**Разработка редактора компонентов для программного комплекса анализа динамики технических систем ПА10.**

**Аннотация**

Работа освещает вопросы реализации инструментального средства разработчика моделей для программного комплекса анализа динамики технических систем ПА10.

В научно-исследовательской части дипломного проекта приведено исследование технологий разработки компонентов и языков описания моделей, применяемых в современных программных комплексах анализа динамики. Освещены вопросы объектно-ориентированного моделирования. Обоснована необходимость разработки редактора компонентов для программного комплекса ПА10, сформулированы основные требования к редактору компонентов.

Конструкторская часть дипломного проекта посвящена разработке системы автоматической генерации программного кода, применяемой для генерации кода компонентов для программного комплекса ПА10.

Технологическая часть дипломного проекта освещает вопросы разработки редактора условных графических обозначений компонентов для программного комплекса ПА10.

**Содержание**

[Введение 5](#_Toc63238)

[1. Научно-исследовательская часть 9](#_Toc63239)

[1.1. Компонентные модели 10](#_Toc63240)

[1.2. Объектно-ориентированное моделирование 13](#_Toc63241)

[1.3. Обзор современных программных средств 15](#_Toc63242)

[1.3.1. Подсистема Simulink пакета Matlab 15](#_Toc63243)

[1.3.2. Язык Modelica 18](#_Toc63244)

[1.3.3. Программные комплексы серии ПА (ПА7-ПА9) 20](#_Toc63245)

[1.3.4. Выводы 21](#_Toc63246)

[1.4. Реализация компонентного моделирования в ПА10 22](#_Toc63247)

[2. Конструкторская часть 30](#_Toc63248)

[2.1. Разработка подсистемы автоматической генерации программного кода 30](#_Toc63249)

[2.1.1. Технология CodeDOM (Code Document Object Model) 32](#_Toc63250)

[2.1.2. Разработка библиотеки классов Special.CodeGen 37](#_Toc63251)

[2.1.2.1. Генерация кода на основе объектной модели 40](#_Toc63252)

[2.1.3. Генерация кода программных моделей элементов технических систем 46](#_Toc63253)

[3. Технологическая часть 51](#_Toc63254)

[3.1. Разработка редактора условных графических обозначений 51](#_Toc63255)

[Приложение 1. Иерархия классов пространства имен System.CodeDOM 60](#_Toc63256)

[Приложение 2. Пример результатов генерации кода программной модели компонента 62](#_Toc63257)

# Введение

Системы моделирования динамических процессов в технических объектах прошли достаточно большой путь развития (работы в этой области начались еще в 60-х годах прошлого века) и использовались при решении ряда сложных технических задач. Первые системы моделирования применялись в научных исследованиях и дорогостоящих технических проектах (космические и ядерные программы, проекты, связанные с созданием новых средств вооружения). В каждом отдельном случае разрабатывалось уникальное программное обеспечение, использовать которое могли лишь высококвалифицированные профессионалы. Системы моделирования того времени имели текстовый интерфейс. Поворотным моментом в развитии компьютерного моделирования стало появление персонального компьютера с графическим дисплеем, сделавшего возможным развитие современных визуальных средств моделирования.

Современные системы визуального моделирования позволяют пользователю задавать описание модели в графической форме – в виде структурной схемы, состоящей из образов функциональных блоков моделируемого устройства и связей между ними.

В процессе проектирования инженеру часто приходится прибегать к перебору и сравнению множества различных вариантов проектируемой системы в условиях жестких временных ограничений. Поэтому простота и скорость реализации отдельного варианта при использовании системы моделирования существенно влияет на качество проектирования. Широкое использование средств моделирования в практике проектирования невозможно, если программные комплексы моделирования не будут обладать удобным интерфейсом пользователя, развитыми графическими средствами представления результатов расчетов, обширной библиотекой моделей компонентов моделируемых технических систем, средствами разработки пользовательских моделей компонентов.

Программный комплекс ПА10 представляет собой систему моделирования динамических процессов в технических объектах, предназначенную для анализа процессов в различных физических системах во временной и частотной области. Моделирование осуществляется на уровне обыкновенных дифференциальных уравнений, математическое ядро программного комплекса обеспечивает как быстроту, так и точность оценки принимаемых проектных решений.

Система моделирования ПА10 разрабатывается с учетом современных требований к программному обеспечению САПР – модульности, адаптируемости, обеспечению объектно-ориентированного подхода к процессу проектирования, наличию удобного графического интерфейса.

Моделируемая техническая система может иметь сложную многоуровневую иерархическую структуру и представляться конечным множеством компонентов (элементов и подсистем) и их связей (Рис.1). Каждая связь объединяет произвольное число внешних контактов компонентов.

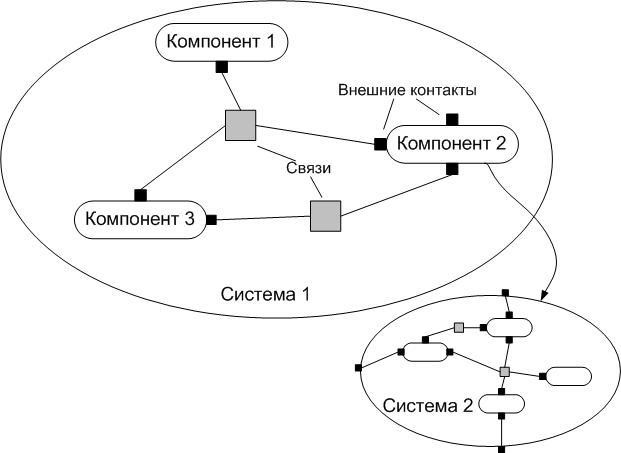


Рис.1. Иерархическое представление моделируемой системы.

Математической моделью технической системы является нелинейная система обыкновенных дифференциальных уравнений. Система уравнений формируется из элементных и топологических уравнений. Элементные уравнения выражают функциональные свойства отдельных элементов. Топологические уравнения выражают фундаментальные физические законы сохранения величин, соответствующих переменным состояния. Такой подход применим для моделирования различных физических систем: электрических, механических, гидравлических, тепловых, а также систем, состоящих из взаимодействующих физически разнородных элементов.

Программный комплекс ПА10 реализует рассмотренную концепцию моделирования. При его разработке был учтён опыт создания предыдущих версий программ серии ПА. ПА10 имеет графический интерфейс, ввод описания моделируемой системы осуществляется при помощи редактора схем, в котором пользователь собирает функциональную схему моделируемой системы из отдельных компонентов.

Одним из основных требований к разрабатываемому программному комплексу являлось обеспечение возможности легкой адаптации библиотеки компонентов к нуждам пользователя. В настоящий момент большинство представленных на рынке программных комплексов этому требованию полностью не удовлетворяет. Так разработка моделей для программных продуктов группы MSC.ADAMS долгое время осуществлялась на языках FORTRAN и C++, в предыдущих версиях комплексов ПА (ПА9, ПА7) для этих целей применялись специальные языки. При этом визуальные средства разработки компонентов отсутствуют, и пользователь, являющийся специалистом в своей предметной области, но не знакомый с технологиями программирования не имеет возможностей для пополнения библиотеки моделей необходимыми элементами.

Для программного комплекса ПА10 было создано специальное визуальное средство разработки компонентов, которое обеспечивает следующие возможности:

* Создание моделей компонентов в схемном редакторе;
* Задание функциональных зависимостей для математических моделей нелинейных элементов;
* Создание новых компонентов путем расширения функциональности уже имеющихся компонентов;
* Создание компонента на основе любого связного участка схемы;
* Автоматическая генерация программного кода модели компонента на одном из языков, поддерживаемых платформой Microsoft .Net;
* Возможность редактирования программного кода модели компонента;
* Создание условного обозначения компонента при помощи специального графического редактора;
* Добавление разработанного компонента в библиотеку моделей.

Данные возможности должны позволить эффективно решать задачи создания математических моделей элементов технических систем с различной степенью точности, адекватности и экономичности. При этом также остается возможность разработки модели компонента непосредственно на языке программирования без использования визуальных средств.

# Научно-исследовательская часть.

При математическом моделировании динамических процессов состояние любого технического объекта (ТО) в конкретный момент времени можно описать множеством некоторых физических величин, имеющих как постоянные значения, так и переменные значения в течение заданного времени моделирования. Обозначим множество переменных модели ТО вектором *V ts*(*t*). Математической моделью динамических процессов в ТО в общем случае будет некоторая система ДАУ, замкнутая относительно этих переменных:

*dV ts* ,*t*) = 0 (1)

*F*(*Vts* , *dt*

Можно решать эту систему с использованием известных математических программ (Matlab, MathCAD, Mapple, Mathematica) или с использованием стандартных подпрограмм интегрирования систем обыкновенных дифференциальных уравнений из соответствующих библиотек (NAG, IMSL). Однако на практике такой подход мало применим, т.к. получение системы уравнений (1) вручную даже для относительно простых ТО оказывается сложной задачей. При этом возможно лишь однократное решение какой-либо задачи моделирования конкретного ТО, для другого ТО придется получать новую систему уравнений.

Между тем эти проблемы можно решить с использованием схемных математических моделей ТО. Любой ТО можно представить как множество элементов, связанных между собой множеством связей, поэтому его можно отобразить схемной моделью в виде некоторой обобщенной формальной схемы (ФС), состоящей из набора элементов, связанных между собой в некоторых узлах этой схемы. При этом система уравнений (1) генерируется программным комплексом моделирования автоматически на основе элементных и топологических уравнений. Топологические уравнения выражают фундаментальные законы сохранения физических величин (токов, сил, моментов, расхода, теплового потока), за их формирование отвечает ядро программного комплекса. Элементные уравнения задают законы функционирования отдельных компонентов системы.

Современные программные комплексы анализа динамики поставляются с библиотеками моделей различных элементов технических систем. Состав библиотек зависит от целевого назначения программного комплекса. Так библиотеки моделей систем проектирования изделий электроники (PSPICE, P-CAD) содержат до нескольких тысяч моделей электронных компонентов, выпускаемых промышленностью. Специфика изделий электронной промышленности такова, что при проектировании практически всегда используется ограниченная элементная база, а математические модели отдельных компонентов хорошо изучены. Эти обстоятельства позволили создать столь обширные библиотеки моделей. Возможность пользователя найти в базе моделей любой требуемый элемент, во многом, определили успех в практике проектирования систем моделирования электронных изделий.

Любой современный ТО состоит из совокупности взаимодействующих физически разнородных систем. Для моделирования таких объектов могут потребоваться самые разные, в том числе, ещё не изученные, основывающиеся на экспериментальных данных модели. В этом случае охватить в библиотеке моделей, поставляемой с программным комплексом, всё многообразие элементов, которые могут потребоваться пользователю при моделировании его ТО невозможно. Следовательно, для поддержки эффективного моделирования необходимо обеспечить пользователя соответствующими средствами разработки компонентов моделей, выработать методологию разработки.

## Компонентные модели.

Обычно моделируемая техническая система имеет естественную структуру, информация о которой должна быть сохранена в модели. Наличие структуры уже само по себе является косвенным свидетельством сложности системы. Поскольку оперативные возможности человека ограничены (по закону Миллера человек в состоянии обрабатывать 72 смысловых единицы информации), то исследователю или инженеру удобно сосредоточится сначала на поведении отдельных компонентов, а затем на их взаимодействиях.

Далее, возможно, что компоненты моделируемой технической системы являются типовыми, и модели этих компонентов уже были разработаны и включены в библиотеки компонентов. В идеальном случае инженер просто «собирает» разрабатываемую систему на виртуальном стенде из готовых компонентов (таких, например, как электродвигатель, редуктор и регулятор) так же, как это происходило бы «в металле», совсем не задумываясь о математических моделях, уравнениях и т.п., и предполагая, что компоненты в его компьютерной модели будут взаимодействовать так же, как они взаимодействовали бы в реальном мире.

Основой любой системы моделирования является блок (компонент, многополюсник), аналог реального устройства или элемента технической системы. Компонент – это независимый, функционирующий по своим внутренним законам в соответствии с определенной математической моделью элемент, взаимодействующий с другими компоненты через «входы-выходы» или «контакты».

Система моделирования должна поддерживать иерархическую структуру моделируемого объекта. Это означает, что любая функциональная схема объекта, а также любая её связная часть сама может рассматриваться как сложный блок (или компонент). Такие сложные блоки со своей внутренней структурой вновь можно соединять между собой, строя иерархические многоуровневые системы.

Принципы компонентного моделирования нашли разную реализацию в существующих системах моделирования. При этом основные различия относятся к тому, каким образом задаются математические модели отдельных компонентов. Здесь можно выделить три подхода:

* Разработка модели осуществляется на одном из языков программирования. Это может быть как специализированный язык программного комплекса моделирования, так и язык программирования общего назначения (Java, C++ и др.). Такой подход реализован в программах серии ПА (ПА4-ПА9).
* Пользователь может задать математическую модель блока, используя естественные математические нотации (алгебродифференциальные уравнения и формулы в естественной математической нотации). Используется при разработке моделей в графической среде Model Vision Studium.
* Пользователь может собрать новый прикладной элементарный блок из имеющихся в библиотеке моделей блоков или вынужден написать его программный код на одном из языков программирования, как и в первом случае. Таким способом выполняется разработка моделей в пакете Matlab Simulink.

Каждый компонент обладает набором внешних или интерфейсных переменных, которые определяют его взаимодействие с другими компонентами. От того, какие ограничения накладываются на интерфейсные переменные блоков зависит то, как будет формироваться совокупная система уравнений при моделировании технического объекта. Здесь также можно выделить несколько подходов:

* Внешние переменные разделяются на входные, которые могут быть изменены только вне блока и выходные, значения которых вычисляются только внутри блока. Такие блоки называются ориентированными. При моделировании явно задаются причинноследственные связи. Ориентированные блоки используются при моделировании в пакете Simulink Matlab.
* Разделения на входные и выходные переменные нет. Значения интерфейсных переменных блока могут меняться как внутри, так и вне блока. Такие блоки называются неориентированными. Подход можно назвать «физическим». Причинно-следственные связи при моделировании явно не задаются. Данный подход реализован в языке

Modelica, в системах проектирования электронных схем.

При реализации второго подхода важен алгоритм формирования общей системы уравнений, пригодной для численного интегрирования. В программных комплексах серии ПА используется концепция физических аналогий. При этом возможно моделирование систем, для которых существуют топологические уравнения, аналогичные законам Кирхгофа, а использование специальных преобразующих компонентов позволяет моделировать физически разнородные системы.

## Объектно-ориентированное моделирование

В настоящее время объектно-ориентированный подход широко используется в программировании. Его преимущества по сравнению с более ранними парадигмами разработки программных систем были описаны в литературе не раз. При моделировании технических систем принципы объектно-ориентированного подхода на сегодняшний день широкого распространения не получили, хотя работы в этом направлении ведутся и термин «объектно-ориентированное моделирование» (ООМ) можно встретить все чаще, но на практике не так уж много пакетов его поддерживают. На сегодняшний день наиболее полно идеи ООМ поддерживает язык Modelica.

Ранее практически все языки моделирования строились как расширения некоторого универсального алгоритмического языка (Fortran, Algol). Основная причина этого заключается в том, что при описании моделей неизбежно возникает потребность в более мощных средствах, чем те которые может предоставить специализированный язык описания моделей. Всегда найдутся модели со сложными алгоритмами обработки данных, для которых потребуются возможности мощного языка программирования. Также часто необходима возможность использования при разработке модели внешних, созданных вне системы моделирования, процедур. Например, в состав модели могут входить компоненты, выполняющие обработку изображений.

С развитием программирования появились такие объектноориентированные языки, как Java, C#. С их помощью возможна реализация концепции объектно-ориентированного моделирования.

Применительно к моделированию понятия объектно-

ориентированного подхода, такие как, объект, инкапсуляция, наследование, полиморфизм имеют ряд важных особенностей. Рассмотрим их.

Объектом называют сущность, которая инкапсулирует в себе данные и методы и взаимодействует с внешним окружением через определенный интерфейс. Каждый объект является экземпляром некоторого класса. При компонентном моделировании блоки, из которых строится функциональная схема технического устройства, являются объектами некоторых классов, при этом существует базовый для всех блоков класс устройства (или компонента). Каждый класс компонента технической системы инкапсулирует определенную математическую модель, в соответствии с которой данный компонент функционирует. В классе компонента могут быть определены необходимые поля с данными и алгоритмические процедуры и функции.

При наследовании все элементы базового класса включаются в производный класс. При этом изменения сделанные в структуре базового класса автоматически отражаются в производном. Целью наследования является модификация базового класса: дополнение его новыми элементами и/или переопределение существующих. Например, пусть требуется создать модель усилителя с насыщением (с нелинейной характеристикой). Естественно строить описание этого специального усилителя на базе обычного линейного усилителя, соответствующим образом модифицировав в производном классе функциональность базового. При этом везде, где используется базовый класс, можно будет использовать и производный.

Полиморфизмом в объектно-ориентированном подходе называется возможность использования вместо объектов одного декларированного класса объектов другого, называемого замещающим, совместно с первым. При этом замещающий класс либо является потомком декларированного класса, либо определен интерфейс и оба класса реализуют этот интерфейс. Таким образом, можно выделить «совместимость по наследованию» и «совместимость по интерфейсу». Интерфейсом называется совокупность спецификаций функций (методов) без их реализации, т.е. совокупность абстрактных методов. Класс реализует некоторый интерфейс, если в определении класса имеются реализации для всех методов, указанных в определении интерфейса. Применительно к объектно-ориентированному моделированию понятие интерфейса имеет также и другую интерпретацию. Каждый блок содержит совокупность внешних контактов, при помощи которых осуществляется взаимодействие блока с другими блоками. Интерфейсом блока является совокупность его внешних контактов.

При разработке модели блока в распоряжении пользователя должна быть графическая среда, а текстовое представление в виде декларации класса должно генерироваться средой автоматически. Подобный подход при разработке программных систем реализуется с использованием языка UML. При создании моделей компонентов технических систем также может быть использован язык UML, но лучше подходят для этой цели специальные формы графического описания модели (эквивалентные схемы, блок-диаграммы, карты состояний).

## Обзор современных программных средств

В данном обзоре освещаются вопросы создания компонентов пользователя в современных программных комплексах анализа динамики технических систем, рассмотрен пример реализации объектноориентированного подхода к моделированию. Уделено внимание вопросам автоматической генерации программного кода компонентов на основе графического описания.

### Подсистема Simulink пакета Matlab

Пакет Matlab приобретает в нашей стране все большую популярность. Первоначально ориентированный на исследовательские задачи, пакет в последние годы стал рабочим инструментом инженеров, преподавателей и студентов. Одной из основных причин широкой распространенности пакета является наличие обширного набора средств для решения разнообразных инженерных и научных задач.

Подсистема Simulink пакета Matlab – это интерактивная среда для моделирования и анализа динамических систем, использующая графический язык блок-диаграмм. Simulink включает в себя обширную библиотеку блоков (непрерывные, дискретные элементы, математические функции, нелинейные элементы, источники сигналов, средства отображения), которые можно использовать при моделировании.

Среда позволяет объединять блок-диаграммы в составные блоки, что обеспечивает иерархическое представление структуры модели. Возможно создание блоков и библиотек, определенных пользователем. Для того чтобы выделить часть системы в составной блок, можно пойти следующим путем: перенести из библиотеки блок Subsystem, представляющий собой составной блок, и, раскрыв его окно, сформировать в нем структуру подсистемы. В пакете Simulink предусмотрен и другой способ: построив блок-диаграмму в основном окне создаваемой модели, можно выделить часть модели, которую необходимо объединить в составной блок и выбрать пункт меню Edit/Create Subsystem. В результате в основном окне будет образован составной блок, включающий выделенную часть блок-диаграмы.

Подсистема Stateflow Similink дает возможность моделировать поведение гибридных (содержащих дискретные и непрерывные компоненты) систем, базируясь на картах состояния Харела. Созданные пользователями пакета Simulink модели рассматриваются как объекты, закон управления которыми реализуется в Stateflow.

Для генерации С-кода на основе диаграмм Stateflow используется подсистема Stateflow Coder. Применяя Stateflow и Stateflow Coder, пользователь может генерировать код на языке С только для управляющих моделью блоков, реализованных при помощи Stateflow. Подсистема Real-Time Workshop дополняет Simulink и Stateflow Coder, обеспечивая автоматическую генерацию С-кода для моделей, разработанных в Simulink.

Подсистема Real-Time Workshop обеспечивает следующие возможности:

* Автоматическая генерация программного кода для различных платформ;
* Интеграция с системой Matlab и пакетом Simulink;
* Графический интерфейс;
* Открытая, расширяемая архитектура.

Процесс генерации кода и компиляции файла модели с использованием Real-Time Workshop иллюстрируют следующие схемы (Рис. 2,

3).

Модель Simulink

Параметры

конфигурации

Шаблон make-

файла

пользователя

Генерация кода

Создание make-файла

Код модели:

model.c

model.h

model\_private.h

...

make-файл

model.mk

Make -f model.mk

Выполняемый модуль:

model.exe

Автоматическая

генерация

Разработка

пользователем

Выполнение

Рис. 2. Разработка приложения.

Simulink

model.mdl

Real-time

Workshop

model.rtw

Компилятор

Make

model.

с

model.exe

model.

mk



Файлы блоков



Файлы со

встраиваемыми S-

функциями



Библиотеки

функций

компилятора



Динамически

созданные файлы

блоков

Файлы поддержки

интерфейса времени

выполнения

Рис. 3. Генерации кода и компиляции файла модели с использованием Real-Time Workshop.

Real-Time Workshop обеспечивает также создание интерфейса времени выполнения, который используется для запуска модели на выполнение (Рис.

4).

Подсистема выполнения кода

моделей, вызовы функций

операционной системы, ввод-

вывод, сохранение результатов

Код модели и S-

функции

Интерфейс модели времени выполнения

Рис. 4. Интерфейс модели.

Пакет Simulink при создании моделей позволяет использовать в качестве блоков ранее созданные модели. Таким образом, модель может содержать ссылки на другие модели. Данная возможность чрезвычайно полезна при создании крупномасштабных приложений для моделирования. Для каждой модели, на которую содержится ссылка в модели верхнего уровня, осуществляется генерация кода и создается бинарный файл, из которого вызываются функции в процессе моделирования.

Версия 6 пакета Simulink позволяет моделировать, тестировать и реализовывать каждый компонент отдельно, еще до вставки в общую модель системы. В версии 6 улучшена по сравнению с предыдущими версиями поддержка шин для задания интерфейсов блоков, введено описание шин, как структур при генерации кода.

### Язык Modelica

Язык Modelica разработан ассоциацией “Modelica” ([www.modelica.org)](http://www.modelica.org/), штаб-квартира которой расположена в городе Линкопинг, Швеция. Язык Modelica является свободно распространяемым, объектно-ориентированным языком для моделирования сложных физически разнородных систем.

Пользователь языка обеспечен возможностями по созданию новых библиотек моделей, при этом возможно многократное использование и модификация существующих моделей. Модели строятся из независимых компонентов, оформленных в виде классов. Особенностью языка является возможность строить модели из компонентов с ненаправленными связями (неориентированные блоки).

В языке Modelica четко разделяются непрерывные и дискретные процессы. Непрерывные процессы описываются в виде уравнений (equation), дискретные – с помощью алгоритмов (algorithm). Форма уравнений может

меняться в зависимости от наступления событий.

Язык Modelica оперирует только с классами. Все переменные описываются с их помощью. В язык введены различные специализированные типы классов – model, connector, record, block, type, package и собственно class.

Вместо class может быть использован любой из перечисленных служебных классов. Рассмотрим назначение специализированных классов:

* record (запись). Данный класс не может содержать уравнений ни в каком из своих компонентов. Аналогичен структуре языка C;
* type (тип). Определяет тип некоторого объекта;
* connector (соединение). Используется для описания соединений компонентов. Не может содержать уравнений;
* block (блок). Класс предназначен для описания моделей типа «вход-состояние-выход»;
* package (пакет). Может содержать только определение классов и констант;
* function (функция). Объектное представление функции. Допускается только одна алгоритмическая секция.

Модель (model) – основная структурная единица, которая может включать в себя:

* поля предопределенных типов, таких как Real, Integer, Boolean,

String;

* структурированные компоненты, необходимые для создания иерархических моделей;
* массивы (в том числе массивы моделей), вектора, матрицы;
* уравнения (equation) и/или алгоритмы (algorithm);
* функции;
* соединения.

При объявлении переменных можно использовать специальные префиксы, которые управляют возможностями модификации значений переменных (constant, input, output и др.).

Язык содержит средства для защиты классов от несанкционированного использования, в том числе при наследовании – заголовки Public, Protected, Replaceable, Final. Конструкция redeclaration позволяет переопределять элементы базового класса.

Соединение компонентов задается при помощи оператора connect, расположенного в секции equation. Оператор должен содержать две ссылки на соединители, которые либо являются элементами того же класса, что и оператор connect, либо элементами одного из его компонентов. Соединение задает равенство двух переменных.

По введенной пользователем модели Modelica строит совокупную систему уравнений. Для этого иерархическое описание переводится в линейное, образуя единую систему уравнений, добавляются уравнения, описанные в соединениях. Результирующая система уравнений может иметь разрывы и переменную структуру.

### Программные комплексы серии ПА (ПА7-ПА9)

В качестве входной информации комплексы серии ПА воспринимают описание эквивалентной схемы моделируемого объекта на текстовом входном языке в виде списка элементов, составляющих объект. Для каждого элемента указывается имя его математической модели, имена узлов подключения и значения параметров. Отметим, что входной язык комплексов ПА допускает иерархическое описание сложных объектов.

В ПА9 для задания описания объекта моделирования используется редактор эквивалентных схем, а текстовое описание топологии схемы генерируется редактором автоматически. Возможно объединение участка эквивалентной схемы в составной блок. На рис.5 приведен пример составного блока.

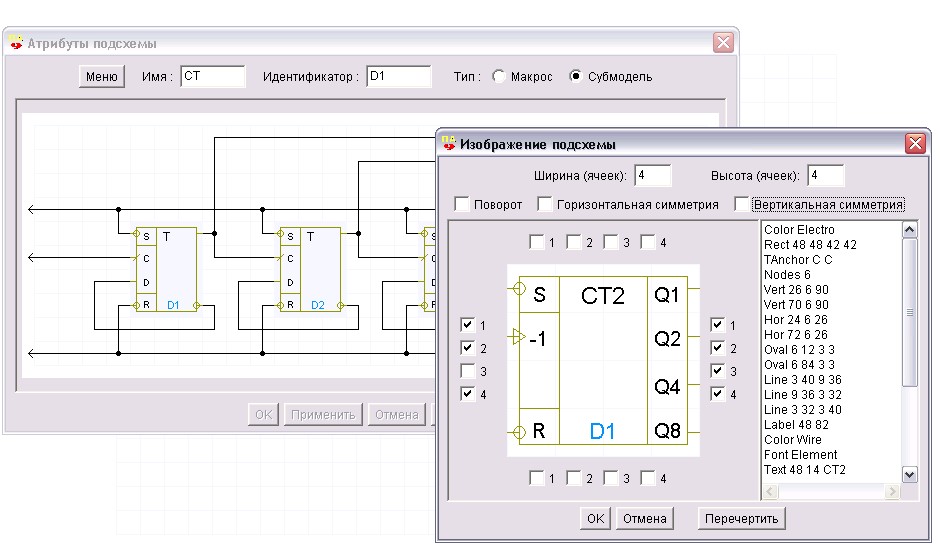


Рис. 5. Составной блок и его условное изображение.

Для разработки компонентов применяется специализированный язык, описание модели на котором транслируется в декларацию класса языка Java. Описание модели состоит из нескольких разделов, в которых задается топология внутренней эквивалентной схемы модели, параметры модели, выражения для производных, необходимые численному решателю.

Визуальные средства, автоматизирующие разработку модели, отсутствуют.

В ПА9 было введено понятие макромодели. Макромодели используются для замены схем сложных вложенных моделей на простое математическое описание, что должно ускорить процесс численного решения системы уравнений при моделировании. Использование макромодели вместо обычной модели указывается в разделе описания топологии схемы. В ПА9 для каждой модели возможно создание только одной макромодели.

### Выводы

В современных пакетах моделирования динамических систем объектно-ориентированные технологии моделирования используются не достаточно широко. Исключение составляют лишь несколько пакетов (например, Modelica). Также в языке Modelica используется раздельное описание дискретного и непрерывного поведения, что позволяет эффективно моделировать гибридные системы. Это, несомненно, большое достоинство языка. В настоящий момент визуальные средства разработки на языке Modelica отсутствуют, хотя ведутся работы по интеграции языка в пакет Matlab. Язык Modelica является интерпретируемым.

На наш взгляд, современный инструмент моделирования, обеспечивающий возможность определения классов компонентов пользователем, должен быть компилирующим. Компиляция модели в исполняемый код дает ряд важных преимуществ по сравнению с интерпретируемыми моделями. Например, открываются возможности по интеграции моделей с внешними приложениями. Использование технологий Java или Microsoft .NET позволяет создавать переносимые между различными платформами модели, поскольку компиляция выполняется в байт-код виртуальной машины.

Практически всегда в компилирующих пакетах моделирования используется некоторый промежуточный язык программирования (Fortran, C, Pascal, Java). Код модели, разработанный на внутреннем языке программного комплекса, транслируется в код на промежуточном языке, который затем компилируется. К примеру, так выполняется разработка моделей в ПА7-ПА9. Более перспективным является подход, при котором пользователь работает с графической средой, а программный код модели генерируется автоматически.

Пакет Matlab поддерживает автоматическую генерацию кода моделей на языке С на основе блок-диаграмм. При этом, хотя и есть возможности генерации кода для разных платформ, сам код между различными платформами непереносим. Кроме того, не поддерживаются объектно-ориентированные технологии разработки моделей.

## Реализация компонентного моделирования в ПА10

Основными достоинствами пакета ПА10 должны стать поддержка объектно-ориентированного моделирования и возможности визуальной разработки компонентов.

Математическая модель компонента может быть представлена в различных видах – это могут быть дифференциальные уравнения в текстовом виде, эквивалентная схема, передаточные функции.

В рамках подхода, реализованного в ПА10, основным способом задания математических моделей компонентов являются эквивалентные схемы, состоящие из базовых двухполюсников (как линейных, так и нелинейных элементов), многополюсников, а также зависимых и независимых источников переменных типа потока и потенциала. Элементы эквивалентной схемы связываются между собой в узлах. Каждый компонент имеет внешние контакты, служащие для его включения в схему. Система уравнений, являющаяся математической моделью компонента будет незамкнутой. При моделировании совокупная замкнутая система уравнений, передаваемая численному решателю, формируется из уравнений компонентов и уравнений связей.

Программной моделью элемента технической системы является класс, производный от базового для всех компонентов класса многополюсника EqComponent. Следует сказать, что в данном случае не идет речь о классе, написанном на некотором конкретном языке программирования (С++, Java или C#), под классом понимается общее понятие объектно-ориентированного моделирования. Элементы эквивалентной схемы модели компонента будут представлены приватными полями класса, а параметры модели – публично доступными свойствами. Инициализация элементов модели, установление связей между ними в соответствии с эквивалентной схемой, задание начальных значений полей будет выполняться в методе класса InitializeComponent, вызываемом из конструктора класса при создании экземпляра компонента.

Компонент должен содержать информацию, задающую его математическую модель, кроме эквивалентной схемы модели компонента это могут быть, к примеру, дифференциальные уравнения в текстовом виде. При этом сам компонент не должен обязательно выполнять вычисления, необходимые при численном решении систем уравнений в процессе моделирования. Для перехода от математической модели компонента в той или иной форме к значениям, необходимым численному решателю, был введен интерфейс контроллера компонента IСomponentController. Объекты, реализующие этот интерфейс, могут, в частности, вычислять элементы матрицы Якоби, формируемой при интегрировании системы дифференциальных уравнений.

Нередко для одного и того же элемента технической системы может потребоваться разработка нескольких математических моделей, различающих степенью точности, адекватности и экономичности. В этом случае для каждой математической модели разрабатывается своя программная модель в виде класса, при этом все классы реализуют общий интерфейс, определенный для данного элемента технической системы, и их экземпляры являются взаимозаменяемыми при сборке эквивалентных схем и в других случаях.

Комплекс ПА10 создавался с использованием технологий Microsoft .NET. Среда Microsoft .NET Framework при реализации компонентного моделирования обеспечивает следующие возможности:

* Поддержка программирования на нескольких языках. Наличие большого числа свободно распространяемых компиляторов различных языков (C#, VB .NET, Fortran и др.);
* С помощью механизма рефлексии (reflection) доступна информация о внешнем интерфейсе объекта, созданного с помощью любого языка, поддерживающего соглашения .NET CLR;
* Генерация программного кода на нескольких языках на основе обобщенных синтаксических конструкций. Компиляция и подключение программных модулей во время выполнения;
* Возможность вызова функций из любых библиотек DLL для платформы Windows. В частности, можно использовать библиотеки на Fortran, содержащие реализации численных методов.

При разработке компонентов для ПА10 осуществляется компиляция их программного кода в .NET-сборки – программные модули (\*.dll, \*.exe), содержащие бинарный выполняемый код для виртуальной машины CLR. Кроме того, такие сборки могут содержать любые необходимые для использования компонента ресурсы – текстовые описания на разных языках, графические изображения, формы пользовательского интерфейса.

Разработанные для ПА10 компоненты в виде бинарных файлов могут быть использованы во внешних приложениях, поддерживающих интерфейс со средой Microsoft .NET. Также возможно создание бинарных сборок не только для отдельных компонентов, но и для «полных» моделей, описываемых замкнутой системой алгебро-дифференциальных уравнений. Такие модели могут использовать возможности математического ядра ПА10 во внешних приложениях.

При этом, например, может осуществляться интеграция моделей, разработанных в ПА10 с системами твердотельного геометрического моделирования (например, с САПР SolidWorks, поддерживающей технологии Microsoft .NET) для создания комплексных электронных моделей изделий, отражающих, как геометрические, так функциональные характеристики объектов проектирования. При этом геометрическая модель изделия, созданная в системе твердотельного моделирования может использоваться для определения параметров функциональной математической модели, а на основе вычисленных в процессе моделирования величин может быть изучено движение различных частей изделия, определены возможные пересечения и т.п. При этом за счет моделирования на уровне обыкновенных дифференциальных уравнений достигается высокая экономичность, обеспечивается быстрота получения результатов.

Для моделей в виде бинарных файлов могут быть разработаны специальные графические интерфейсы, при помощи которых пользователь будет осуществлять управление моделированием, исследовать полученные результаты. Такой подход к изучению поведения моделей иногда называют концепцией «виртуального испытательного стенда».

Модели в виде .NET-сборок могут использоваться в составе ASPXстраниц и Web-сервисов, что дает возможности удаленного взаимодействия с ними через Internet.

Программный код компонента для ПА10 может быть написан на одном из языков, для которого существует реализация компилятора для платформы Microsoft .NET (C#, VB .NET, J#, Fortran и др.). Кроме того, для программного комплекса ПА10 было создано специальное визуальное средство разработки компонентов (Редактор компонентов), используя которое разработчик моделей сможет создавать новые компоненты, не прибегая к написанию программного кода. Программный код компонента в виде декларации класса будет сгенерирован автоматически.

Перечислим основные возможности Редактора компонентов для ПА10:

* Создание моделей компонентов в редакторе эквивалентных схем;
* Задание функциональных зависимостей для математических моделей нелинейных элементов;
* Создание новых компонентов путем расширения функциональности уже имеющихся компонентов;
* Создание компонента на основе связного участка любой эквивалентной схемы;
* Автоматическая генерация программного кода модели компонента на любом языке, поддерживаемом платформой

Microsoft .Net;

* Возможность редактирования программного кода модели компонента;
* Создание условного обозначения компонента при помощи специального графического редактора;
* Добавление разработанного компонента в библиотеку моделей.

Данные возможности должны позволить пользователям эффективно решать задачи разработки новых компонентов.

Процесс создание новой модели элемента технической системы заключается в получении эквивалентной схемы элемента, состоящей из двухполюсников и многополюсников, в визуальном редакторе (Рис. 9), точно также выполняется создание эквивалентных схем для комплексных технических систем. Следует учесть, что в случае модели элемента часть контактов являются внешними. При помощи внешних контактов модель элемента впоследствии будет включаться в схему. На рис.6 приведена эквивалентная схема диода, внешние контакты обозначены буквами А (анод) и К (катод).

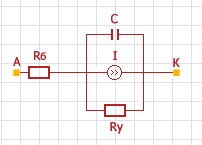


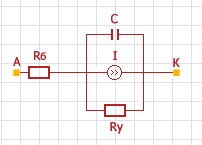
Рис. 6. Эквивалентная схема диода.

Пользователь должен иметь возможность определения параметров модели элемента. В качестве параметров модели могут выступать некоторые параметры элементов эквивалентной схемы, а также заданные пользователем величины (Рис. 8). Для модели диода такими параметрами будут являться: сопротивления Rб, Rу и ряд величин, учитывающих свойства p-n перехода. Для нелинейных элементов должна быть предусмотрена возможность задания функциональной зависимости между переменными типа потока и потенциала (для диода это будет зависимость тока I от напряжения, основанная на аппроксимации вольт-амперной характеристики диода и зависимость значения ёмкости С от напряжения, учитывающая динамические свойства p-n перехода). Процесс создания модели элемента завершается получением условного обозначения для нового элемента (Рис.7).



Рис. 7. Условное обозначение диода.

После этого разработанная модель элемента может быть добавлена в библиотеку моделей программного комплекса и использована при составлении эквивалентных схем технических объектов.



Модель диода

1

. Создание модели компонента

I=I(U)

C=C(U)

Параметры модели

Rб

Rу

m

fi

C

o

fi

0

Внешние контакты

2

. Автоматическая генерация

программного кода класса

3

. Разработка условного обозначения



компонента

4

. Компиляция, включение

компонента в библиотеку

моделей



Рис. 8. Разработка компонента.

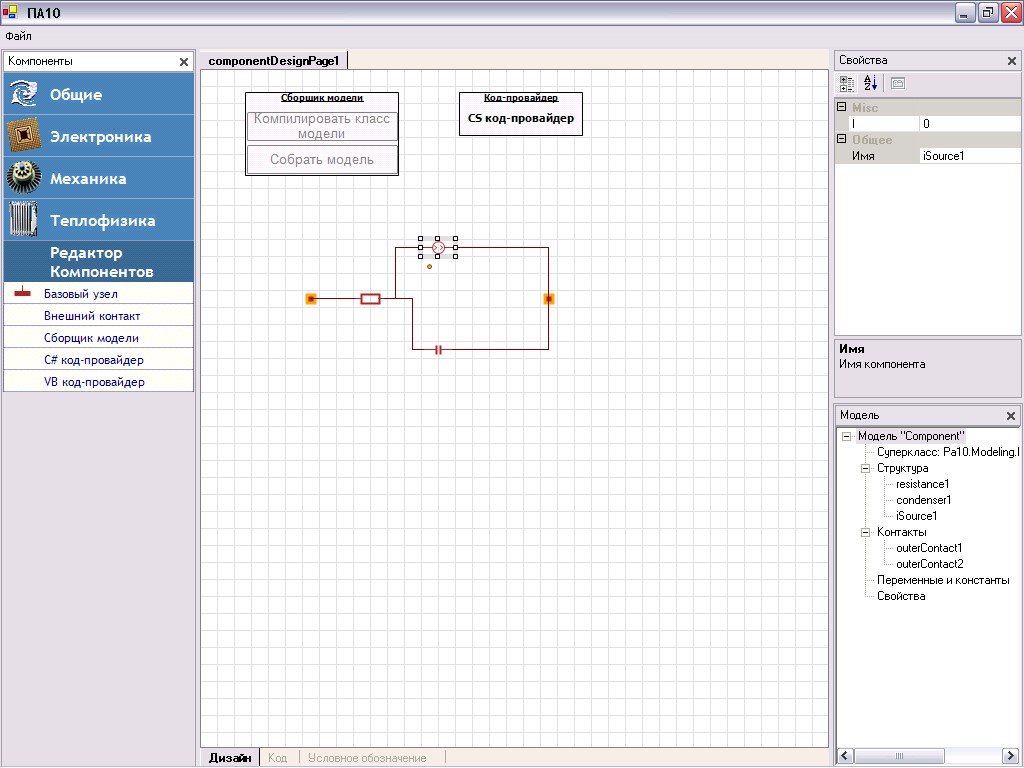


Рис. 9. Интерфейс Редактора компонентов. Создание эквивалентной схемы.

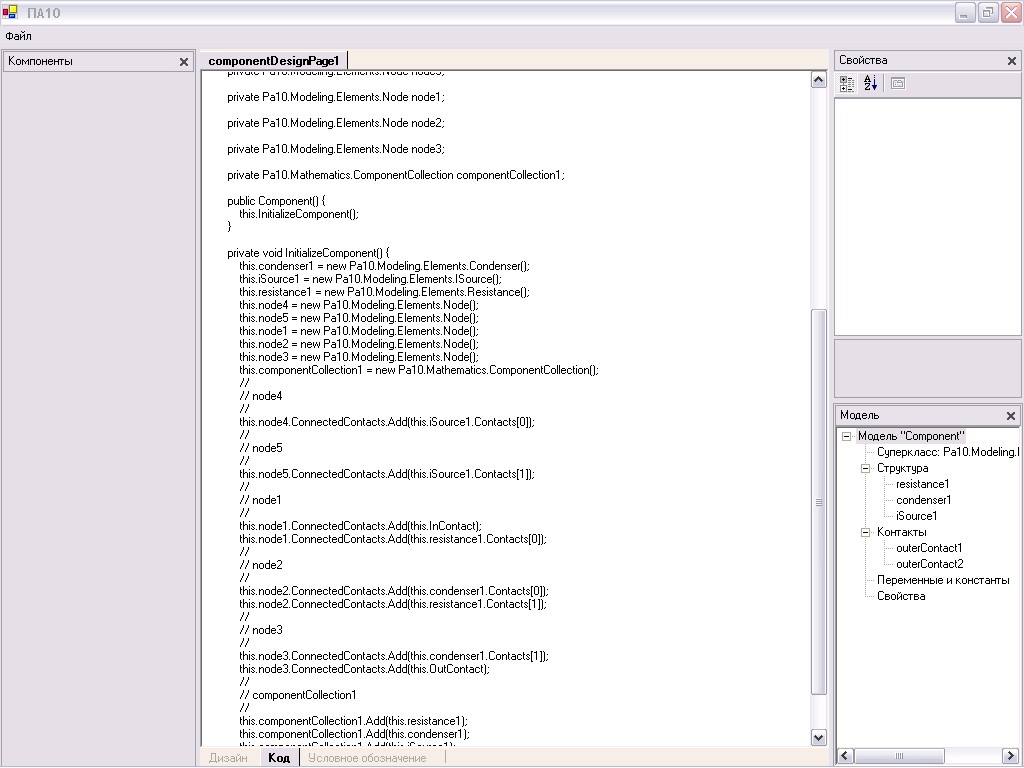


Рис. 10. Интерфейс Редактора компонентов. Страница редактирования программного кода.

# Конструкторская часть.

## Разработка подсистемы автоматической генерации программного кода.

Довольно часто автоматическая генерация программного кода (кодогенерация) применяется как средство, позволяющее избежать рутинной работы, избавится от которой обычными приемами объектноориентированного программирования не удается. Например, при помощи кодогенерации может быть автоматизировано создание типизированных коллекций (в особенности для языков, не имеющих средств аналогичных шаблонам), бизнес-классов, прокси-классов для удаленных объектов и т.д. В некоторых случаях кодогенерация применяется в целях оптимизации. Иногда автоматически генерируется код, предназначенный для выполнения на платформах, отличных от той, на которой выполняется разработка (код для промышленных контроллеров, встраиваемых систем и т.п.).

В последнее время интенсивно развиваются средства автоматизированной разработки программных продуктов (CASE-средства), в которых в той или иной форме используется автоматическая генерация программного кода. Чаще всего в CASE-средствах код программных модулей (декларации классов, процедуры) генерируется на основе диаграмм (например, диаграммы языка UML), применяется также генерация на основе шаблонов. В качестве примеров CASE-средств можно привести продукты компании

Rational Software.

Средства быстрой разработки приложений (Rapid Application Development – RAD) позволяют конструировать формы пользовательского интерфейса с помощью визуальных редакторов, при этом программный код, выполняющий инициализацию элементов интерфейса, генерируется средой разработки автоматически. К таким средствам относятся Microsoft Visual Studio, Borland Delphi и др.

Перспективным направлением является разработка систем, генерирующих код программных модулей многоуровневых приложений (бизнес-логика, взаимодействие с базами данных, Web-сервисы) на основе модели приложения. Например, пакет Visible Developer, работающий со средой Visual Studio.NET, позволяет автоматически сгенерировать до 90% кода

для бизнес-логики и работы с базами данных, основываясь на модели приложения. Visible Developer 3.0 позволяет генерировать полнофункциональные трехуровневые приложения на базе таких технологий, как XML, Web-службы, ADO.NET и Microsoft Visual Basic .NET. Обеспечивается представление физической структуры базы данных в виде одного самостоятельного объекта бизнес-логики.

В Microsoft .NET Framework кодогенерация также применяется довольно часто. Например, при помощи кодогенератора на основе xsd-файлов со схемой данных создаются типизированные наборы данных (typed datasets), создаются прокси-классы для веб-сервисов. Во время выполнения генерируют код для

повышения производительности классы System.Xml.Serialization.

XmlSerializer,System.Text.RegularExpressions.Regex,

System.Web.UI.PageParser.

В нашем случае генерация кода применяется в целях автоматизации разработки программных моделей элементов технических систем (компонентов). На основе графического описания математической модели компонента в виде эквивалентной схемы осуществляется генерация объявления класса компонента. Данная возможность позволяет выполнять разработку компонентов для программного комплекса ПА10 в графической среде без написания исходного кода на языке программирования вручную.

Для выполнения генерации кода была разработана специальная библиотека классов Special.CodeGen для среды Microsoft .NET. Данная библиотека не является ориентированной только на решение задач, связанных с генерацией кода компонентов для программного комплекса ПА10, при её разработке ставились задачи создания подсистемы генерации кода, допускающей применение в различных приложениях.

К данному моменту разработано большое число систем генерации кода, для многих из них характерны следующие недостатки:

* Возможность генерации кода только на основе определенного представления входных данных (диаграммы, язык шаблонов);
* Поддержка генерации кода только для определенного класса приложений (например, графический интерфейс, доступ к базам данных, Web-сервисы);
* Ориентация на конкретные языки программирования. Данная особенность характерна для систем генерации кода на основе шаблонов.

При разработке библиотеки классов Special.CodeGen были

использованы подходы, призванные преодолеть указанные недостатки.

Для представления элементов программного кода в виде объектной модели независимой от конкретного языка программирования Microsoft была предложена технология CodeDOM (Code Document Object Model). При реализации подсистемы кодогенерации Special.CodeGen был сделан упор на использование технологии CodeDOM.

### Технология CodeDOM (Code Document Object Model)

Одним из преимуществ платформы Microsoft.NET является поддержка программирования на нескольких языках. Для генерации программного кода на нескольких языках следует воспользоваться разработанной Microsoft технологией CodeDOM (Code Document Object Model). Данная технология обеспечивает также возможности компиляции программных модулей во время выполнения. Полученные программные модули (сборки) могут быть динамически загружены во время выполнения программы и использованы точно так же, как если бы они изначально входили в состав программного комплекса (используется т.н. позднее связывание).

Для представления обобщенных, не зависимых от языка, синтаксических конструкций служат классы из пространства имен System.CodeDom (Приложение 1). На каждую конструкцию в этом

пространстве имен присутствует соответствующий класс. Например, объявление метода описывает класс CodeMemberMethod, использование поля CodeFieldReferenceExpression. Все классы являются производными от общего базового класса CodeObject.

Важную роль среди классов пространства имен System.CodeDom играет класс CodeCompileUnit. В объекты данного класса должны быть иерархически вложены все остальные синтаксические конструкции

(пространства имен, классы и т.д.).

Процесс генерации кода выглядит следующим образом. Сначала выстраивается дерево обобщенных синтаксических конструкций языка, которое затем CodeDOM-провайдером конкретного языка преобразуется в исходный текст. CodeDOM-провайдер представляет собой объект класса CodeDomProvider, и служит для получения экземпляров генераторов кода и компилятора (Таблица 1).

Таблица 1. Основные методы класса CodeDomProvider.

|  |  |
| --- | --- |
| **Метод** | **Описание** |
| CreateCompiler | Создание компилятора |
| CreateGenerator | Создание генератора кода |
| CreateParser | Создание синтаксического анализатора кода |

Для обеспечения компиляции CodeDOM-провайдер конкретного языка при вызове метода CreateCompiler возвращает объект, реализующий специальный интерфейс System.CodeDom.Compiler.ICodeCompiler. У этого интерфейса есть несколько методов, осуществляющих компиляцию (Таблица 2).

Таблица 2. Методы интерфейса System.CodeDom.Compiler.

ICodeCompiler.

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  | | --- | | **Метод** | | CompileAssemblyFromSource | | CompileAssemblyFromDom | | CompileAssemblyFromFile | | |  | | --- | | **Выполняемое действие** | | Компиляция из исходного кода, представленного в виде строки. | | Компиляция из CodeDOM дерева. | | Компиляция файла, содержащего исходный | |
|  | |  | | --- | | код. | |

Для настройки компиляции в эти методы передаются настройки компилятора, представленные классом System.CodeDom.Compiler.

CompilerParameters. Вот основные свойства этого класса:

Таблица 3. Свойства класса System.CodeDom.Compiler.

CompilerParameters.

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  | | --- | | **Свойство** | | GenerateExecutable | | MainClass | | OutputAssembly | | |  | | --- | | **Значение** | | Генерировать запускаемый файл или библиотеку | | Имя класса, содержащего метод Main() | | Имя результирующей сборки | |
| |  | | --- | | IncludeDebugInformation | | |  | | --- | | Определяется, нужно ли включать отладочную информацию | |
| |  | | --- | | ReferencedAssemblies | | |  | | --- | | Коллекция, в которую нужно добавить все сборки, на которые производятся ссылки | |
| |  | | --- | | GenerateInMemory | | |  | | --- | | Генерировать в память. Опция, позволяющая не создавать при компиляции файл на диске. | |
| |  | | --- | | WarningLevel | | |  |  | | --- | --- | | Уровень выдаваемых  предупреждений | компилятором | |
| |  | | --- | | TreatWarningsAsErrors | | |  | | --- | | Воспринимать предупреждения как ошибки. Запрещает компилятору завершать компиляцию при наличии предупреждений. | |
| |  | | --- | | CompilerOptions | | |  | | --- | | Дополнительные настройки, специфичные для конкретного компилятора. Например, опция  /optimize для компилятора C# | |
| |  | | --- | | TempFiles | | |  | | --- | | В этой коллекции сохраняется список временных | |
|  | |  | | --- | | файлов, образованных при компиляции и не удаленных. | |

Схема алгоритма кодогенерации приведена на рис. 11.

В настоящий момент существует более 40 бесплатных реализаций компиляторов различных языков для платформы .NET, включая такие языки как FORTRAN и Smalltalk.

ICodeProvider provider =

new

CSharpCodeProvider();

compiler.CompileAssemblyFromSource

generator.GenerateCodeFromCompileUnit

ICodeGenerator generator =

provider.CreateGenerator()

compiler.CompileAssemblyFromD

om

ICodeCompiler compiler =

provider

.CreateCompiler();

Построение

синтаксического дерева

VB. NET

Java

C#, VB .NET, J#

Создание экземпляра CodeDOM-провайдера для

конкретного языка (C#, VB. NET, и др.)

Получение экземпляра компилятора

Компиляция на основе синтаксического

дерева

Получение экземпляра

генератора кода

Генерация кода на основе

синтаксического дерева

Компиляция из исходного кода

Сборка (AssemblyName.dll, AssemblyName.exe)

Рис. 11. Алгоритм кодогенерации.

Следующий пример иллюстрирует генерацию кода на основе синтаксического дерева программы, представленного в виде объекта класса CodeCompileUnit, и компиляцию исходного текста:

|  |
| --- |
| public static void GenerateGraph(CodeCompileUnit compileunit) {  // Создание CodeDomProvider’а языка C#  CSharpCodeProvider provider = new CSharpCodeProvider();  // Получение кодогенератора ICodeGenerator из объекта класса // CodeDomProvider.  ICodeGenerator gen = provider.CreateGenerator();  // Создание StreamWriter’а для записи в файл  StreamWriter sw = new StreamWriter("TestGraph.cs", false);  // Генерация исходного кода с использованием кодогенератора gen.GenerateCodeFromCompileUnit(compileunit, tw, new  CodeGeneratorOptions());  // Закрытие файла sw.Close();  }  public static CompilerResults CompileCode(string filepath) {  // Создание CodeDomProvider’а языка C#  CSharpCodeProvider provider = new CSharpCodeProvider();  // Получение компилятора ICodeCompiler из объекта класса // CodeDomProvider.  ICodeCompiler compiler = provider.CreateCompiler();  // Настройка параметров компилятора  CompilerParameters cp = new CompilerParameters(new string[]  {"System.dll"}, filepath.Substring(0,  filepath.LastIndexOf(".")+1)+"exe", false);  // Обозначает, что должен быть создан выполняемый файл (\*.exe),  // а не \*.dll  cp.GenerateExecutable = true;  // Запуск компиляции  CompilerResults cr = compiler.CompileAssemblyFromFile(cp,  filepath);  // Возврат результатов компиляции return cr; |

}

### Разработка библиотеки классов Special.CodeGen

При помощи разработанной библиотеки классов Special.CodeGen

может осуществляться построение синтаксических деревьев из не зависимых от языка конструкций, представленных классами пространства имен System.CodeDom. Формирование синтаксического дерева программы

осуществляется на основе определенной пользователем библиотеки объектной модели. При этом возможно применение библиотеки в различных приложениях.

Процесс построения синтаксического дерева CodeDOM заключается в последовательном выполнении набора директив кодогенерации, каждая директива отвечает за создание некоторого фрагмента синтаксического дерева.

Директивы могут быть вложенными друг в друга.

Для директив кодогенерации используется следующий интерфейс:

/// <summary>

/// Директива кодогенерации.

/// Классы, реализующие данный интерфейс, отвечают

/// за выполнение некоторого определенного задания /// в процессе кодогенерации. Например, "создать новый тип", /// "добавить метод класса". /// </summary>

public interface ICodeGenDirective

{

/// <summary>

/// Выполняет задание по кодогенерации.

/// </summary>

/// <param name="host">Хост кодогенерации,

/// содержит необходимую настроечную информацию.</param>

/// <returns>Результат.</returns>

CodeObject Generate(CodeGenHost host);

}

Данный интерфейс реализуют следующие классы:

* BuildTypeDirective отвечает за создание декларации нового типа

(класса, структуры). Содержит коллекцию вложенных директив,

выполнение которых приводит к заполнению создаваемого типа различными элементами, такими как, методы, конструкторы, свойства.

* CreateConstructor - создание конструктора. Вложенные директивы позволяют выполнить генерацию кода, который должен содержаться в теле конструктора.
* CreateMemberMethod – создание метода класса. Вложенные директивы позволяют выполнить генерацию кода, который должен содержаться в теле метода.
* CreateInitialization – производный от CreateMemberMethod, отвечает за создание метода, выполняющего сериализацию объектов “в код”. Функциональность этого класса мы рассмотрим далее.
* CreateProperty – создание свойства.
* CreatePropertyFromField – производный от CreateProperty, выполняет публикацию некоторого поля объекта в качестве свойства создаваемого в процессе генерации класса.

При работе с библиотекой этот список может быть расширен. Класс CompileUnitBuilder отвечает за создание единицы компиляции (CodeCompileUnit), которая затем может быть передана компилятору для компиляции сборки, а также кодогенератору для получения исходного текста

(см. рис. 11).

public class CompileUnitBuilder

{

public CompileUnitBuilder()

{

}

/// <summary>

/// Экземпляр CodeCompileUnit.

/// </summary>

public CodeCompileUnit CompileUnit

{

get { return this.compileUnit; }

}

/// <summary>

/// Коллекция директив кодогенерации, выполнение

/// которых в методе BuildCompileUnit приводит

/// к заполнению экземпляра CodeCompileUnit

/// пространствами

/// имён и типами.

/// Результатом выполнения каждой директивы из коллекции

/// должно являться описание пространства

/// имён(CodeNamespace),

/// либо объявление типа(CodeTypeDeclaration), в этом

/// случае новое объявление типа будет добавлено в

/// пространство имён принимаемое по-умолчанию

/// (задаётся при помощи свойства DefaultNamespace).

/// </summary>

public DirectiveCollection Directives

{

get { return this.directives; }

}

/// <summary>

/// Название пространства имён принимаемого /// по-умолчанию. /// </summary>

public string DefaultNamespace

{

get { return this.defaultNamespace; } set { this.defaultNamespace = value; }

}

private StringCollection imports= new StringCollection();

/// <summary>

/// Наименования импортируемых пространств имен, /// например System.WindowsForms, System.Xml, ...

/// </summary>

public StringCollection Imports

{

get { return this.imports; }

}

private CodeNamespace defNS = null;

/// <summary>

|  |
| --- |
| /// Выполняет построение единицы компиляции  /// (CodeCompileUnit)  /// в соответствии с заданными директивами кодогенерации.  /// </summary>  /// <param name="host">Хост кодогенерации, содержащий  /// необходимые  /// настройки и сервисы.</param> /// <returns></returns>  public CodeCompileUnit BuildCompileUnit(CodeGenHost host)    } |

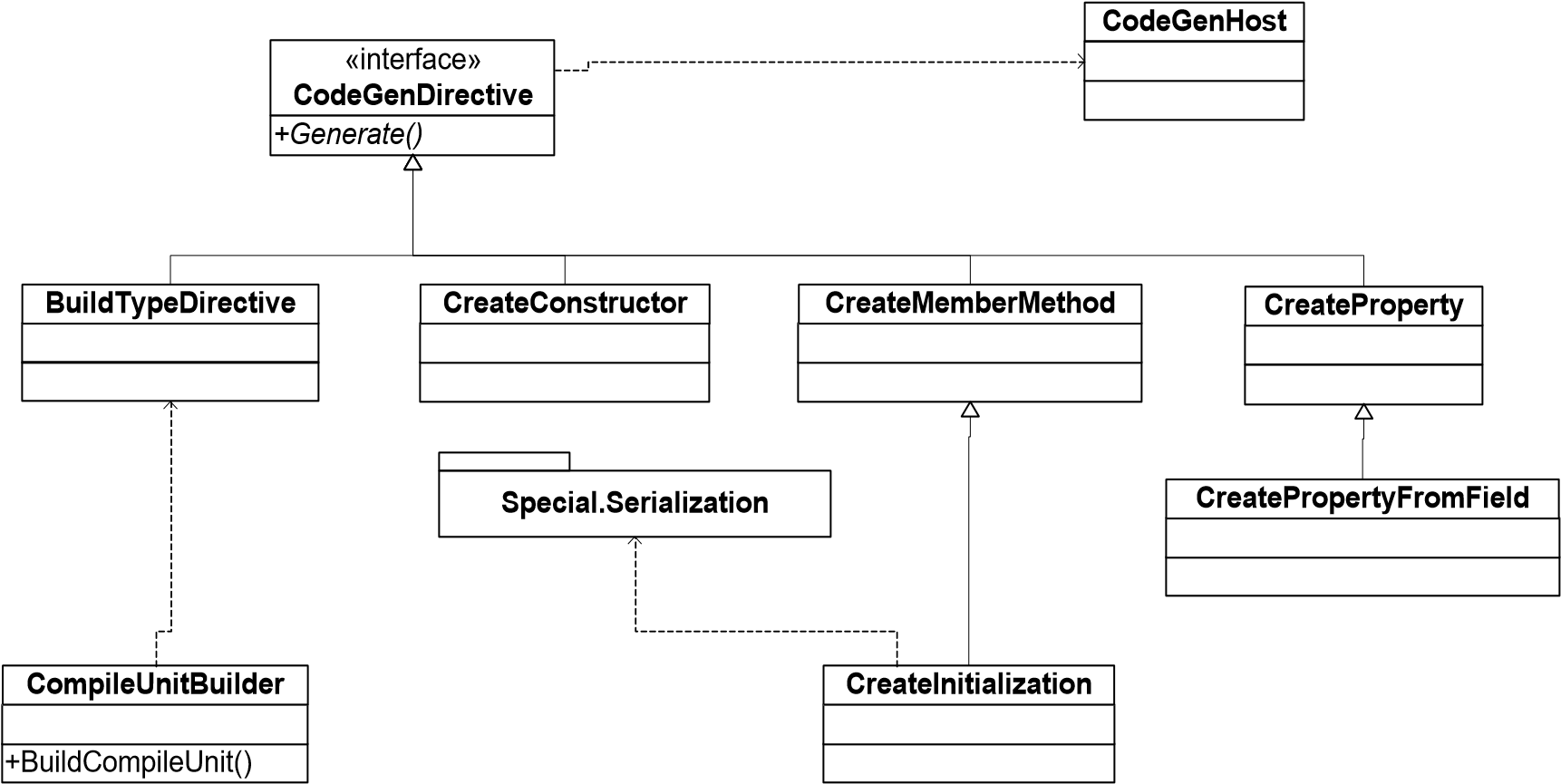


Рис. 12. Диаграмма классов библиотеки Special.CodeGen (представлены основные классы).

Объект класса CodeGenHost («хост» кодогенерации) предназначен для хранения различной настроечной информации, управляющей процессом кодогенерации, а также для предоставления в процессе кодогенерации необходимых сервисов (например, класс, реализующий интерфейс INameService отвечает за получение имен для генерируемых полей классов, переменных).

#### Генерация кода на основе объектной модели

Объектная модель – это структурированное в виде множества взаимосвязанных объектов представление информации из некоторой предметной области. Объектная модель описывает структуру объектов, составляющих систему, их атрибуты, операции, взаимосвязи с другими объектами. В объектной модели находят отражения те понятия и объекты реального мира, которые важны для разрабатываемой системы. Под предметной областью обычно понимается определенное прикладное направление, для автоматизации деятельности в котором разрабатывается данная система. Кроме этого можно говорить об объектной модели, отражающей понятия, используемые при разработке программы и важные для её функционирования, но не находящие аналогов в прикладной предметной области. Например, объектная модель пользовательского интерфейса.

Во время выполнения программы объектная модель представлена в памяти ЭВМ в виде совокупности взаимосвязанных объектов. Все объекты являются экземплярами некоторых классов. Поля объектов хранят определенные данные, объекты содержат ссылки друг на друга. Множество объектов образует ориентированный граф, вершинам графа соответствуют сами объекты, а ребрам – ссылки.

Задача генерации кода на основе объектной модели заключается в получении программного кода на основе существующего во время выполнения программы представления объектной модели в виде графа объектов. Возможны различные способы формирования графа объектов, служащего основой для генерации кода. Например, пользователь CASE-средства создает при помощи визуального редактора диаграммы, моделирующие разрабатываемую систему, при этом в памяти формируется граф объектов, отражающий заданную пользователем информацию, на основе которого выполняется генерация отдельных фрагментов программного кода разрабатываемой системы. При конструировании форм пользовательского интерфейса с помощью RAD-средств информация о расположении, атрибутах, вложенности элементов управления, окон и т.п. хранится в памяти также в виде графа объектов. Кроме графической формы представления исходной информации возможны и другие формы, например, текстовое описание в виде XML-документа.

obj1

obj2

obj3

obj4

Объектная модель

Исходная форма

представления информации

**Window**



aaaa

>

<

?xml version="1.0" encoding="utf-8"?

>

<

Root

<Root\_element>

<Properties>

<Properties\_element>

<Title>ssss</Title>

<Description />

<DocLang>ru-RU</DocLang>

Class1

Class2

Class1

Данные для

кодогенерации

Код

Рис.13. Генерация кода на основе объектной модели

При генерации кода на основе объектной модели выполняется сбор информации о состоянии объектов, на основе которых выполняется генерация.

Для этого используется разработанная нами библиотека Special.Serialization.

Сбор информации об объектах тесно связан с задачей сериализации. Сериализация (от англ. serial – последовательный) – преобразование графа объектов в последовательную форму. Результатом такого преобразования является набор записей, характеризующих состояние объектов в момент сериализации. Последовательная форма состояния объектов может быть сохранена на постоянном носители информации, переда по каналам связи, использована для создания копии исходных объектов. Процессом обратным сериализации является десериализация, при которой линейный набор записей преобразуется в объекты. Совместное использование обоих механизмов позволяет сохранять и передавать любые данные с которыми работает приложение.

.NET Framework предоставляет две технологии сериализации:

* Бинарная сериализация. Используется для полного и точного сохранения состояния объектов;
* Сериализация в XML. Могут быть сохранены только значения общедоступных (public) полей и свойств. Благодаря использованию открытого формата может использоваться для обмена данными между приложениями.

Классы сериализуемых объектов должны быть помечены специальным атрибутом Serializable, либо реализовывать интерфейс ISerializable. Т. е. возможность серализации должна быть предусмотрена при проектировании классов.

Для обеспечения сериализации любых объектов была разработана библиотека классов Special.Serialization, дополняющая возможности сериализации, предоставляемые .NET Framework. Использование данной библиотеки должно обеспечить большую гибкость при сериализации.

Сбор информации в процессе сериализации производится следующим образом: на основании полученной у сериализуемого объекта информации о типе (здесь используется механизм рефлексии – получении информации об интерфейсе объекта во время выполнения) извлекаются значения полей (и/или свойств) объекта. Если извлеченное значение представляет собой ссылку на другой объект, то объект, на который была найдена ссылка, также включается в процесс сериализации и подвергается процедуре сбора содержащихся в нем данных, в том числе ссылок на другие объекты. Для коллекций и массивов выполняется обход всех содержащихся в них элементов. Процедура сбора информации может быть настроена программистом – пользователем библиотеки, какие-то объекты могут быть из неё исключены, для определённых типов может быть указан состав полей и свойств, подлежащих сериализации и др.

Результатом работы описанного алгоритма является коллекция объектов типа ComponentSerializationData. Каждая запись коллекции содержит информацию об одном сериализуемом объекте (состав полей/свойств подлежащих сериализации). В такой форме информация об объекте может быть записана в поток, при этом вместо ссылок на объекты будут использоваться числовые идентификаторы. При кодогенерации собранная информация в поток не записывается, она используется для построения формальной модели программы в виде синтаксического дерева.

Преобразование в код осуществляется за несколько этапов. На первом этапе формируется модель данных, отражающая состояние исходного графа объектов. Основное назначение этого этапа – приведение данных о сериализуемых объектах в вид, удобный для последующего использования при построении синтаксического дерева программы. Также на данном этапе могут выполняться заданные пользователем преобразования данных. Для представления данных используются следующие классы:

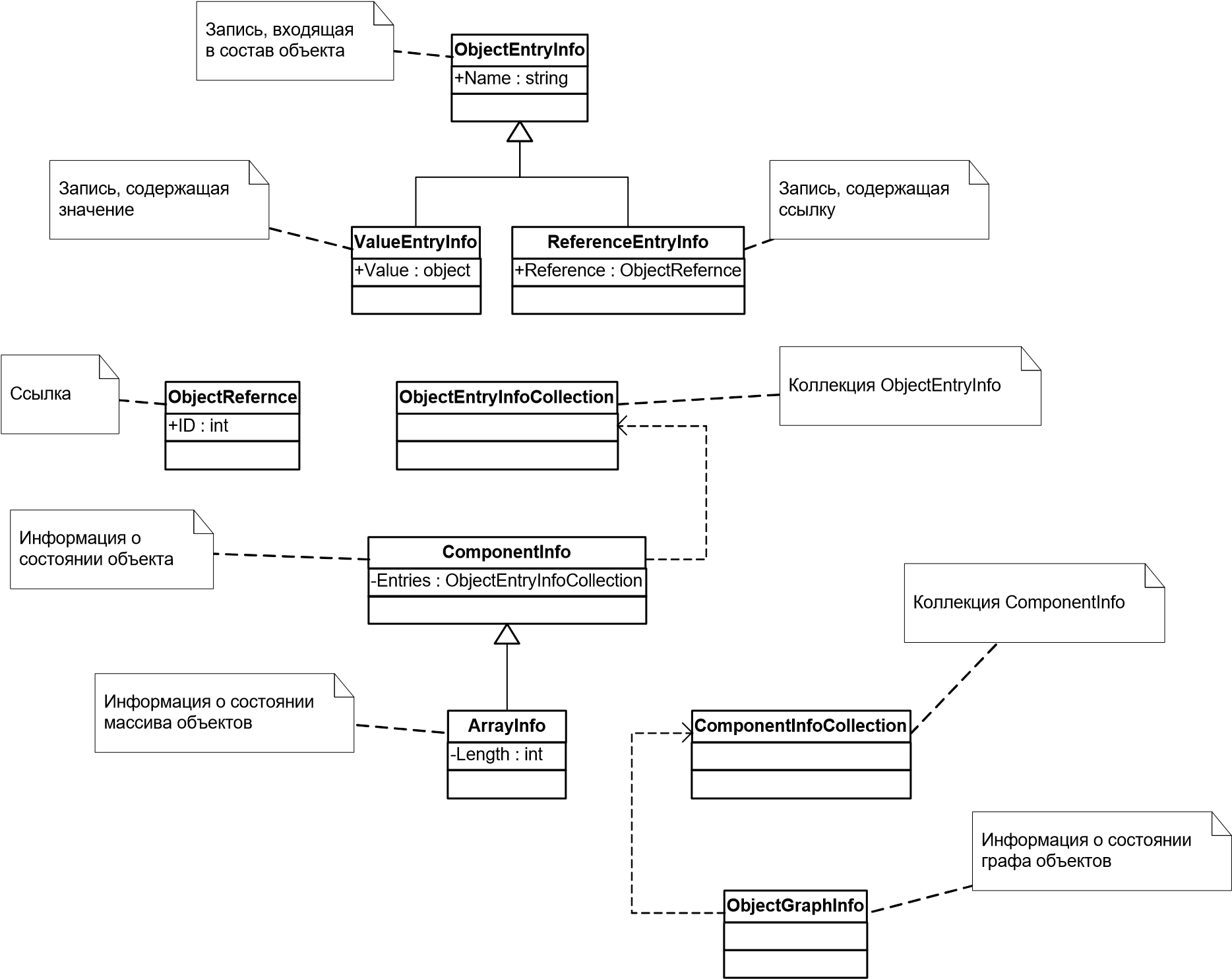


Рис. 14. UML-диаграмма классов, используемых для представления состояния графа объектов.

Далее осуществляется генерация узлов CodeDOM-дерева на основании построенной модели. Если на предыдущем этапе не было применено заданное программистом преобразование данных, то генерируется код, исполнение которого должно приводить к созданию в памяти копии исходного графа объектов. Т. е. выполняется сериализация “в код”. Задача построения CodeDOMдерева возлагается на методы класса ObjectGraphInfoTranslator. Эта задача также решается в несколько этапов.

В начале собирается информация о том, какие поля, свойства объектов какими данными следует инициализировать, какими объектами следует заполнять коллекции. Для каждого объекта находится инициализированный по умолчанию объект того же типа, что и данный. Производится создание экземпляра объекта при помощи конструктора по умолчанию, либо если ссылка на объект была получена как значение поля (свойства) другого объекта, то происходит обращение к полю (свойству) объекта того же типа, что и объектконтейнер, инициализированного по умолчанию, и извлечение искомого объекта из него. Следует учесть, что для объектов некоторых типов способ создания должен задать дополнительно программист-пользователь библиотеки.

Далее производится сравнение значений свойств и полей инициализированного по умолчанию и целевого объектов, при разнице значений данное поле (свойство) подлежит инициализации соответствующим значением, либо ссылкой на объект, также подлежащий созданию и инициализации в коде. Собираемая информация позволяет получить сведения о том, какие объекты потребуются для создания и инициализации данного объекта. Это должно позволить получить код, выполняющий создание и инициализацию объектов в корректном порядке в соответствии с исходной моделью данных и заданными преобразованиями. Также возможна дополнительная настройка порядка инициализации полей (свойств) для объектов определенных типов.

Далее вся собранная информация о порядке инициализации объектов транслируется в узлы CodeDOM-дерева. Для получения имен переменных используется класс, реализующий интерфейс INameService , экземпляр которого хранится в «хосте» кодогенерации, передаваемом ObjectGraphInfoTranslator’у. Объекты, для которых следует создать поля в генерируемой в данный момент декларации класса, регистрируются в текущем контексте типа (TypeContext), получаемом у «хоста» кодогенерации (при условии, что в данный момент выполняется генерация объявления класса). Для них экземпляр BuildTypeDirective, выполняющий генерацию объявления класса, создаст требуемые поля.

### Генерация кода программных моделей элементов технических систем.

Задача генерации кода программной модели элемента технической системы сводится к построению CodeDOM-дерева, содержащего декларацию класса модели элемента, для свойств класса отражающих параметры модели в синтаксическое дерево должны быть добавлены соответствующие узлы. Также должно быть создано поддерево для функции инициализации компонента InitializeComponent, данная функция отвечает за инициализацию внутренних элементов компонента в соответствии с эквивалентной схемой модели, задание начальных значений параметров. Общая схема создаваемого синтаксического дерева приведена на рис. 15.

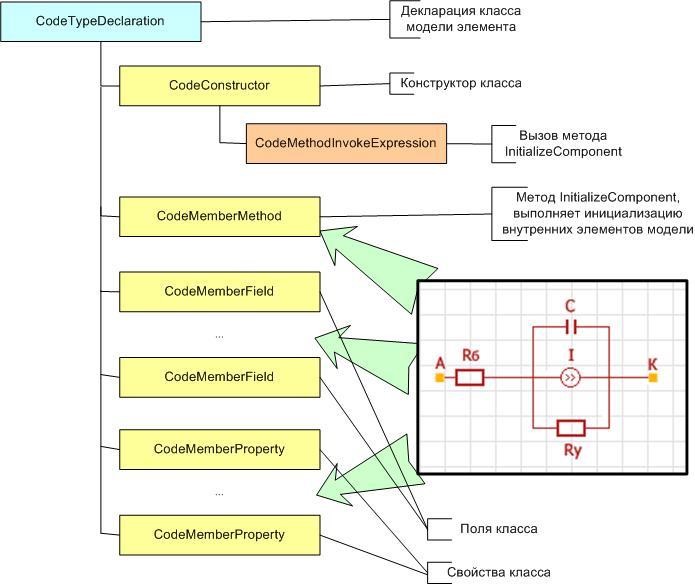


Рис. 15. Схема синтаксического дерева декларации класса компонента.

Создание эквивалентной схемы элемента технической системы происходит в визуальном редакторе путем перетаскивания изображений условных обозначений двух- и многополюсников из панели компонентов на поле редактирования. Между элементами схемы устанавливаются связи. Также при помощи визуального редактора определяется набор свойств программной модели элемента.

За процесс редактирования схемы отвечают классы-дизайнеры. Информация, необходимая для выполнения кодогенерации, запрашивается у классов-дизайнеров. Выполняется сбор всех компонентов, содержащихся на странице редактирования, при этом дизайнеры компонентов могут указать способы инициализации редактируемых ими объектов, дать иные указания кодогенератору. Далее выполняется процедура построения синтаксического CodeDOM-дерева, которое имеет структуру, приведенную на рис. 15. За

создание элементов дерева, соответствующих коду метода

InitializeComponent отвечает класс CreateInitialization из пространства имён Special.CodeGen, его работа была описана в предыдущем разделе.

На рис. 16 приведена эквивалентная схема элемента технической системы, в Приложении 2 приведен полученный кодогенератором на её основе код.

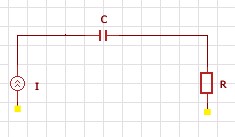


Рис. 16. Эквивалентная схема.

Для настройки кодогенерации и компиляции сборки, содержащей код модели, используется элемент управления «Сборщик модели», размещаемый на странице редактирования (Рис. 17).

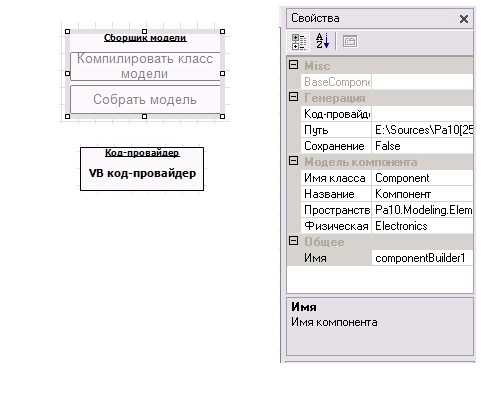


Рис. 17. Настройка свойств Сборщика модели.

При выборе Сборщика модели в окне Свойства доступны для редактирования следующие значения:

* Код-провайдер – позволяет выбрать экземпляр CodeDOMпровайдера, который будет использоваться для получения исходного кода и компиляции. Для создания экземпляра CodeDOM-провайдера следует разместить на странице

редактирования элемент управления «Код-провайдер» желаемого языка (Рис. 18);

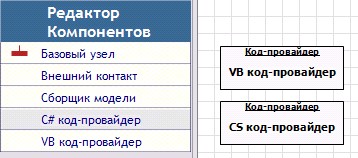


Рис. 18. Размещение на странице редактирования элементов управления «Код-провайдер» для языков VB .NET и C#.

* Путь – выбор пути для сохранения результата компиляции компонента;
* Сохранение – выполнять или нет сохранение компилированной сборки на диск;
* Имя класса – позволяет задать наименование класса компонента;
* Название – наименование модели, которое должно отображаться в панели компонентов;
* Пространство имен – задает пространство имен для класса компонента;
* Физическая подсистема – выбор, к какой физической подсистеме (электрическая, механическая, тепловая и т.д.) будет относиться разрабатываемый компонент.

Элемент управления Сборщик модели содержит две кнопки:

* Собрать модель – предназначена для генерации исходного кода модели. Текст декларации класса модели на выбранном языке отображается на странице редактирования исходного кода;
* Компилировать класс модели – выполняется компиляция исходного кода модели в бинарную сборку.

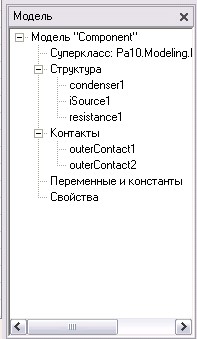


Рис. 19. Окно Модель.

В окне Модель отображается структура модели (Рис. 19).

# Технологическая часть.

## Разработка редактора условных графических обозначений.

При создании эквивалентной схемы технического устройства в схемном редакторе, каждый компонент, включаемый в состав схемы, представлен на странице редактирования в виде графического элемента, содержащего, как правило, условное графическое изображение (УГО) соответствующего элемента. Например, резистор или диод будут отображаться в виде их общепринятых графических обозначений. Графические изображения элементов присутствуют также на панели компонентов, откуда при помощи операции drag-n-drop пользователь может перетащить требуемые компоненты на страницу редактирования (Рис. 20).

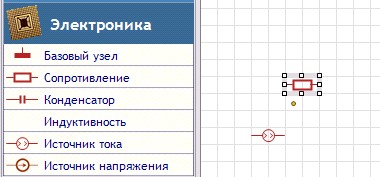


Рис. 20. Примеры условных графических обозначений.

Графический элемент (Glyph, от англ. символ – глиф), размещаемый на странице редактирования, представляет собой аналог элемента управления и может содержать не только статическое изображение, но и вложенные элементы управления (кнопки, выпадающие списки и др.), с которыми пользователь может взаимодействовать и при помощи которых может осуществляться настройка параметров модели (Рис. 21).

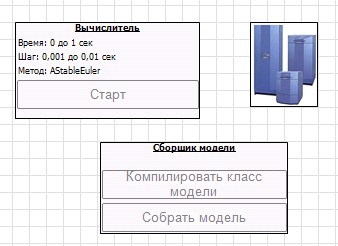


Рис. 21. Примеры глифов.

Графические элементы (глифы) принимают сообщения от клавиатуры и мыши. Для вложенных глифов выполняется рассылка сообщений. Базовый для всех глифов класс Glyph не является наследником класса стандартного элемента управления System.WindowsForms.Control. Глифы не используют системные ресурсы, такие как дескрипторы окон (HWND) и не имеют

собственной оконной функции обработки сообщений, существующей у каждого «настоящего» элемента управления, окна. Рассылка сообщений для множества глифов осуществляется посредством их приема от одного «настоящего» элемента управления, которому они все принадлежат. Благодаря такому подходу достигается значительная экономия ресурсов и повышение быстродействия по сравнению с использованием стандартных элементов управления. Кроме того, глифы могут реализовывать функциональность, недоступную для стандартных элементов управления. Например, достаточно просто могут быть созданы глифы непрямоугольной формы, обеспечен поворот на произвольный угол и др.

Редактор условных графических обозначений (УГО) предназначен для создания графических элементов (глифов), используемых в качестве условных обозначений для разрабатываемых компонентов. Редактор позволяет создавать глифы, содержащие элементы управления (Рис. 22).

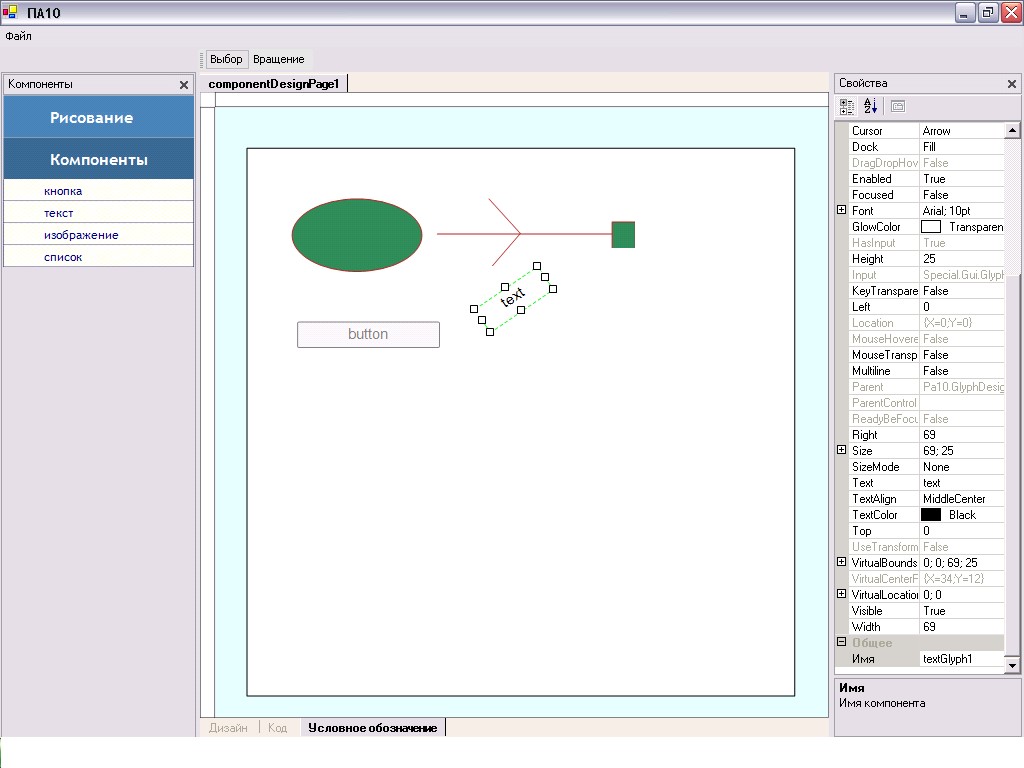


Рис. 22. Окно Редактора условных графических обозначений.

Процесс редактирования заключается в расположении на странице редактора графических фигур, таких как линии, прямоугольники, эллипсы, текст, растровые изображения, а также элементов управления – кнопок, списков и т.д. Окно редактора содержит панель, предназначенную для отображения и изменения свойств имеющихся на странице элементов.

Выбор графического элемента осуществляется при помощи щелчка мыши по его изображению на странице редактирования. При помощи растягиваемой рамки возможен выбор нескольких фигур сразу. Свойства выбранных элементов отображаются в панели свойств. Вокруг выбранных фигур отображаются рамки, содержащие в углах захваты, перемещение захватов приводит к изменению размеров выбранной фигуры. Все элементы могут быть перемещены в пределах страницы редактирования и повернуты на заданный угол вокруг центра вращения. При этом центр вращения может быть перемещен в любую точку (Рис. 23).

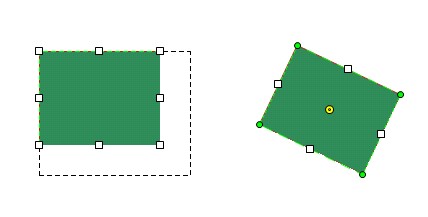


Рис. 23. Изменение размеров и поворот фигуры.

Основой при разработке Редактора послужили созданные нами библиотеки классов Special.Gui и Special.Design, предоставляющие каркас для приложений визуального редактирования. Для представления абстракции приложения редактирования используется класс GlyphDesignApp производный от класса DesignApplication. Процессом редактирования управляют классыдизайнеры. Для каждого графического элемента, с которым работает пользователь редактора, создаются экземпляры классов-дизайнеров: VisualDesigner, служащий для поддержки процессов визуального редактирования, и ModelDesigner, управляющий редактированием свойств элемента при помощи панели свойств. Класс приложения GlyphDesignApp отвечает за создание экземпляров классов-дизайнеров для размещаемых на странице редактирования элементов. Диаграмма основных классов редактора приведена на рис. 24.

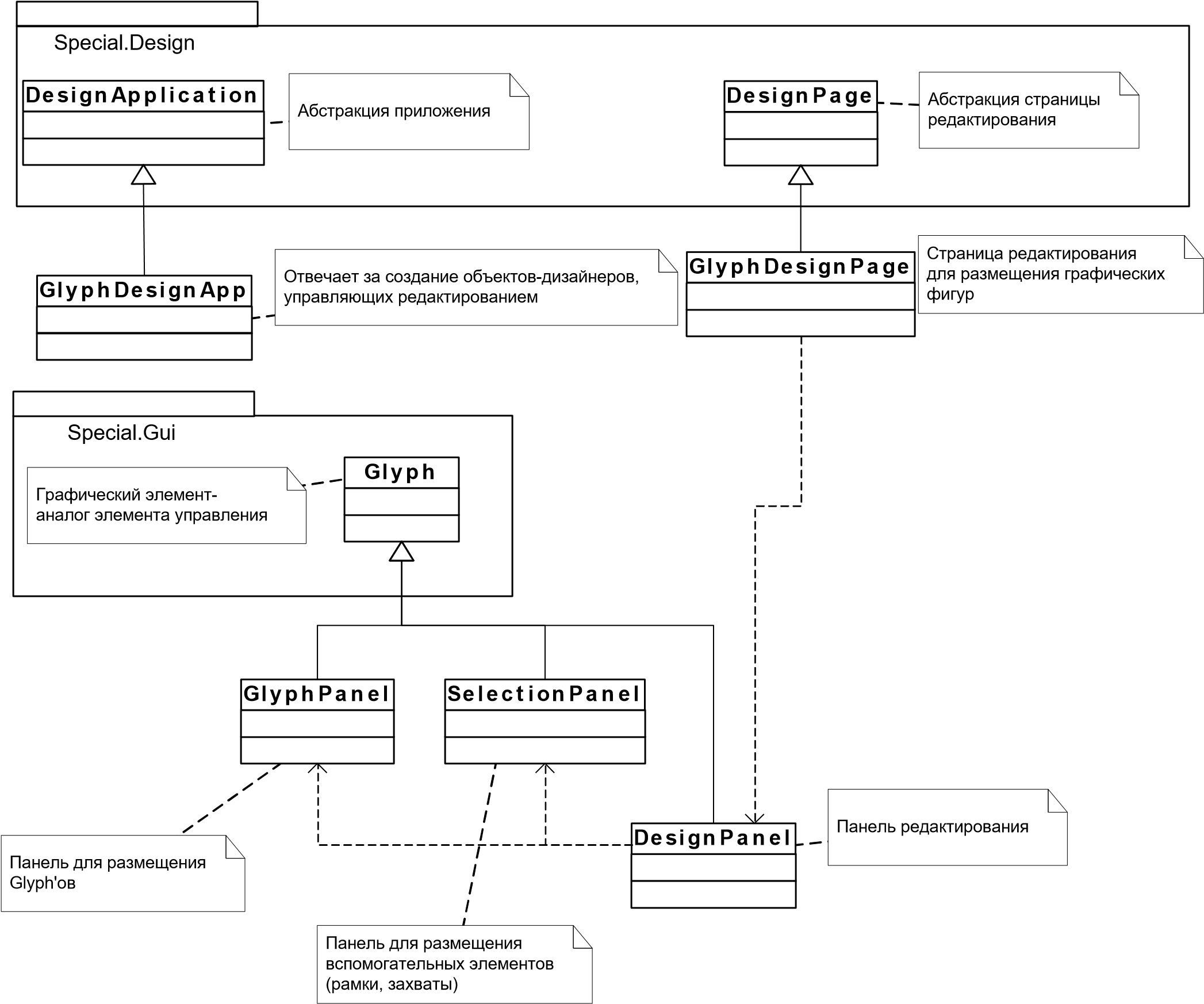


Рис. 24. UML диаграмма основных классов Редактора.

Для представления атрибутов, общих для всех редактируемых геометрических фигур, был введен абстрактный класс ShapeComponent, являющийся базовым для классов прямоугольника, эллипса и т.д.:

public abstract class ShapeComponent

{

/// Возвращает объект геометрической фигуры

/// Интерфейс IGraphicObject введен для обозначения /// геометрических фигур, которые могут быть отрисованы /// на некоторой поверхности.

public abstract IGraphicObject Shape

{ get;

|  |
| --- |
| }    /// Ограничивающий прямоугольник  public abstract System.Drawing.Rectangle BoundingBox  {  get;  }    /// Карандаш для отрисовки контура фигуры public IPen ShapePen  {  get { return this.shapePen; } set { this.shapePen = value; }  }    /// Кисть для заполнения фигуры public IBrush ShapeBrush  {  get { return this.shapeBrush; } set { this.shapeBrush = value; }  }  } |

Класс ShapeGlyph предназначен для представления геометрической фигуры на странице редактирования в виде визуального элемента (глифа), с которым могут выполняться различные манипуляции (выделение, перемещение, поворот). Классы, производные от класса ShapeGlyph – RectGlyph, LineGlyph и др. – представляют на странице редактирования соответствующие геометрические фигуры. При создании экземпляра класса одной из геометрических фигур на страницу редактора попадает соответствующий глиф, который будет осуществлять отрисовку фигуры, и с которым может взаимодействовать пользователь. На рис. 25 изображена диаграмма классов графических фигур, с которыми работает редактор, приведена иерархия классов-дизайнеров фигур.

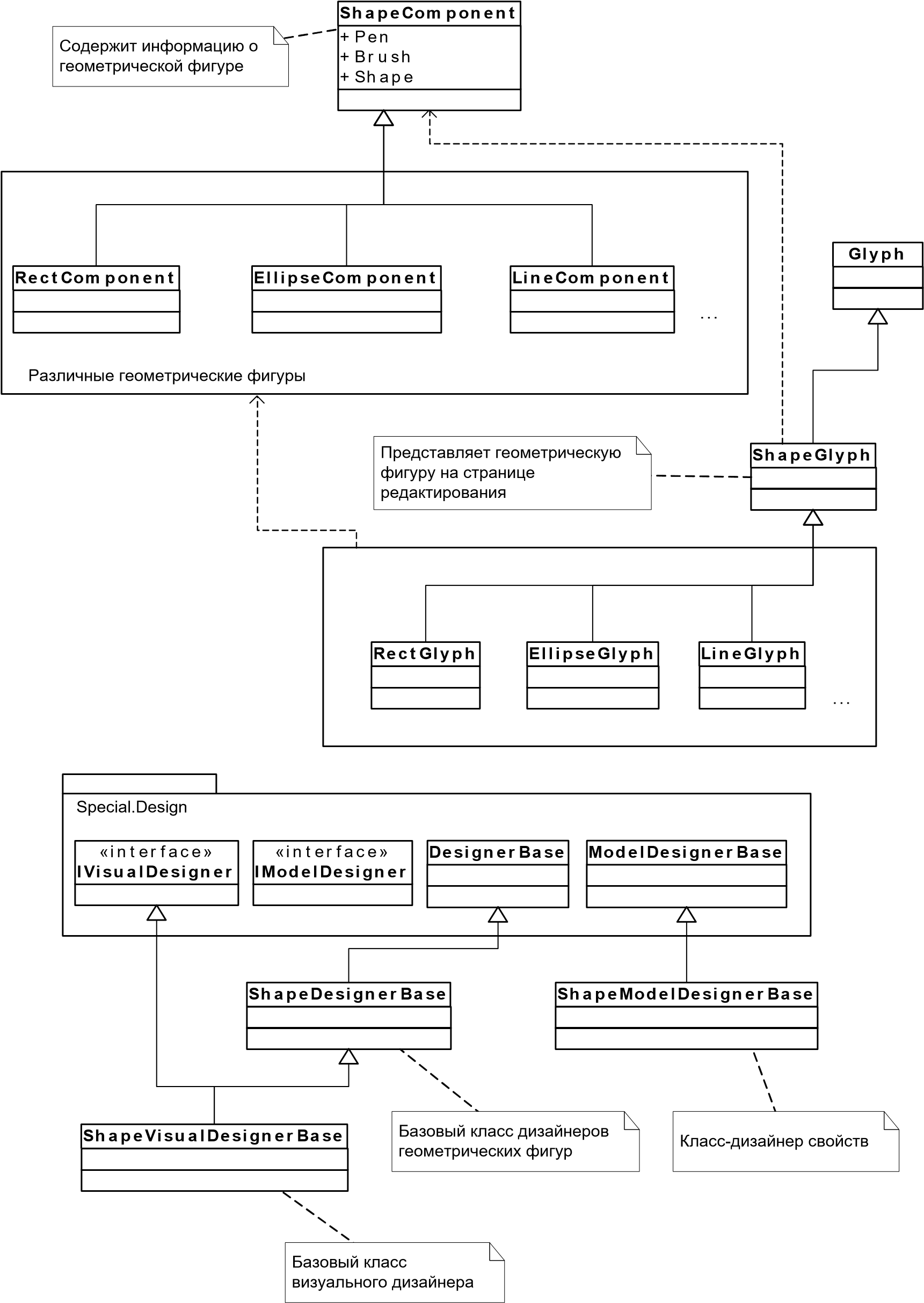


Рис. 25. Классы геометрических фигур.

Пример:

|  |
| --- |
| /// Базовый класс для Glyph’ов, представляющих /// на странице редактирования геометрические фигуры public class ShapeGlyph: Glyph  {  /// В конструктор должен быть передан экземпляр /// класса геометрической фигуры  public ShapeGlyph(ShapeComponent shape)  {  }  public ShapeComponent Shape  {  get  }  }    /// Глиф, представляющий эллипс public class EllipseGlyph: ShapeGlyph  {    /// Конструктор  public EllipseGlyph(EllipseComponent ellipse)  : base(ellipse)  {  }    /// Возвращает отображаемый данным глифов эллипс public new EllipseComponent Shape  {  get { return base.Shape as EllipseComponent; }  }    /// Проверка принадлежности точки эллипсу.  /// Нужна для придания глифу нестандартной /// эллиптической формы.  public override bool ContainsPoint(System.Drawing.Point  point)  {  int X = point.X - Ellipse.Center.X; |
| int Y = Ellipse.Center.Y - point.Y; float xa = X/Ellipse.RadiusOX;    if(Y\*Y<=(Ellipse.RadiusOY\* Ellipse.RadiusOY)\*(1-xa\*xa))  return true;    return false;  }    /// Отрисовка эллипса с использованием /// выбранной кисти и карандаша.  protected override void OnPaint(RenderingPaintEventArgs e)  {  base.OnPaint (e);    e.Graphics.DrawEllipse(  this.Shape.ShapePen, this.Shape.ShapeBrush, this.Shape.Ellipse.Center, this.Shape.Ellipse.RadiusOX, this.Shape.Ellipse.RadiusOY);  }    } |

# Приложение 1. Иерархия классов пространства имен System.CodeDOM.

System.CodeDom.CodeObject

System.CodeDom.CodeComment

System.CodeDom.CodeCompileUnit

System.CodeDom.CodeExpression

System.CodeDom.CodeArgumentReferenceExpression

System.CodeDom.CodeArrayCreateExpression

System.CodeDom.CodeArrayIndexerExpression

System.CodeDom.CodeBaseReferenceExpression

System.CodeDom.CodeBinaryOperatorExpression

System.CodeDom.CodeCastExpression

System.CodeDom.CodeDelegateCreateExpression

System.CodeDom.CodeDelegateInvokeExpression

System.CodeDom.CodeDirectionExpression

System.CodeDom.CodeEventReferenceExpression

System.CodeDom.CodeFieldReferenceExpression

System.CodeDom.CodeIndexerExpression

System.CodeDom.CodeMethodInvokeExpression

System.CodeDom.CodeMethodReferenceExpression

System.CodeDom.CodeObjectCreateExpression

System.CodeDom.CodeParameterDeclarationExpression

System.CodeDom.CodePrimitiveExpression

System.CodeDom.CodePropertyReferenceExpression

System.CodeDom.CodePropertySetValueReferenceExpression

System.CodeDom.CodeSnippetExpression

System.CodeDom.CodeThisReferenceExpression

System.CodeDom.CodeTypeOfExpression

System.CodeDom.CodeTypeReferenceExpression

System.CodeDom.CodeVariableReferenceExpression

System.CodeDom.CodeNamespace

System.CodeDom.CodeNamespaceImport

System.CodeDom.CodeStatement

System.CodeDom.CodeAssignStatement

System.CodeDom.CodeAttachEventStatement

System.CodeDom.CodeCommentStatement

System.CodeDom.CodeConditionStatement

System.CodeDom.CodeExpressionStatement

System.CodeDom.CodeGotoStatement

System.CodeDom.CodeIterationStatement

System.CodeDom.CodeLabeledStatement

System.CodeDom.CodeMethodReturnStatement

System.CodeDom.CodeRemoveEventStatement

System.CodeDom.CodeSnippetStatement

System.CodeDom.CodeThrowExceptionStatement

System.CodeDom.CodeTryCatchFinallyStatement

System.CodeDom.CodeVariableDeclarationStatement

System.CodeDom.CodeTypeMember

System.CodeDom.CodeMemberEvent System.CodeDom.CodeMemberField

System.CodeDom.CodeMemberMethod

System.CodeDom.CodeMemberProperty

System.CodeDom.CodeSnippetTypeMember

System.CodeDom.CodeTypeDeclaration

System.CodeDom.CodeTypeReference

# Приложение 2. Пример результатов генерации кода программной модели компонента

|  |
| --- |
| namespace Pa10.Modeling.Elements { using System; using Pa10.Mathematics; using Pa10.Modeling; using Pa10.Modeling.Elements;    public class RC\_Component : Pa10.Modeling.Elements.  SimpleEqComponent {    private Pa10.Modeling.Elements.Node node1;    private Pa10.Modeling.Elements.Node node2;    private Pa10.Modeling.Elements.Condenser condenser1;    private Pa10.Modeling.Elements.Resistance resistance1;    private Pa10.Modeling.Elements.ISource iSource1;    private Pa10.Modeling.Elements.Node node3;    private Pa10.Modeling.Elements.Node node4;    private Pa10.Mathematics.ComponentCollection  componentCollection1;    public RC\_Component() { this.Initialize();  }  public System.Double R { get {  return this.resistance1.R;  } set { |

|  |
| --- |
| this.resistance1.R = value;  }  }  public System.Double I { get {  return this.iSource1.I;  } set {  this.iSource1.I = value;  }  }  public System.Double C { get {  return this.condenser1.C;  } set {  this.condenser1.C = value;  }  }  private void Initialize() {    this.node1 = new Pa10.Modeling.Elements.Node(); this.node2 = new Pa10.Modeling.Elements.Node(); this.condenser1= new Pa10.Modeling.Elements.Condenser(); this.resistance1 = new Pa10.Modeling.Elements.  Resistance();  this.iSource1 = new Pa10.Modeling.Elements.ISource(); this.node3 = new Pa10.Modeling.Elements.Node(); this.node4 = new Pa10.Modeling.Elements.Node(); this.componentCollection1=new  Pa10.Mathematics.ComponentCollection();  //  // node1  //    this.node1.ConnectedContacts.Add(this.resistance1.OutContact) ; |

|  |
| --- |
| //  // node2  //    this.node2.ConnectedContacts.Add(this.iSource1.InContact);  //  // iSource1  //  this.iSource1.I = 1;  //  // node3  //    this.node3.ConnectedContacts.Add(this.iSource1.OutContact);    this.node3.ConnectedContacts.Add(this.condenser1.InContact);  //  // node4  //    this.node4.ConnectedContacts.Add(this.condenser1.OutContact);    this.node4.ConnectedContacts.Add(this.resistance1.InContact);  //  // componentCollection1  //  this.componentCollection1.Add(this.node4); this.componentCollection1.Add(this.node3); this.componentCollection1.Add(this.iSource1); this.componentCollection1.Add(this.resistance1); this.componentCollection1.Add(this.condenser1); this.componentCollection1.Add(this.node2); this.componentCollection1.Add(this.node1); this.node2.ConnectedContacts.Add(this.InContact); this.node1.ConnectedContacts.Add(this.OutContact);  }  }  } |