ИНСТРУМЕНТАЛЬНЫЕ СРЕДСТВА РАЗРАБОТКИ И СОПРОВОЖДЕНИЯ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ НА ОСНОВЕ ГЕНЕРАЦИИ КОДА

А.Е. Александров,
доктор технических наук, профессор кафедры персональных ЭВМ и сетей Московского государственного университета приборостроения и информатики (МГУПИ)

В.П. Шильманов,
аспирант кафедры персональных ЭВМ и сетей Московского государственного университета приборостроения и информатики (МГУПИ)

Адрес: г. Москва, ул. Стромынка, д. 20
E-mail: femsystem@yandex.ru

В статье рассмотрены инструментальные средства разработки и сопровождения программного обеспечения на основе автоматической генерации кода, реализованного на языке Python с использованием комбинаторной библиотеки PyParsing. Наполнение компонентной библиотеки и формирование конкретной реализации программного приложения происходит в результате генерации программного кода, формирующего исходный код на алгоритмическом языке C++. Для разделения сгенерированного и рукописного кода применен механизм разделения класса на три подкласса связанных отношением наследования.

Ключевые слова: языко-ориентированное программирование, программный компонент, библиотека программных компонентов, генерация кода.

1. Введение

В индустрии программирования наметилась устойчивая тенденция к переходу от объектно-ориентированной технологии программирования к новой парадигме, основанной на теории понятий и языко-ориентированных методах программирования [1]. Причиной этому являются неоправданные затраты для объектно-ориентированной технологии на перевод высокоуровневых понятий и методик предметной области в низкоуровневые конструкции, отвечающие требованиям используемых языков программирования. Этот период разработки программного обеспечения (ПО) является весьма длительным, не особенно творческим, и к тому же практически не нужным в дальнейшем при сопровождении и развитии программного кода – все этапы разработки ПО приходится повторять заново. Альтернативой этому подходу становится новый, использующий предметно-ориентированные языки, которые выполняют роль своеобразного ин-

терфейса, связывая высокоуровневые понятия с непосредственной программной реализацией. Разработанные предметно-ориентированные языки дают возможность сформировать программный код, как непосредственное описание реализованной модели, используя его в дальнейшем для накопления опыта работы и необходимой модификации и развития.

На *рис. 1* изображены два варианты разработки и развития программного обеспечения: первый из них основан на классическом (объектно-ориентированном) подходе, второй – предполагает использование языко-ориентированного программирования, новой парадигмы. Для сравнения проанализированы стадии разработки программного обеспечения, включающие проектирование, программирование, а также его развитие и сопровождение.

Далее, для варианта а), основанного на классическом объектно-ориентированном подходе, приведено описание действий при реализации каждой из стадий.

Проектирование. При проектировании ПО создается объектно-ориентированная модель, на основе низкоуровневых объектно-ориентированных абстракций – классов и методов. Процесс отображения предметных понятий на объектно-ориентированные абстракции не всегда производится однозначно, при переводе высокоуровневых требований в низкоуровневые программные конструкции очень часто теряется важная информация о предметной области.

Программирование. Программирование предполагает отображение сформированной объектно-ориентированной модели на выбранный язык программирования, как правило на C++. Существенный объем работ по переводу требований к системе в исходный код программистам приходится выполнять самостоятельно. Рассматривая программный код на алгоритмическом языке, трудно восстановить реальную модель, отвечающую требованиям предметной области.

Особые трудности возникают при формировании программистами библиотеки классов, поскольку им приходится с помощью алгоритмического языка формировать предметно-ориентированные высокоуровневые абстракции, вручную проводить предметно-ориентированную оптимизацию и решать многие другие задачи, которые требуют знаний предметной области. Библиотеки классов позволяют разрабатывать не одно программное приложение, а семейство программных систем, и выполняют функцию расширения языка программирования. Разработка объектно-ориентированной библиотеки классов и формирование на основе разработанной библиотеки программного приложения производятся ручным способом.

Сопровождение и развитие. Поддержка и сопровождение, как самой библиотеки, так и разработанных приложений становятся весьма сложной задачей, так как требуют привлечения квалифицированных программистов, к тому же обладающих опытом и знаниями используемой предметной области. Каждое отдельное приложение может составлять до 100 тысяч

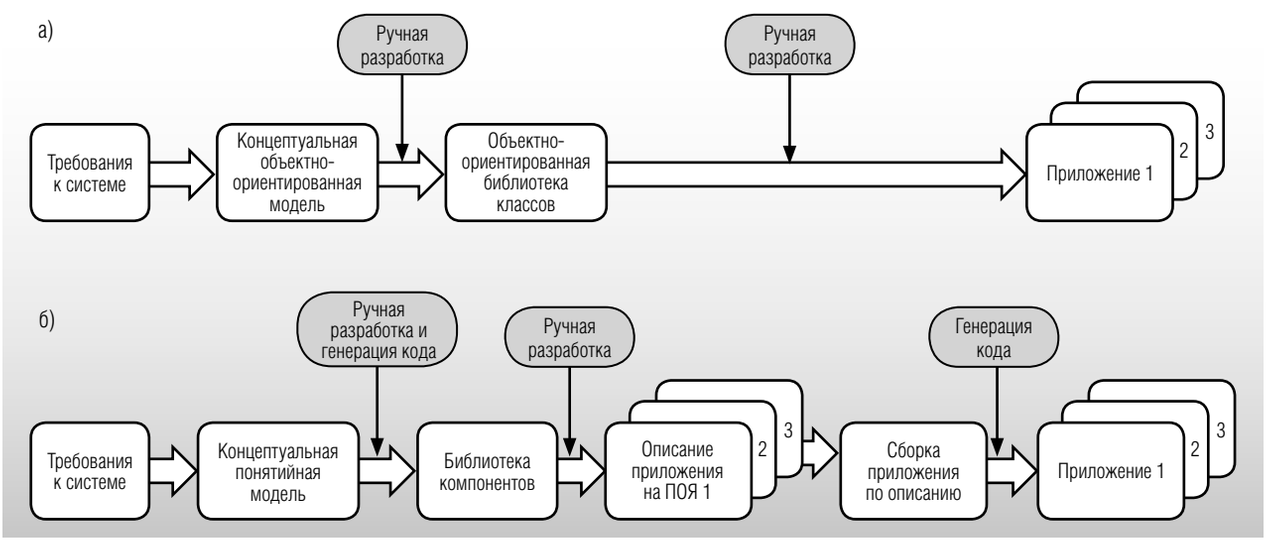

Рис. 1. Варианты разработки программного обеспечения:
а) классический объектно-ориентированный подход; б) языко-ориентированный подход

строк программного кода на алгоритмическом языке.

Развитие систем, требует повторения вышеназванных стадий разработки: проектирование и программирование, практически заново, поскольку реализацию ПО необходимо выполнять на основе низкоуровневых программных конструкций. Отсутствие средств стандартизации делает этот процесс весьма трудоемким. Единственный процесс, который может быть автоматизирован в данном случае, это компиляция исходного кода, написанного на алгоритмическом языке (C++).

Для варианта б), основанного на языко – ориентированном подходе, анализируемые стадии требуют выполнения следующих действий.

Проектирование. При проектировании ПО используется высокоуровневая модель предметной области, основанная на понятиях, которые более приближены к предметной области, чем классы и методы. Формирование высокоуровневой модели позволяет представить реализацию программной системы в виде предметно – ориентированных нотаций, что существенно облегчает процесс ее сопровождения и развития. В этом случае возникают реальные предпосылки для автоматизации процесса программирования на основе использования предметно – ориентированных языков и генераторов кода.

Программирование. Программирование предполагает разработку предметно-ориентированного языка (ПОЯ), который основан на понятиях предметной области. В этом случае, код программного приложения становится легко читаемым экспертом без участия программиста и легко сопровождаемым. Объем приложения, реализованного на ПОЯ становится на два три порядка меньше, чем реализованного на алгоритмическом языке. Практически вся необходимая информация с точки зрения эксперта может быть сохранена в коде программного приложения. Сам код приложения, в этом случае, становится документом, который может быть классифицирован по различным признакам и храниться в архиве, сохраняя всю необходимую информацию о разработанных и используемых моделях.

Основой для разработки ПОЯ являются компоненты, объединенные в библиотеку. Понятие программных компонентов предполагает совместимость их при включении в программное приложение и многократное их использование. Наличие компонентной библиотеки позволяет формировать из программных компонентов, как из блоков, различные программные приложения. Разработка программного компонента в этом случае производится с расчетом на многократное его использование. Формирование библиотеки компонентов может быть автоматизировано на основе использования генераторов кода.

Сопровождение и развитие. Высокоуровневая реализация системы в виде предметно-ориентированных нотаций, как уже было замечено выше, существенно облегчают как процесс сопровождения, так и расширение функциональности системы.

2. Постановка задачи

Сопоставление рассмотренных выше стадий разработки ПО показывает, что второй вариант обладает существенными преимуществами при моделировании сложных систем, для которых неоднозначны сами модельные представления по описанию процессов в них. Поскольку в этом случае, процесс разработки ПО превращается в непрерывный процесс модификации и развития кода, связанный с необходимостью сравнения различных модельных представлений и выбору оптимальных, а ПО становится в этом случае инструментом, конфигурация которого меняется в зависимости от требований эксперта. К такому классу систем относится программная система «Прогноз», предназначенная для анализа безопасности объектов ядерной энергетики [2], применительно к которой и были разработаны инструментальные средства, поддерживающие процессы формирования компонентной библиотеки и программного приложения.

Для обеспечения совместимости по данным компонентов при формировании из них программного приложения, авторами была разработана архитектура компонентной библиотеки, исключающая их непосредственное взаимодействие друг с другом – связь компонентов по данным осуществляется с помощью объектов – посредников. В этом случае, объект, реализующий отдельный алгоритм, взаимодействует только с объектом хранилища данных, получает входные данные из хранилища и возвращает их обратно.

Однако при формировании компонентной библиотеки в рамках разработанной архитектуры появляется дополнительный код в функциональном компоненте (алгоритме). В этом случае, алгоритм включает в свой состав не только ядро алгоритма, реализующего непосредственно его функцию, но и

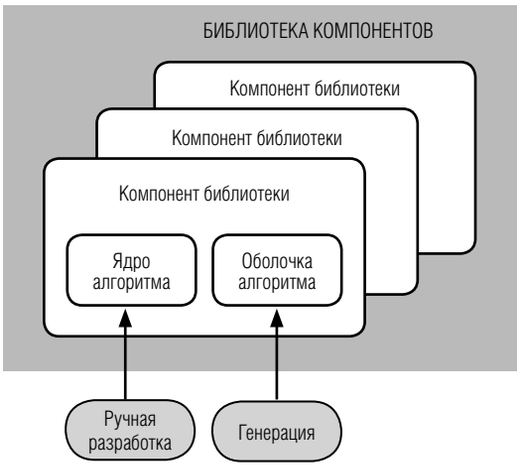

Рис. 2. Структура компонента

оболочку алгоритма, выполняющую связь алгоритма с хранилищем и диском *рис. 2.*

Объем оболочки (количество строк кода) зависит от объема данных алгоритма. В среднем одно данное алгоритма требует написания 50-ти строк программного кода в дополнительных служебных компонентах, реализующих взаимодействие алгоритма с хранилищем.

Автоматизация генерации кода при формировании компонентной библиотеки в такой постановке является нестандартной задачей. Не существует готовых инструментальных средств для автоматизации задач такого класса.

Генерация кода, за исключением самых простых случаев, обычно строится на основе семантической модели.

Важной особенностью применения семантической модели является разделение семантики и синтаксического анализа. Можно проверить семантику наполнив непосредственно модель и протестировав её, или можно проверить синтаксический анализатор предметно – ориентированного языка, определив, наполняет ли он семантическую модель корректным образом.

Семантическая модель позволяет проверить корректность данных описанных на предметно –ориентированном языке. Например, нет ли алгоритмов, у которых не задан вход или выход, проверить корректность имен данных.

В рассматриваемой варианте семантическая модель представляется в виде структуры данных, хранящей таблицу, содержащую описания входных и выходных данных алгоритма вычислений.

Ниже рассматриваются два основных подхода, используемых при генерации кода: генерация с помощью преобразователя и шаблонная генерация.

3. Генерация с помощью преобразователя

Генерация с помощью преобразователя основана на написании программы, которая получает на вход семантическую модель, а на выходе дает исходный текст для целевой языковой среды. Различают преобразователи управляемые вводом и управляемые выводом. Управляемые выводом преобразования начинаются с требуемого вывода и погружаются в семантическую модель для сбора необходимых данных. Преобразование, управляемое входом, проходит по всей семантической модели и генерирует вывод.

На практике часто используются комбинированные преобразователи. В таком варианте преобразователь управляемый выводом используется для разделения выходного документа на логические разделы, а внутренняя часть каждого раздела генерируется на основе конкретных входных данных семантической модели.

Преимуществами рассмотренного метода являются:

1. Простота написания генератора при условии, что генерируется большая часть выходного текста.

2. Возможность реализации многоступенчатого преобразователя, в случаях сложных взаимоотношений между входом и выходом, каждый этап может обрабатывать различные аспекты проблемы.

3. Простое отображение в коде преобразователя обработки сложных случаев при генерации выхода, таких как генерация вывода для элементов коллекции, использующая операторы циклов или условные операторы, когда в зависимости от значения в контексте могут генерироваться различные результаты.

В качестве недостатков выделим следующие:

1. По исходному коду преобразователя сложно понять, как будет выглядеть результат генерации.

2. Структура выходных файлов с исходным кодом жестко прописана внутри кода преобразователя и обычно разбита на малые части для упрощения преобразователя, что усложняет внесение изменений при изменении формата вывода.

3. При необходимости иметь разные виды выходных файлов на основе одной семантической модели, требуются разные преобразователи.

4. Больший объем кода генератора по сравнению с подходом на основе шаблонов.

4. Шаблонная генерация

Шаблонная генерация основана на трех компонентах: обработчик шаблонов, шаблон и контекст. Шаблон представляет собой исходный текст выходного файла, в котором динамические части представлены маркерами-идентификаторами. Последние являются ссылками на контекст, который будет использоваться для заполнения динамических элементов в процессе генерации. Контекст выступает в роли источника динамических данных. Он может быть простой структурой данных или сложной семантической моделью. Разные инструменты используют контексты разных видов. Обработчик шаблонов является инструментом, который связывает шаблон и контекст для генерации выхода.

Управляющая программа будет выполнять шаблонную программу с определенным контекстом и шаблоном для получения выходного файла. Также может использоваться вариант с одним контекстом и разными шаблонами или одним шаблоном и разными контекстами, при этом будут генерироваться разные выходные файлы.

Отметим следующие преимущества, рассматриваемого метода генерации:

1. Взглянув на файл шаблона можно сразу представить, как будет выглядеть сгенерированный результат.

2. Простота написания обработчика шаблонов, при условии преобладания статического содержимого выходных файлов над динамическими частями.

3. Простота и ясность шаблонов, при условии, что динамическое содержимое достаточно простое.

4. Вынесение описания вывода в отдельные файлы шаблоны позволяет иметь один генератор и набор шаблонов для всех возможных случаев выходных файлов.

В качестве недостатков перечислим следующие:

1. Запутанность шаблонов при сложном динамическом содержимом. Чем больше используется циклов, условий и прочих возможностей языка шаблонов, тем труднее представить, как будет выглядеть результат обработки шаблона.

Рис. 3. Схема генерации заготовок исходного кода для функционального компонента (алгоритма)

2. Усложнение и потеря ясности шаблонов при преобладании динамического содержимого над статическим.

3. Неприменимость многоступенчатого подхода в случае сложных взаимоотношений между контекстом и динамическим содержимым шаблонов, только усложнение самого шаблона.

5. Генерация кода при формировании компонентной библиотеки

Для автоматизации процесса формирования кода функционального компонента (алгоритма) в рамках разработанной архитектуры компонентной библиотеки была использована генерация оболочки алгоритма по предварительно сформированному шаблону описания его входных и выходных данных. Схема процесса генерации заготовки кода для алгоритма представлена на *рис. 3.*

Для описания шаблона входных и выходных данных была использована таблица, сформированная в текстовом редакторе (Word). В качестве примера ниже приведена *таблица*, содержащая перечень данных одного из алгоритмов вычислений (*табл. 1*), входящего в расчет одного из объектов ядерной энергетики – корпуса реактора.

Приведенные *в табл. 1* поля имеют следующие пояснения:

✦ имя блока вычислений – имя блока в библиотеке

Таблица 1.

Имя блока вычислений: CriticalTempOfBrittleness
Комментарий: Расчет критической температуры хрупкости

Имя данного	Тип данного	Входное/Выходное	Комментарий к данному
CHEMICAL_COMPOSITION_ CONTENT	double	Input	Значения содержания химических компонентов корпуса
AF_CL	double	Input	Значение коэффициентов Ci в формуле радиационного охрупчивания
CRITICAL_TEMP_OF_ BRITTLENESS	double	Output	Величина критической температуры хрупкости

```
alg
  name: CriticalTempOfBrittleness #Расчет критической температуры хрупкости
  CHEMICAL_COMPOSITION_CONTENT double Input #Значения содержания химических компонентов корпуса
  AF_CI double Input #Значение коэффициентов Ci в формуле радиационного охрупчивания
  CRITICAL_TEMP_OF_BRITTLENESS double Output #Величина критической температуры хрупкости
end
```

Рис. 4. Пример файла расширенного описания поступающего на вход генератора

компонентов и имя файла заготовки;

✦ комментарий к алгоритму вычислений – описание, которое будет добавлено в файл заготовки;

✦ имя данного – имя переменной (объекта данного);

✦ тип данного – тип переменной (объекта данного);

✦ входное/выходное – указание является ли описываемое данное входным или выходным;

✦ комментарий – комментарий, который будет добавлен к строке определения объекта данного.

На вход генератора кода поступает файл специального формата, сформированный на основе *табл.1* и представленный на *рис. 4.*

Для разбора данного файла была использована библиотека PyParsing [3], являющаяся комбинаторной библиотекой для синтаксического анализа и реализованной на языке Python. Ниже, на *рис. 5*, представлена программа описания грамматики для разбора входного файла генератора.

Далее приведено описание реализации самой программы.

Первая строка – импортируем информацию из модуля PyParsing.

Третья строка – определяем комментарий как любой текст от символа «#» до символа конца строки. Выражение Suppress(«#») означает, что сам символ «#» не будет включен в результат.

Четвертая строка – определяем строку описания имени алгоритма как строку, начинающуюся с ключевого слова «name:» за которым через пробел следует имя алгоритма определенное как слово, состоящее из букв или букв и цифр и/или символа подчеркивания. Строка описания имени алгоритма завершается комментарием. Ключевое слово также не будет включено в результат.

Пятая и шестая строка – определяем строку таблицы как запись, состоящую из четырех компонентов разделенных пробелом:

1. Имя данного определяем как слово, состоящее из букв или букв и цифр и/или символа подчеркивания.
2. Тип данного определяем как слово, состоящее из букв или букв и цифр и/или символа подчеркивания и/или символа «*».
3. Признак входного или выходного данного определяем как слово, состоящее только из букв.
4. Комментарий к данному определяем, ссылаясь на определенное ранее правило для комментария.

Седьмая и восьмая строка – определяем блок описания таблицы для одного алгоритма вычислений, как правило algname и правило datainfo, лежащие между двумя ключевыми словами «alg» и «end». Выражение OneOrMore означает, что фраг-

```
from pyparsing import *
#определяем грамматику
comment = Suppress("#") + restOfLine
algname = Suppress("name:")+ Word(alphas, alphanums+'_')+ comment
datainfo = Word(alphas, alphanums+'_') + Word(alphas, alphanums+'_*') \
           + Word(alphas) + comment
alginfo = Suppress("alg") + algname + Group(OneOrMore(Group(datainfo))) \
          + Suppress("end")
file = OneOrMore(Group(alginfo))

# запускаем разбор входного файла
input_file = open('Algorithm_table.txt').read()
result = file.parseString(input_file)
```

Рис. 5. Программа описания грамматики в генераторе

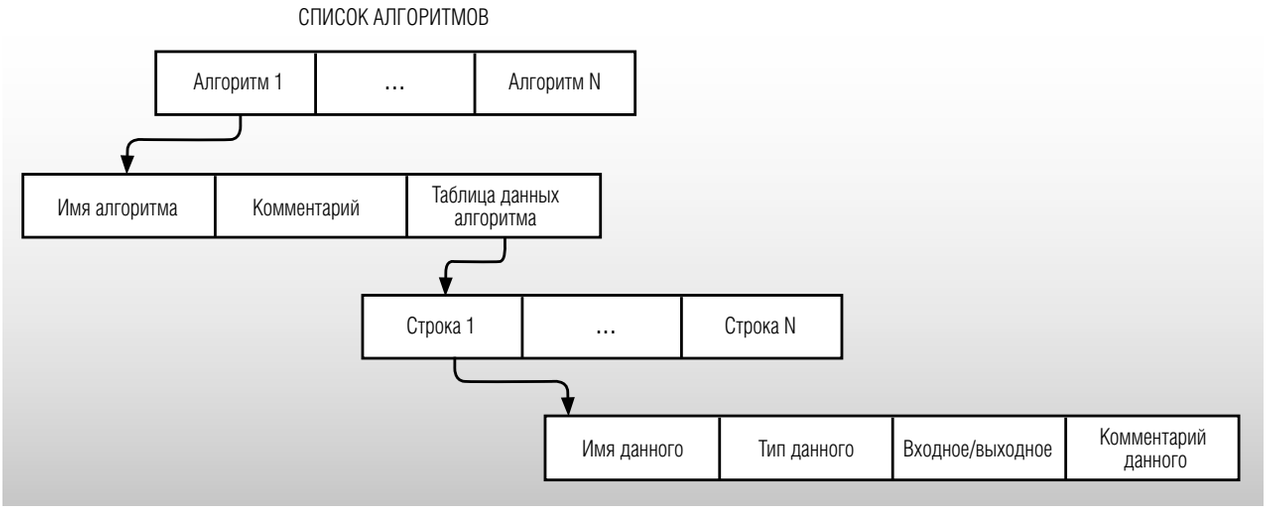

Рис. 6. Структура выходного списка

мент текста, описываемый правилом datainfo, может встречаться один или более раз. Выражение Group() означает группировку результатов полученных внутри её вызова.

Девятая строка – определяем правило, описывающее весь входной файл, поскольку в одном файле могут присутствовать описания для нескольких алгоритмов.

Двенадцатая и тринадцатая строка – производим чтение входного файла и вызов анализатора. Вызов анализатора осуществляется путем вызова метода parseString у объекта с именем верхнего правила грамматики. В качестве параметра передаем ему содержимое входного файла.

Результатом работы анализатора является список следующего вида, представленного на *рис. 6.*

На основе полученного списка строится работа генератора кода. Для каждого выходного файла генератора имеется шаблон, описывающий его структуру. Каждый шаблон является файлом с исходным кодом на C++, содержащим метки-идентификаторы для всех варьирующихся частей. Генератор обрабатывает шаблон и заполняет метки-идентификаторы на основе данных из расширенного описания алгоритмов вычислений для создания заготовки выходного файла.

Сгенерированные заготовки файлов с исходным кодом дополняются рукописным кодом, реализующим функциональную часть алгоритма вычисления, и, после этого, записываются в библиотеку компонентов. Вместе с ними хранится и копия расширенного описания входных и выходных данных.

6. Формирование программного приложения на основе библиотеки компонентов

Сформированная библиотека компонентов позволяет автоматизировать сборку программного приложения из готовых элементов компонентной библиотеки. Для формирования программного приложения необходимо выбрать из библиотеки компонентов требуемые алгоритмы вычислений, интегрировать их в служебные компоненты и написать управляющий код.

Например, для системы, состоящей из одного алгоритма вычислений, необходимо выполнить следующие действия:

1). В компонент хранения добавить определения всех объектов данных, являющихся входными и выходными данными вычислительного алгоритма, а также написать методы доступа к этим данным.

2). В компоненте управления написать код, реализующий загрузку до начала расчета и выгрузку после окончания расчета для всех данных присутствующих в компоненте хранения, путем вызова для каждого данного соответствующего метода из компонента загрузки или сохранения.

Интеграция алгоритмов в служебные компоненты является стандартным процессом, поэтому может быть выполнена с помощью генератора кода, подобного генератору описанному выше. При этом сохраняется простота кода готовой системы, но разработчики избавляются от утомительного написания большого количества повторяющихся блоков кода.

На вход генератора поступает файл, содержащий

перечень имен алгоритмов вычислений, требуемых к включению в систему, а также имя новой системы.

На основе этого перечня генератор строит обращения к библиотеке компонентов для получения заготовок файлов исходного кода служебных компонентов, готовых файлов алгоритмов, а также файлов расширенного описания алгоритмов.

Далее происходит генерация кода для интеграции компонентов и помещение его внутрь файлов с исходным кодом служебных компонентов.

После этого все участвующие файлы с исходным кодом размещаются в структуре каталогов, и генерируется файл проекта для используемой среды разработки.

Последним действием является написание рукописного кода в компоненте управления, реализующего взаимодействие алгоритмов вычислений между собой по заранее определенной методике в соответствии с реализуемой задачей. В дальнейшем предполагается разработать предметно – ориентированной язык для формирования управляющей программы для управления работой программного приложения.

Для исключения внесения исправлений в сгенерированный код было проведено разделение класса, содержащего как рукописный, так и сгенерированный код на три подкласса:

1. Рукописный базовый подкласс, содержащий код, не зависящий от параметров генерации.

2. Сгенерированный подкласс, содержащий код, формируемый автоматически на основании параметров генерации.

3. Рукописный конкретный подкласс, содержащий код, который не может быть сгенерирован, но использует уже сгенерированный код. К этому подклассу может обращаться другой код.

Описанный способ дает возможность создавать один класс, разделенный на отдельные файлы, чтобы хранить сгенерированный код отдельно и переопределять любой аспект сгенерированного кода в подклассе. Рукописный код может легко вызывать любые сгенерированные методы, а сгенерированный – написанные вручную компоненты с помощью механизма абстрактных методов или перехвата методов. Обращение к такому классу всегда ведется через рукописный конкретный класс.

7. Заключение

Авторами использован нетрадиционный подход для разработки программной системы «Прогноз», основанный на теории понятий, и включающий разработку высокоуровневой модели предметной области и компонентной библиотеки, а также формирование программного приложения на основе разработанной компонентной библиотеки. Каждая реализация программного приложения формируется из отдельных программных компонентов с помощью высокоуровневого описания.

Для формирования компонентной библиотеки использован разработанный авторами генератор кода на языке Python с использованием комбинаторной библиотеки PyParsing. Разработанный генератор позволяет генерировать программный код алгоритма по предварительно сформированному шаблону описания его входных и выходных данных. Приведен соответствующий пример применения разработанного генератора в рамках разработанной программной среды для формирования компонентной библиотеки и программного приложения. Для разделения сгенерированного и рукописного кода применен механизм разделения класса на три подкласса связанных отношением наследования.

Представлена последовательность действий при проектировании программного приложения на основе разработанных инструментальных средств. ■

Литература

1. Фаулер М. Предметно – ориентированные языки программирования.: Пер. с англ. – М.: Вильямс, 2011.
2. Александров А.Е. и др. Отчет по верификации программного комплекса «Прогноз» по расчету вероятности повреждаемости перемычек коллектора парогенератора РУ ВВЭР-1000. Рег. номер №01201056086. М.: МГУПИ. 2010, с. 167 с: ил.
3. Документация по библиотеке PyParsing (http://pyparsing).