотя бы один нетерминальный символ. Цепочка символов β может состоять из произвольной последовательности терминальных и нетерминальных символов.

В рассмотренном выше примере продукциями являются:

<Число> $\rightarrow$ <2> или <3>;

<Сумма> $\rightarrow$ <Число> + <Число>.

Совокупность терминальных и нетерминальных символов называется *объединенным словарем грамматики G.*

Множество всех цепочек терминальных символов, выводимых из аксиомы грамматики, называется языком, порождаемой этой грамматикой. Аксиомой грамматики в рассматриваемом примере будет <Сумма>, которую мы определили. Нетерминал <Число> является промежуточным.

Указанная форма описания языка задает правило порождения цепочек, но не фиксирует порядок применения правил подстановки и, таким образом, не обеспечивает детерминированности создаваемых текстов. Это дает возможность компактного описания грамматики языка.

К примеру, по сформированной грамматике можно вывести выражения «2*2», «2*3», «3*2» и «3*3». Выражения «2» и «3» не являются целевыми цепочками языка, поскольку не определяются аксиомой грамматики (<Сумма>).

1.5 Классификация Хомского

Кроме простой подстановки $\alpha \rightarrow \beta$, существует несколько типов правил порождающих грамматик, которые лингвист Хомский предложил классифицировать следующим образом.

Класс 0. Правила вывода грамматики имеют форму $\alpha \rightarrow \beta$ без ограничений на строки α и β. Типичным представителем этого класса являются естественные языки.

Класс 1. Все правила подстановки имеют вид $A\alpha B \rightarrow A\beta B$, где A и B — цепочки, составленные из объединенного словаря грамматики G. То есть замена α на β возможна, только если цепочка α находится между цепочек A и B или, как говорят, «в контексте цепочек A и B».

Порождающая грамматика с такими правилами называется грамматикой *непосредственно составляющих (НС),* а генерируемые ею языки — *контекстно-зависимыми.*

Также языки класса 1 могут порождаться грамматиками, левая часть которых не длиннее правой части. Такая грамматика называется *неукорачивающей.*

Класс 2. Правила подстановки имеют вид $A \rightarrow \beta$, где A — нетерминальный символ, а β — непустая цепочка из объединенного словаря грамматики G. В данном случае замена происходит без контекста, поэтому такие грамматики называют контекстно-свободными или КС-грамматиками.

Класс 3. Правила подстановки имеют вид $A \rightarrow \beta B$ и $A \rightarrow \beta$, где A и B — нетерминальные символы, а β — терминальный символ. То есть правая часть правила является либо терминальным символом, либо нетерминальным символом, за которым следует терминальный символ. Такие языки называют языками с конечным числом состояний, или *автоматными (регулярными)* языками.

Взаимосвязь между классами языков формулируется следующей теоремой:

Если язык регулярный (класс 3), то он контекстно-свободный (класс 2). Если язык контекстно-свободный (класс 2), то, исключив из него пустой символ, мы получим НС-язык (класс 1). Если язык является НС-языком (класс 1), то он также является языком класса 0.

Связь между приведенными классами и проблемой распознавания языков формулируется теоремой: язык, порождаемый неукорачивающей грамматикой, легко распознаваем.

Наибольший практический интерес при разработке трансляторов представляют языки класса 2. В этот класс входят укорачивающие контекстно-свободные грамматики, которые используются для описания языков программирования. С их помощью можно определить большую часть синтаксической структуры языка.

Языки класса 3 используются для описания синтаксиса языка программирования, поскольку они легко представимы конечными автоматами и решают подавляющее большинство проблем распознавания синтаксических конструкций языков программирования.

Практическим применением механизма грамматик при описании языков программирования являются метаязыки, такие как форма Бэкуса-Наура (БНФ), который является прямым эквивалентом КС-грамматики.

Форма Бэкуса—Наура (БНФ) — формальная система описания синтаксиса, в которой одни синтаксические категории последовательно определяются через другие категории. БНФ используется для описания контекстно-свободных формальных грамматик.

1.6 Проектирование лексических анализаторов

Лексика языка и его синтаксис определяют внешний вид программы на языке программирования. Они задают способ записи команд и конструкций языка. При проектировании транслятора необходимо создать формальное описание лексики и синтаксиса языка с тем, чтобы гарантировать однозначную трансляцию программы.

Метасинтаксические языки позволяют описывать синтаксис других языков. Одним из первых метасинтаксических языков является *форма Бэкуса-Наура* (БНФ). Этот язык состоит из металингвистических формул, представляющих правила написания конструкций описываемого языка программирования.

Каждая металингвистическая формула состоит из двух частей. В левой части находится металингвистическая переменная, описывающая конструкцию языка. Затем следует символ "::=", являющийся *металингвистической связкой,* которая читается как «есть». В правой части формулы указывается один или несколько вариантов построения рассматриваемой конструкции.

При этом используется ряд правил. В частности:

- варианты разделяются символом ')';
- имена металингвистических переменных заключаются в символы "<>".

Запись целого положительного целого числа в БНФ:

```
1: <Положительное целое число> ::=  <Цифра>
                        | <Положительное целое число><Цифра>
2: <Цифра> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

В строке 2 указано, что цифра — это любой символ от 0 до 9. В строке 1 указано правило формирования положительного целого числа. Это либо цифра, либо положительное целое число, к которому дописана цифра. Данное правило не определяет семантики (значения) числа.

Оно лишь указывает, что число — это либо одиночная цифра, либо число можно получить добавлением цифры к другому числу. Таким образом, может быть получено любое целое положительное число.

В данном случае в терминологии формальных грамматик отдельные цифры (0..9) являются терминальными, т. е. «конечными», символами, которые невозможно разложить на составляющие. Металингвистические переменные <Цифра> и <Положительное целое число являются нетерминальными символами, т. е. теми, которые можно разложить на составляющие согласно правилам подстановки, выраженным металингвистическими формулами. Аксиомой грамматики или целевым нетерминальным символом является <Положительное целое число>, которое требовалось выразить в БНФ.

Использование рекурсий в металингвистических формулах (строка 1) позволяет с помощью небольшого числа формул описать бесконечное число комбинаций символов, которые могут возникать в реальных языках.

Если потребуется определить целое число со знаком, то достаточно ввести всего лишь одну дополнительную металингвистическую формулу:

```
1: <Целое число> ::= <Положительное целое число> |
2:                   +<Положительное целое число> |
                     -<Положительное целое число>
```

Формула указывает, что целое число — это число, перед которым может находиться символ '+' или '-'.

Число с фиксированной точкой может быть определено следующим образом:

```
1: <Число с фиксированной точкой> ::=
2:     <Целое число>.<Положительное целое число> |
3:     .<Положительное целое число> | <Целое число>.
```

В данном случае указаны три варианта — десятичная точка в центре числа, до начала числа или в его конце.

Существуют расширения нотации БНФ, позволяющие сократить запись формул.

1.7 Лексический и синтаксический анализ формальных языков

Трансляция программы состоит в переводе ее с одного языка на другой с сохранением ее семантики. Трансляция состоит из двух основных фаз: анализа и синтеза. В ходе анализа программы происходит ее формальное представление в терминах транслятора. Затем по этим промежуточным данным синтезируется целевая программа на выходном языке.

Первым этапом анализа является разделение текста программы на лексемы (слова). Лексемами могут быть названия операторов, имена функций, числа, строки (заключенные в кавычки) и даже отдельные символы. Этот этап называется *лексическим анализом,* или *сканированием.* Цель лексического анализа — распознание лексем и перевод их в унифицированный вид. Обычно текстовые лексемы при этом заменяются условными кодами (числовыми значениям), работа с которыми на последующих этапах происходит значительно быстрее и легче формализуется. Информация организуется в виде таблицы, в которой хранится имя лексемы, ее код и специфичные данные. Например, для переменной хранится ее тип и значение. Для функции хранится адрес, по которому она расположена.

Рассмотрим пример кода на BASIC-образном языке:

```
1:   A = 0.5 + N * SIN(ALPHA)
```

В этом случае код будет разделен на 10 независимых лексем: «A», «=», «0.5», «+», «N», «*», «SIN», «(», «ALPHA», «)». Лексема является неделимым элементом, и нам необходимо иметь способ указать транслятору, как понять, что «0.5» является лексемой, в то время как «SIN(ALPHA)» является набором из четырех лексем.

Для этого можно использовать описание языка в БНФ. Продолжая рассмотренный ранее пример, запишем формулы для новых лексем:

```
1: <Инструкция> ::= <Идентификатор> = <Выражение>
2: <Выражение> ::= <Операнд>|<Выражение><Оператор><Операнд>
3: <Оператор>::= + | - | * | /
4: <Операнд>::= <Идентификатор> | <Вызов функции> |
5:              <Число с фиксированной точкой> | <Целое число>
6: <Вызов функции>::= <Идентификатор>(<Выражение>)
7: <Идентификатор>::= <Буква>|<Идентификатор><Буква>|
8:                    <Идентификатор><Цифра>
9: <Буква>::= A|...|Z
```

Такое описание указывает, что «0.5» является лексемой, определенной как «Число с фиксированной точкой». Учитывая это описание, транслятор, прочитав символ '0', будет ожидать либо появления

числового символа, либо появления символа 7, как это указано в правилах для «Целое число», «Положительное целое число» и «Число с фиксированной точкой».

Вторым этапом анализа является *грамматический анализ* (parsing), который обеспечивает группировку лексем исходной программы в грамматические фразы, которые будут использоваться транслятором при формировании выходной программы.

В рассмотренном примере, прочитав символ S, транслятор может предположить, что это начало идентификатора (правило «Идентификатор» в строке 7). Прочитав идентификатор, он может также получить дополнительную информацию о том, что это — имя функции, поскольку символ '(' входит в правило «Вызов функции» (строка 6). Используя БНФ, транслятор получает информацию не только о формате, в котором представляются лексемы, но и о структуре выражения.

В данном случае уже известно, что в выражении (строка 2) «Вызов функции» может использоваться только как неделимая единица. То есть нельзя вычислить «N*SIN», а затем вычислять «(ALPHA)», поскольку «SIN(ALPHA)» является понятием, которое должно быть обработано (вычислено) до включения его в выражение.

Правильное описание БНФ для языка программирования позволяет определить как формат операторов языка, так и приоритет операций в выражениях.

Для того чтобы задать приоритет основных операций, строки 2 и 3 «Выражение» и «Оператор» необходимо заменить следующими:

```
1: <Выражение> ::= <Произведение>|<Выражение>+<Произведение> |
2:                 <Выражение>-<Произведение>
3: <Произведение> ::= <Операнд>|<Произведение>*<Операнд> |
4:                    <Произведение>/<Операнд>
```

Здесь указывается, что «Произведение» есть либо операнд, либо «Произведение», умноженное или разделенное на операнд. А «Выражение» определяется аналогичным образом через «Произведение». Два приведенных правила формируют важную закономерность: прежде чем использовать какие-то аргументы в операциях вычитания или сложения, они предварительно должны являться результатом «Произведения», если это возможно.

В правиле «Произведение» первая подстановка «Операнд» позволяет использовать в качестве «Произведения» одиночный операнд, если операторов умножения или деления не было, как в случае «1+2».

Название «Произведение» в данном случае не строгое, оно лишь указывает, что выполнены все операции данного уровня приоритета.

Используя такую форму записи, можно выразить приоритеты всех операций. После синтаксического анализа транслятор формирует представление программы в виде дерева операций и операндов, называемого *деревом разбора выражения.*

Для приведенного примера дерево будет выглядеть следующим образом (рис. 1).

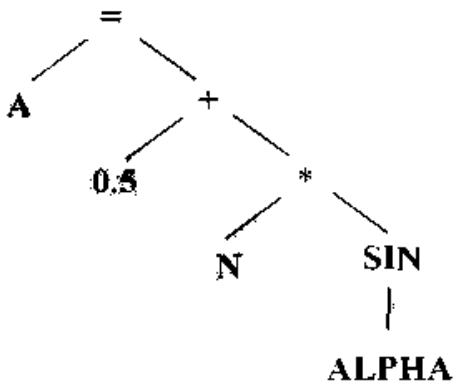


Рис. 1. Дерево разбора выражения

Лексический и грамматический анализ являются этапами синтаксического анализа программы.

После построения иерархического представления программы проводится фаза семантического анализа, в которую входят такие важные проверки, как соответствие типов данных и возможность применения определенных операторов к конкретным данным.

2 ПАКЕТЫ ПРИКЛАДНЫХ ПРОГРАММ (ППП) КАК ФОРМА ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

2.1 Понятие ППП

Многочисленные программные средства для решения различных типов вычислительных задач можно разделить на 4 группы:

- отдельные прикладные программы;
- библиотеки прикладных программ;
- пакеты прикладных программ;
- интегрированные программные системы.

Рассмотрим по порядку каждую из этих групп.

Отдельная прикладная программа пишется, как правило, на некотором универсальном языке программирования (Basic, Pascal, Си и т.п.) и предназначается для решения конкретной прикладной задачи. Примерами могут служить программа решения системы алгебраических уравнений тем или иным численным методом, программа вычисления матриц и т.д.

Библиотека представляет собой набор отдельных программ, каждая из которых решает некоторую прикладную задачу или выполняет определенные вспомогательные функции (управление памятью, обмен с внешними устройствами и т.п.). Библиотеки программ зарекомендовали себя эффективным средством решения вычислительных задач. Они интенсивно используются при решении научных и инженерных задач с помощью ЭВМ. Условно их можно разделить на библиотеки широкого применения и специализированные библиотеки.

Программы, входящие в состав библиотеки широкого применения, предназначены для решения задач из различных предметных областей.

Специализированные библиотеки ориентированы на решение отдельных, порой достаточно узких, классов задач. В качестве примера можно привести программный комплекс для реакторных расчетов, комплекс программ для расчета аэродинамических характеристик крыльев сверхзвуковых самолетов.

Переход от разработки отдельных прикладных программ к созданию библиотек программ поставил перед разработчиками ряд проблем как системного, так и прикладного характера. К их числу относятся проблемы систематизации (1), документирования (2), тестирования(3) и переносимости(4).

1. Проблема систематизации состоит в разбиении библиотеки на разделы и подразделы в соответствии с классификацией задач предметной области и

методов их решения. Сюда входит выбор единых правил наименования программ (с учетом модификации развития библиотеки), единых форм представления и наименования математически сходных объектов, единой схемы контроля ошибок и т.д.

2. Проблема документирования заключается в составлении единых правил описания программ библиотеки. Наличие качественной документации существенно упрощает доступ к отдельным программам, организацию взаимодействия между программами, включение новых программ. Важную роль при решении указанной проблемы играют средства автоматизации документирования, обеспечивающие широкое применение шаблонов (для титульных листов, фрагментов текста и т. п.), использование текстов программ библиотеки для автоматизированного составления документации и т. д.

3. Тестирование библиотеки состоит в проверке программ на специально подготовленных тестовых данных. Результаты тестирования в большой мере зависят от правильности и полноты набора тестов. Тестирование, в частности, включает проверку соответствия текста программы выбранному стандарту языка программирования, определение области применимости программы и качества ее диагностического аппарата, выявление разнообразных количественных характеристик программы (скорость работы, точность получаемых результатов и т.д.), сравнение с другими программами для решения той же задачи.

4. Проблема переносимости состоит в разработке методов и средств, обеспечивающих возможность использования программ библиотеки в различных вычислительных условиях (на различных типах ЭВМ, в различных операционных системах и т.д.). Эта проблема включает в себя такие аспекты, как следование стандарту языка программирования, организация работы с машинно-зависимыми константами, создание инструментальных средств, позволяющих автоматизировать перенос программ из одной вычислительной среды в другую, и некоторые другие.

Доступ к программам библиотеки осуществляется с использованием штатных средств того или иного языка программирования. Таким образом, как и при работе с отдельными прикладными программами, от пользователя требуется определенная квалификация в области программного обеспечения ЭВМ и умение программировать.

Разработка библиотек программ обычно осуществляется силами прикладных программистов. При этом нередко в библиотеку включаются программы, написанные в разное время и разными авторами. Отсутствие в составе библиотеки специализированного системного обеспечения позволяет в большинстве случаев при ее конструировании обойтись без какой-либо существенной помощи системных программистов.

Пакеты прикладных программ. Одной из главных особенностей является ориентация ППП не на отдельную задачу, а на некоторый класс задач, включающий и специфические задачи предметной области. Отсюда следует

необходимость модульной организации ППП как основного технологического принципа его конструирования. Суть этого принципа состоит в оформлении общих фрагментов используемых алгоритмов в виде самостоятельных модулей. Решение сформулированной пользователем задачи осуществляется некоторой цепочкой таких модулей. Организация же пакета по принципу библиотеки (т.е. в виде совокупности программ, каждая из которых решает содержательную прикладную задачу) во многих случаях оказывается неэффективной из-за наличия значительного числа общих частей в алгоритмах решения различных задач одного класса.

Другой особенностью ППП является наличие в его составе специализированных языковых средств, обеспечивающих удобную работу пользователя с пакетом. Как правило, развитый пакет обладает несколькими входными языками, ориентированными на выполнение различных функций и различные типы пользователей. Язык может предназначаться для формулировки исходной задачи описания алгоритма решения и начальных данных, организации доступа и поддержания базы данных или информационной базы ППП, разработки программных модулей, описания модели предметной области, управления процессом решения в диалоговом режиме и других целей. Входные языки ППП разрабатываются с учетом специфики профессиональной терминологии пользователей и принятого в предметной области способа выполнения работ. Проблемно-ориентированные языки часто предназначаются для пользователей, не владеющих программированием. Возможность работы с пакетом пользователей различных типов (по видам выполняемых работ и по уровню прикладной и программистской квалификации) достигается не только за счет включения в его состав различных входных языков, но и посредством реализации различных режимов их использования. Например, применение принципа умолчания дает возможность не указывать значения отдельных параметров задач: им автоматически присваиваются некоторые стандартные значения. В качестве иллюстрации можно привести средства задания метода решения той или иной математической задачи: квалифицированный пользователь может указать метод решения, учитывающий особенности решения задачи; если же метод не задан, то по умолчанию используется некоторый стандартный и, возможно, не лучший для данной задачи метод.

Еще одна особенность ППП состоит в наличии специальных системных средств, обеспечивающих принятую в предметной области дисциплину работы. К их числу относятся специализированные банки данных, средства информационного обеспечения, средства взаимодействия пакета с информационной системой.

Наконец, ***интегрированной программной системой*** назовем комплекс программ, элементами которого являются различные пакеты и библиотеки программ. Примером служат системы автоматизированного проектирования, имеющие в составе несколько ППП различного назначения. Часто в подобной

системе решаются задачи, относящиеся к различным классам или даже к различным предметным областям.

Следует указать на отсутствие четких и однозначных границ между перечисленными формами прикладного программного обеспечения.

2.2 Структура и основные компоненты ППП

Несмотря на большое разнообразие конкретных пакетных разработок, можно выделить следующие основные компоненты ППП:

- входные языки;
- предметное обеспечение;
- системное обеспечение.

Рассмотрим функции каждого из компонентов ППП.

Входные языки представляют собой средство общения пользователя с пакетом.

Можно выделить следующие основные типы пользователей ППП:

- разработчик ППП, осуществляющий его модификацию и развитие с учетом изменения круга пользователей, класса решаемых задач (появление новых типов задач, развитие численных методов, модификация форм проведения работ и т.д.), а также состава аппаратного и программного обеспечения ЭВМ;
- ответственный за сопровождение, в функции которого входит поддержание пакета в работоспособном состоянии в условиях конкретной вычислительной системы (обеспечение сохранности программ и массивов данных, своевременное дублирование информационных файлов, выявление ошибок в программах пакета);
- администратор, отвечающий за организацию доступа пользователя к пакету, содержимое базы данных, защиту информации от несанкционированного доступа;
- конечный пользователь, применяющий пакет для решения конкретных прикладных задач.

Входные языки отражают объем и качество предоставляемых пакетом средств, а также удобство их пользования. Таким образом, с точки зрения конечного пользователя именно входной язык является основным показателем возможностей ППП.

В качестве входных языков могут использоваться как универсальные, так и специализированные языки программирования. В то же время входной язык конечного пользователя в развитом пакете, как правило, является языком

качественно более высокого уровня по сравнению с универсальными языками. Подобные языки называют проблемно-ориентированными, или предметно-ориентированными.

Перейдем теперь к рассмотрению других компонентов ППП. Конкретная деятельность характеризуется 2-мя факторами:

- Классом решаемых задач и используемых для этой цели методов;
- Дисциплиной работы, т.е. совокупностью правил, соглашений и технологических приемов, принятых при разработке, отладке и эксплуатации программ.

Предметное обеспечение представляет собой компонент пакета, отражающий особенности первого из этих факторов, т.е. особенности конкретной предметной области.

Предметное обеспечение включает:

- программные модули, реализующие алгоритмы (или их отдельные фрагменты) решения прикладных задач;
- средства сборки программ из отдельных модулей.

Наиболее распространено в настоящее время оформление каждого модуля в виде программной единицы на том или ином уровне программирования. Такой модуль обеспечивает решение некоторой самостоятельной задачи и связан с другими модулями в виде лишь входной и выходной информацией. Организация предметного обеспечения в виде библиотеки программ характерна для большинства существующих ППП.

Для сборки программ из модулей могут использоваться различные средства: от ручной сборки, выполняемой самим пользователем, то работа с пакетом по существу не отличается от работы с библиотекой программ. Однако даже при ручной сборке пользователю могут предоставляться средства автоматизации этих процессов.

Автоматическая сборка предполагает наличие в составе предметного обеспечения ППП так называемой модели предметной области. Это может быть совокупность справочных таблиц, моделей отдельных понятий и объектов и, в частности, информация о применимости того или иного модуля для решения поставленной задачи. Модель предметной области используется для автоматизации планирования вычислений, т.е. для построения алгоритма решения задачи и синтеза целевой программы из отдельных модулей. Эти действия выполняются специальной программой пакета, называемой планировщиком вычислений. ППП с автоматическим планированием вычислений часто называют интеллектуальным.

Следует различать статическое и динамическое планирование вычислений.

При статическом планировании построение алгоритма решений целиком осуществляется до начала вычислений. Другими словами, способ решения задачи однозначно определяется исходными данными.

При динамическом планировании выбор алгоритма решения производится в ходе самого решения с учетом полученных промежуточных результатов.

Системное обеспечение представляет собой совокупность системных средств (программы, файлы, таблицы и т.д.), обеспечивающих определенную дисциплину работы пользователя при решении прикладных задач. По своей роли в составе ППП и выполняемым функциям системное обеспечение по существу является специализированной операционной системой, определяющей операционное окружение пакета. Несмотря на многообразие способов реализации системного обеспечения в рамках конкретных пакетных разработок, можно выделить его следующие основные компоненты:

- монитор, управляющий процессом решения и взаимодействием всех компонентов ППП;
- трансляторы с входных языков;
- средства работы с данным;
- средства информационного обеспечения, реализующие выдачу разнообразно справочной информации как по запросам пользователей, так и по запросам различных компонентов пакета (например, сведения о свойствах модулей предметного обеспечения, необходимые планировщику вычислений);
- различные служебные программы, в том числе реализующие взаимодействие пакета с операционной системой (работа с внешней памятью, средства ввода/вывода, драйверы специализированных внешних устройств и др.) .

В конкретном ППП, как правило, отсутствует четкое структурное разделение программ на предметное и системное обеспечение.

2.3 Показатели качества ППП

Можно выделить две группы показателей качества ППП. К первой относятся показатели, отражающие функциональные возможности ППП и имеющие важное значение с точки зрения пользователей. Вторую группу составляют показатели, характеризующие структурные свойства пакета; эти показатели существенны в первую очередь для разработчиков и влияют на простоту сопровождения модификации ППП. Рассмотрим показатели каждой из этих групп.

К показателям качества функциональных возможностей вычислительных ППП в первую очередь относятся: точность результатов, время работы,

используемые ресурсы ЭВМ. Эти показатели наиболее просто формализуются и чаще всего используются при сравнении различных пакетов.

Точность работы пакета характеризуется расхождением между точным математическим результатом и результатом, полученным в ходе вычислений. Определение разумной меры точности может оказаться относительно несложной проблемой в случаях, когда сама задача допускает достаточно простую математическую постановку. Однако для некоторых задач (например для задач вычислительного эксперимента) определение меры точности является непростой проблемой, связанной с физической интерпретацией полученных результатов.

Определение такого показателя, как время работы ППП, не вызывает затруднений. Однако в ряде случаев возможна очень сложная зависимость времени счета от входных данных. Если в процессе своей работы программы пакета обращаются к другим программам или внешним устройствам, то более интересной и важной может оказаться такая характеристика ППП, как количество подобных обращений к внешним объектам. Например, время работы пакета минимизации чаще всего характеризуется количеством вычислений минимизируемой функции, необходимым для получения заданного результата. Время же работы пакета, оперирующего большим количеством информации, расположенной на внешних носителях, целесообразно характеризовать числом обращений к таким носителям.

В некоторых случаях оказываются существенными не абсолютные показатели скорости работы и объема используемой памяти, а их относительные значения – процент использования быстродействия и памяти ЭВМ. Например, для ряда пакетных разработок авиационного и космического назначения экспериментально установлено резкое увеличение стоимости программ при использовании памяти и производительности ЭВМ более, чем на 70%. Попытки же использовать ресурсы машины более, чем на 90% приводят к увеличению затрат на разработку в 3-4 раза, что обычно не учитывается при планировании работ. Такое снижение производительности труда разработчиков программ обусловлено прежде всего необходимостью многократных переработок алгоритмов и программ с целью решения всей заданной совокупности задач с достаточно высоким качеством при ограниченных ресурсах ЭВМ.

Остановимся теперь на показателях качества, отражающих удобство использования ППП, которые в равной степени относятся к программам пакета и к документации.

Мы рассмотрим следующие показатели удобства использования ППП:

- возможность возникновения аварийных ситуаций в процессе работы пакета;
- квалификация пользователя, необходимая для полноценной эксплуатации пакета;

- степень пригодности пакета для решения некоторого заданного класса задач;
- гарантии разработчиков, предоставляемые пользователям.

Если аварийная ситуация возникает в результате неправильно заданных пользователем входных данных (например, входные данные выходят за рамки области допустимых значений), то это может быть оправдано лишь в случае, когда такая возможность явно указана в документации (т.е. пакет вычисляет верный результат для допустимых входных данных, в противном случае результат работы заранее не определен). Однако с точки зрения удобства эксплуатации ППП при неверно заданных входных данных более целесообразным представляется не выход на аварийную ситуацию, а выдача пользователю соответствующего диагностического сообщения. Существуют тем не менее, случаи, когда выход на аварийную ситуацию является оптимальным решением. Например, если некоторая программа пакета предназначена для многократного использования во внутренних циклах, то проверки (внутри этой программы) на возможное переполнение существенно увеличивают время ее работы. В этом случае программа, работающая быстрее и изредка дающая переполнение, может оказаться предпочтительнее медленно работающей программы, в которой переполнений не возникает.

При этом конечно пользователь должен быть предупрежден о возможных аварийных ситуациях и у него должны иметься средства для продолжения своей работы при возникновении таких ситуаций.

Из сказанного следует, что нет необходимости говорить о таком специальном свойстве ППП, как способность обходить аварийные ситуации. Необходимо лишь различать следующее: возникает аварийная ситуация при допустимых или при недопустимых значениях входных данных, описана или нет возможность появления аварийной ситуации в документации, оправдано ли возникновение аварийных ситуаций какими-либо причинами (например, скоростью работы).

Другой важный показатель качества ППП связан с квалификацией пользователя, необходимой для эффективной работы с пакетом. Примером ситуации, в которой этот показатель надо учитывать является решение вопроса о том, какие управляющие параметры (влияющие, например, на выбор метода решения задачи) следует делать доступными пользователю. Часто бывает так, что пакет, предоставляющий широкие возможности по управлению решением, может эффективно применяться лишь квалифицированным пользователем, который способен правильно выбрать значения указанных параметров для своей конкретной задачи. При этом малоквалифицированный пользователь чаще всего задает неадекватные управляющие параметры. Опыт показывает, что он, как правило, задает значения параметров, которые использовались для решения простых модельных примеров, приведенных в описании лишь для иллюстрации различных возможностей пакета.

Еще один показатель качества связан со степенью пригодности пакета для решения некоторого конкретного класса задач той или иной группой пользователей. это важная характеристика ППП, которая отражает возможность легкого комбинирования пакета с другими программами или программными комплексами, возможность использования пакета именно для тех данных, которые специфичны для конкретного приложения, соблюдение необходимых согласованностей значений точности и скорости работы пакета и многие другие более тонкие характеристики. Важно, например, решает ли пакет целиком некоторую самостоятельную задачу или является одним из нескольких программных комплексов, используемых в рамках специализированного программного обеспечения, создаваемого самим пользователем.

Последним из перечисленных показателей удобства использования ППП является гарантия разработчиков, что пакет соответствует своему описанию, приведенному в документации. Смысл этой гарантии конечно не в том, что на самом деле имеется полное такое соответствие(для достаточно сложной программы и тем боле для пакета этого никогда нельзя гарантировать), а в том, что организация-разработчик (или некоторая другая организация, занимающаяся сопровождением пакета) обязуется предпринять определенные действия в случае, если в процессе эксплуатации пакета будут выявлены какие-либо несоответствия. Такими действиями могут быть разного рода компенсации за безуспешные попытки применения "неисправного" пакета, обязательство разобраться в найденной ошибке и исправить ее и т. д. Когда в научной литературе говорят о сертификации программ, то имеют в виду именно наличие такой гарантии.

Перейдем теперь к показателям качества структурных свойств ППП. Анализ процессов разработки, сопровождения и модернизации ППП приводит к необходимости использования самостоятельной группы показателей качества. Эти показатели отражают прежде всего строгое следование на указанных этапах жизненного цикла ППП некоторой системе стандартов.

Наличие стандартов - неотъемлемый элемент любой промышленной технологии. По этой причине особое значение вопросы стандартизации имеют сегодня, когда производство программ и их документации носит промышленный характер. Введение стандартов преследует следующие цели:

- упрощение процессов разработки и эксплуатации программ и, следовательно, повышение их надежности и уменьшение их стоимости;
- облегчение чтения и понимания программ, что в свою очередь упрощает их тестирование, позволяет снизить число ошибок, облегчает модернизацию.

Важным объектом стандартизации при разработке ППП являются используемые языковые средства. Можно выделить два аспекта этой проблемы:

1)выбор и строгое соблюдение стандарта языка программирования (проверка выполнения этого требования может быть осуществлена вручную либо с помощью специального процессора)

2)разработку правил использования языковых конструкций в рамках выбранного стандарта.

Важным элементом стандартизации языковых средств является выявление типовых ситуаций, встречающихся в программах пакета, и определение языковых конструкций для их программирования. Это позволяет программисту сконцентрировать внимание на наиболее сложных и ответственных решениях при написании программ, а техническую работу (выбор управляющих структур, работу с внешней памятью, организацию контроля и т. д.) выполнять автоматически.

Перечислим некоторые другие объекты стандартизации в программах пакета:

- способы представления данных в памяти ЭВМ;
- диагностика аварийных ситуаций, возникающих на различных этапах решения задачи;
- правила наименования программ и используемых объектов данных (система наименований должна создаваться в соответствии с некоторой классификацией задач предметной области и допускать модификацию и развитие пакета);
- соглашения по использованию констант, в частности машинно-зависимых констант, значения которых определяются величиной разрядной сетки конкретной ЭВМ;
- соглашения по работе с внешними устройствами, в том числе правила использования операторов ввода/вывода;
- схема управления памятью разных уровней;
- соглашения по использованию внешних программ, к числу которых относятся программы пользователя, различные компоненты операционной системы и системы программирования, отдельные служебные программы и т. д.

Самостоятельным объектом стандартизации является программная документация. В настоящее время стандарты на документацию значительно лучше разработаны, чем на другие элементы программного обеспечения. Наличие таких стандартов существенно облегчает обмен программными изделиями и применение ранее созданных программ и алгоритмов в новых разработках. Кроме того, появляется возможность автоматизировать подготовку документации путем использования специализированных программных средств.

2.4 Входные языки ППП

Входные языки представляют собой средство общения пользователя с пакетом программ. Пакет может иметь несколько входных языков. Например, язык может предназначаться для логического описания задачи пользователя, задания алгоритма решения и исходных данных, организации доступа и поддержания базы данных или информационной базы ППП, разработки модулей предметного обеспечения, описания модели предметной области, управления процессом решения задачи в диалоговом режиме и т. д. Использование универсальных языков программирования для решения перечисленных задач во многих случаях не может удовлетворить ни пользователей, ни разработчиков ППП.

Использование универсального языка требует определенной программистской подготовки пользователя: знания особенностей конкретного транслятора и других многочисленных деталей системного обеспечения ЭВМ. С точки зрения разработчика ППП необходимость создания специализированного входного языка может быть продиктована стремлением сделать пакет независимым от операционной системы или системы программирования, обеспечить возможность дальнейшего развития и модификации пакета, уменьшить используемые ресурсы ЭВМ. Таким образом разработка специализированного языка в большинстве случаев объясняется необходимостью предоставить более эффективные средства доступа к пакету, учитывающие особенности решаемых задач и требования пользователей.

Рассмотрим некоторые характерные особенности и свойства входных языков современных ППП.

Входной язык, как правило, ориентирован на достаточно узкий круг задач.

По мере развития предметного и системного обеспечения ППП (включение новых методов решения задач, изменение режимов работы пакета и т. п.) должны вноситься соответствующие изменения во входной язык.

Входной язык обычно обладает существенно менее развитым набором средств по сравнению с современными языками программирования. В первую очередь это объясняется ориентацией на ограниченный класс задач, составляющих предметную область пакета. Кроме того, создание развитого языка требует больших временных и материальных затрат.

С точки зрения режима использования можно выделить три группы проблемно-ориентированных языков:

- универсальные языки;
- специализированные проблемно-ориентированные языки;
- диалоговые языки.

Универсальные языки. Использование того или иного алгоритмического языка в качестве входного характерно для пакетов библиотечного типа. Такой

подход обеспечивает относительно простую реализацию и нередко применяется в тех случаях, когда разработчиками ППП являются прикладные программисты. При этом использование определенных приемов позволяет в какой-то мере приспособить алгоритмический язык для конкретного приложения. К числу типичных недостатков рассматриваемого подхода относятся:

- несоответствие изобразительных средств универсальных алгоритмических языков специфике предметной области;
- необходимость знакомства пользователя с соответствующим языком программирования.

Специализированные проблемно-ориентированные языки. С точки зрения пользователей ППП можно выделить два основных типа специализированных входных языков:

- встроенные языки, которые могут использоваться в рамках того или иного базового языка;
- автономные (самостоятельные) языки, применяемые независимо от других языков.

В основе разработки встроенного языка лежит идея расширения некоторого базового языка путем включения его в состав дополнительных средств, отражающих специфику конкретной предметной области. При этом базовым языком может быть как универсальный язык программирования (Basic, Pascal, C), так и некоторый специализированный язык. Реализованный таким образом входной язык ППП может использоваться в рамках соответствующего базового языка. Другими словами, фрагменты на входном языке могут быть "встроены" в текст программы на базовом языке (отсюда и возник термин "встроенный язык"). Существуют два основных способа реализации такого расширения:

- средствами базового языка;
- путем разработки дополнительного программного обеспечения.

Языки второй группы (автономные (самостоятельные) языки) предназначены для решения самостоятельных задач. Такой язык разрабатывается в случае, когда требуются специальные изобразительные средства, отсутствующие в имеющихся языках. В нем обычно используется терминология, синтаксические и семантические конструкции, отражающие особенности соответствующей предметной области, в частности принятые в ней методы формулировки и решения задач.

Диалоговые языки. Интерактивное взаимодействие пользователя с компьютером является на сегодняшний день наиболее распространенным режимом решения различных прикладных задач.

Необходимым компонентом ППП является в этом случае диалоговый монитор, реализующий взаимодействие с пользователем.

Важной особенностью диалогового режима работы является возможность оперативного вмешательства пользователя в процесс решения задачи. Такое вмешательство может быть организовано с помощью так называемых контрольных точек, в которых решение задачи прерывается и инициируется диалог с пользователем. Это необходимо в случаях, когда пакету требуется помощь для определения дальнейших действий. Примером является ситуация, в которой необходимо выбрать метод решения той или иной задачи, причем отсутствует формализованный алгоритм выбора и требуются дополнительные указания со стороны пользователя. Контрольные точки должны быть заранее предусмотрены в программах ППП при его конструировании. Другим способом вмешательства является прерывание процесса решения по инициативе пользователя с помощью специальной команды. Прервав решение, пользователь может проанализировать ситуацию и принять какое либо решение (продолжить решение в диалоговом или пакетном режиме, использовать другой метод, закончить работу и т. п.)

Можно выделить три наиболее распространенных типа диалога: "команды", "меню" и взаимодействие на ограниченном естественном языке.

2.5 Средства сборки программ

Неотъемлемым этапом работы ППП является сборка из имеющихся модулей целевой программы решения задачи пользователя. Сборка может выполняться либо самим пользователем (в этом случае говорят о ручной сборке), либо с применением специализированных программных средств, называемых средствами сборки.

Этап сборки целевой программы включает:

- определение последовательности модулей, обеспечивающих решение сформулированной задачи (эти действия называют планированием вычислений);
- организацию выполнения модулей, в частности сопряжение модулей по входным и выходным параметрам.

Существуют два основных режима использования модулей: компиляция и интерпретация. В режиме компиляции этапу выполнения программы предшествует ее полная сборка из имеющихся модулей. В режиме интерпретации решение задачи осуществляется управляющей программой пакета, которая производит последовательный вызов необходимых модулей. В некоторых пакетах указанные способы комбинируются, например, модули, вызываемые управляющей программой, могут подвергаться определенной предварительной обработке (редактированию, сборке из отдельных фрагментов и т.п.)

При ручной сборке работа пользователя с пакетом по существу не отличается от работы с библиотекой программ.

Автоматизация процесса сборки программ предполагает наличие в составе ППП определенных сведений как о решаемой задаче, так и о свойствах отдельных модулей. При этом существует 2 источника таких сведений:

- внутренний источник, т.е. информация, тем или иным способом хранимая в рамках пакета (в виде модели предметной области, паспортов модулей и т. п.);
- внешний источник, представляющий собой совокупность сведений, взятых из запроса пользователя на входном языке пакета.

При автоматизированной сборке используется следующий механизм. Для каждого типа задачи заранее зафиксирована последовательность вызываемых модулей. Входные параметры модуля формируются на основе информации пользователя, заданной во входном запросе. Если какие-либо параметры не заданы пользователем, то им по умолчанию присваиваются определенные стандартные значения.

Таким образом, управляющая программа пакета после завершения синтаксического и семантического анализа входной информации инициирует выполнение определенной последовательности модулей, избавляя пользователя от необходимости их явного указания. Заметим, что хранимые в управляющей программе сведения о соответствии между различными типами задач и цепочками решающих их модулей и являются тем внутренним источником информации, который используется при сборке.

Повышение степени автоматизации сборки программ (в том числе реализация полностью автоматической сборки) может быть достигнуто за счет включения в состав ППП описания модели предметной области. Такая модель используется для построения алгоритма решения задачи и синтеза целевой программы из имеющихся модулей. Информация, содержащаяся в модели, представляет собой различные сведения о решаемых задачах и имеющихся модулях: объекты предметной области и связь между ними, возможные способы решения задач, применимость того или иного модуля для решения конкретной задачи и т. д.

Модель может задаваться различными способами в зависимости от особенностей задач и используемых методов: набором макроопределений, справочными таблицами, графом информационных и логических связей между модулями, базой знаний и т. д. Наиболее развитые и эффективные способы задания таких моделей в виде баз знаний применяются в современных системах искусственного интеллекта.

2.6 Технологические аспекты разработки ППП

В процессе разработки больших программных систем, в том числе и пакетов программ, можно выделить 6 основных этапов, составляющих "цикл жизни" программного обеспечения:

- анализ требований, предъявляемых к программной системе;
- определение спецификаций;
- проектирование;
- программирование;
- тестирование;
- эксплуатация и сопровождение.

Рассмотрим содержание каждого из этих этапов.

На этапе ***анализа требований*** определяется целесообразность использования ЭВМ для решения поставленной задачи.

На этапе ***определения спецификаций*** производится точное описание функций системы.

На этапе ***проектирования*** определяется структура системы и разрабатываются все необходимые алгоритмы. Так же, на данном этапе, определяются способы реализации системы на основе разработанных ранее спецификаций. Определяются структура системы и алгоритмы, необходимые для ее функционирования. Осуществляется выделение модулей и фиксируются межмодульные связи. Определяются необходимые технические и программные инструментальные средства, языки программирования и т. п.

Этап ***программирования*** включает в себя написание и отладку программ, а так же подготовку необходимой документации.

Цель этапа ***тестирования*** состоит в проверке правильности системы (отсутствие ошибок) и ее соответствия имеющимся спецификациям.

Наконец последним этапом жизненного цикла программной системы являются ее ***эксплуатация и сопровождение***. На этом этапе могут обнаруживаться ошибки, пропущенные при тестировании.

Особое значение имеет вопрос разработки документации на программную систему. Основное правило, которого необходимо придерживаться, состоит в том, что разработка документации должна производиться параллельно разработке самой программной системы, начиная с самых ранних этапов. Наличие качественной документации существенно облегчает как разработку системы (взаимодействие между программистами, подключение новых разработчиков и т. д.), так и ее эксплуатацию и сопровождение. Кроме того, хорошая документация выполняет роль рекламы проекта, поскольку пользователями часто берутся на вооружение не объективно лучшие разработки, а качественно и вовремя документированные.

3 ПРАКТИЧЕСКАЯ ЧАСТЬ

3.1 Visual C++ и Microsoft Foundation Classes (MFC)

Основные различия программирования под MS-DOS и Windows

MS-DOS	Windows
Активную роль выполняет приложение. В перерывах между вычислениями компьютер простаивает в ожидании ввода пользователя. Одновременно может быть запущено только одно приложение.	Активную роль выполняет пользователь. В перерывах между вычислениями приложение отдаёт управление операционной системе и другим приложениям.
Ввод данных осуществляется самим приложением. Приложение запрашивает ввод в известный ему момент и приостанавливает выполнение до окончания ввода.	Ввод данных инициируется устройством ввода. Приложение должно быть готово к обработке команд пользователя в любой момент. Порядок их поступления не гарантируется.
Взаимодействие с устройствами ввода (клавиатура, мышь) осуществляется посредством опроса состояния соответствующих устройств.	Взаимодействие с устройствами ввода (клавиатура, мышь) осуществляется посредство сообщений, которые генерируются соответствующими устройствами и передаются оконной процедуре Windows-приложения.
Приложение безраздельно владеет устройством вывода (например, экраном компьютера). Единожды отображённая на экране информация может быть изменена только этим приложением. Выводимые на экран данные можно не сохранять, если они не требуются для дальнейших вычислений.	Приложение и использует лишь часть экрана – окно, которое в любой момент может быть перекрыто окном другого приложения. Для восстановления утраченной при перекрытии информации приложение должно обеспечивать возможность перерисовать содержимое окна в любой момент времени по запросу операционной системы. Выводимые на экран данные должны сохраняться во время работы, поскольку они могут понадобиться в любой момент времени для перерисовки окна.
Приложение полностью самостоятельно формирует пользовательский интерфейс, определяя его представление и содержание.	Пользовательский интерфейс наполняется приложением и отображается операционной системой по определённым правилам, гарантирующим однообразие пользовательского интерфейса во всех приложениях.

Создание приложения с использованием WinAPI является достаточно сложным. Для упрощения программирования в операционной системе Windows была разработана библиотека MFC, которая обеспечивает удобную объектно-ориентированную обёртку вокруг функций и структур данных WinAPI. Так же компилятор Visual C++ предоставляет ряд мастеров настроек, которые позволяют автоматизировать часть рутинных операций, выполняемых при создании приложения.

Создание приложения с использованием MFC

Запустим интегрированную среду разработки Microsoft Visual Studio 2008 и создадим в ней новый проект: *File* > *New* > *Project...*

В открывшемся окне мастера создания новых проектов (New Project) необходимо выбрать тип проекта: *Visual C++ / MFC / MFC Application*

В поле **"Name"** необходимо указать имя проекта. Назовём его: "Spheres".

В поле **"Location"** необходимо указать расположение проекта (например: "C:\Temp").

После нажатия клавиши “OK” появится диалоговое окно мастера настройки проекта MFC.

Основные настройки надо оставить в их начальном состоянии. Необходимо изменить лишь следующие настройки:

Application Type / Application Type надо выбрать "Single Document"

Application Type / Resource Language проверить, что выбран "Английский (США)"

Advanced Features / Advanced features отключить флажки "Printing and print preview" и "ActiveX Controls".

После этого нужно нажать на клавишу **Finish** и приложение будет сгенерировано.

Слева появится окно с закладками (в нижней части окна) Solution Explorer / Class View / Property Manager. По умолчанию должна быть активна закладка Solution Explorer, на которой показывается дерево файлов проекта.

На первом этапе нам будет интересен файл SpheresView.cpp, отвечающий за отображение информации приложения в окне.

Дважды щёлкнув на указанном файле, мы откроем его на редактирование.

Функция (CSpheresView::OnDraw) является функцией-обработчиком сообщения WM_PAINT, которое операционная система присылает приложению всякий раз, когда приложение должно перерисовать содержимое своего окна:

```
// CSpheresView drawing
void CSpheresView::OnDraw(CDC* /*pDC*/)
{
	CSpheresDoc* pDoc = GetDocument();
	ASSERT_VALID(pDoc);
	if (!pDoc)
```

```
        return;
    // TODO: add draw code for native data here
}
```

Это означает, что в ответ на данное сообщение мы должны отобразить требуемую информацию. Первые четыре строки этой функции требуются для работы с документом в архитектуре Doc/View с которой мы познакомимся позже.

Как указано в комментарии – мы можем добавлять свой код после строки //TODO: add draw code for native data here.

Задача 1: Создать десять случайных окружностей и вывести их на экран.

Попробуем решить задачу максимально простым способом. Нарисуем окружности прямо в функции OnDraw (не забыв снять комментарий с указателя на контекст устройства вывода, в нашем случае дисплея /*pDC*/):

```
// CSpheresView drawing
void CSpheresView::OnDraw(CDC* pDC)
{
    CSpheresDoc* pDoc = GetDocument();
    ASSERT_VALID(pDoc);
    if (!pDoc)
        return;
    // TODO: add draw code for native data here
    const int nCircles = 10;
    const int area_width=800;
    const int area_high =600;
    const int max_radius=50;
    for (int i=0; i < nCircles; i++)
    {
        int x=rand()%area_width;
        int y=rand()%area_high;
        int radius = rand()%max_radius;
        pDC->Ellipse(x-radius,y-radius,x+radius,y+radius);
    }
}
```

Данный код вызывает появление окружностей на экране, однако, при изменении размера окна приложения или любом другом действии, вызывающем перерисовку окна, происходит создание новых окружностей, поскольку информация об окружностях теряется.

По идеологии Windows необходимо сохранять информацию, выводимую на экран для того, чтобы иметь возможность в любой момент перерисовать содержимое экрана. Для этого мы можем сохранить данные в массиве. Для того, чтобы массив инициализировался (наполнялся данными о конкретных окружностях) мы введём флаг, проверяющий, была ли сгенерирована информация об окружностях (is_circles_created).

```
struct Circle {
     int x,y;
     int radius;
};

// CSpheresView drawing
void CSpheresView::OnDraw(CDC* pDC)
{
     CSpheresDoc* pDoc = GetDocument();
     ASSERT_VALID(pDoc);
     if (!pDoc)
          return;
     // TODO: add draw code for native data here
     const int nCircles = 10;
     const int area_width = 800;
     const int area_high  = 600;
     const int max_radius = 50;
     static bool is_circles_created=false;
     static Circle circles[nCircles];
     if ( !is_circles_created ) {
          for (int i=0; i < nCircles; i++) {
               circles[i].x=rand()%area_width;
               circles[i].y=rand()%area_high;
               circles[i].radius=rand()%max_radius;
          }
          is_circles_created=true;
          // Установить таймер
          // SetTimer(1, 100, NULL);
     }
     for (int i=0; i < nCircles; i++)
     pDC->Ellipse(  circles[i].x-circles[i].radius,
                    circles[i].y-circles[i].radius,
                    circles[i].x+circles[i].radius,
                    circles[i].y+circles[i].radius);
}
```

Переменные is_circles_created и circles должны быть static, поскольку они должны сохранять свои значения между вызовами функции OnDraw.

Теперь, при изменении размера окна приложения или любом другом действии, вызывающем перерисовку окна, создание новых окружностей не происходит, а осуществляется перерисовка уже существующих окружностей, с использованием имеющейся о них информации.

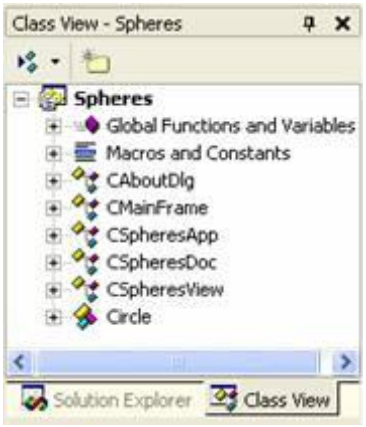
Задача 2: Добавим элемент интерактивности - окружности должны уменьшаться.

Нам потребуется анимировать сцену – показать уменьшающиеся окружности. Для этого воспользуемся таймером.

Во-первых, таймер необходимо установить. Это удобно сделать в том же блоке if, в котором мы создаём окружности, поскольку как создание окружностей,

так и установку таймера необходимо выполнить только один раз(чтобы установить таймер, нужно снять комментарий со строки SetTimer(1, 100, NULL); в приведенной выше функции OnDraw).

Для этого мы воспользуемся функцией SetTimer. Первый аргумент этой функции – номер создаваемого таймера, второй – время между прерываниями от таймера (в миллисекундах). Третий параметр – адрес функции-обработчика прерываний таймера в нашем случае будет NULL, что означает, что необходимо использовать не адрес функции, а передачу приложению сообщения WM_TIMER, обработчик которого мы напишем.



Для того, чтобы создать обработчик сообщения WM_TIMER, необходимо переключиться на закладку Class View и перейти к классу CSpheresView.

Правым кликом на CSpheresView вызвать окно свойств (Properties) этого класса.

В окне свойств необходимо нажать кнопку ***Messages***, найти WM_TIMER и щёлкнуть в поле справа от сообщения WM_TIMER. Появится треугольник, обозначающий открывающийся список. Щелчок по этому треугольнику вызовет появление предложения создать обработчик: <Add> OnTimer. Щелчок по этой надписи приведёт к созданию соответствующего обработчика и появления его в окне редактирования (код обработчика добавится в SpheresView.cpp):

```
void CSpheresView::OnTimer(UINT nIDEvent)
{
	// TODO: Add your message handler code here and/or call default
	CView::OnTimer(nIDEvent);
}
```

В текущем примере нам будет достаточно, чтобы по сообщению таймера вызывалась перерисовка экрана. Всю оставшуюся работу мы проведём в функции OnDraw. Для вызова перерисовки окна достаточно вызывать функцию Invalidate.

```
void CSpheresView::OnTimer(UINT nIDEvent)
{
     // TODO: Add your message handler code here and/or call default
     Invalidate();
     CView::OnTimer(nIDEvent);
}
```

Уменьшение размера окружностей проведём, как и планировали в функции OnDraw, заменив последние строки на:

```
for (int i=0; i < nCircles; ++i) {
     pDC->Ellipse(  circles[i].x-circles[i].radius,
                    circles[i].y-circles[i].radius,
                    circles[i].x+circles[i].radius,
                    circles[i].y+circles[i].radius);

     if (circles[i].radius > 0) circles[i].radius--;
}
```

3.2 Visual C++ и MFC. Архитектура документ/представление (Doc/View)

Библиотека MFC (Microsoft Foundation Classes) предлагает поддержку модели документ/представление, в которой логика работы приложения явным образом отделяется от кода, ответственного за пользовательский интерфейс.

Разделение происходит как на уровне файлов (этот код содержится в разных единицах трансляции), так и на уровне классов – за хранение данных приложения и работу с ними отвечает один класс, а за реализацию пользовательского интерфейса – другой.

В нашем случае код этих классов размещён в следующих файлах:

Класс	Файл	Назначение
CSpheresDoc	SpheresDoc.h	Заголовочный файл документа, хранящего данные приложения.
	SpheresDoc.cpp	Исходный файл документа, хранящего данные приложения.
CSpheresView	SpheresView.h	Заголовочный файл вида.
	SpheresView.cpp	Исходный файл вида, отвечающий за отображение информации.

Определение
Рефакторингом называется внесение изменений в архитектуру и код проекта, призванных улучшить его основные показатели: повторное использование, надёжность, расширяемость, понятность и т.д.

Задача 3. Рефакторинг кода - разделение отображения и данных.

На первой стадии создания приложения мы поместили код логики приложения и код интерфейса пользователя в одной точке – методе OnDraw класса CSpheresView. Однако, по предлагаемой MFC модели приложения (Document/View), выбранной нами при создании приложения необходимо отделить код и данные, относящиеся к логике приложения от кода, отвечающего за отображение информации. Сделать это желательно до того, как мы стали увеличивать объём кода с тем, чтобы потом не переделывать больший объём кода.

Параллельно с этим мы перейдём от использования массива к более удобному контейнеру – вектору из стандартной библиотеки шаблонов (Standard Template Library (STL).

Сразу перед началом работы уберём весь код, относившийся к эксперименту с уменьшением размера окружностей – он нам больше не понадобится.

В файл SpheresDoc.h добавляем:

```
#include<vector>
```

В классе CSpheresDoc (файл SpheresDoc.h) добавим:

```
// Attributes
public:
	std::vector<Circle> m_circles;
```

Массив Circle circles[nCircles] заменяется вектором, благодаря чему мы получаем следующие преимущества:

- Нам не требуется заранее знать размер массива – он будет определён по мере наполнения его окружностями
- Мы получаем возможность в любой момент добавлять новые окружности

Поскольку окружности теперь находятся в документе, нам потребуется интерфейс, позволяющий получить доступ к окружностям извне этого класса (например, из класса вида). Это приводит к необходимости добавить в класс CSpheresDoc методы (файл SpheresDoc.h):

```
// Operations
int GetCircleIndex(const CPoint& pt) const;
Circle& GetCircleById(int id);
```

Первый метод позволяет по точке в плоскости получить индекс окружности, которая содержит эту точку. Как мы обсуждали на предыдущем занятии, для этого проверяются все окружности в порядке обратном созданию –

чтобы в случае накладывающихся окружностей выбрать ту, которая отображается сверху.

Реализация этих методов выглядит следующим образом (файл SpheresDoc.cpp):

```
int CSpheresDoc::GetCircleIndex(const CPoint& pt) const
{
     for (int i=(int)m_circles.size()-1; i >= 0; i--)
     if ( pow(pt.x-m_circles[i].pos.x,2) +
          pow(pt.y-m_circles[i].pos.y,2) <
          pow(m_circles[i].radius,2))
          return i;
     return -1;
}
```

Метод GetCircleById возвращает ссылку на окружность по её индексу. Возвращается именно ссылка, поскольку в этом случае пользователь получает возможность не только считывать, но и изменять данные окружности.

```
Circle& CSpheresDoc::GetCircleById(int id)
{
     return m_circles[id];
}
```

Генерация начального набора окружностей теперь будет происходить в документе. Логично, что это должно быть сделано при создании нового документа. Для этого в документе (CSpheresDoc) существует метод OnNewDocument. В нём можно расположить весь код, отвечающий за инициализацию документа при создании.

До перехода к реализации метода OnNewDocument следует уточнить, что для физического моделирования движения окружностей, которое мы планируем, необходимо хранить координаты и скорости окружностей в переменных с плавающей точкой. В этом случае для представления координат будет полезно разработать структуру, представляющую точку на плоскости, где координаты представлены величинами типа double.

Эта структура (которую нужно поместить в файл SpheresDoc.h) будет выглядеть следующим образом:

```
struct vec2d {
     vec2d () {}
     vec2d (double x, double y) : x(x), y(y){}
     vec2d (vec2d const& other) : x(other.x), y(other.y){}
     // ACCESSORS
     double length() const;
     // MANIPULATORS
     vec2d& operator+=(vec2d const& other);
     vec2d& operator-=(vec2d const& other);
     vec2d& operator*=(double value) ;
```

```
    vec2d& operator/=(double value) ;

    vec2d& normalize();

    double x,y;
};
```

С учётом вышесказанного структура окружности (файл SpheresDoc.h):

```
struct Circle {
    Circle() : velocity(1,0) {}
    vec2d pos;
    vec2d velocity;
    double radius;
};
```

Инициализация документа (метод расположен в файле SpheresDoc.cpp) выглядит следующим образом:

```
BOOL CSpheresDoc::OnNewDocument()
{
    if (!CDocument::OnNewDocument())
        return FALSE;

    // TODO: add reinitialization code here
    // (SDI documents will reuse this document)

    if (!CDocument::OnNewDocument()) return FALSE;

    const int min_radius=4;
    const int max_radius=100;
    const int nCircles = 30;
    const int nTries = 1000;  // Сколько раз можно пытаться
                              // поставить окружность,
                              // которая не пересекается с другими
                              //(чтобы не зациклиться)

    // Размеры виртуальной области
    m_area = CRect(0,0,800,600);

    const int area_width = m_area.right - m_area.left;
    const int area_high  = m_area.bottom - m_area.top;

    m_circles.clear();
    for (int tries=0 ; tries < nTries &&
        m_circles.size() < nCircles; tries++) {
        Circle cir;
        cir.pos.x = (int)(rand()%(area_width-
                    2*max_radius)+max_radius);
        cir.pos.y = (int)(rand()%(area_high-
                    2*max_radius)+max_radius);
        cir.radius= rand()%(max_radius-min_radius)+min_radius;
```

```
            // Проверка на пересечение с существующими окружностями
            if (!IsIntersect(cir)) {
                // Пересечения нет – ставим окружность
                m_circles.push_back(cir);
            }

        }

        return TRUE;
}
```

Рассмотрим взаимодействие вида и документа на примере обновлённой функции OnDraw (она реализована в файле SpheresView.cpp):

```
void CSpheresView::OnDraw(CDC* pDC)
{
        // Получение указателя на документ и проверка его валидности
        CSpheresDoc* pDoc = GetDocument();
        ASSERT_VALID(pDoc);
        if (!pDoc)
            return;

        // Рисование окружностей
        pDoc->DrawCircle(pDC);
 }
```

Основным средством прорисовки окружностей в окне является метод DrawCircle, который представлена далее.

В класс CSpheresDoc (файл SpheresDoc.h) добавляем описание метода DrawCircle, таким образом, участок кода класса CSpheresDoc на данный момент выглядит следующим образом:

```
// Operations
public:
        int GetCircleIndex(const CPoint& pt) const;
        Circle& GetCircleById(int id);

        // Описание метода DrawCircle для прорисовки окружностей в окне
        void DrawCircle(CDC *pDC);
        // Проверка на пересечение окружностей
        bool IsIntersect(Circle);
```

Реализация самого метода DrawCircle, код которого необходимо добавить в файл SpheresDoc.cpp, представляет собой:

```
void CSpheresDoc::DrawCircle(CDC *pDC)
{
        int x, y, r;
        const int min_drawable_radius=2;
```

```
	for(unsigned int i=0; i<m_circles.size(); i++)
	{
		x = (int)m_circles[i].pos.x;
		y = (int)m_circles[i].pos.y;
		r = (int)max(m_circles[i].radius, min_drawable_radius);
		pDC->Ellipse(x-r,y-r,x+r,y+r);
	}
}
```

Так же, в SpheresDoc.cpp добавляем реализацию метода проверки на пересечение окружностей (он используется выше, при инициализации документа OnNewDocument(), и обеспечивает отсутствие наложений окружностей друг на друга при их создании):

```
// Проверка на пересечение с существующими окружностями
bool CSpheresDoc::IsIntersect(Circle cir)
{
	for(unsigned int i=0; i<m_circles.size(); i++)
	 if (sqrt(pow(cir.pos.x - m_circles[i].pos.x, 2) +
		pow(cir.pos.y - m_circles[i].pos.y, 2)) <=
		cir.radius+m_circles[i].radius)
	  return true;

	return false;
}
```

Таким образом, поскольку исходя из концепции Doc/View, хранить данные в классе CSpheresView нельзя, данные об окружностях перенесены в класс CSpheresDoc, для чего потребовалось изменить код остальных обработчиков с тем, чтобы они получали данные через интерфейс класса CSpheresDoc..

Задача 4. Перемещение окружностей

Для того, чтобы иметь возможность ручного перемещения окружностей потребуется перехватить сообщения (аналогично тому, как мы это делали для таймера) WM_MOUSEMOVE, WM_LBUTTONDOWN, WM_LBUTTONUP.

Добавим в классе CSpheresView (SpheresView.h) переменную int selected_circle;

```
// Attributes
public:
	int selected_circle;
```

которая будет указывать, какую окружность (индекс в массиве) мы в данный момент выбрали.

В конструкторе класса установим ей значение -1, означающее, что ни одна окружность не выбрана (файл SpheresView.cpp):

```
CSpheresView::CSpheresView() : selected_circle(-1)
```

```
{
     // TODO: add construction code here
}
```

Далее требуется в обработчике WM_LBUTTONDOWN определять, на какую окружность указывает курсор мыши. Программный код всех обработчиков будет формироваться в файле SpheresView.cpp:

```
void CSpheresView::OnLButtonDown(UINT nFlags, CPoint point)
{
     // Получение указателя на документ и проверка его валидности
     CSpheresDoc* pDoc = GetDocument();
     ASSERT_VALID(pDoc);
     if (!pDoc)
          return;

     // Запросить индекс окружности, в которой находится точка
     selected_circle = pDoc->GetCircleIndex(point);

     CView::OnLButtonDown(nFlags, point);
}
```

В обработчике WM_MOUSEMOVE перемещать окружность вызывать перерисовку клиентской области окна (Invalidate):

```
void CSpheresView::OnMouseMove(UINT nFlags, CPoint point)
{
     // Получение указателя на документ и проверка его валидности
     CSpheresDoc* pDoc = GetDocument();
     ASSERT_VALID(pDoc);
     if (!pDoc)
          return;

     if(selected_circle != -1){
          // Перемещение окружности в точку, указанную мышью
          pDoc->GetCircleById(selected_circle).pos.x = point.x;
          pDoc->GetCircleById(selected_circle).pos.y = point.y;
          Invalidate();
     }

     CView::OnMouseMove(nFlags, point);
}
```

В обработчике WM_LBUTTONUP надо сбрасывать индекс выделенной окружности:

```
void CSpheresView::OnLButtonUp(UINT nFlags, CPoint point)
{
     selected_circle = -1;
     CView::OnLButtonUp(nFlags, point);
```

```
}
```

Продемонстрированный вариант имеет один недостаток: в начале движения (вне зависимости от того, за какую точку в окружности был произведён захват мышью) окружность перемещается центом к курсору. Происходит хорошо заметный «прыжок». Как избавиться от этого эффекта – оставляется в качестве самостоятельной работы.

Задача 5. Разработка диалоговой формы настройки окружности

Решим следующую задачу: при запросе свойств объекта вызывать диалоговое окно, в котором можно будет настроить радиус окружности.

Вначале мы должны разработать соответствующий диалог. Для этого необходимо переключиться в ClassView и щёлкнув правой клавишей мыши на корне дерева проекта выбрать пункт выпадающего меню Add->Class…

В появившемся диалоге следует выбрать Visual C++/MFC/MFC Class и нажать клавишу “Add”. В появившейся форме заполним поля:

Class name: DlgCircleProps

Base class: CDialog

Все остальные поля будут заполнены автоматически. Нажмём клавишу Finish.

Будет создан класс DlgCircleProps, отвечающий за работу диалога и ресурс IDD_DLGCIRCLEPROPS.

Чтобы открыть созданную форму диалога необходимо переключиться на закладку Resource View. Если её нет – выберите пункт основного меню View/Resource View.

В ResourceView надо выполнить двойной щелчок левой клавишей мыши на ресурсе Spheres/Speres.rc/Dialog/ IDD_DLGCIRCLEPROPS.

Диалоговое окно будет открыто и доступно для редактирования. В правой части экрана появится закладка Toolbox, которая содержит палитру элементов управления Windows, которые можно расположить на форме.

Требуется добавить Static Text, перетащив соответствующий элемент управления из палитры на разрабатываемое диалоговое окно. В свойствах этого элемента свойство Caption необходимо установить в «&Radius:» (без кавычек). Знак ‘&’ перед буквой R означает, что с помощью комбинации “Alt+R” можно перейти к элементу управления, связанному с данным текстом.

Далее необходимо на диалоговую форму установить поле для ввода данных – Edit Control. В этом поле будет показываться текущий радиус выбранной окружности, и в него пользователь сможет вводить новый радиус. Для того, чтобы получить доступ к значению, хранимому в этой ячейке требуется создать соответствующую переменную в созданном нами классе диалога. Для этого надо вызвать контекстное меню установленного элемента Edit Control (кликнуть правой кнопкой мыши на Edit Control) и выбрать пункт Add Variable...

Появится диалоговое окно создания переменной. Заполним поля:
Access: public
Category: Value
Variable type: double
Variable name: m_radius
Min value: 1
Max value: 100

После нажатия клавиши Finish будет создана переменная, значение которой будет автоматически синхронизироваться со значением в поле Edit Control.

Разработку диалогового окна на этом можно считать завершённой. Теперь необходимо вызвать это диалоговое окно.

Запрос свойств выполняется нажатием правой клавиши мыши. Для обработки соответствующего события необходимо перехватить сообщение WM_RBUTTONUP.

Обработчик будет выглядеть следующим образом:

```
void CSpheresView::OnRButtonDown(UINT nFlags, CPoint point)
{
	CSpheresDoc* pDoc = GetDocument();
	ASSERT_VALID(pDoc);
	if (!pDoc)
		return;

	int circle = pDoc->GetCircleIndex(point);

	// Создание диалога
	DlgCircleProps dlg;

	if (circle != -1) {
		// Инициализация данных диалога
		dlg.m_radius=pDoc->GetCircleById(circle).radius;

		// Отображение диалога и проверка выхода по нажатию «ОК»
		if (dlg.DoModal() == IDOK)

			// Если радиус был изменён
			if ( pDoc->GetCircleById(circle).radius !=
				dlg.m_radius ) {
				// Обновить радиус
				pDoc->GetCircleById(circle).radius =
				dlg.m_radius;
				// Перерисовать окно
				Invalidate();
			}
	}

	CView::OnRButtonDown(nFlags, point);
}
```

В заключении, необходимо в файле SpheresView.cpp подключить хидр-файл:

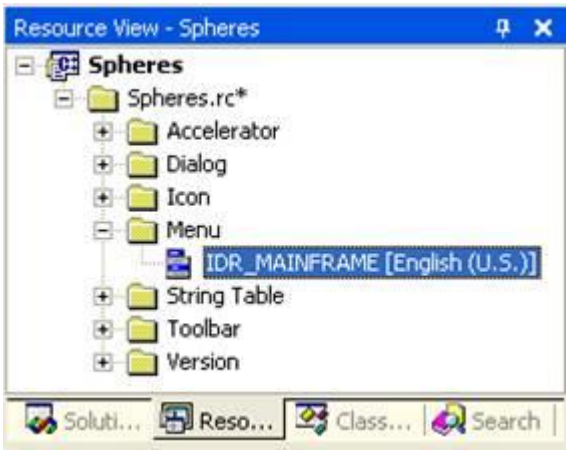
```
#include "DlgCircleProps.h"
```

3.3 Меню и диалоговые окна

Использование меню

Работа с меню начинается с разработки ресурса меню. Создать его можно в окне Resource View, которое вызывается по команде меню View/Resource View. Первым этапом мы отредактируем существующее меню приложения, а затем создадим своё.

Открываем ветку Menu в окне Resource View. В ней присутствует созданный по умолчанию ресурс IDR_MAINFRAME:



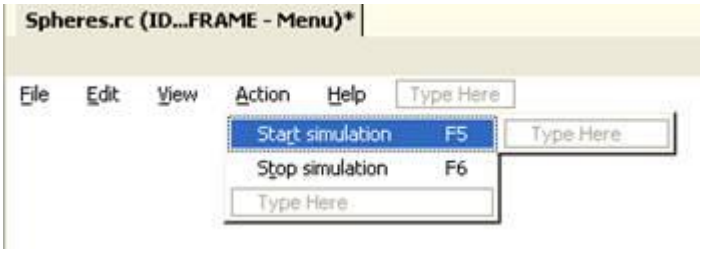
Откроем этот ресурс на редактирование (двойной щелчок левой клавишей мыши):

Для того, чтобы добавить пункт меню перед пунктом «Help», выберем пункт меню Help, нажмём правую клавишу мыши и в появившемся контекстном меню выберем пункт «Insert New» (того же эффекта можно добиться просто

нажав клавишу “Ins”). В появившемся поле введём название пункта меню – “&Action”. Символ “&” перед словом Action означает, что буква «A», следующая за ним является клавишей быстрого доступа для данного пункта меню. То есть его можно будет вызвать по сочетанию клавиш Alt+A.

После этого можно ввести пункты самого ниспадающего меню Action: Для этого введём текст в каждом из пунктов. В первом пункте напишем “Sta&rt simulation\tF5”, а во втором “S&top simulaton\tF6”. Необычное расположение символа “&” (не перед первой буквой) вызвано тем, что для разных пунктов меню должны использоваться разные клавиши доступа. В нашем же случае обе команды начинаются с одной буквы. Чтобы избежать дублирования мы выбрали разные буквы быстрого доступа. Запись “\tF5” в конце первой команды обозначает, что после текста команды необходимо отступить одну позицию табуляции и написать, что клавишей быстрого доступа для данного пункта меню будет F5. При нажатии этой клавиши будет автоматически выполняться данный пункт меню. Но написать этот текст в пункте меню – не достаточно. Необходимо ещё прописать реальную взаимосвязь этой клавиши с выполняемым пунктом меню. Инструкция по настройке акселераторов будет дана позже.

Меню приобрело вид, показанный на рисунке:



Теперь, нужно задать, какие обработчики в нашей программе будут вызваны в ответ на выбор соответствующего пункта меню. Для этого надо выбрать нужный пункт меню, нажать правую клавишу мыши и в появившемся контекстном меню выбрать пункт “Add Event Handler…”. В ответ появится диалоговое окно мастера создания функции-обработчика.

Выбранный вариант “Message type – COMMAND” нас вполне устраивает и его не следует изменять. Ниже его находится поле ввода имени обработчика. Для пункта меню “Start simulation” логично использовать имя *OnActionStartsimulation*, чтобы впоследствии легко определять к какому пункту меню относится данный обработчик. В правом окне выводится список классов вашего приложения. В большинстве случаев функции обработки меню разумнее размещать в классе вида (View). В нашем случае это CSpheresView.

После того, как обработчики будут добавлены, в них можно разместить код, изменяющий поведение программы. В простейшем случае в этих методах достаточно разместить код:

```
void CSpheresView::OnActionStartsimulation()
{
     SetTimer(1,20,NULL);
```

```
}

void CSpheresView::OnActionStopsimulation()
{
     KillTimer(1);
}
```

Эти методы будут создавать и уничтожать таймер. После выбора пункта меню “Start simulation” вы будете получать сообщения в методе OnTimer. Поскольку в архитектуре Doc/View данными приложения владеет объект документа, то задачей функции OnTimer будет вызвать очередной шаг моделирования в документе:

```
void CSpheresView::OnTimer(UINT_PTR nIDEvent)
{
     CSpheresDoc* pDoc = GetDocument();
     ASSERT_VALID(pDoc);
     if (!pDoc)
          return;

     pDoc->DoSimulation();

     CView::OnTimer(nIDEvent);
}
```

Объект документа может соответствующим образом производить шаг моделирования движения окружностей (в соответствием с вариантами заданий). Необходимо помнить, что в этом случае объект документа после выполнения метода DoSimulation должен вызывать метод UpdateAllViews(NULL) с единственным аргументом NULL для того, чтобы вызвать перерисовку объекта вида для отображения обновлённой информации.

Наибольший интерес представляет метод DoSimulation(), который собственно и воплощает алгоритм перемещения окружностей, в первоначальном варианте реализации метод осуществляет перемещение окружностей слева направо с появлением окружностей в левой части окна, когда они вылетают за правую границу:

```
void CSpheresDoc::DoSimulation()
{
     for(unsigned int i=0; i<m_circles.size(); i++)
     {
          if(m_circles[i].pos.x > m_area.right)
               m_circles[i].pos.x = 0

          m_circles[i].pos.x += m_circles[i].velocity.x;
     }

     UpdateAllViews(NULL);
}
```

Создание клавиатурных акселераторов

Вернёмся к клавишам F5 и F6, которые мы планируем использовать для быстрого доступа к соответствующим пунктам меню. Для того, чтобы задать требуемое поведение этих клавиш, необходимо в Resource View открыть папку Accelerator и в ней открыть акселераторы для существующего меню IDR_MAINFRAME:

ID	Modifier	Key	Type
ID_EDIT_COPY	Ctrl	C	VIRTKEY
ID_EDIT_COPY	Ctrl	VK_INSERT	VIRTKEY
ID_EDIT_CUT	Shift	VK_DELETE	VIRTKEY
ID_EDIT_CUT	Ctrl	X	VIRTKEY
ID_EDIT_PASTE	Ctrl	V	VIRTKEY
ID_EDIT_PASTE	Shift	VK_INSERT	VIRTKEY
ID_EDIT_UNDO	Alt	VK_BACK	VIRTKEY
ID_EDIT_UNDO	Ctrl	Z	VIRTKEY
ID_FILE_NEW	Ctrl	N	VIRTKEY
ID_FILE_OPEN	Ctrl	O	VIRTKEY
ID_FILE_SAVE	Ctrl	S	VIRTKEY

Нажать клавишу “Ins” и в появившейся строке в поле “ID” выбрать идентификатор пункта меню, в поле “Modifier” выбрать пустое поле, в поле “Key” выбрать значение VK_F5.

В результате должно получиться следующее:

ID	Modifier	Key	Type
ID_EDIT_COPY	Ctrl	C	VIRTKEY
ID_EDIT_COPY	Ctrl	VK_INSERT	VIRTKEY
ID_EDIT_CUT	Shift	VK_DELETE	VIRTKEY
ID_EDIT_CUT	Ctrl	X	VIRTKEY
ID_EDIT_PASTE	Ctrl	V	VIRTKEY
ID_EDIT_PASTE	Shift	VK_INSERT	VIRTKEY
ID_EDIT_UNDO	Alt	VK_BACK	VIRTKEY
ID_EDIT_UNDO	Ctrl	Z	VIRTKEY
ID_FILE_NEW	Ctrl	N	VIRTKEY
ID_FILE_OPEN	Ctrl	O	VIRTKEY
ID_FILE_SAVE	Ctrl	S	VIRTKEY
ID_ACTION_STOPSIMULATION	None	VK_F6	VIRTKEY
ID_ACTION_STARTSIMULATION	None	VK_F5	VIRTKEY

Теперь для доступа к пункту меню “Action->Start Simulation” достаточно нажать клавишу F5, а к пункту меню “Action->Stop Simulation” - клавишу F6.

Всплывающие (popup) контекстные меню

Для того, чтобы создать всплывающее меню, необходимо выполнить следующие действия. В окне Resource View щелкнуть правой клавишей мыши на папке Menu и в контекстном меню выбрать пункт *Insert Menu* для того, чтобы добавить новое меню. Затем его следует переименовать в IDR_MENU_POPUP. Двойным щелчком на имени меню откроем меню на редактирование.

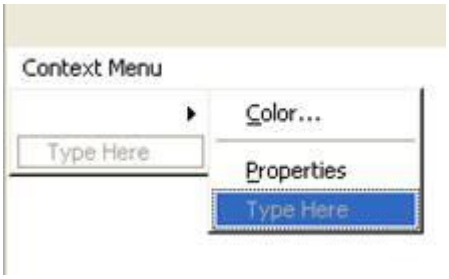
В редакторе показывается пустое меню. Но в данный момент меню отображается в режиме обычного меню приложения. А нам требуется разработать всплывающее меню. Для отображения его в соответствующем виде нажмём

правую кнопку мыши, находясь в редакторе меню, и выберем пункт *View As Popup*. Теперь меню показывается в нужном нам режиме.

По соглашению, принятому в Windows API отображается не само контекстное меню, а его первое (с нулевым индексом) подменю. Именно поэтому, создавая меню, мы должны заполнять не основную его часть, а первое вложенное меню.

Создадим три пункта: “Color…”, разделитель, и “Properties…”. Троеточие в конце текста означает, что выбор данного пункта меню будет приводить к открытию диалогового окна, а не к немедленному выполнению действия.

По окончании разработки меню должно выглядеть следующим образом:



Для того, чтобы добавить разделитель между пунктами “Color…” и “Properties…” необходимо установить курсор мыши на пункте “Properties…” и, нажав правую кнопку мыши, в контекстном меню выбрать пункт “Insert Separator”.

Задание функций-обработчиков для каждого пункта меню осуществляется также, как и в случае с обычным меню.

Вызов контекстного меню осуществляется в ответ на событие WM_RBUTTONDOWN, измененный код OnRButtonDown теперь выглядит следующим образом (в SpheresView.cpp):

```
void CSpheresView::OnRButtonDown(UINT nFlags, CPoint point)
{
     CSpheresDoc* pDoc = GetDocument();
     ASSERT_VALID(pDoc);
     if (!pDoc)
          return;

     CPoint screen_point(point);

     ClientToScreen(&screen_point);
     selected_circle = pDoc->GetCircleIndex(point);

     if (selected_circle != -1) {
          // Popup menu
          CMenu popup;
          popup.LoadMenu(IDR_MENU_POPUP);
          CMenu* real_menu=popup.GetSubMenu(0);
          real_menu->TrackPopupMenu(TPM_LEFTALIGN,screen_point.x,
                                    screen_point.y,this);
```

```
    }
    CView::OnRButtonDown(nFlags, point);
}
```

Код, отвечающий за загрузку и отображение меню, выполняет следующие действия: создаёт объект меню и загружает в него данные по имени ресурса. Далее запрашивается указатель на вложенное меню нулевого уровня и для этого вложенного меню вызывается метод TrackPopupMenu отображающий меню на экране. Следует учесть, что для отображения контекстного меню на экране снова требуются экранные координаты.

При вызове контекстного меню переменная selected_circle будет хранить индекс окружности, для которой было вызвано контекстное меню.

Рассмотрим обработчик пункта “Properties…” всплывающего меню:

```
void CSpheresView::PopupMenuCircleProps()
{
    CSpheresDoc* pDoc = GetDocument();
    ASSERT_VALID(pDoc);
    if (!pDoc)
        return;

    // Создание диалога
    DlgCircleProps dlg;
    // Инициализация данных диалога
    dlg.m_radius=pDoc->GetCircleById(selected_circle).radius;

    // Отображение диалога и проверка выхода по нажатию «ОК»
    if (dlg.DoModal() == IDOK)
        // Если радиус был изменён
        if ( pDoc->GetCircleById(selected_circle).radius !=
            dlg.m_radius ) {
            // Обновить радиус
            pDoc->GetCircleById(selected_circle).radius =
            dlg.m_radius;
            // Перерисовать окно
            Invalidate();
        }
    selected_circle = -1;
}
```

Использование стандартных диалогов

В обработчике “Color…” всплывающего меню требуется выбрать цвет окружности. Для выбора цвета в Windows API существует стандартный диалог, которым можно воспользоваться. В MFC он представлен классом CColorDialog.

```
void CSpheresView::PopupMenuCircleColor()
{
    CSpheresDoc* pDoc = GetDocument();
    ASSERT_VALID(pDoc);
```

```
    if (!pDoc)
        return;

    CColorDialog dlg;

    if (dlg.DoModal() == IDOK)
    if ( pDoc->GetCircleById(selected_circle).color !=
        dlg.GetColor() ) {
        pDoc->GetCircleById(selected_circle).color =
        dlg.GetColor();
        Invalidate();
    }
    selected_circle = -1;
}
```

Как видно из примера, использование этого диалога ничем не отличается от использования созданного нами ранее пользовательского диалога.

Для поддержания функциональности установки цвета окружности требуется лишь завести соответствующее поле в структуре данных окружности, теперь она будет выглядеть следующим образом (объявлена в SpheresDoc.h):

```
struct Circle {
    Circle() : velocity(1,0), color(RGB(0,0,0)) {}
    vec2d pos;
    vec2d velocity;
    // Переменная color хранит информацию о цвете окружности
    COLORREF color;
    double radius;
};
```

Так же, необходимо использовать соответствующий цвет при рисовании окружностей, для этого нужно дополнить метод DrawCircle (в SpheresDoc.cpp):

```
void CSpheresDoc::DrawCircle(CDC *pDC)
{
    int x, y, r;
    const int min_drawable_radius=2;

    for(unsigned int i=0; i<m_circles.size(); i++)
    {
        x = (int)m_circles[i].pos.x;
        y = (int)m_circles[i].pos.y;
        r = (int)max(m_circles[i].radius, min_drawable_radius);

        CPen pen(PS_SOLID,1,m_circles[i].color);
        pDC->SelectObject(&pen);

        pDC->Ellipse(x-r,y-r,x+r,y+r);
    }
}
```

4 КОНТРОЛЬНАЯ РАБОТА

4.1 Указания к выполнению и оформлению контрольной работы

Общие требования к работе:

- Работы выполняются студентами индивидуально.
- Номер варианта выбирается в соответствии со значением полученным от деления порядкового номера студента в списке группы на десять.
- При выполнении работы помимо задания должны быть учтены требования и комментарии к работе.
- Выполненная работа демонстрируется преподавателю. В процессе демонстрации студент должен понимать и уметь объяснить все этапы работы.
- В разделе «Оформление» указывается, что должен содержать отчёт по работе.
- На зачёте студент должен запрограммировать мини-задание и ответить на вопросы преподавателя.

Требования к оформлению отчетов:

- Отчёт оформляется на листах формата А4, шрифтом Tunes New Roman, 14 размера, одиночным интервалом.
- Допускается двусторонняя печать.
- Рукописные тексты и рисунки в отчётах не допускаются.
- Допускаются скриншоты экранов в качестве демонстрации работы программы.

Требования к оформлению листингов программ:

- Перед каждой разработанной функцией должен быть комментарий, поясняющий, что эта функция делает, что возвращает и какие аргументы принимает.
- Листинги программ оформляются шрифтом Courier 12 одиночным интервалом.
- В начале листинга должна быть информация о том, кто написал программу: ФИО, номер группы и назначение программы.
- В теле функций должны быть поясняющие комментарии, если части алгоритма не являются очевидными.

При выполнении данной работы необходимо использовать знания, полученные из курсов «Технологии программирования», «Лингвистическое и программное обеспечение автоматизированных систем». Разработка программного кода осуществляется в среде программирования Microsoft Visual Studio 2008.

4.2 Варианты заданий на контрольную работу

Реализовать следующие варианты поведения окружностей:

1. Окружности перемещаются справа налево, при достижении левой границы они перемещаются скачком к правой границе и случайным образом меняется радиус окружностей в пределах 50 - 80.
2. Окружности перемещаются сверху вниз, при достижении нижней границы они перемещаются скачком к верхней границе и случайным образом меняется радиус окружностей в пределах 4 - 100.
3. Окружности перемещаются по диагонали в одну определенную сторону, при достижении границ они появляются в новом случайном месте и меняют свой цвет.
4. Окружности перемещаются сверху вниз, при достижении нижней границы они перемещаются скачком к верхней границе, во время движения увеличивается, а потом уменьшается радиус окружностей в пределах 50 - 80.
5. Окружности перемещаются снизу вверх, при достижении верхней границы они начинают перемещаться вниз, во время движения уменьшается, а потом увеличивается радиус окружностей в пределах 4 - 100.
6. Окружности разлетаются в разные стороны, при достижении границ они появляются в новом случайном месте.
7. Окружности разлетаются в разные стороны, при достижении границ они отталкиваются от них.
8. Окружности разлетаются в разные стороны, при достижении границ они появляются в новом случайном месте и меняют цвет.
9. Вместо окружностей используются квадраты, они разлетаются в разные стороны, при достижении границ они отталкиваются от них.
10. Окружности перемещаются слева направо и ускоряются, при достижении правой границы они перемещаются скачком к левой границе и ускорение сбрасывается.

4.3 Пример выполнения контрольной работы

Постановка задачи:

Используя библиотеку MFC, необходимо разработать программу, которая позволит решить следующие задачи:

1. Создать 30 окружностей со случайными параметрами (координаты центра, радиус, цвет) и вывести их на экран.
2. Обеспечить интерфейс изменения параметров окружностей.
3. Созданные окружности должны перемещаться по определенному закону в соответствии с вариантами заданий.

Исходные данные:

Исходные данные при создании нового документа инициализируются случайным образом, первоначальные радиусы окружностей генерируются в пределах от 4 до 100. Количество окружностей 30.

Особые ситуации:

1. При инициализации окружности не должны накладываться друг на друга.
2. В процессе перемещения окружности не должны вылетать за указанные координаты экранной области.

Алгоритм решения задачи:

Для реализации движения окружностей был написан алгоритм приводящий к их перемещению слава направо с последующим появлением окружностей в левой части экрана, при достижении ими правой границы. Алгоритм представляет собой простейший вариант имитации движения окружностей.

Структура программы:

Библиотека MFC (Microsoft Foundation Classes) предлагает поддержку модели документ/представление, в которой логика работы приложения явным образом отделяется от кода, ответственного за пользовательский интерфейс.

Разделение происходит как на уровне файлов (этот код содержится в разных единицах трансляции), так и на уровне классов – за хранение данных приложения и работу с ними отвечает один класс, а за реализацию пользовательского интерфейса – другой.

В нашем случае код этих классов размещён в следующих файлах:

Класс	Файл	Назначение
CSpheresDoc	SpheresDoc.h	Заголовочный файл документа, хранящего данные приложения.
	SpheresDoc.cpp	Исходный файл документа, хранящего данные приложения.
CSpheresView	SpheresView.h	Заголовочный файл вида.
	SpheresView.cpp	Исходный файл вида, отвечающий за отображение информации.

Форматы представления данных:

В программе использовались следующие константы:

Имя	Тип	Значение	Описание
min_radius	const int	4	Минимальный радиус окружности
max_radius	const int	100	Максимальный радиус окружности
nCircles	const int	30	Число создаваемых окружностей
nTries	const int	1000	Число попыток установить окружность

Для хранения координат точки на экране (точками задаются центры окружностей), а также, для задания скорости смещения по осям X и Y во время перемещения используется структура: vec2d.

Для задания параметров окружностей используется структура Circle, в которой определены:

`pos` - структура vec2d, центр окружности (`pos.x` и `pos.y` координаты X и Y соответственно);

`radius` - радиус окружности;

`velocity` - структура vec2d, скорость перемещения (`velocity.x` и `velocity.y` скорости перемещения по осям);

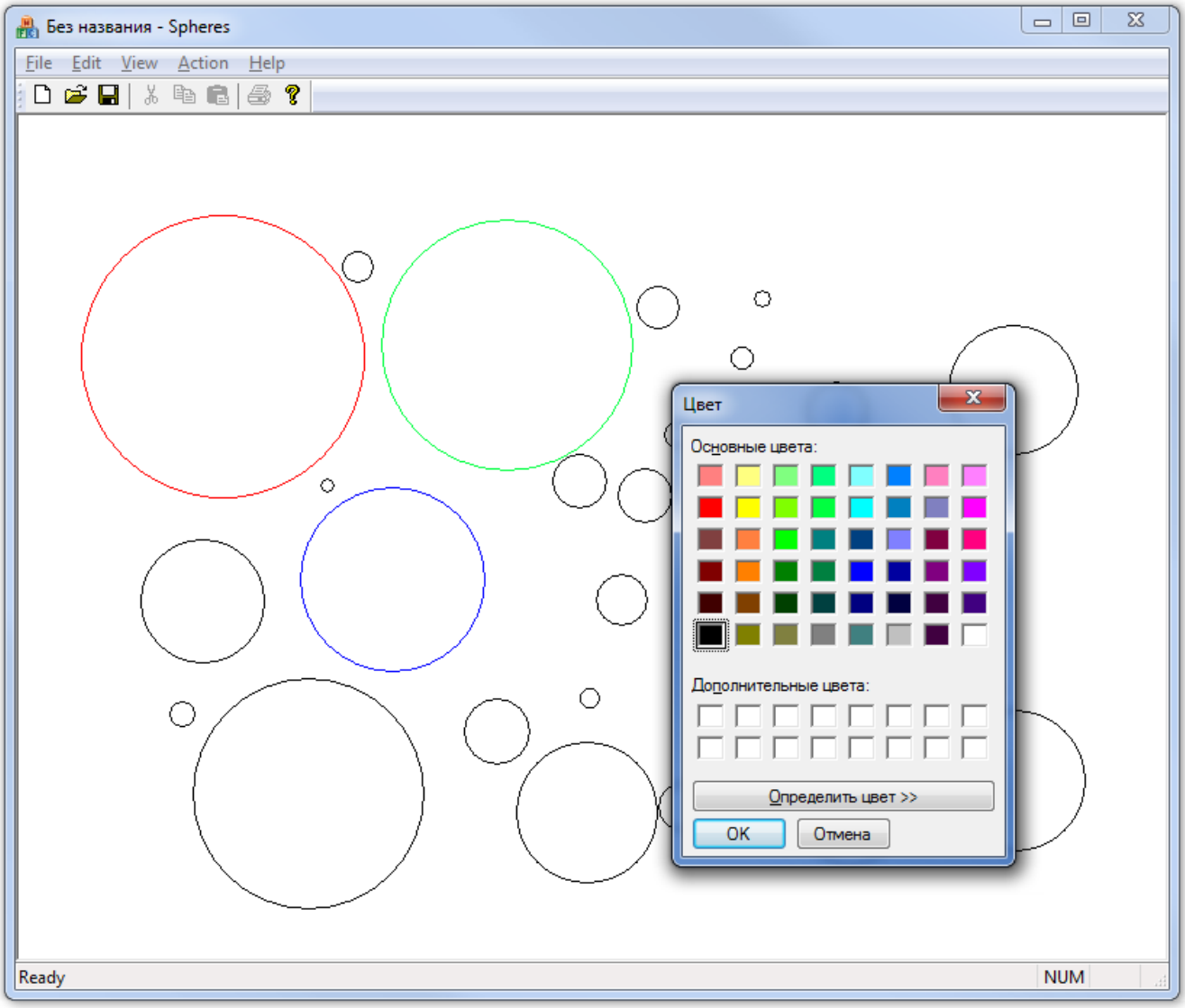
`color` - цвет окружности.

В программе использовались следующие функции:

Имя	Описание
SpheresDoc.cpp	
`int GetCircleIndex(const CPoint& pt) const`	Позволяет по точке в плоскости получить индекс окружности, которая содержит эту точку
`Circle& GetCircleById(int id)`	Возвращает ссылку на окружность по её индексу
`void DrawCircle(CDC *pDC)`	Отрисовка окружностей
`bool IsIntersect(Circle)`	Определение пересечения окружностей
`void DoSimulation()`	Функция содержит алгоритм движения
SpheresView.cpp	
`afx_msg void OnTimer(UINT_PTR nIDEvent)`	Функция по таймеру осуществляет вызов `DoSimulation()` таким образом реализуя один шаг имитации движения
`afx_msg void OnLButtonDown(UINT nFlags, CPoint point)`	Отслеживание нажатия левой кнопки мыши
`afx_msg void OnMouseMove(UINT nFlags, CPoint point)`	Отслеживание перемещения мыши
`afx_msg void OnLButtonUp(UINT nFlags, CPoint point)`	Отслеживание отпускания левой кнопки мыши
`afx_msg void OnActionStartsimulation()`	Запустить имитацию движения окружностей посредством пункта меню или нажатия F5
`afx_msg void OnActionStopsimulation()`	Остановить имитацию движения окружностей посредством пункта меню или нажатия F6
`afx_msg void OnRButtonDown(UINT nFlags, CPoint point)`	Отслеживание нажатия правой кнопки мыши для вызова контекстного меню
`afx_msg void PopupMenuCircleProps()`	Вызов окна изменения радиуса окружностей
`afx_msg void PopupMenuCircleColor()`	Вызов окна изменения цвета окружностей

Результаты работы программы:

При запуске программы создаются окружности. Их можно перемещать курсором мыши, изменять радиус и цвет. Запустить/остановить имитацию движения окружностей можно с помощью соответствующего пункта меню Action или нажатием функциональных клавиш F5/F6.



4.4 Листинг программы

SpheresDoc.h

```
// SpheresDoc.h : interface of the CSpheresDoc class
//

#include<vector>
#pragma once


struct vec2d {
	vec2d () {}
	vec2d (double x, double y) : x(x), y(y){}
	vec2d (vec2d const& other) : x(other.x), y(other.y){}
	// ACCESSORS
	double length() const;
	// MANIPULATORS
	vec2d& operator+=(vec2d const& other);
	vec2d& operator-=(vec2d const& other);
	vec2d& operator*=(double value);
	vec2d& operator/=(double value);

	vec2d& normalize();

	double x,y;
};

struct Circle {
	Circle() : velocity(1,0), color(RGB(0,0,0)) {}
	vec2d pos;
	vec2d velocity;
	// Переменная color хранит информацию о цвете окружности
	COLORREF color;
	double radius;
};

class CSpheresDoc : public CDocument
{
protected: // create from serialization only
	CSpheresDoc();
	DECLARE_DYNCREATE(CSpheresDoc)

// Attributes
public:
	std::vector<Circle> m_circles;
	RECT m_area;


// Operations
public:
	int GetCircleIndex(const CPoint& pt) const;
	Circle& GetCircleById(int id);
```

```
	// Описание метода DrawCircle для прорисовки окружностей в окне
	void DrawCircle(CDC *pDC);
	// Проверка на пересечение окружностей
	bool IsIntersect(Circle);

	// Моделирование движения окружностей
	void DoSimulation();

// Overrides
public:
	virtual BOOL OnNewDocument();
	virtual void Serialize(CArchive& ar);

// Implementation
public:
	virtual ~CSpheresDoc();
#ifdef _DEBUG
	virtual void AssertValid() const;
	virtual void Dump(CDumpContext& dc) const;
#endif

protected:

// Generated message map functions
protected:
	DECLARE_MESSAGE_MAP()
};
```

SpheresDoc.cpp

```
// SpheresDoc.cpp : implementation of the CSpheresDoc class
//

#include "stdafx.h"
#include "Spheres.h"

#include "SpheresDoc.h"
//#include "SpheresView.h"

#include <math.h>
#include <algorithm>

#ifdef _DEBUG
#define new DEBUG_NEW
#endif


// CSpheresDoc

IMPLEMENT_DYNCREATE(CSpheresDoc, CDocument)

BEGIN_MESSAGE_MAP(CSpheresDoc, CDocument)
```

```
END_MESSAGE_MAP()


// CSpheresDoc construction/destruction

CSpheresDoc::CSpheresDoc()
{
     // TODO: add one-time construction code here

}

CSpheresDoc::~CSpheresDoc()
{
}

BOOL CSpheresDoc::OnNewDocument()
{
     if (!CDocument::OnNewDocument())
          return FALSE;

     // TODO: add reinitialization code here
     // (SDI documents will reuse this document)

     if (!CDocument::OnNewDocument()) return FALSE;

     const int min_radius=4;
     const int max_radius=100;
     const int nCircles = 30;
     const int nTries = 1000;  // Сколько раз можно пытаться
поставить окружность,
                               // которая не пересекается с другими
(чтобы не зациклиться)

     // Размеры виртуальной области
     m_area = CRect(0,0,800,600);

     const int area_width = m_area.right - m_area.left;
     const int area_high  = m_area.bottom - m_area.top;

     m_circles.clear();
     for (int tries=0 ; tries < nTries && m_circles.size() <
nCircles; tries++) {
          Circle cir;
          cir.pos.x = (int)(rand()%(area_width-
2*max_radius)+max_radius);
          cir.pos.y = (int)(rand()%(area_high-
2*max_radius)+max_radius);
          cir.radius= rand()%(max_radius-min_radius)+min_radius;

          // Проверка на пересечение с существующими окружностями
          if (!IsIntersect(cir)) {
               m_circles.push_back(cir); // Пересечения нет - ставим
окружность
```

```
            }

      }

      return TRUE;
}


// CSpheresDoc serialization

void CSpheresDoc::Serialize(CArchive& ar)
{
      if (ar.IsStoring())
      {
            // TODO: add storing code here
      }
      else
      {
            // TODO: add loading code here
      }
}


// CSpheresDoc diagnostics

#ifdef _DEBUG
void CSpheresDoc::AssertValid() const
{
      CDocument::AssertValid();
}

void CSpheresDoc::Dump(CDumpContext& dc) const
{
      CDocument::Dump(dc);
}
#endif //_DEBUG


// CSpheresDoc commands
int CSpheresDoc::GetCircleIndex(const CPoint& pt) const
{
      for (int i=(int)m_circles.size()-1; i >= 0; i--)
     if ( pow(pt.x-m_circles[i].pos.x,2) +
            pow(pt.y-m_circles[i].pos.y,2) <
pow(m_circles[i].radius,2))
           return i;
      return -1;
}

Circle& CSpheresDoc::GetCircleById(int id)
{
      return m_circles[id];
}
```

```
// Проверка на пересечение с существующими окружностями
bool CSpheresDoc::IsIntersect(Circle cir)
{
     for(unsigned int i=0; i<m_circles.size(); i++)
      if (sqrt(pow(cir.pos.x - m_circles[i].pos.x, 2) +
            pow(cir.pos.y - m_circles[i].pos.y, 2)) <=
            cir.radius+m_circles[i].radius)
       return true;

     return false;
}

void CSpheresDoc::DrawCircle(CDC *pDC)
{
     int x, y, r;
     const int min_drawable_radius=2;

     for(unsigned int i=0; i<m_circles.size(); i++)
     {
          x = (int)m_circles[i].pos.x;
          y = (int)m_circles[i].pos.y;
          r = (int)max(m_circles[i].radius, min_drawable_radius);

          CPen pen(PS_SOLID,1,m_circles[i].color);
          pDC->SelectObject(&pen);

          pDC->Ellipse(x-r,y-r,x+r,y+r);
     }
}

void CSpheresDoc::DoSimulation()
{
     for(unsigned int i=0; i<m_circles.size(); i++)
     {
          if(m_circles[i].pos.x > m_area.right) m_circles[i].pos.x =
0;
          m_circles[i].pos.x += m_circles[i].velocity.x;
     }

     UpdateAllViews(NULL);
}
```

SpheresView.h

```
// SpheresView.h : interface of the CSpheresView class
//


#pragma once
```

```
class CSpheresView : public CView
{
protected: // create from serialization only
     CSpheresView();
     DECLARE_DYNCREATE(CSpheresView)

// Attributes
public:
     CSpheresDoc* GetDocument() const;
     int selected_circle;

// Operations
public:

// Overrides
public:
     virtual void OnDraw(CDC* pDC);  // overridden to draw this view
     virtual BOOL PreCreateWindow(CREATESTRUCT& cs);
protected:

// Implementation
public:
     virtual ~CSpheresView();
#ifdef _DEBUG
     virtual void AssertValid() const;
     virtual void Dump(CDumpContext& dc) const;
#endif

protected:

// Generated message map functions
protected:
     DECLARE_MESSAGE_MAP()
public:
     afx_msg void OnTimer(UINT_PTR nIDEvent);
     afx_msg void OnLButtonDown(UINT nFlags, CPoint point);
     afx_msg void OnMouseMove(UINT nFlags, CPoint point);
     afx_msg void OnLButtonUp(UINT nFlags, CPoint point);
     afx_msg void OnActionStartsimulation();
     afx_msg void OnActionStopsimulation();
     afx_msg void OnRButtonDown(UINT nFlags, CPoint point);
     afx_msg void PopupMenuCircleProps();
     afx_msg void PopupMenuCircleColor();
};

#ifndef _DEBUG  // debug version in SpheresView.cpp
inline CSpheresDoc* CSpheresView::GetDocument() const
   { return reinterpret_cast<CSpheresDoc*>(m_pDocument); }
#endif
```

SpheresView.cpp

```
// SpheresView.cpp : implementation of the CSpheresView class
//
```

```
#include "stdafx.h"
#include "Spheres.h"

#include "SpheresDoc.h"
#include "SpheresView.h"
#include "DlgCircleProps.h"

#ifdef _DEBUG
#define new DEBUG_NEW
#endif


// CSpheresView

IMPLEMENT_DYNCREATE(CSpheresView, CView)

BEGIN_MESSAGE_MAP(CSpheresView, CView)
//    ON_WM_TIMER()
ON_WM_LBUTTONDOWN()
ON_WM_MOUSEMOVE()
ON_WM_LBUTTONUP()
//ON_WM_CONTEXTMENU()
ON_COMMAND(ID_ACTION_STARTSIMULATION,
&CSpheresView::OnActionStartsimulation)
ON_COMMAND(ID_ACTION_STOPSIMULATION,
&CSpheresView::OnActionStopsimulation)
ON_WM_TIMER()
ON_WM_RBUTTONDOWN()
ON_COMMAND(ID__PROPERTIES, &CSpheresView::PopupMenuCircleProps)
ON_COMMAND(ID__COLOR, &CSpheresView::PopupMenuCircleColor)
END_MESSAGE_MAP()

// CSpheresView construction/destruction

CSpheresView::CSpheresView() : selected_circle(-1)
{
     // TODO: add construction code here
}

CSpheresView::~CSpheresView()
{
}

BOOL CSpheresView::PreCreateWindow(CREATESTRUCT& cs)
{
     // TODO: Modify the Window class or styles here by modifying
     //  the CREATESTRUCT cs
     return CView::PreCreateWindow(cs);
}
// CSpheresView drawing

void CSpheresView::OnDraw(CDC* pDC)
```

```
{
	// Получение указателя на документ и проверка его валидности
	CSpheresDoc* pDoc = GetDocument();
	ASSERT_VALID(pDoc);
	if (!pDoc)
		return;

	// Рисование окружностей
	pDoc->DrawCircle(pDC);

}

// CSpheresView diagnostics

#ifdef _DEBUG
void CSpheresView::AssertValid() const
{
	CView::AssertValid();
}

void CSpheresView::Dump(CDumpContext& dc) const
{
	CView::Dump(dc);
}

CSpheresDoc* CSpheresView::GetDocument() const // non-debug version
is inline
{
	ASSERT(m_pDocument->IsKindOf(RUNTIME_CLASS(CSpheresDoc)));
	return (CSpheresDoc*)m_pDocument;
}
#endif //_DEBUG

void CSpheresView::OnLButtonDown(UINT nFlags, CPoint point)
{
	// Получение указателя на документ и проверка его валидности
	CSpheresDoc* pDoc = GetDocument();
	ASSERT_VALID(pDoc);
	if (!pDoc)
		return;

	// Запросить индекс окружности, в которой находится точка
	selected_circle = pDoc->GetCircleIndex(point);
	CView::OnLButtonDown(nFlags, point);
}

void CSpheresView::OnMouseMove(UINT nFlags, CPoint point)
{
	// Получение указателя на документ и проверка его валидности
	CSpheresDoc* pDoc = GetDocument();
	ASSERT_VALID(pDoc);
	if (!pDoc)
		return;
```

```
	if(selected_circle != -1){
		// Перемещение окружности в точку, указанную мышью
		pDoc->GetCircleById(selected_circle).pos.x = point.x;
		pDoc->GetCircleById(selected_circle).pos.y = point.y;
		Invalidate();
	}
	CView::OnMouseMove(nFlags, point);
}

void CSpheresView::OnLButtonUp(UINT nFlags, CPoint point)
{
	selected_circle = -1;
	CView::OnLButtonUp(nFlags, point);
}

void CSpheresView::OnActionStartsimulation()
{
	SetTimer(1,20,NULL);
}

void CSpheresView::OnActionStopsimulation()
{
	KillTimer(1);
}

void CSpheresView::OnTimer(UINT_PTR nIDEvent)
{
	CSpheresDoc* pDoc = GetDocument();
	ASSERT_VALID(pDoc);
	if (!pDoc)
		return;

	pDoc->DoSimulation();
	CView::OnTimer(nIDEvent);
}

void CSpheresView::OnRButtonDown(UINT nFlags, CPoint point)
{
	CSpheresDoc* pDoc = GetDocument();
	ASSERT_VALID(pDoc);
	if (!pDoc)
		return;

	CPoint screen_point(point);
	ClientToScreen(&screen_point);
	selected_circle = pDoc->GetCircleIndex(point);

	if (selected_circle != -1) {
		// Popup menu
		CMenu popup;
		popup.LoadMenu(IDR_MENU_POPUP);
		CMenu* real_menu=popup.GetSubMenu(0);
```

```
            real_menu-
>TrackPopupMenu(TPM_LEFTALIGN,screen_point.x,screen_point.y,this);
      }
      CView::OnRButtonDown(nFlags, point);
}

void CSpheresView::PopupMenuCircleProps()
{
      CSpheresDoc* pDoc = GetDocument();
      ASSERT_VALID(pDoc);
      if (!pDoc)
            return;

      // Создание диалога
      DlgCircleProps dlg;
      // Инициализация данных диалога
      dlg.m_radius=pDoc->GetCircleById(selected_circle).radius;

      // Отображение диалога и проверка выхода по нажатию «ОК»
      if (dlg.DoModal() == IDOK)
            // Если радиус был изменён
            if ( pDoc->GetCircleById(selected_circle).radius !=
dlg.m_radius ) {
                  // Обновить радиус
                  pDoc->GetCircleById(selected_circle).radius =
dlg.m_radius;
                  // Перерисовать окно
                  Invalidate();
            }
      selected_circle = -1;
}

void CSpheresView::PopupMenuCircleColor()
{
      CSpheresDoc* pDoc = GetDocument();
      ASSERT_VALID(pDoc);
      if (!pDoc)
            return;

      CColorDialog dlg;

      if (dlg.DoModal() == IDOK)
      if ( pDoc->GetCircleById(selected_circle).color !=
dlg.GetColor() ) {
            pDoc->GetCircleById(selected_circle).color =
dlg.GetColor();
            Invalidate();
      }
      selected_circle = -1;
}
```

ЛИТЕРАТУРА

1. Лингвистическое и программное обеспечение автоматизированных систем: учеб.пособие / Д.Ш. Тенишев : под ред. д-ра техн. наук, проф. Т.Б. Чистяковой. - СПб.: ЦОП "Профессия", 2010. - 408с.
2. Страуструп Б. Язык программирования С++. Специальное издание. М.;СПб.: «Издательство БИНОМ» - «Невский Диалект», 2008 г. - 1104с.
3. Н. Вирт. Алгоритмы и структуры данных. М.: «Издательство ДМК Пресс», 2011 г. - 272с.
4. Кнут Э. Искусство программирования. Том 1. Основные алгоритмы. М.: «Издательство Вильямс», 2010 г. - 720 с.
5. Кент Д. С++. Основы программирования. М.: «Издательство НТ Пресс», 2008 г. - 368с.
6. И.Г. Сидоркина. Системы искусственного интеллекта. М.: «Издательство КноРус», 2011г. - 248с.
7. Телло Э.Р. Объектно-ориентированное программирование в среде Windows. М.: Наука-Уайли, 1993.347 с.
8. Давыдов В.Г. Технологии программирования С++: Учебное пособие, рекомендовано УМО. / СПб: bhv-питер, 2005.- 672 с.
9. Иванова Г.С. Технологии программирования: Учебник для вузов. / М.: Издательство МГТУ им. Н.Э. Баумана, 2003.- 320 с.
10. Стивен Прата Язык программирования Си. Лекции и упражнения. 5-е изд. / пер. с англ. / М.: изд. дом «Вильямс», 2006.- 960 с.
11. Дейтел Х.М., Дейтел П.Дж. Как программировать на Си. 4-е изд. / пер. с англ. / М.: «Бином-Пресс», 2006.- 912 с.
12. Романов Е.Л. Практикум по программированию на С++: Учебное пособие. / СПб: bhv-питер, 2004.- 432 с.

Кафедра систем автоматизированного проектирования и управления

Учебное пособие

для студентов заочной формы обучения

направления подготовки «Информатика и вычислительная техника»

ЛИНГВИСТИЧЕСКОЕ И ПРОГРАММНОЕ ОБЕСПЕЧЕНИЕ АВТОМАТИЗИРОВАННЫХ СИСТЕМ

Олег Владимирович Проститенко

Александр Юрьевич Рогов

Отпечатано с оригинал макета. Формат 60x90 $^{1}/_{16}$
Печ. л. 4,25. Тираж 100 экз.

Государственное образовательное учреждение
высшего профессионального образования
Санкт-Петербургский государственный технологический институт
(технический университет)

190013, Типография издательства СПбГТИ(ТУ), тел. 49-49-365
Санкт-Петербург, Московский пр., 26
