Министерство образования и науки Российской Федерации

Федеральное государственное бюджетное образовательное учреждение
высшего профессионального образования
«Комсомольский-на-Амуре государственный технический университет»

В. Д. Бердоносов, А. А. Животова

ОБЪЕКТНО-ОРИЕНТИРОВАННОЕ ПРОГРАММИРОВАНИЕ

Допущено Учебно-методическим объединением по образованию
в области прикладной информатики в качестве учебного пособия
для студентов высших учебных заведений, обучающихся по направлению
«Прикладная информатика»

Комсомольск-на-Амуре
2015

УДК 004.652.5(07)
ББК 32.973.2-018.1я7
Б483

Рецензенты:
Кафедра «Автоматика и системотехника»
ФГБОУ ВПО «Тихоокеанский государственный университет»,
зав. кафедрой доктор технических наук, профессор Чье Ен Ун;
В. А. Ханов, кандидат технических наук,
заместитель начальника отдела информационных технологий
ОАО «Амурский судостроительный завод»

Бердоносов, В. Д.

Б483 Объектно-ориентированное программирование : учеб. пособие / В. Д. Бердоносов, А. А. Животова. – Комсомольск-на-Амуре : ФГБОУ ВПО «КнАГТУ», 2015. – 135 с.

ISBN 978-5-7765-1173-8

В учебном пособии изложены основные принципы объектно-ориентированного программирования, применяемые в различных языках программирования. Пособие дает общее представление о предпосылках и движущих силах возникновения объектно-ориентированной парадигмы программирования, подробно рассмотрена эволюция средств реализации базовых принципов объектно-ориентированного подхода.

Приводятся примеры использования объектно-ориентированных возможностей в различных языках, таких как C++, Java и др. В учебном пособии даны задачи для самостоятельного решения, а также рекомендации по их решению и вопросы для закрепления пройденного материала.

Учебное пособие предназначено для студентов технических специальностей, изучающих курс объектно-ориентированного программирования.

УДК 004.652.5(07)
ББК 32.973.2-018.1я7

ISBN 978-5-7765-1173-8

ОГЛАВЛЕНИЕ

ВВЕДЕНИЕ

Целью учебного пособия является развитие системного подхода к освоению знаний, умений, навыков в объектно-ориентированном программировании.

На сегодняшний день практически все широко используемые языки поддерживают объектно-ориентированное программирование (ООП), развивая и совершенствуя прикладные средства реализации объектно-ориентированных возможностей. В связи с большим разнообразием языков программирования и критериев классификации, трудно объективно оценивать целесообразность и эффективность использования того или иного языка для решения тех или иных задач.

Особенно актуальна проблема большого объема неструктурированной информации об объектно-ориентированных языках программирования в образовательном процессе. В силу ограниченности времени при изучении ООП рассматриваются, как правило, возможности какого-либо одного языка, что приводит к игнорированию полезных и эффективных механизмов ООП в других языках. В то же время очевидно, что на сегодняшний день при обучении требуется передавать огромный объём знаний за ограниченное время в режиме непрерывного творчества.

В настоящий момент компетентностный подход является ключевым подходом, используемым в образовательном процессе. Наиболее быстро набирает популярность в мире в рамках компетентностного подхода методов обучения проблемно-ситуативное обучение с использованием кейсов [1]. Одной из разновидностей метода кейсов можно считать ТРИЗ-эволюционный подход к обучению [2].

ТРИЗ-эволюционный подход позволяет не только систематизировать и структурировать знания при помощи ТРИЗ-эволюционных карт, но и значительно повысить эффективность представления и изучения знаний в виде систематизированных блоков. Следует отметить, что данный подход может применяться не только при групповом обучении, но и при индивидуальном.

Концепция ТРИЗ-эволюционности знаний позволяет наметить подходы к разрешению основного противоречия образования между объёмом передаваемых знаний и временем на их освоение [3]. ТРИЗ-эволюционный подход к искусственным объектам был предложен аналогично фрактальному подходу [4]. Между ТРИЗ-эволюционностью и фрактальным подходом в исследовании объектов различной природы есть сильная связь. Укрупненно развитие (эволюция) фрактального объекта происходит следующим образом. Исходный объект (паттерн) в соответствии с правилами эволюции (законами эволюции), используя ресурсы окружающей среды,

многократно воспроизводится (копируется), увеличивая при этом свою «сложность» (рис. 1).

Рис. 1. Развитие фрактального объекта

В свою очередь, ТРИЗ-эволюционный подход может быть применён к эволюции знаний. Сначала для выбранной области знаний определяются исходные положения – аксиомы, что эквивалентно паттернам. Затем выявляются и оцениваются ресурсы соответствующей области знаний. Наконец, выявляются правила «строительства», используя инструментарий ТРИЗ.

В целом процесс исследования ТРИЗ эволюции состоит:

– из описания исходного объекта;

– выявления противоречий у выбранного объекта;

– определения инструментов ТРИЗ, позволяющих разрешить выявленные противоречия;

– описания последующих объектов, в которых разрешены отдельные противоречия;

– повторения предыдущих действий для всех наиболее значимых объектов исследуемой области;

– построения и анализа ТРИЗ эволюционной карты.

Такой ТРИЗ-эволюционный подход был использован при структурировании знаний по численным методам [5], по CASE системам [6], парадигмам программирования [7].

В рамках методологии использования ТРИЗ-эволюционного подхода в обучении предполагается, что обучение проводится при помощи ТРИЗ-эволюционных карт, которые отражают систематизированные знания конкретной предметной области.

Укрупненно процесс обучения с использованием ТРИЗ-эволюционных карт включает следующие этапы [3]:

1) Сначала студенты изучают все инструменты ТРИЗ. Если по каким-то причинам они не смогут изучить все инструменты, то тогда изучают только приёмы разрешения противоречий.

2) После этого студенты начинают изучение дисциплины с самого простого набора базовых знаний (исходные объекты ТРИЗ-эволюции). Студентам предлагается решить самую простую задачу.

3) Затем увеличивается сложность задачи, и снова предлагается студентам её решить.

4) Далее студенты определяют противоречия и предпринимают попытку разрешить эти противоречия инструментами ТРИЗ. То есть они должны предложить новый механизм решения задачи или, по крайней мере, определить свойства, которыми должен обладать этот механизм. Таким образом, студенты «открывают» для себя все последующие элементы ТРИЗ-эволюционной карты.

Можно сказать, что данная модель представляет собой циклическую последовательность (рис. 2). Глубина цикла же зависит от объема знаний, положенных к изучению (количества элементов ТРИЗ-эволюционной карты).

Рис. 2. ТРИЗ-эволюционный подход

На каждой итерации студент выполняет следующие шаги:

1 Постановка и формализация задачи.

2 Решение задачи средствами исходного объекта ТРИЗ-эволюции.

3 Анализ решения:

а) выявление недостатков;

б) формулировка противоречий.

4 Разрешение противоречий с использованием приемов разрешения противоречий.

5 Описание предложенных при разрешении противоречия средств.

6 Решение задачи с использованием предложенных средств.

Структура пособия определена в соответствии с принципами, лежащими в основе ТРИЗ-эволюционного подхода в образовании, и предполагает изучение дисциплины по описанной технологии. Пособие включает в себя следующие пять основных разделов.

Первый раздел «Место объектно-ориентированной парадигмы программирования в системе языков программирования» посвящен описанию истории зарождения объектно-ориентированного подхода. Рассматриваются как предпосылки создания объектной модели, так и зарождение объектно-ориентированной парадигмы программирования в ходе эволюции парадигм программирования.

Второй раздел «Объектно-ориентированная парадигма программирования» описывает основные принципы объектно-ориентированного подхода к программированию. В разделе приведена информация об объектно-ориентированных языках программирования и их классификации, дано определение объектно-ориентированного языка программирования; описано историческое развитие объектно-ориентированных языков, определены связи между ними и подходы к их классификации.

Третий раздел «Абстракция и наследование» и четвертый раздел «Инкапсуляция и полиморфизм» описывают основные средства реализации данных принципов в языках программирования, начиная с языка Simula-67 – первого объектно-ориентрованного языка программирования. В разделах проанализирована эволюция средств реализации данных принципов ООП, приведены примеры реализации в различных языках, рассмотрены практические задачи использования данных средств.

Пятый раздел «Эволюция прочих механизмов в объектно-ориентированных языках программирования» описана эволюция выразительных средств языков программирования, а также механизмов отладки приложений. Эволюция данных механизмов может быть прослежена и на группе языков любой другой парадигмы, так как не характеризует реализацию объектно-ориентированных возможностей.

В целом же в пособии в той или иной степени рассмотрены следующие языки программирования: Simula-67; Smalltalk; C++; Eiffel; Python; Perl 6; Java.

Каждый раздел содержит практические задания, в рамках которых студенты самостоятельно проводят анализ программного кода, определяют противоречия и предлагают варианты их решения. Каждый раздел сопровождается заданием на лабораторную работу, которое выполняется по мере изучения подразделов.

В пособии также описаны варианты заданий на курсовую работу и рекомендации по ее выполнению, приведены примеры выполнения лабораторных работ и тесты для самоконтроля, экзаменационные вопросы и задания. Кроме того, в приложении описан пример использования ТРИЗ-эволюционного подхода в рамках самостоятельного изучения языков программирования и их возможностей на примере изучения языка Python, а именно фреймворка Django и средств создания сайтов и веб-приложений.

1. МЕСТО ОБЪЕКТНО-ОРИЕНТИРОВАННОЙ ПАРАДИГМЫ ПРОГРАММИРОВАНИЯ В СИСТЕМЕ ЯЗЫКОВ ПРОГРАММИРОВАНИЯ

1.1. История развития представлений о программировании и формирования объектной модели

Выразительная сила языка влияет на глубину мыслей и соответственно позволяет наиболее полно их описать с помощью управляющих структур языка и его особенностей. Сложно осмыслить структуры, которые нельзя описать устно или письменно, поэтому для выражения абстрактных идей стало необходимо разработать языки представления этих идей и те управляющие структуры, которые могли быть использованы для их описания.

Для выражения программных идей были созданы языки программирования, которые в зависимости от поставленной задачи стали обладать присущим только данному языку разнообразием свойств.

С момента появления первых электронно-вычислительных машин подходы к разработке программного обеспечения прошли большой путь: от восхищения фактом возможности написать хоть какую-нибудь программу до осознания того, что именно технология разработки программного обеспечения определяет прогресс в вычислительной технике [8].

Современные системы программного обеспечения намного масштабнее и сложнее своих предшественников. Возрастание сложности стимулировало многочисленные прикладные исследования по методологии проектирования программного обеспечения, особенно в области декомпозиции, абстракции и иерархии. Возникла тенденция перехода от языков, указывающих компьютеру, что делать, к языкам, описывающим ключевые абстракции проблемной области и связи между ними.

П. Вегнер (P. Wegner) [9] сгруппировал некоторые из наиболее известных языков высокого уровня по поколениям в зависимости от того, какие языковые конструкции впервые в них появились (рис. 1.1). В каждом следующем поколении менялись поддерживаемые языками механизмы абстракции.

Языки первого поколения (1954 - 1958) ориентировались на научно-инженерные применения, и словарь этой предметной области был почти полностью математическим.

Рис. 1.1. Классификация П. Вегнера

Во втором поколении языков (1959 - 1961) основной тенденцией стало развитие алгоритмических абстракций. В это время мощность компьютеров быстро росла, а компьютерная индустрия позволила расширить области их применения. Главной задачей стало инструктировать машину, что делать. Второе поколение характеризовалось возможностями использования подпрограмм, раздельной компиляции, типов данных, обработки списков, наличием указателей и сборки мусора.

Начиная с середины 60-х годов стали осознавать роль подпрограмм как важного промежуточного звена между решаемой задачей и компьютером. Использование подпрограмм как механизма абстрагирования имело три существенных последствия. Во-первых, были разработаны языки, поддерживавшие разнообразные механизмы передачи параметров. Во-вторых, были заложены основания структурного программирования, что выразилось в языковой поддержке механизмов вложенности подпрограмм и в научном исследовании структур управления и областей видимости. В-третьих, возникли методы структурного проектирования, стимулирующие разработчиков создавать большие системы, используя подпрограммы как готовые строительные блоки.

В конце 60-х годов с появлением транзисторов, а затем интегральных схем, стоимость компьютеров резко снизилась, а их производительность росла почти экспоненциально. Развитие языков и методов программирования, хотя и существенно облегчило и ускорило разработку программных средств, не успевало за растущими с еще большей скоростью потребностями в программах. Единственным реальным способом удовлетворить необходимость в резком ускорении разработки являлся метод многократного использования кода. Разумеется, структурный подход к программированию предоставлял такую возможность. Однако ни гибкость этого метода, ни масштабы использования не позволяли существенно ускорить программирование. Также появилась необходимость в упрощении сопровождения и модификации разработанных систем. Требовалось радикально изменить способ построения систем, с тем, чтобы локальные модификации не могли нарушать работоспособность всей системы, и было легче производить изменения поведения системы. Наиболее существенная проблема, которую требовалось решить, – облегчение проектирования систем. Далеко не все задачи поддавались алгоритмическому описанию и тем более алгоритмической декомпозиции. Требовалось приблизить структуру программ к структуре решаемых задач.

В результате возникли языки третьего поколения (1962 - 1970), поддерживающие абстракцию данных. Теперь разработчики получили возможность описывать свои собственные виды данных (т.е. создавать пользовательские типы). Значение абстрактных типов данных в разрешении проблемы сложности систем хорошо выразил Шан-кар (Shan-Kar) [10]: «Абстрагирование, достигаемое посредством использования процедур, хорошо подходит для описания абстрактных действий, но не годится для описания абстрактных объектов. Это серьезный недостаток, так как во многих практических ситуациях сложность объектов, с которыми нужно работать, составляет основную часть сложности всей задачи». Первым, кто указал на важность построения систем в виде многоуровневых абстракций, был Дейкстра (Dijkstra). Позднее Парнас (Parnas) ввел идею сокрытия информации, то есть инкапсуляции. В 1970-х годах ряд исследователей, главным образом Лисков и (Zilles), Гуттаг (Guttag) и Шоу (Shaw), разработали механизмы абстрактных типов данных. Хоар (Hoare) дополнил эти подходы теорией подклассов. Несмотря на то, что технологии построения баз данных развивались независимо от языков программирования, они также внесли свой вклад в разработку объектной модели, в основном с помощью идей подхода «сущность-отношение» к моделированию данных. В данной модели, предложенной Ченом (Chen), моделирование происходит в терминах сущностей, их атрибутов и взаимоотношений. Свой вклад в развитие объектно-ориентированных абстракций внесли и исследователи в области искусственного интеллекта, разрабатывавшие методы представ-

ления знаний. Объектная модель получила широкое распространение лишь в конце 1980-х и начале 1990-х годов.

Идеи и принципы объектной модели, которые легли в основу объектно-ориентированной парадигмы программирования, позволили справиться со сложностью, характерной для широкого круга систем. По этой причине подход был признан объединяющей концепцией компьютерной науки, которую можно применять не только в языках программирования, но и при разработке пользовательских интерфейсов, баз данных и даже компьютерной архитектуры.

Чтобы получить более полное представление о причинах и движущих силах возникновения объектно-ориентированной парадигмы программирования рассмотрим эволюцию парадигм программирования.

1.2. Эволюция парадигм программирования

Прежде чем приступить к анализу эволюции парадигм программирования, рассмотрим основные определения, принятые в тексте.

Сформируем понятие «**язык программирования**», основываясь на мнениях разных авторов. Известно, что понятие «язык программирования» – это частный случай понятия «язык», которое является средством взаимодействия между передатчиком сообщения и приемником с целью реализации программной идеи [11]. В области программирования в роли передатчиков и приемников сообщений могут выступать как компьютер, так и человек, а взаимодействовать они будут с помощью команд и операторов, являющихся взаимопонятными. Понятность набора команд и операторов для человека зависит от конкретного языка программирования, с помощью которого человек реализует программную идею, то есть в каждом языке программирования, должны быть разработаны адекватные методы реагирования на сообщения автора.

Программная идея в данном случае будет являться направлением деятельности, на основе которого мы описываем задачу программирования. Задача программирования – обеспечение взаимодействия компьютера и человека.

Таким образом, определим понятие «**программирование**» как процесс описания программной идеи с помощью языка программирования, понятного приемнику и передатчику.

Существуют также такие понятия, как «концепция программирования» и «парадигма программирования».

Концепция программирования – это движущая сила для возникновения новой группы языков программирования или, другими словами, это противоречие или совокупность противоречий, разрешение которых привело к возникновению парадигмы программирования. Впервые ввёл поня-

тие «парадигма» Томас Кун (Thomas Kuhn) [12]. Кун называл парадигмами устоявшиеся системы научных взглядов, в рамках которых ведутся исследования.

Таким образом, **парадигма программирования** – это совокупность идей и понятий, сформулированных на основе концепции программирования и определяющих стиль программирования.

Очевидно, что каждая новая парадигма программирования не может являться принципиально новой, а будет сохранять некоторые признаки парадигм-предшественниц.

Для наглядного представления результатов анализа эволюции парадигм программирования опишем парадигмы программирования в виде формализованного макета, представленного на рис. 1.2.

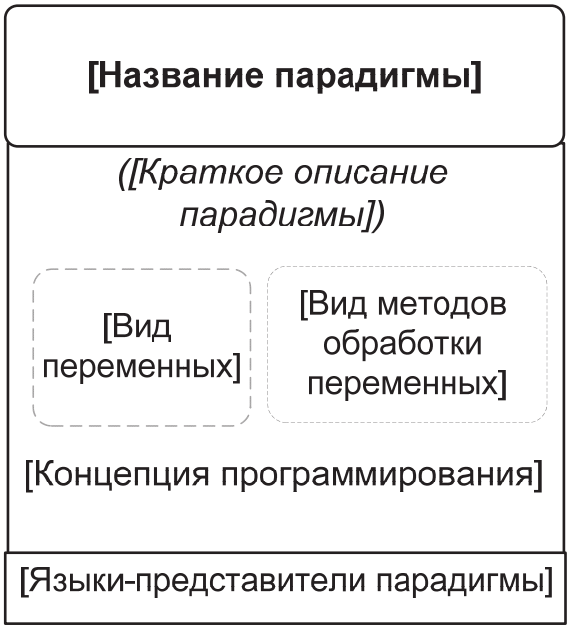

Рис. 1.2. Макет описания
парадигм программирования

Для того чтобы продемонстрировать эволюцию парадигм программирования выберем конкретную задачу, которую попробуем решить с помощью различных языков программирования. Задача будет заключаться в вычислении факториала числа. Данная задача выбрана по следующим критериям: простота реализации, наличие различных приемов вычисления даже в рамках одного языка программирования.

1.2.1. Машинное кодирование

Машинное кодирование – система команд вычислительной машины, которая интерпретируется для конкретного микропроцессора. Это самая первая и самая примитивная парадигма программирования (рис. 1.3).

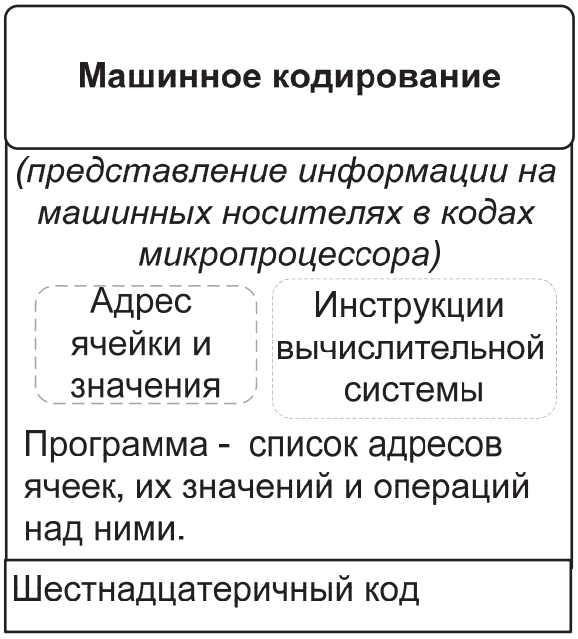

Рис. 1.3. Парадигма
«Машинное кодирование»

Далее представлен код программы в машинных кодах для вычисления факториала числа.

```
ADR  CODE           LINE      COMMENT
                    1     ; вычисление факториала
A000 DB             2     ; значение N
                    3     ;
4000 8B 06 00 A0    4     ; загрузка N в аккумулятор
4004 3D 01 00       5     ; if (@N=1) or (@N=0) then
4007 77 00 41       6     ; переход на вычисление
400A B8 01 00       7     ; factorial:=1
400D 33 12          8     ; очистка регистра
400F C3             9     ; возврат
                    10    ; Вычисление
4100 48             11    ; else
4101 50             12    ; аккумулятор в стек
4102 E8 00 40       13    ; factorial := factorial(@N-1)
4105 F7 E6          14    ; * @N;
4107 C3             15    ; возврат
```

Описанная парадигма является исходным объектом ТРИЗ-эволюции и как любой «первенец» обладает рядом недостатков. Основным ограничением данной парадигмы является сложность поставленной задачи, так как для решения более сложных задач потребуется написать огромное количество строк кода. Таким образом, можно выделить следующие противоречия.

Противоречие 1.1: при увеличении сложности задач недопустимо увеличивается объем машинного кода (машинных операций).

Противоречие 1.2: при увеличении сложности задач программирования недопустимо усложняется структура процессора.

Противоречие 1.3: при увеличении сложности задач недопустимо увеличивалось время программирования.

Для разрешения возникших противоречий применимы принцип копирования и принцип объединения. Таким образом, произошел переход к новой парадигме программирования – ассемблированию.

1.2.2. Ассемблирование

Рассмотрим конкретные решения, устраняющие выявленные в парадигме «Машинное кодирование» противоречия.

Решение 1.1: используя принцип копирования, заменим понятие адреса ячеек и их значения понятием «операнд».

Решение 1.2: используя принцип объединения, соединили однородные машинные инструкции в мнемонические команды.

Благодаря разрешению противоречий появился *новый механизм*: возможность преобразовать машинные инструкции в мнемонический код. Данный ресурс лег в основу новой концепции, а соответственно и парадигмы программирования – ассемблирования.

Ассемблирование (рис. 1.4) – парадигма, при которой происходит компиляция исходного текста программы, написанной на языке Ассемблер, в программу на машинном языке. Обращение к процессору выполняется через мнемонические команды, которые соответствуют инструкциям процессора вычислительной системы [13].

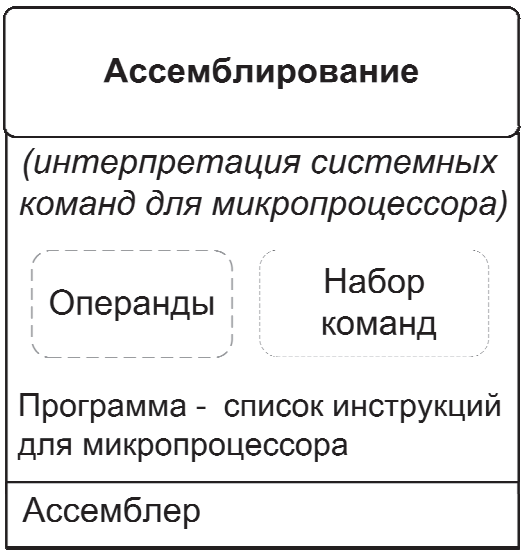

Рис. 1.4. Парадигма «Ассемблирование»

Далее представлен код программы для вычисления факториала числа.

```
Factorial           PROC   ; function
factorial(@@N:Word):DWord  ;
   arg    @@N:word          ; begin
   mov    ax,@@N
   cmp    ax,1              ; if (@@N=1) or (@@N=0) then
   ja     @@calc
   mov    ax,1              ; factorial:=1
   xor    dx,dx
   ret
@@calc:
   dec    ax                ; else
   push   ax
   call   Factorial         ; factorial := factorial(@@N-1)
   mul    @@N               ; * @@N;
   ret                      ; end of calc;
endp
```

На примере оператора “mov” можно убедиться, что использование мнемонического кода, введенного благодаря решению 1.2 противоречий 1.1, 1.2 и 1.3, уменьшило не только время программирования, но и объем программного кода, не усложняя при этом структуру процессора.

Оператор “mov” выполняет функцию пересылки следующих операндов: reg, reg; mem, reg, mem, immed и так далее. В свою очередь, например, операнд reg может содержать в себе один из 8-, 16- или 32-разрядных регистров из списка: AH, AL, BH, BL, CH, AX, BX, CX, DX, SI, DI, BP, SP, EAX, EBX, ECX, EDX, ESI, EDI, EBP, ESP, а оператор mem – содержать ссылку на адрес ячейки в памяти. Таким образом, вместо подробного описания машинных инструкций и адресов памяти получаем одну мнемоническую команду “mov” с подчиненными операндами.

С развитием информационных технологий и появлением новых задач, стало ясно, что и данная парадигма не удовлетворяет требованиям, предъявляемым к языкам программирования. В силу машинной ориентации языка ассемблера человеку сложнее читать, понимать и разрабатывать программу, соответственно возрастают затраты на разработку программ. Для решения сложных задач необходимо привлекать высококвалифицированного специалиста. Кроме того, программа, написанная на языке ассемблера, может быть запущена только на компьютере, обладающем архитектурой, под которую программа разрабатывалась. Далее приведены основные противоречия данной парадигмы.

Противоречие 1.4: при увеличении разнообразия аппаратных платформ недопустимо увеличивается количество несовместимых частей программных команд и процессора.

Противоречие 1.5: при увеличении сложности задач программирования недопустимо увеличивается объем программного кода.

Противоречие 1.6: при увеличении сложности задач недопустимо увеличивалось время программирования.

Противоречие 1.7: при увеличении количества реализованных математических функций недопустимо нарушается логика вычислений.

Противоречие 1.8: при увеличении сложности реализованных математических функций недопустимо нарушается логика вычислений.

Противоречие 1.9: при доказательстве математических теорем недопустимо нарушается логика вычислений.

Для разрешения возникших противоречий были применены принципы объединения, вынесения, предварительного действия, что привело к развитию новых парадигм программирования – процедурной, логической и функциональной.

1.2.3. Процедурная парадигма

Рассмотрим конкретные решения, устраняющие противоречия 1.4, 1.5 и 1.6 парадигмы «Ассемблирование».

Решение 1.3: используя принцип объединения, соединили однородные машинные инструкции в приказы (императивы), для этого наборы машинных команд для микропроцессора и совокупности элементарных операций были объединены и ассоциированы с командным словом.

При этом появился новый механизм написания программного кода: возможность заменить машинный код императивом.

Процедурное программирование (рис. 1.5) – программирование на императивном языке, при котором последовательно выполняемые операторы можно собрать в подпрограммы, то есть более крупные целостные единицы кода, с помощью механизмов самого языка.

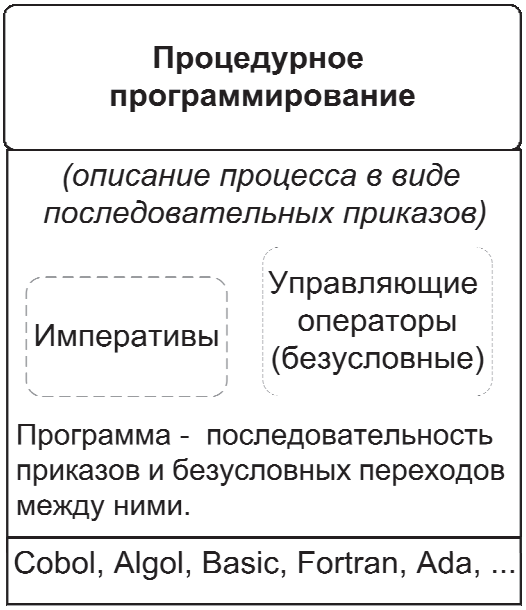

Рис. 1.5. Процедурная парадигма

При этом императивный язык – язык программирования, который описывает процесс вычисления в виде инструкций, изменяющих состояние программы [11]. Императивная программа представляет собой приказы,

выражаемые повелительным наклонением в естественных языках, то есть это последовательность команд для компьютера.

Составим программу на Basic для вычисления факториала числа.

```
10 n=0
20 j=0
30 k=0
40 cls
50 input "Введите значение n = ", n$
60 if n<1 goto 90
70 k=1
80 for j=1 to n
90 k=k*j
100 next j
110 print " Факториал равен= "; k$
120 END
```

Важным событием для программирования более сложных задач стало использование в программном коде императивов, представляющих собой те же машинные инструкции, только замененные ассоциативным словом, например, императив input, print, next и другие.

Однако это введение имело и отрицательную сторону. В данном примере продемонстрирован механизм безусловного перехода: оператор «go to» в случае соответствия условию $n < 1$ отсылает к определенной точке программы. Для масштабных задач стало невозможным использование таких переходов и со временем проявились следующие противоречия [14].

Противоречие 1.10: механизм безусловного перехода к определенной точке программы привёл к увеличению неструктурированных программных конструкций, вследствие чего недопустимо увеличилось время компилирования программы.

Противоречие 1.11: при увеличении количества безусловных переходов недопустимо уменьшается взаимозависимость фрагментов кода.

Для разрешения возникших противоречий применим принцип предварительного действия, что повлекло за собой появление парадигмы структурного программирования.

1.2.4. Логическая парадигма

Рассмотрим конкретные решения, устраняющие противоречия 1.7, 1.8 и 1.9 парадигмы «Ассемблирование».

Решение 1.4: используя принцип вынесения, отказались от операторов присваивания и управляющих операторов.

Решение 1.5: используя принцип предварительного действия, программу стали представлять в виде высказываний в символьной логике.

В результате разрешения противоречий появились новые механизмы, определяющие концепцию логической парадигмы программирования: возможность использования программного вычисления предикатов и автоматического доказательства теорем.

Логическое программирование – парадигма программирования, основанная на автоматическом доказательстве теорем [12]. Логическое программирование основано на теории и аппарате математической логики с использованием математических принципов (рис. 1.6).

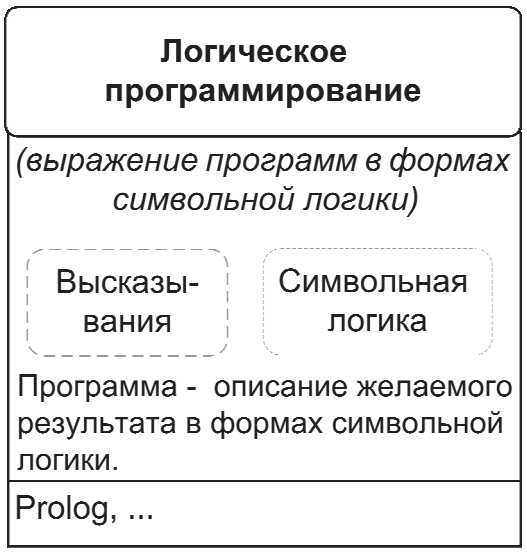

Рис. 1.6. Логическая парадигма

Посмотрим, как будет выглядеть программа рассматриваемой задачи на логическом языке Turbo Prolog.

```
domains
       N,F=real
predicates
       factorial(N,F)
       result
clauses
       factorial(0,1).
       factorial(N,F):-N>0,N1=N-1,factorial(N1,F1),
                       F=F1*N.
       result:-write("Input N"),nl,
       write("N="),readreal(N),factorial(N,F),
       write(N,"!=",F).
goal
       result
```

Программа демонстрирует, как рассматриваемая задача решается с помощью введения принципов логического вывода информации на основе заданных фактов и правил вывода. Подобно доказательству теорем, выделяются секции *domains*, *predicates*, *clauses* и *goal*, с помощью которых и формируется алгоритм решения задачи.

Так как данная парадигма была создана для описания математических задач, то именно логический подход стал наиболее применим и эффективен для решения задач дискретной математики и доказательства теорем. Кроме того, этот подход всегда выступал в качестве альтернативы императивному подходу в программировании, и, в частности, процедурному программированию. Но с повышением уровня задач программирова-

ния для полноценного решения задачи требовалось введение объектов и привязки к ним алгоритмов математической логики.

Противоречие 1.12: при увеличении сложности задач, требующих обобщенного подхода, недопустимо усложняется структура программы, заключающаяся в имитации объектных свойств языка.

Данное противоречие было разрешено с развитием парадигм программирования и возникновением объектно-ориентированной парадигмы.

1.2.5. Функциональная парадигма

Рассмотрим конкретные решения, устраняющие противоречия 1.7, 1.8 и 1.9 парадигмы «Ассемблирование».

Решение 1.6: используя принцип вынесения, отказались от понятия «переменной», операторов присваивания и соответственно от хранения состояний программы, а также отказались от метода вычисления последовательности изменений состояний функций.

Данное решение предполагает создание нового механизма – модели вычислений без состояний. Данная модель предполагает, что программа ни целиком, ни частями состояния не имеет и побочных эффектов не производит. В результате появилась парадигма «Функциональное программирование».

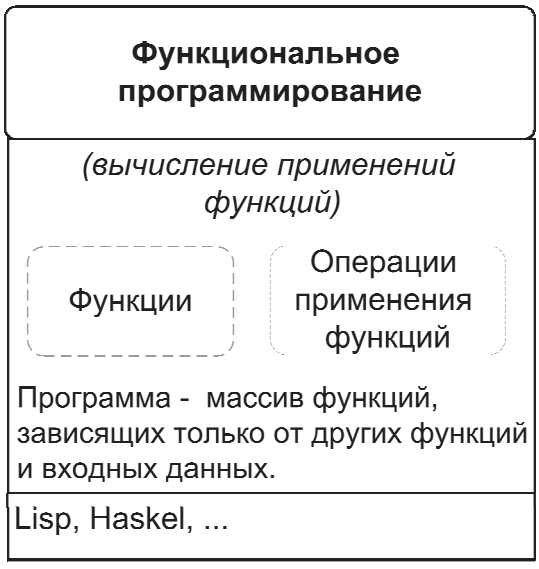

Рис. 1.7. Функциональная парадигма

Функциональное программирование – это парадигма программирования, в которой процесс вычисления трактуется как вычисление значений функций в математическом понимании (рис. 1.7).

Функциональное программирование предполагает обходиться вычислением результатов функций от исходных данных и результатов других функций, и не предполагает явного хранения состояния программы. Соответственно, не предполагает оно и изменяемость этого состояния.

Программа вычисления факториала на функциональном языке Common Lisp будет следующей:

```
(defun fact (n)
       (if (eql n 0)
            1
            (* n (fact (- n 1)))))
(fact 10)
```

Анализируя принцип построения кода программы, можно убедиться, что в отличие от процедурной парадигмы, описывающей шаги, ведущие к достижению цели, функциональная парадигма описывает математические отношения между данными и целью [15].

Например, в данном примере отношение между входными данными и результатом устанавливаются с помощью описания соответствующей функции нахождения факториала числа. Эта функция описывает типичный алгоритм работы функциональной программы – если входное число это 0, то результат будет равен 1, в остальных случаях результат будет находиться рекурсивно * *n (fact (- n 1))*. Отметим, что функциональный язык не подразумевает сравнений в условиях, так как в таком языке отсутствует понятие переменной и понятие присваивания [16].

В традиционном понимании переменная должна хранить изменяющиеся состояния ячейки данных, что противоречит понятию переменной как хранилища неизменных входных данных. Тем не менее ничто не мешает использовать для традиционного понимания переменной функцию, которая бы и явилась хранилищем переменных значений. В этом заключается мультипарадигменная основа функционального языка, которая применяется в языках обобщенного программирования.

Недостатки функционального программирования вытекают напрямую из его особенностей. Отсутствие присваиваний и замена их на порождение новых данных приводят к необходимости постоянного выделения и автоматического освобождения памяти, поэтому в системе исполнения функциональной программы обязательным компонентом становится высокоэффективный «сборщик мусора». Кроме того, при решении комплексных задач приходится тратить много времени на воспроизведение возможностей императивных языков программирования.

Противоречие 1.13: при увеличении сложности задач, требующих мультипарадигменного подхода, недопустимо усложняется структура программы, заключающаяся в создании повторных конструкций имитируемого языка.

Данное противоречие было разрешено с развитием парадигм программирования и возникновением объектно-ориентированной парадигмы.

1.2.6. Структурная парадигма

Рассмотрим конкретные решения, устраняющие противоречия 1.10 и 1.11 процедурной парадигмы программирования.

Решение 1.7: используя принцип предварительного действия, заранее выставляем части кода так, чтобы они выполнялись по наиболее логичной структуре, используя условные переходы.

Появился новый механизм: возможность представления кода в виде иерархической структуры с помощью логических операторов.

Структурное программирование – это парадигма программирования, в основе которой лежит представление программы в виде иерархической структуры блоков (рис. 1.8).

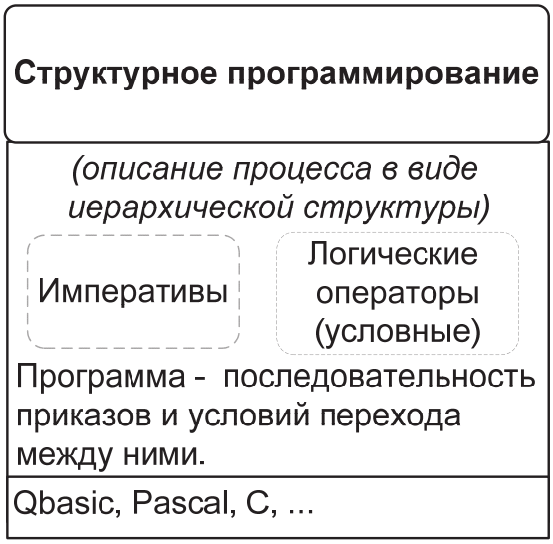

Рис. 1.8. Структурная парадигма

Следование принципам структурного программирования сделало тексты программ более читабельными. Серьёзно облегчилось понимание программ, появилась возможность разработки программ в нормальном промышленном режиме, когда программу может без особых затруднений понять не только её автор, но и другие программисты.

Алгоритм на языке Pascal для вычисления факториала числа будет отражать структурную организацию кода.

```
function fact(n : integer) : longint;
  begin
    if n <= 1 then
      fact := 1
    else
      fact := n * fact(n - 1);
  end;
```

На данном примере видно, как отказ от безусловного перехода привел к повышению структурированности кода. Были введены условия *if [ ] then [ ] else, do [ ] loop while* и другие, обеспечивающие выполнение определённой команды только при условии истинности некоторого логического выражения, либо выполнение одной из нескольких команд в зависимости от значения некоторого выражения [17].

Однако структурная парадигма также обладает рядом недостатков.

Увеличение сложности решаемой задачи приводит к использованию большого количества однотипных операторов, например, операторов *begin* и *end*, предварительных объявлений переменных и так далее. Наиболее сильной критике со стороны разработчиков структурного подхода к программированию подвергся оператор безусловного перехода. Неправильное использование произвольных переходов в тексте программы приводит к получению запутанных, плохо структурированных программ, по тексту которых практически невозможно понять порядок исполнения и взаимозависимость фрагментов.

Противоречие 1.14: увеличение в программном коде множества однотипных функций, применяемых для разных объектов, недопустимо увеличивало время разработки и отладки программы.

Данное противоречие также было разрешено с возникновением объектно-ориентированной парадигмы.

1.2.7. Объектно-ориентированная парадигма

Рассмотрим конкретные решения, устраняющие противоречия 1.12, 1.13 и 1.14 структурной, логической и функциональной парадигм программирования.

Решение 1.8: используя принцип местного качества, перешли от однородной структуры кода к неоднородной, для чего объединили однородные части кода.

Решение 1.9: с помощью принципа объединения соединили абстрактные типы данных в классы, а экземпляры классов в объекты.

Решение 1.10: с помощью принципа универсальности сделали возможным использование одинаковых функций для разных объектов за счет наследования свойств одного класса у другого.

Решение 1.11: с помощью принципа «матрешки» ввели понятия родительского класса и наследного класса, при этом наследный класс стал динамически связан с родительским, свойства класса-родителя стали переходить и на класс-наследник.

Появился новый механизм: объектная модель, которая предоставляет возможность написания кода с более высокой организацией структуры.

Объектно-ориентированное программирование – это парадигма программирования, в которой основными концепциями являются понятия объектов и классов (рис. 1.9).

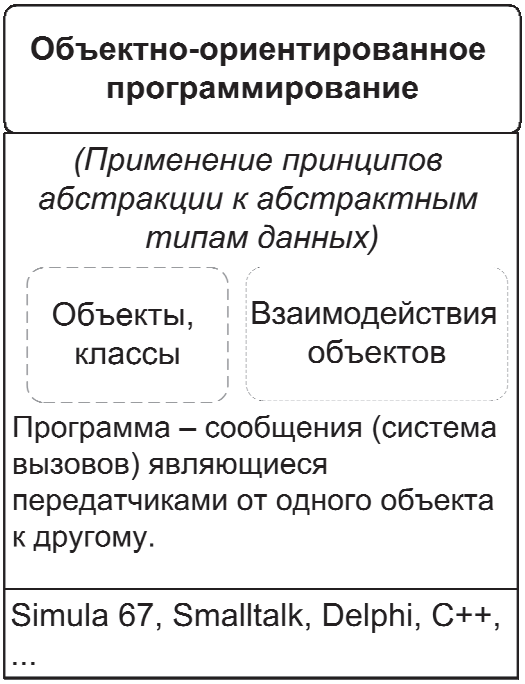

Рис. 1.9. Объектно-ориентированная парадигма

Программный код задачи, решаемой с помощью языка C++, будет выглядеть следующим образом:

```
unsigned long long fact(unsigned int n)
  { if(n <= 1) {return 1;}
else {return n * fact(n - 1);} }
```

Этот пример показывает, что отказ от использования однотипных конструкций структурных языков привел к введению визуально легких конструкций. Для примера, операторы *{ }* заменили аналогичные операторы начала и конца функции *begin* и *end*, синтаксис условных конструкций сократился до использования лишь *if (условие)* и *else*, наконец, стало возможным записывать условие одной строкой, не нарушая при этом логики программы. Как следствие, получили более высокую организацию кода.

Таким образом, место объектно-ориентированной парадигмы в системе языков программирования показано на рис. 1.10.

Рис. 1.10. ТРИЗ эволюционная карта парадигм программирования

Объектно-ориентированный подход, который лежит в основе объектно-ориентированной парадигмы программирования, принципиально отличается от подходов, которые связаны с методами структурного анализа, проектирования и программирования, и в свое время был воспринят разработчиками как «панацея» благодаря своим явным преимуществам по сравнению с другими подходами.

Во-первых, объектная декомпозиция дает возможность создавать программные системы меньшего размера путем использования общих механизмов, обеспечивающих необходимую экономию выразительных средств.

Во-вторых, объектная декомпозиция упрощает процесс создания сложных систем ПО, которое представляет собой набор относительно небольших подсистем.

В-третьих, процесс интеграции системы происходит параллельно с процессом разработки, а не превращается в единовременное событие.

Поскольку структура программного обеспечения, реализованного при помощи объектно-ориентированного подхода, отражает реальный или воображаемый мир, он позволяет размышлять о задаче в терминах существующих в этом мире объектов, а не в терминах языка программирования.

К тому же основные операции, заложенные в программное обеспечение, имеют тенденцию изменяться намного медленнее, чем информационные потребности определенных групп людей или учреждений. Это означает, что программное обеспечение, основанное на общих моделях, будет существовать и успешно работать много дольше, чем то, которое написано для решения определенной, сиюминутной проблемы.

В настоящее время объектно-ориентированный подход к программированию используется в подавляющем большинстве промышленных проектов.

2. ОБЪЕКТНО-ОРИЕНТИРОВАННАЯ ПАРАДИГМА ПРОГРАММИРОВАНИЯ

2.1. Концептуальная база объектно-ориентированной парадигмы

Каждый подход к программированию имеет свою концептуальную базу и требует своего способа восприятия решаемой задачи. Для объектно-ориентированного подхода концептуальная база – это объектная модель. Она строится на четырех основных принципах (абстрагирование, инкапсуляция, полиморфизм и наследование) [10]:

1) **Абстрагирование** – выделение существенных характеристик некоторого объекта, отличающих его от всех других видов объектов и, таким образом, четко определяет его концептуальные границы с точки зрения наблюдателя. Абстракция концентрирует внимание на внешнем представлении объекта и позволяет отделить существенные особенности поведения от их реализации. Абельсон и Суссман назвали такое разделение поведения от его реализации принципом минимальных обязательств, в соответствии с которым интерфейс объекта должен обеспечивать только существенные аспекты его поведения и ничего больше. Существует также дополнительный принцип наименьшего удивления, согласно которому абстракция должна полностью описывать поведение объекта, ни больше и ни меньше, и не порождать сюрпризы или побочные эффекты, выходящие за пределы абстракции.

2) **Инкапсуляция** – процесс отделения друг от друга элементов объекта, определяющих его устройство и поведение. Инкапсуляция служит для того, чтобы изолировать контрактные обязательства абстракции от их реализации. Абстракция и инкапсуляция дополняют друг друга. В центре внимания абстракции находится наблюдаемое поведение объекта, а инкапсуляция сосредоточена на реализации, обеспечивающей заданное поведение. Как правило, инкапсуляция осуществляется с помощью сокрытия информации (а не просто сокрытия данных), т.е. утаивания всех несущественных деталей объекта. Обычно скрываются как структура объекта, так и реализация его методов.

3) **Полиморфизм** – обозначение различных действий одним именем и свойство объекта отвечать на направленный к нему запрос сообразно своему типу. Полиморфизм позволяет обойтись без операторов выбора, при этом связь метода и имени определяется только в процессе выполнения программ.

4) **Наследование** – механизм, позволяющий определять новые типы данных на основе существующих таким образом, что данные и функции существующих классов становятся членами наследуемых классов.

Без любого из них модель не будет объектно-ориентированной. С момента зарождения объектно-ориентированного подхода концептуальная база ООП развивалась и эволюционировала вместе с языками, реализующими данный подход к программированию. Описанные принципы представляют собой группы механизмов, которые их реализуют в различных языках программирования. Таким образом, обозначим следующие определения:

Группа механизмов ООП – набор выразительных средств языка или программных инструментов, которые объединены по признаку реализации одного из принципов ООП.

Механизм – некоторый инструмент или выразительное средство языка, которое может включать в себя модели, алгоритмы, синтаксические единицы, принципы построения программы и т.д.

Каждый более поздний элемент в группе механизмов расширяет функциональные возможности языка в рамках реализации ООП.

Кроме того, имеются еще дополнительные принципы:

– **Модульность** – свойство системы, которая была разложена на внутренне связанные, но слабо связанные между собой модули. Правильный выбор модулей для решения поставленной задачи является почти такой же сложной задачей, как и выбор правильного набора абстракций. Существует несколько эмпирических приемов и правил, позволяющих обеспечить разумную модульность. Бриттон (Britton) и Парнас утверждают: «Конечной целью декомпозиции программы на модули является снижение затрат на программирование, благодаря независимому проектированию и тестированию. Структура каждого модуля должна быть достаточно простой для понимания, допускать независимую реализацию других модулей и не влиять на поведение других модулей, а также позволять легкое изменение проектных решений».

– **Типизация** – это правила использования объектов, не допускающие или ограничивающие взаимную замену объектов разных классов. Типизация заставляет проектировщиков выражать свои абстракции так, чтобы язык программирования, используемый для реализации системы, поддерживал принятые проектные решения.

– **Параллелизм** – возможность различным объектам действовать одновременно.

– **Устойчивость** – способность объекта существовать во времени, переживая породивший его процесс, и (или) в пространстве, перемещаясь из своего первоначального адресного пространства.

Эти принципы важны в объектной модели, но не обязательны.

По мнению Алана Кея (Alan Key) [18], создателя языка Smalltalk, которого считают одним из «отцов-основателей» объектно-ориентированного программирования, объектно-ориентированный подход к программированию заключается в следующем наборе следующих ключевых положений:

1) Всё является объектом.

2) Вычисления осуществляются путём взаимодействия (обмена данными) между объектами, при котором один объект требует, чтобы другой объект выполнил некоторое действие. Объекты взаимодействуют, посылая и получая сообщения. Сообщение – это запрос на выполнение действия, дополненный набором аргументов, которые могут понадобиться при выполнении действия.

3) Каждый объект имеет независимую память, которая состоит из других объектов.

4) Каждый объект является представителем (экземпляром) класса, который выражает общие свойства объектов.

5) В классе задаётся поведение (функциональность) объекта. Тем самым все объекты, которые являются экземплярами одного класса, могут выполнять одни и те же действия.

6) Классы организованы в единую древовидную структуру с общим корнем, называемую иерархией наследования. Память и поведение, связанное с экземплярами определённого класса, автоматически доступны любому классу, расположенному ниже в иерархическом дереве.

В целом, согласно объектно-ориентированному подходу программирования, декомпозиция программы должна непосредственно отображать структуру прикладной предметной области, в которой выделяются вполне различимые сущности – объекты, обладающие индивидуальностью, проявляемой как их состояние и поведение.

Наряду с методологией построения программного обеспечения, несомненно, важны особенности конкретного языка программирования, поскольку в конечном счете наши конструкции должны быть выражены на каком-то языке.

Выделим группу механизмов, которая определяет особенности разработки программ на том или ином языке программирования:

Синтаксис – механизм языка программирования, который описывает структуру программ в виде наборов символов, слов, операторов.

Структура программы – механизм, определяющий строение программного кода.

Отладка – механизм, упрощающий процесс выявления и устранения ошибок в программе.

Использование объектно-ориентированного подхода не ограничено каким-либо одним языком программирования – он применим к широкому спектру объектных и объектно-ориентированных языков. Объектно-ориентированный подход стал основой объектно-ориентированной парадигмы программирования, и в настоящее время количество прикладных языков программирования, реализующих данную парадигму, является наибольшим по отношению к другим парадигмам.

2.2. Определение объектно-ориентированного языка программирования

Ренч (Rench) однажды предсказал: «В 1980-х годах объектно-ориентированное программирование будет занимать такое же место, какое занимало структурное программирование в 1970-х. Оно всем будет нравиться. Каждая фирма будет рекламировать свой продукт как созданный по этой технологии. Все программисты будут писать в этом стиле, причем все по-разному. Все менеджеры будут рассуждать о нем. И никто не будет знать, что же это такое» [10].

На сегодняшний день нет точного определения объектно-ориентированного языка программирования (ООП). В различной литературе авторы дают различное разъяснение этим терминам.

Г. Буч (G.Booch), главный исследователь корпорации Rational Software, признанный всем международным сообществом разработчиков программного обеспечения благодаря его основополагающим работам в области объектно-ориентированных методов и приложений, дал следующее определение ООП: «Объектно-ориентированное программирование – это метод реализации, в котором программы организуются в виде взаимодействующих наборов объектов, каждый из которых представляет собой экземпляр класса, а классы являются членами иерархии, связанной отношением наследования» [10].

В соответствии с этим определением не все языки программирования являются объектно-ориентированными, хотя теоретически возможна имитация объектно-ориентированного программирования на таких языках, как Pascal, COBOL или Ассемблер, но это крайне затруднительно. Б. Страуструп (B. Stroustrup), создатель языка C++, утверждает: «Если написание программ в стиле ООП требует специальных усилий или оно невозможно совсем, то этот язык не отвечает требованиям ООП» [19].

Л. Карделли (L. Cardelli) и П. Вегнер говорят, что: «язык программирования является объектно-ориентированным тогда и только тогда, когда выполняются следующие условия:

– поддерживаются объекты, то есть абстракции данных, имеющие интерфейс в виде именованных операций и собственные данные, с ограничением доступа к ним;

– объекты относятся к соответствующим типам (классам);

– типы (классы) могут наследовать атрибуты супертипов (суперклассов)» [10].

Таким образом, определим *объектно-ориентированный язык программирования* как язык программирования, в качестве базовых элементов которого выступают объекты, имеющие собственные параметры и именованные операции и образующие иерархически организованные классы объектов.

Данное определение в дальнейшем будем использовать в качестве *рабочего*.

2.3. Объектно-ориентированные языки программирования

Первым объектно-ориентрованным языком был язык Simula (1967). Этот язык был предназначен для моделирования различных объектов и процессов, и объектно-ориентированные черты появились в нем именно для описания свойств модельных объектов.

Популярность объектно-ориентированному программированию принес язык Smalltalk, созданный в 1980 году. Язык предназначался для проектирования сложных графических интерфейсов и был первым по-настоящему объектно-ориентированным языком. В нем классы и объекты — это единственные конструкции программирования [20].

Объектно-ориентированное программирование развилось в доминирующую методологию программирования в начале и середине 1990 годов, когда стали широко доступны поддерживающие ее языки программирования, такие как C++ и Delphi. Доминирование этой методологии поддерживалось ростом популярности графических интерфейсов пользователя, которые основывались на техниках ООП.

Наиболее известные объектно-ориентированные языки программирования: Simula-67 (1967 г.); Smalltalk (1980 г.); C++ (1983 г.); Eiffel (1986 г.); Python (1990 г.); Java (1995 г.); Delphi (1995 г.); Perl 6 (2000 г.); C# (2001 г.); Scala (2003 г.). На рис. 2.1 представлена генеалогическая зависимость между указанными языками программирования.

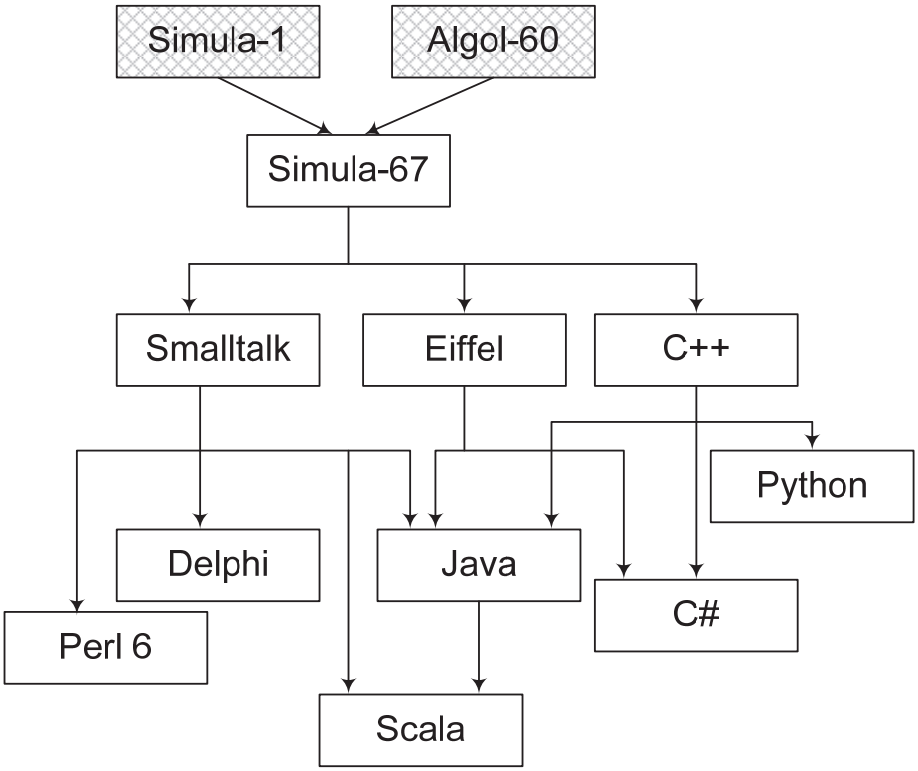

Рис. 2.1. Генеалогическая зависимость между языками объектно-ориентированной парадигмы

2.4. Классификация объектно-ориентированных языков программирования

Рассмотрим некоторые виды классификаций объектно-ориентированных языков программирования. Традиционно объектно-ориентированные языки разделяют на «чистые» и «гибридные».

Чистые – это те, которые позволяют использовать только одну модель программирования – объектно-ориентированную. Чистыми объектно-ориентированными языками являются Eiffel, Smalltalk.

C++ и Delphi, наоборот, типичные примеры гибридных языков, которые позволяют программистам использовать при необходимости возможности нескольких парадигм программирования.

Выделяют также «урезанные» языки, которые появились в результате удаления из гибридных языков наиболее опасных и ненужных с объектно-ориентированной точки зрения конструкций (Java, C#).

В работе Саундерса (Saunders) дан обзор более восьмидесяти объектно-ориентированных языков. Автор выделил в них семь категорий:

1) *Actor-языки*. Языки, поддерживающие механизм делегирования.

2) *Параллельные языки*. Объектно-ориентированные языки, нацеленные на параллелизм.

3) *Распределенные языки*. Объектно-ориентированные языки, нацеленные на обработку распределенных объектов.

4) *Основанные языки*. Языки, поддерживающие теорию фреймов.

5) *Гибридные языки*. Объектно-ориентированные надстройки над обычными языками.

6) *Языки типа Smalltalk*. Smalltalk и его диалекты.

7) *Идеологические языки*. Приложения объектной ориентированности к другим областям [10].

Классификации подвергаются также основные механизмы исполнения языков программирования, например, по принципу преобразования написанного кода в программу выделяют интерпретируемые и компилируемые языки, по характеру типизации выделяют языки со слабой или строгой типизацией. Критериев можно найти огромное количество и по каждому из них оценить существующий язык программирования. Совокупность различных критериев в том или ином языке определяет не только его функциональные возможности, но и его эффективность и производительность.

Из-за большого разнообразия языков программирования и их диалектов, которых в общей сложности насчитывается более 2500 тысяч, и критериев классификации программисту трудно объективно оценивать эффективность использования того или иного языка для решения поставленных задач, поэтому выбор языка чаще обоснован субъективными мотивами. Это приводит к нерациональному использованию одних языков и игнорированию других. В результате программисты пишут менее эффективные и менее изящные программы, чем могли бы.

3. АБСТРАКЦИЯ И НАСЛЕДОВАНИЕ

3.1. Абстракция как группа механизмов объектно-ориентированного программирования

Абстракция данных является важнейшей составной частью объектно-ориентированного программирования и мощным инструментом современного программирования. Этот концептуальный подход позволяет объединить тип данных с множеством операций, которые допустимо выполнять над этим типом данных.

Абстракция данных позволяет рассматривать необходимые объекты данных и операции, которые должны выполняться над такими объектами, без необходимости вникать в несущественные детали. Кроме того, **абстрактный тип данных** – это совокупность данных вместе с множеством операций, которые можно выполнять над этими данными.

Абстракция выделяет существенные характеристики некоторого объекта, отличающие его от всех других видов объектов и, таким образом, четко определяет его концептуальные границы с точки зрения наблюдателя (рис. 3.1) [10].

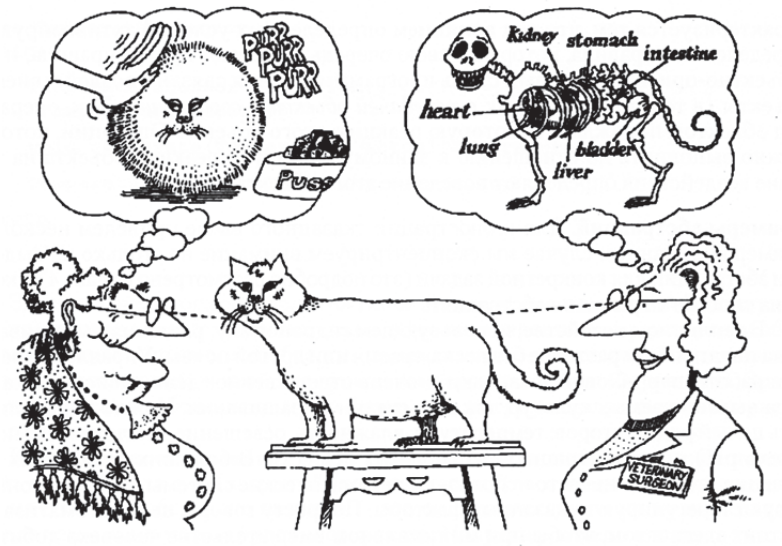

Рис. 3.1. Абстрагирование

Выбор правильного набора абстракций для заданной предметной области представляет собой главную задачу объектно-ориентированного проектирования.

Каждый язык программирования содержит ряд механизмов для реализации данного принципа ООП. Далее рассмотрим их более подробно.

3.2. Классы и объекты

Рассмотрим основные характеристики классов и объектов.

Класс – абстрактное описание свойств и методов для совокупности похожих объектов, представители которой называются экземплярами класса.

Объект – это модель или абстракция реальной сущности в программной системе (экземпляр класса).

Свойство объекта – объявленный в классе параметр, характеризующий объект.

Метод объекта – объявленная в классе процедура, описывающая поведение объекта.

Далее приведен пример описания класса на языке Simula-67 – первом объектно-ориентированном языке программирования.

```
! Объявление класса «Человек»
Class C_Person(PName,age); Text Pname; Integer age;!Свойства
объекта
Begin
    !Методы объекта
     Procedure Show;
     Begin
         ...
     End of Show;
End of C_Person;
```

Для того чтобы создать новый объект данного класса, необходимо выполнить следующие команды:

```
Ref(C_Person) P0; ! Объявление переменной
p0:-New C_Person("J.Dyllan",22); ! Создание экземпляра класса
«Человек»
```

Поскольку объект создается при его появлении в области видимости и с помощью оператора *new*, то в этих случаях будет вызываться функция инициализации экземпляра класса.

Вопрос: *Какими недостатками обладает описанный выше способ объявления класса?*

Основной недостаток, порождающий массу противоречий – это то, что нет возможности определить порядок создания класса и изменить функцию создания класса. В случае, например, если мы хотим создать класс на основе данных, введенных пользователем, или сделать какие-либо преобразования с данными – нам необходимо сначала создать объект класса с любыми значениями параметров и затем производить дополнительные манипуляции. Это неудобно и приводит к усложнению кода и увеличению его объема. То есть возникает *противоречие 3.1*: при увеличении количества способов инициализации класса недопустимо увеличивается сложность разрабатываемой программы.

Задание 3.1. *Представьте себя разработчиком языка программирования. Подумайте и опишите, как можно разрешить представленное противоречие.*

Противоречие 3.1 было разрешено в языках программирования при помощи приемов «принцип заранее подложенной подушки» и «принцип динамичности».

В целом для заполнения полей объекта при его создании и для освобождения полей при его удалении в языках программирования предусмотрены специальные средства, которые состоят из составных функций, называемых конструкторами и деструкторами.

Конструктором называется составная функция класса, вызываемая при создании объекта.

Основные свойства конструктора:

– Класс может иметь несколько конструкторов. Имя каждого конструктора совпадает с именем класса, для которого этот конструктор определен. Конструкторы могут быть переопределены и иметь параметры по умолчанию.

– Конструктор не имеет возвращаемого значения.

– Конструктор может быть определен вне тела класса.

Некоторые языки программирования различают несколько особых типов конструкторов:

– конструктор по умолчанию – конструктор, не принимающий аргументов;

– конструктор с параметрами – конструктор, который принимает в качестве аргументов значения свойств класса.

При удалении объектов используются деструкторы, которые программист может переопределить.

В отличие от конструктора деструктор не принимает параметров.

Деструктором называется составная функция класса, которая вызывается перед разрушением объекта [21]. Это означает, что деструктор вызывается в следующих случаях:

– при выходе из области видимости;

– при выполнении операции delete для объектов, размещенных в динамической памяти;

– непосредственно как составная функция.

Класс может иметь несколько конструкторов, все эти конструкторы имеют одинаковое имя, совпадающее с именем класса. Приведём правила определения деструкторов:

– класс имеет ровно один деструктор;

– имя деструктора совпадает с именем класса с добавленным впереди символом тильды «~»;

– деструктор не имеет аргументов и не имеет возвращаемого значения.

Пример вызова деструктора:

```
~C_Person(void) //указывает деструктор
{
// Операторы деструктора
}
```

Данное решение, в частности, применяется в языке C++. Программист может использовать различные способы задания алгоритмов инициализации объектов. В то же время, если программист не описывает конструктор по умолчанию самостоятельно, компилятор создает его. Функции конструктора, созданные компилятором, называются неявными.

Рассмотрим пример инициализации класса и описания явного конструктора по умолчанию на языке C++ для класса «Человек».

```
//объявление класса «Человек»
class C_Person
{
//описание полей класса
protected: string Name;
           int Age;
//описание методов класса
public:    C_Person ()// конструктор по умолчанию
           {
             cout<<"\nEnter name: "; //запрос информации у
пользователя
              getline(cin,Name);
              cout<<"\nEnter Age: "; //запрос информации у
пользователя
              cin>>Age;

           }
          void Show()
           {
              ...
           }
};
```

Создание экземпляра класса в таком случае выполняется следующими командами:

```
C_Person *P0; //создаем указатель на объект
P0 = new C_Person(); //инициализируем экземпляр класса
```

Конструктор с параметрами в отличие от конструктора по умолчанию принимает аргументы.

Пример:

```
C_Person(string N, int A) //конструктор с параметрами
           {
              Name=N; Age=A;
           }
```

Создание экземпляра класса в таком случае выполняется следующими командами:

```
C_Person *P0; //создаем указатель на объект
P0 = new C_Person("J.Bin",19); //инициализируем экземпляр
класса
```

Рассмотри еще одну особенность C++. По умолчанию при инициализации одного объекта другим C++ выполняет побитовое копирование. Это означает, что точная копия инициализирующего объекта создается в целевом объекте. Хотя в большинстве случаев такой способ инициализации объекта является вполне приемлемым, имеются случаи, когда побитовое копирование не может использоваться. Например, такая ситуация имеет место, когда объект выделяет память при своем создании. Рассмотрим в качестве примера два объекта A и B класса C_Person, выделяющего память при создании объекта. Предположим, что объект A уже существует. Это означает, что объект A уже выделил память. Далее предположим, что A использовался для инициализации объекта B:

```
C_Person B = A;
```

Если в данном случае используется побитовое копирование, то B станет точной копией A. Это означает, что B будет использовать тот же самый участок выделенной памяти, что и A, вместо того, чтобы выделить свой собственный. Ясно, что такая ситуация нежелательна. Например, если класс C_Person включает в себя деструктор, освобождающий память, то тогда одна и та же память будет освобождаться дважды при уничтожении объектов A и B.

Проблема того же типа может возникнуть еще в двух случаях. Первый из них возникает, когда копия объекта создается при передаче в функцию объекта в качестве аргумента. Второй случай возникает, когда временный объект создается функцией, возвращающей объект в качестве своего значения.

Задание 3.2. *Сформулируйте противоречия на основе полученной информации о проблемах копирования объектов в C++.*

Воспользуемся следующим противоречием в качестве рабочего: при увеличении удобства копирования объектов недопустимо ухудшается рациональность использования памяти для хранения объектов (*противоречие 3.2*).

Противоречие 3.2 было разрешено в языке C++ при помощи приема «принцип заранее подложенной подушки».

Для решения подобных проблем язык C++ позволяет создать конструктор копирования, который используется компилятором, когда один объект инициализирует другой. При наличии конструктора копирования побитовое копирование не выполняется. Рассмотрим пример такого конструктора:

```
// Конструктор копирования.
C_Person(C_Person &obj){
      Name = obj.Name;
     Age=obj.Age;
}
```

Копирование объекта выполняется следующим образом:

```
C_Person P1;
C_Person P2 = P1; // Вызов конструктора копирования.
```

Таким образом, мы рассмотрели эволюцию методов создания и удаления объектов (механизм группы «Абстракция») (рис. 3.2).

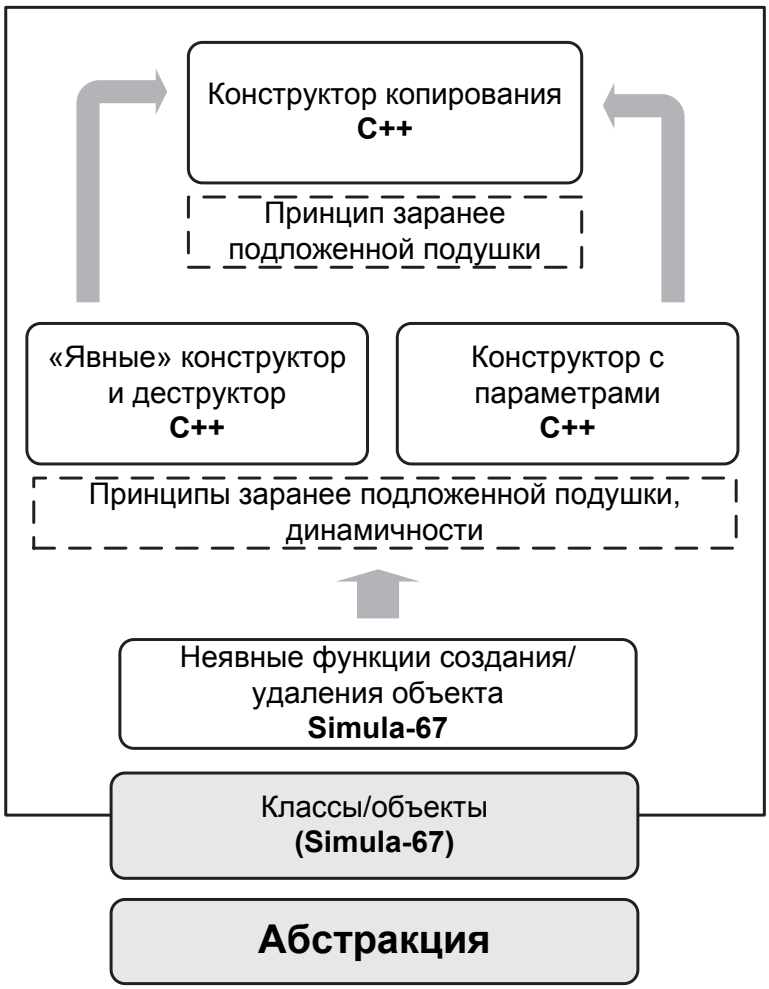

Рис. 3.2. Эволюция средств создания и удаления объектов

Лабораторная работа 1

КЛАССЫ И ОБЪЕКТЫ

Задание 1

На основе полученных знаний необходимо описать класс в соответствии с вариантом и реализовать следующие функции:

1) Инициализация экземпляра класса конструктором по умолчанию с параметрами копирования.

2) Удаление объекта при помощи деструктора.

3) Вывод на экран данных об объекте.

4) Вывод на экран списка объектов.

5) Вывод на экран информации о количестве созданных/ удаленных/ активных объектов класса.

6) Хранение двух нумерованных списков объектов. Каждый из списков должен иметь наименование и описание.

Напишите программу, демонстрирующую работу указанных функций. Проанализируйте полученное решение. Насколько эффективно оно решает поставленную задачу? Насколько оно удобно и трудоемко? Сформулируйте противоречия.

Вариант определяется по табл. 3.1. Студент может также выбрать свой вариант, согласовав его с преподавателем.

Таблица 3.1

Варианты заданий на лабораторную работу 1

Номер варианта	Класс, который необходимо описать
1	Символьный массив
2	Массив целых чисел
3	Массив длинных целых чисел
4	Массив типа float
5	Массив коротких целых чисел
6	Массив типа double

3.3. Статические элементы класса

Использование стандартного функционала классов и объектов недостаточно для решения ряда задач. Экземпляры класса представляют собой переменные пользовательского типа данных и никак не взаимосвязаны между собой. Однако зачастую необходимо решать такие задачи, как хранение информации о созданных/удаленных объектах, хранение переменных, являющихся общими для всех экземпляров класса, на основании которых заполняются поля (свойства) объектов и т.д. Такие переменные можно инициализировать в коде программы, однако, с увеличением количества таких переменных и функций их обработки недопустимо снижается наглядность кода (*противоречие 3.3*).

Данное противоречие было разрешено в C++ при помощи приема «принцип объединения». Чтобы получить область, которая является общей для всех объектов класса, достаточно описать поля этой области с атрибутом *static*. Описанные таким образом поля называются статическими. Более точно, **статическими** называются элементы класса, которые являются общими для всех объектов этого класса.

Статическими могут быть как переменные, так и функции.

Например:

```
class C_Person
{
//описание полей класса
protected: string Name;
           int Age;
//описание методов класса
public:
     static int count; //количество созданных объектов
     static int get_count() { return count; }
     C_Person ()// конструктор по умолчанию
           {//запрос информации у пользователя
             cout<<"\nEnter name: ";
getline(cin,Name);
              cout<<"\nEnter Age: ";
              cin>>Age;
            count++;
           }
};
```

3.4. Контейнерные классы и шаблоны

Рассмотрим ситуацию – необходимо хранить информацию о созданных объектах каждого класса в виде списков. Нет возможности описания списков и множеств.

Для этой цели удобно использовать созданный класс, который будет хранить указатель на первый элемент и массив объектов. В данном случае структура списка будет неизменной для любого вида создаваемых объектов, изменится только тип элементов списка.

Использование стандартного функционала классов и объектов также недостаточно для решения подобной задачи. Следовательно, возникает *противоречие 3.4*: с увеличением эффективности управления массивами объектов недопустимо увеличивается объем разрабатываемого кода.

Противоречие 3.4 создает трудности при модификации и масштабировании программ.

Задание 3.3. *Какими средствами можно разрешить противоречие А? Какими свойствами должен обладать механизм, который бы позволил разрешить противоречие?*

3.4.1. Абстрактные структуры данных

Противоречие 3.4 может быть разрешено при помощи приемов «принцип универсальности» и «принцип матрешки» путем описания класса-структуры, который бы хранил данные о некотором множестве объектов другого класса. Такие классы называются контейнерами. **Контейнерными классами** в общем случае называются классы, в которых хранятся организованные данные.

Абстрактные структуры данных также являются контейнерами.

Один раз разработанный и отлаженный контейнерный класс можно использовать для различных типов данных. Существует ряд абстрактных структур данных [22]:

– **массив** – конечная совокупность однотипных величин. Занимает непрерывную область памяти и предоставляет прямой (произвольный) доступ к элементам по индексу;

– **линейный список** – динамическая структура данных, в которой каждый элемент связан со следующим и, возможно, с предыдущим. Количество элементов в списке может изменяться в процессе работы программы. Также списки различаются по способу добавления/удаления элементов списков;

– **бинарное дерево** – динамическая структура данных, состоящая из узлов, каждый из которых содержит помимо данных не более двух ссылок на различные бинарные поддеревья. На каждый узел имеется ровно одна ссылка. Начальный узел называется корнем дерева. Узел, не имеющий поддеревьев, называется листом. Исходящие узлы называются предками, входящие – потомками. Высота дерева определяется количеством уровней, на которых располагаются его узлы;

– **дерево поиска** – динамическая структура данных, организованная таким образом, что для каждого узла все ключи его левого поддерева меньше ключа этого узла, а все ключи его правого поддерева – ассоциативный массив. В дереве поиска можно найти элемент по ключу, двигаясь от корня и переходя на левое или правое поддерево в зависимости от значения ключа в каждом узле, одинаковые ключи не допускаются;

– **ассоциированный массив** – массив, доступ к элементам которого осуществляется не по номеру, а по ключу (т.е. это таблица, состоящая из пар «ключ-значение»).

Рассмотрим инициализации класса-структуры, который будет хранить список объектов класса C_Person в виде линейного списка.

```
Class ListP
{
     C_Person value;          // Элемент списка
     List * nextElement       // Указатель на следующий элемент
```

```
public:
     void add(C_Person);       // Добавление элемента
     C_Person firstElement();  // Получение первого элемента
};
```

Вопрос: *Какими недостатками обладает описанный выше способ объявления класса-структуры?*

В данном случае мы создаем класс-структуру для конкретного типа объектов C_Person, и объектом данного класса может быть только список объектов класса C_Person. В случае если понадобится хранить в виде структуры данных объекты других классов, придется описывать дополнительный контейнерный класс, что неудобно. Это порождает *противоречие 3.5*: при увеличении количества организованных списков объектов недопустимо увеличивается объем кода.

Задание 3.4*. Какими средствами можно разрешить противоречие 3.5? Какими свойствами должен обладать механизм, который бы позволил разрешить противоречие?*

3.4.2. Контейнеры в языках с динамической типизацией

Противоречие 3.5 может быть разрешено при использовании приемов «Принцип динамичности», «Принцип универсальности» и «Принцип копирования» путем создания механизма описания контейнерных классов, не привязанных к конкретному типу хранимых в нем данных.

В языке Smalltalk можно описать классы-структуры данных один раз и впоследствии использовать его для разных типов объектов, за счет того, что язык обладает динамической типизацией. **Динамическая типизация** – приём, при котором переменная связывается с типом в момент присваивания значения, а не в момент объявления переменной. Таким образом, в различных участках программы одна и та же переменная может принимать значения разных типов. При этом можно обрабатывать итерации без того, чтобы выставлять на обозрение внутреннюю структуру контейнеров [23].

В языке Smalltalk операторы могут быть сгруппированы в конструкцию, называемую block. Она во многом аналогична функции. Подобно функции, блок может иметь список аргументов. Стандартный способ выполнения итерации в языке Smalltalk – это передать блок как аргумент вместе с сообщением структуре, к которой осуществляется доступ:

```
linkDo: aBlock
   " выполнить блок, передать следующему элементу списка"
   aBlock value: value.
   nextLink notNil
   ifTrue: [ nextLink linkDo: aBlock ]
```

Класс списка просто пересылает блок классу элемента. Каждый элемент вызывает блок с использованием своего текущего значения, и затем пересылает блок следующему элементу. В этом заключается принцип копирования. Преимущество данного подхода в том, что контейнеры имеют совершенно общий вид и могут даже содержать разнородные наборы данных различных типов [18]. Однако при этом уменьшается производительность программ. Б. Страуструп утверждает [19]: «Данный подход ведет к сложной программной реализации, средства компиляции и компоновки тоже излишне сложны». Таким образом, возникает еще *противоречие 3.6*: при повышении эффективности повторного использования кода недопустимо увеличивается сложность реализации.

3.4.3. Шаблоны классов

В языках со статической типизацией – связывание переменной с типом в момент объявления статической переменной – также могут быть организованы контейнерные классы статической структуры.

Так, противоречия 3.4, 3.5 и 3.6 были разрешены в языке C++ приемом «Принцип универсальности» совместно с приемом «Принцип наоборот» путем добавления к языку возможности описания шаблонов, при инициализации аргументами выступают типы значений, а не сами значения. В целом **шаблоны** – средство языка, предназначенное для кодирования обобщённых алгоритмов, без привязки к некоторым параметрам. При помощи шаблонов в C++ можно описывать функции, классы и т.д. Рассмотрим пример реализации шаблона списка элементов.

```
template <SomeType> class List
{
     SomeType value;          // Элемент списка
     List * nextElement       // Указатель на следующий элемент
public:
     void add(SomeType);      //Добавление элемента
     SomeType firstElement(); //Получение первого элемента
};
```

В этом примере идентификатор SomeType используется как обозначение типа. Каждый экземпляр класса List содержит значение типа SomeType и указатель на следующий элемент списка. Функция-член add добавляет новый элемент в список. Первый элемент в списке возвращается функцией firstElement.

Для создания класса, описывающего список, например, объектов класса «Человек», необходимо прописать функцию `List <C_Person> ListP`.

Шаблоны классов не могут быть вложены в другие классы. Шаблоны классов могут иметь нетипизированные параметры; значения, указанные для этих параметров, должны быть константами.

3.4.4. Шаблоны функций

Следует отметить, что не только классы могут быть описаны в качестве шаблонов, но и функции. **Шаблон функции** – это обобщенное описание поведения функций, которые могут вызываться для объектов разных типов; другими словами, шаблон функции представляет семейство функций. Шаблон очень похож на обычную функцию, разница только в том, что некоторые элементы этой функции не определены и являются параметризованными [24].

Шаблон дает алгоритм, используемый для автоматической генерации экземпляров функций с различными типами. Программист параметризует все или только некоторые типы в интерфейсе функции (т.е. типы формальных параметров и возвращаемого значения), оставляя ее тело неизменным. Функция хорошо подходит на роль шаблона, если ее реализация остается инвариантной на некотором множестве экземпляров, различающихся типами данных, как в случае min() [25]. Так определяется шаблон функции min() в языке C++:

```
template <class Type>
   Type min( Type a, Type b ) {
      return a < b ? a : b;
}
```

Недостатки имеются и у шаблонов. Реализация шаблонов сильно варьируется в отношении легкости использования и качества получаемого кода в различных компиляторах. Большинство из них не делают ничего, кроме интерпретации шаблонов в виде сложных макросов, так что для каждого нового параметра-типа создается совершенно новое определение класса и полностью независимые тела методов. Это приводит к значительному увеличению размера кода. Тем не менее, поскольку шаблоны освобождают программиста от большого количества дополнительной работы, они пользуются большой популярностью [18].

То есть при повышении производительности труда программиста с использованием механизма описания шаблонов недопустимо увеличивается объем компилируемого кода.

Таким образом, мы рассмотрели первую и вторую итерации ТРИЗ-эволюции механизмов группы «Абстракция» (рис. 3.3).

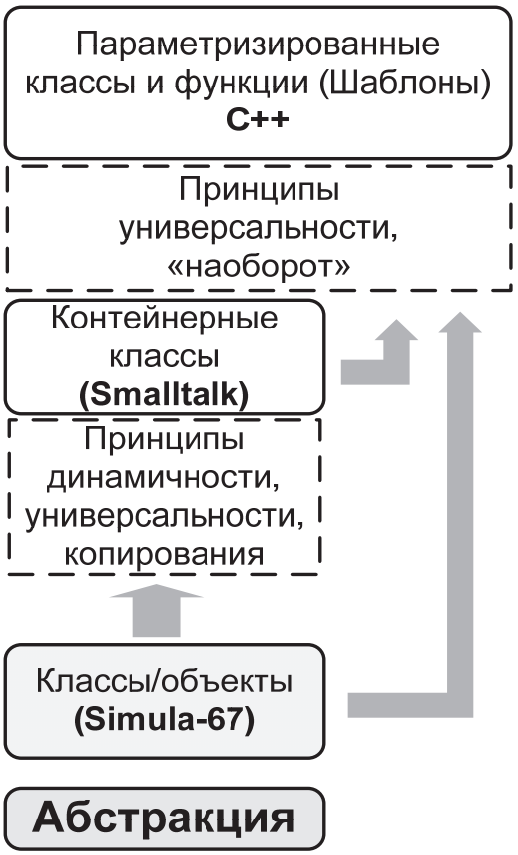

Рис. 3.3. Первая и вторая итерации ТРИЗ-эволюции
механизмов группы «Абстракция»

Задание 2

Используя полученные знания о статических переменных и функциях, контейнерных классах и шаблонах, выполните задания 1.5 и 1.6 с использованием новых механизмов.

Оформите отчет о лабораторной работе, который должен иметь следующую структуру:

1) Постановка задачи (задание 1 на лабораторную работу 1).
2) Ход решения задачи.
3) Анализ полученного решения (выявление недостатков, формулировка противоречий, описание предполагаемого способа разрешения противоречий).
4) Решение исходной задачи средствами, полученными в ходе разрешения противоречий (задание 2 на лабораторную работу 1).

3.5. Наследование как группа механизмов объектно-ориентированного программирования

Наследование является фундаментальной концепцией ООП.

Цель ООП состоит в повторном использовании созданных вами классов, что экономит ваше время и силы. Если вы уже создали некоторый класс, то возможны ситуации, что новому классу нужны многие или даже все особенности уже существующего класса, и необходимо добавить один или несколько элементов данных или функций. В таких случаях ООП позволяет вам строить новый объект, используя характеристики уже существующего объекта. Другими словами, новый объект будет наследовать элементы существующего класса (называемого базовым классом). Когда вы строите новый класс из существующего, этот новый класс часто называется производным классом.

Итак, **наследование** – это механизм языка, позволяющий описать новый класс на основе уже существующего (родительского, базового) класса. Класс-потомок (производный класс) может добавить собственные методы и свойства, а также пользоваться родительскими методами и свойствами; позволяет строить иерархии классов.

3.6. Одиночное наследование

Рассмотрим задачу. Пусть существует некоторое учебное заведение. Необходимо разработать приложение для вывода на экран информации о студентах и сотрудниках учреждения.

Студенты и сотрудники имеют общие параметры: «Имя» и «Возраст», а также уникальные параметры: студенты – «Год обучения», сотрудники – «Размер заработной платы» и «Стаж работы». Опишем три класса: класс «Человек», который будет описывать общие параметры; классы «Студент» и «Сотрудник», которые будут хранить уникальные параметры.

В данном случае удобно использовать наследование от класса «Человек», который будет хранить общие для всех классов параметры.

При этом классы «Студент» и «Сотрудник» будут производными от класса «Человек». Иерархия классов представлена на рис. 3.4.

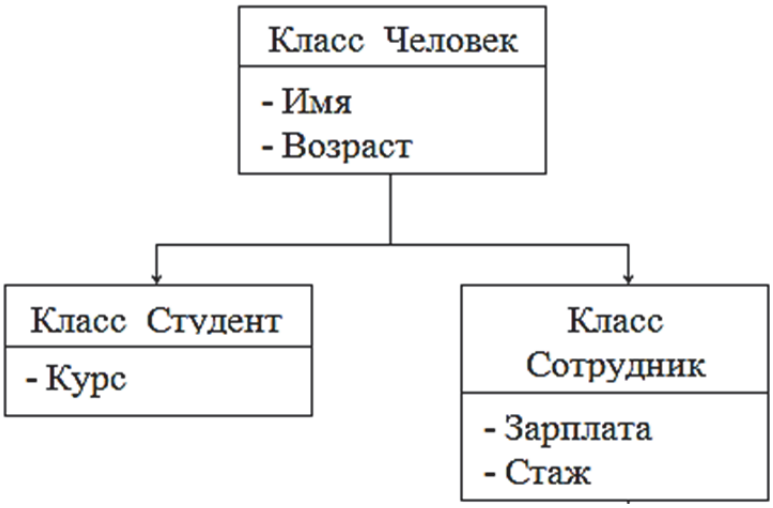

Рис. 3.4. Иерархия наследования классов

Усложним поставленную задачу. Пусть студенты могут являться сотрудниками учебного заведения. Необходимо описать дополнительный класс «Студент-Сотрудник».

Вопрос: *Каким образом организовать иерархию наследования, учитывая возможности одиночного наследования?*

Иерархия производных классов, образованных от класса «Человек» с учетом усложнения задачи, представлена на рис. 3.5.

Задание 3.5. *Определите, какими недостатками обладает данная иерархия наследования. Сформулируйте противоречия.*

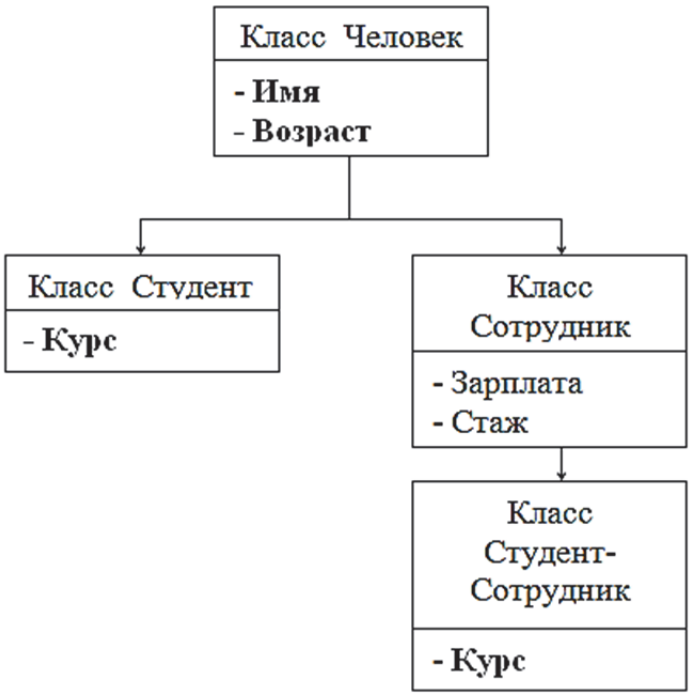

Рис. 3.5. Иерархия производных класса «Человек»

По рис. 3.5 видно, что нерационально инициализировать объект два раза: как экземпляр класса «Студент» и как экземпляр класса «Студент-Сотрудник», ведь в таком случае появится два одинаковых объекта класса «Человек». Кроме того, дублируется свойство «Курс». Возникает *противоречие 3.7*: с увеличением количества возможных классов-родителей недопустимо увеличивается объем дублируемых данных.

Задание 3.6. *Подумайте и опишите, какими средствами можно разрешить указанное противоречие.*

3.7. Множественное наследование

Противоречие 3.7 может быть разрешено путем применения приема «Принцип объединения» при объединении дублируемых параметров в одном из классов и введения возможности объекту наследовать несколько классов.

Таким образом, иерархия подклассов класса «Человек» принимает вид, показанный на рис. 3.6.

Данный механизм наследования сокращает количество дублируемых данных. Инициализировав экземпляр класса «Студент-Сотрудник», свойства классов родителей определяются один раз, и не происходит повторение информации.

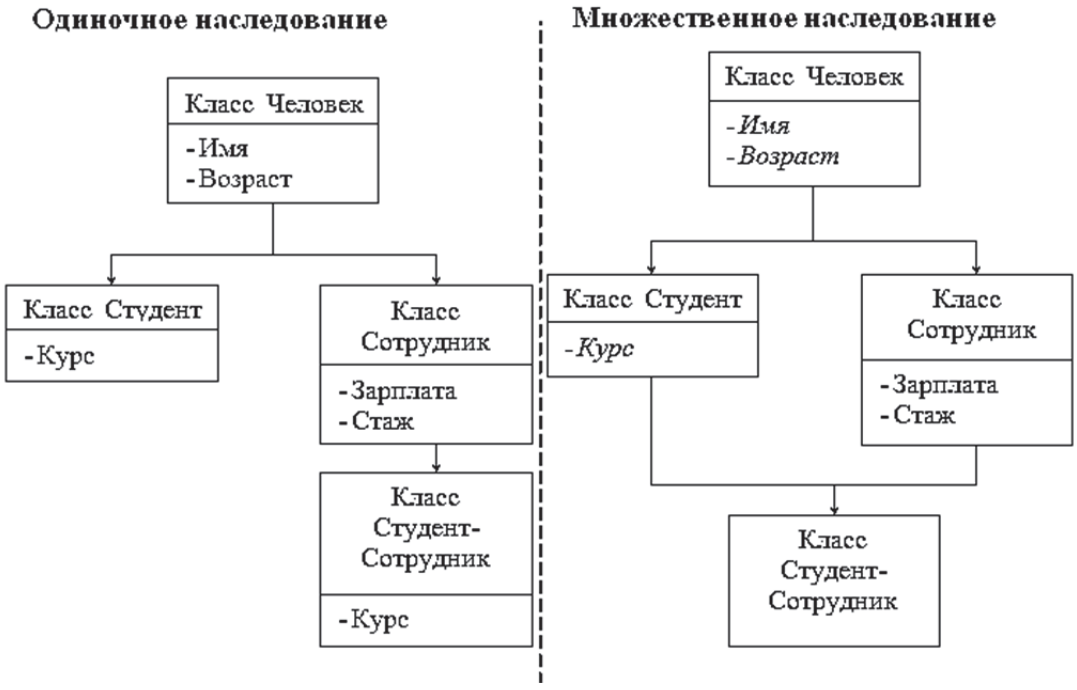

Рис. 3.6. Иерархия подклассов класса «Человек»
при использовании множественного наследования

При использовании одиночного наследования при создании экземпляра класса «Сотрудник-Студент» необходимо будет также создать экземпляр класса «Студент» с идентичными характеристиками. В противном случае, информация о студентах будет неполной. Однако получается, что одна и та же информация будет храниться в двух разных объектах.

Такое решение было использовано в языке C++. Произошла первая итерация ТРИЗ-эволюции группы механизмов ООП «Наследование» (рис. 3.7).

Рис. 3.7. Первая итерация ТРИЗ-эволюции группы механизмов «Наследование»

Далее приведен код программы, решающей указанную задачу.

```
#include <vcl.h>//библиотека визуальных компонентов
#include <conio.h>//для использования функции getch()
#include <iostream.h>//заголовочный файл потокового вво-
да/вывода
//---------------------------------------------------
//объявление класса «Человек»
class C_Person
{
//описание полей класса
protected: string Name;
           int Age;
//описание методов класса
public:    C_Person ()// конструктор по умолчанию
           {
              cout<<"\nEnter name: ";
              getline(cin,Name);
              cout<<"\nEnter Age: ";
              cin>>Age;
           }
           C_Person(string N, int A) //конструктор с параметрами
           {
              Name=N; Age=A;
           }
// вывод информации об объекте на экран
//функция является виртуальной и может быть переопределена
        void Show()
           {
              cout<<"\n"<<Name<<", "<<Age;
           }
};
//---------------------------------------------------
```

```
//Объявление класса «Студент».
//Класс является производным от класса «Человек»
class C_Student : virtual public C_Person
{
protected: int Year;

public:    C_Student() // конструктор по умолчанию
           {
              cout<<"\nEnter Year: ";
              cin>>Year;
           }
           //конструктор с параметрами
           C_Student(string N, int A, int Y): C_Person(N,A)
           {
              Year=Y;
           }
           void Show() // Вывод информации об объекте на экран
           {
              // Вызов метода Show() из базового класса
              C_Person::Show();
              cout<<", "<<Year;
           }
};
//-------------------------------------------------
//Объявление класса «Студент».
//Класс является производным от класса «Человек»
class C_Employee : virtual public C_Person
{
protected: int Salary;
           int Experience;

public:    C_Employee()// конструктор по умолчанию
           {
              cout<< "\nEnter Salary: ";
              cin>>Salary;
              cout<< "\nEnter Experience: ";
              cin>>Experience;
           }
           //конструктор с параметрами
           C_Employee(string N, int A, int S, int E):
C_Person(N,A)
           {
              Salary=S;
              Experience=E;
           }
           void Show()// Вывод информации об объекте на экран
           {
              C_Person::Show();
              cout<<", "<<Experience<<", "<<Salary;
           }
```

```
};
//-------------------------------------------------
//Объявление класса «Студент-Сотрудник».
//Класс является производным от классов «Студент» и «Сотруд-
ник»
class C_StudentEmployee : public C_Student, public C_Employee
{
// конструктор по умолчанию
public:    C_StudentEmployee(): C_Student(), C_Employee()
           {}

           //конструктор с параметрами
           C_StudentEmployee(string N, int A, int S, int E,
int Y): C_Student(N,A,Y), C_Employee(N,A,S,E)
           {}
           void Show() // вывод информации об объекте на экран
           {
             C_Student::Show();
              cout<<", "<<Experience<<", "<<Salary;
           }
};
//-------------------------------------------------
void main ()
{
 cout << "***Demonstration program***";
//Создание объектов
 C_Person *P0,*P1,*P21,*P22;
        P0 = new C_Student("J.Bin",19,2);
        P1 = new C_Employee("D.Grant",39,6,2000);
        P21 = new C_StudentEmployee("A.Clark",22,300,1,5);
//Вывод информации об объектах на экран
 cout<<"\nNew working student...";
        P22 = new C_StudentEmployee();
 cout<<"\n\nStudents:\n";
        P0->Show();
 cout<<"\n\nWorking students:\n";
        P21->Show();
        P22->Show();
 cout<<"\n\nEmployees:\n";
        P1->Show();
//Ожидание нажатия клавиши
 getch();
}
```

Лабораторная работа 2

НАСЛЕДОВАНИЕ

Задание 1

На основе полученных знаний необходимо разработать иерархию классов в соответствии с вариантом и следующими критериями:

1) Иерархия классов должна демонстрировать возможности одиночного и множественного наследования.

2) Количество классов в иерархии не должно быть меньше 4.

3) Каждый класс должен обладать свойствами и методами.

4) В каждом классе должен быть реализован метод Show(), который выводит на экран полную информацию об объекте класса.

5) В каждом классе должны быть реализованы не менее 2 конструкторов и деструктор.

6) В каждом классе должны быть описаны не менее 3 методов, реализующих различные операции с объектами класса.

Напишите программу, демонстрирующую работу указанных функций. Проанализируйте полученное решение. Насколько эффективно оно решает поставленную задачу? Насколько оно удобно и трудоемко? Сформулируйте противоречия.

Вариант определяется по табл.3.2. Студент может также выбрать свой вариант, согласовав его с преподавателем.

Таблица 3.2

Варианты заданий на лабораторную работу 2

Номер варианта	Иерархия, которую необходимо описать
1	Музыкальные инструменты
2	Домашние животные
3	Телевизионные программы
4	Растения
5	Комплектующие ПК
6	Геометрические фигуры

Недостатки множественного наследования

Использование технологии множественного наследования порождает проблемы, связанные с наглядностью кода, неоднозначностью выбора из одноименных методов родительских классов, и влечет за собой появление ошибок. Например, если вызвать метод Show() для объекта класса «Студент-Сотрудник», и в классе не окажется такого метода, но в классе «Человек» будет присутствовать метод Show(), по-своему переопределен-

ный как в классе «Сотрудник», так и в классе «Студент», то какой из методов должен быть вызван (рис. 3.8)?

Таким образом, возникает *противоречие 3.8*: с увеличением количества классов-предков при множественном наследовании недопустимо увеличивается неоднозначность выбора одноименных методов.

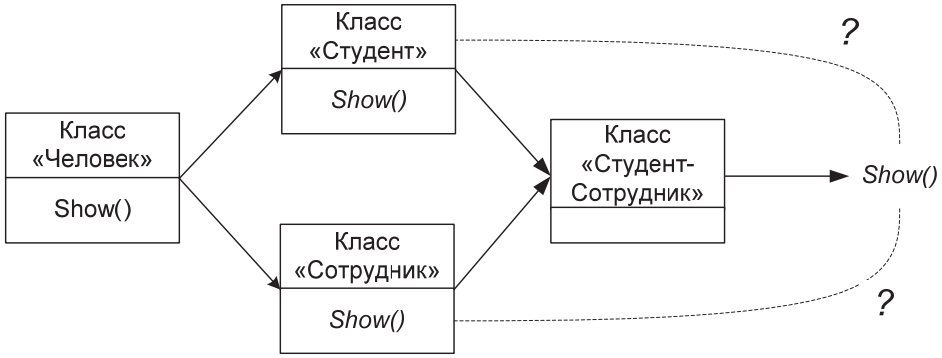

Рис. 3.8. Проблема множественного наследования в C++

Задание 3.7*. Подумайте и опишите, какими средствами можно разрешить указанное противоречие.*

Интересное решение было использовано при создании языка Eiffel. Противоречие 3.8 было разрешено при помощи приема «принцип предварительного действия». Если необходимо использовать один из одноименных методов двух родительских классов, то в описании класса используется оператор select. Рассмотрим пример использования данного оператора.

```
     -- Описание класса «Студент-сотрудник»
class
     STUDENT_EMPLOYEE
inherit
     -- Наследование от класса «Студент»
     STUDENT
          redefine
               make
          end
     -- Наследование от класса «Сотрудник»
     EMPLOYEE
          redefine
               make
          end
          -- Указание на вызов метода
          select
               show
          end
...
end
```

В данном примере класс «Студент-сотрудник» является наследником классов «Студент» и «Сотрудник», каждый из которых содержит метод «Show». Для того чтобы при вызове данного метода выбрать объектом класса «Студент-сотрудник», используется оператор select. При наследовании от класса «Сотрудник» данный оператор явно указывает, что метод «Show» должен быть вызван из этого класса.

Если необходимо использовать несколько одноименных методов родительских классов, то используется функция rename для того, чтобы переименовать в классе методы родительских классов. При этом вызов метода из программы возможен только по новому имени. Рассмотрим далее пример использования данной функции.

```
     -- Описание класса «Студент-сотрудник»
class
     STUDENT_EMPLOYEE
inherit
     -- Наследование от класса «Студент»
     STUDENT
          redefine
               make
          end
          rename
               show as show_like_student
          end
     -- Наследование от класса «Сотрудник»
     EMPLOYEE
          redefine
               make
          end
     -- Переименование родительского метода
          rename
               show as show_like_employee
          end
...
end
```

Данное решение повышает надежность, а также легкость создания программ.

3.8. Абстрактные классы и интерфейсы

Противоречие 3.8, возникающее при использовании множественного наследования, можно разрешить приемом «принцип частичного или избыточного действия» путем создания механизма описания так называемых спецификаций – некоторых сущностей, которые хранят описания свойств и методов, которые должны быть реализованы в производном классе.

3.8.1. Абстрактные классы

Так, например, в C++ появилась возможность описания абстрактных классов [24]. **Абстрактный класс** – это класс, на основе которого не могут создаваться объекты. Абстрактный класс может иметь определённые методы, а также свойства. Абстрактный метод не реализуется для класса, в котором объявлен, однако должен быть реализован для его неабстрактных потомков.

Для того чтобы превратить класс в абстрактный, перед его именем надо указать модификатор abstract.

Рассмотрим пример абстрактного класса C_Person и производного от него C_Student:

```
abstract class C_Person
{
//описание полей класса
string Name;
     int Age;
//описание методов класса
     void Show()
           {
              cout<< Name <<", "<< Age;
           }
};
class C_Student: C_Person {}
```

При этом запрещено создавать объекты абстрактного класса, поэтому методы класса C_Person будут доступны только после инициализации объекта. Также методы абстрактного класса могут быть переопределены в классе C_Student. Методы абстрактного класса также могут быть абстрактными.

Абстрактный метод – метод, который не имеет реализации в данном классе. Объявление абстрактного метода выглядит следующим образом:

```
abstract void AbstractMethod();
```

Для объектов того класса, где метод описан, метод использовать нельзя, но если унаследовать класс и в потомках переопределить метод, задав там его описание, то для объектов классов потомков метод можно будет вызывать (и работать будут описанные в классах потомков реализации).

3.8.2. Интерфейсы

В языке Java [26] была добавлена возможность описания интерфейсов (рис. 3.9). Данное решение позволяет получить многие преимущества множественного наследования, не реализуя его в полном объёме.

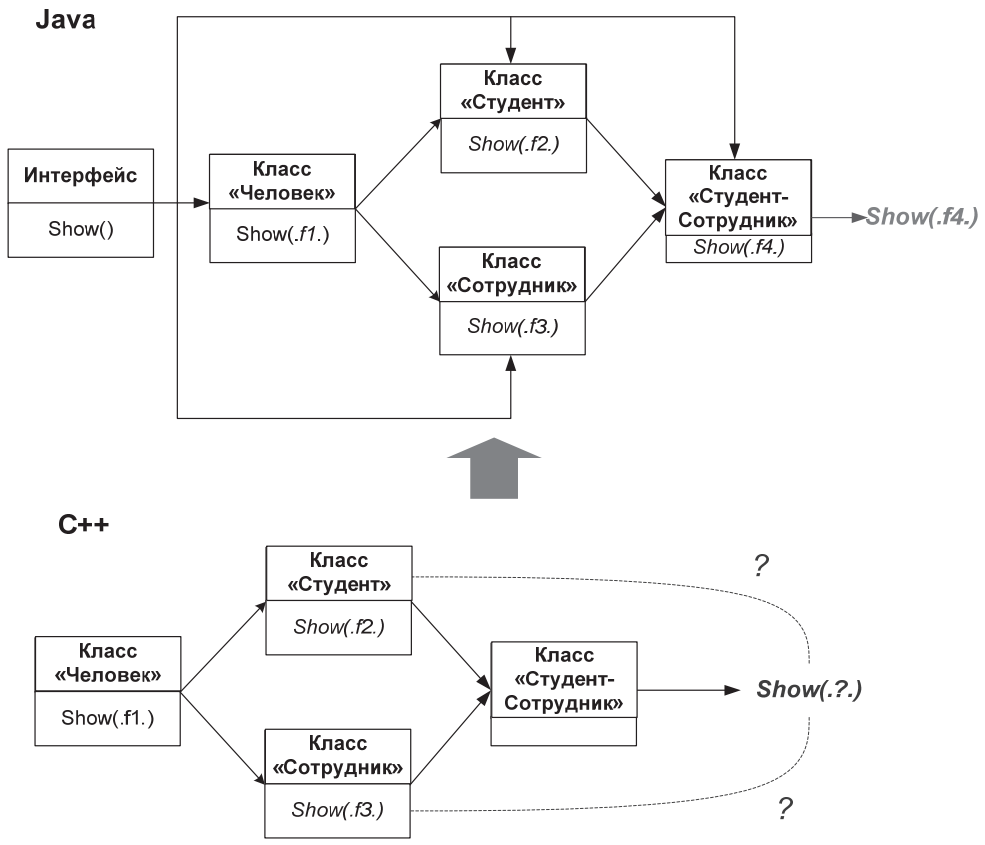

Рис. 3.9. Разрешение проблемы множественного наследования в C++ механизмом наследования интерфейсов на языке Java

Интерфейсы – это конструкции, определяющие границу взаимодействия между классами или компонентами, специфицируя определенную абстракцию, которую осуществляет реализующая сторона.

Интерфейсы Java могут описывать только абстрактные публичные методы и статические константные свойства. То есть так же, как и на основе абстрактных классов, на основе интерфейсов нельзя порождать объекты.

Кроме того, один интерфейс может быть наследником другого интерфейса.

Классы могут реализовывать интерфейсы (т. е. получать от интерфейса список методов и описывать реализацию каждого из них), притом, что особенно важно, один класс может реализовывать сразу несколько интерфейсов.

Перед описанием интерфейса указывается ключевое слово interface. Когда класс реализует интерфейс, то после его имени указывается ключевое слово implements и далее через запятую перечисляются имена тех интерфейсов, методы которых будут полностью описаны в классе. Например:

```
interface C_Person {
    public void Show();
}
class C_Student implements C_Person {
    public void Show () {
        System.out.println("Я – студент!");
    }
}
class C_Employee implements C_Person {
    public void Show () {
        System.out.println("Я – работник!");
    }
}
```

Поскольку все свойства интерфейса должны быть константными и статическими, а все методы общедоступными, то соответствующие модификаторы перед свойствами и методами разрешается не указывать.

Java не поддерживает множественное наследование классов. Это объясняется тем, что такое наследование порождает ряд проблем.

Вместо множественного наследования классов в Java введено множественное наследование интерфейсов, которое частично решает проблемы. Также можно сказать, что интерфейсы являются новым видом абстракции, то есть являются механизмом группы «Абстракция», а наследование интерфейсов является механизмом группы «Наследование». Таким образом, произошла следующая итерация ТРИЗ-эволюции (рис. 3.10).

Задание 3.8. *Механизм наследования интерфейсов не является идеальным. Какими недостатками он обладает? Сформулируйте противоречия.*

В языке Java использование интерфейсов грозит тем, что придётся раз за разом переписывать реализацию методов для каждого класса, использующего интерфейс, что может быть достаточно трудоемким процессом. То есть возникает *противоречие 3.9*: с повышением эффективности использования механизма наследования путем описания интерфейсов недопустимо увеличивается объем разрабатываемого кода.

Кроме того, при работе с интерфейсами отсутствует возможность уточнения имени интерфейса при переопределении одноименных методов разных интерфейсов. Компилятор предлагает только один вариант – сделать общую реализацию двух разных методов, что не всегда корректно.

Возникает *противоречие 3.10*: при повышении количества наследуемых интерфейсов, специфицирующих одноименные методы, недопустимо снижается эффективность использования интерфейсов.

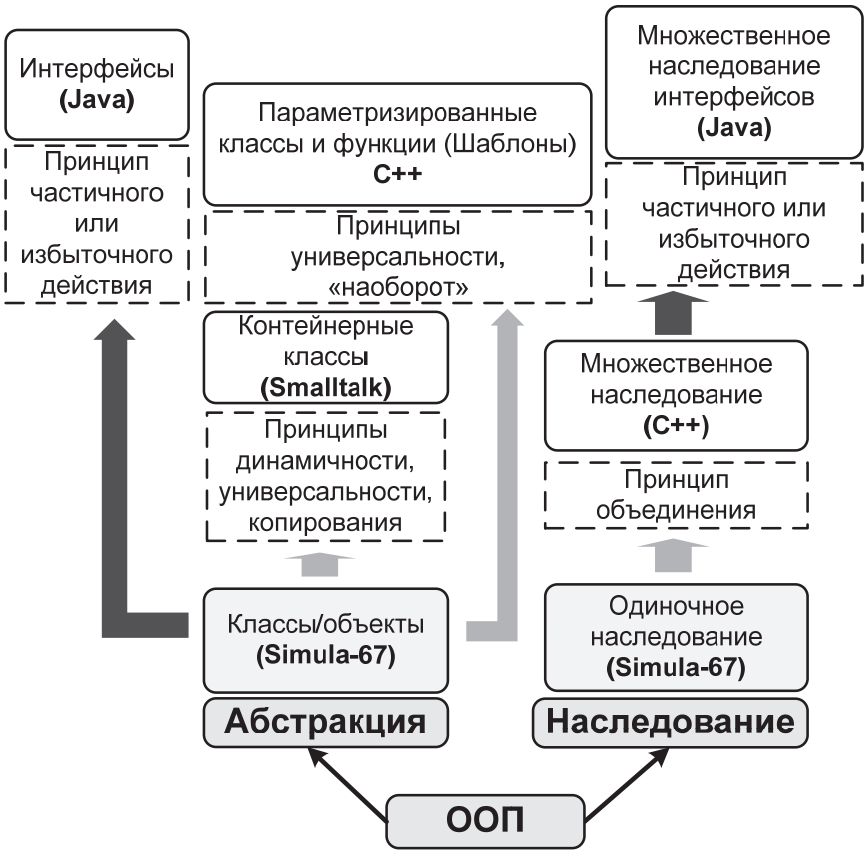

Рис. 3.10. Третья итерация ТРИЗ-эволюции группы механизмов ООП «Абстракция» и вторая итерация ТРИЗ-эволюции группы механизмов ООП «Наследование»

Задание 3.9. *Подумайте и опишите, какими средствами можно разрешить указанные противоречия.*

3.9. Роли (типажи)

В скриптовом языке Perl 6 было разрешено противоречие 3.9 языка Java – при использовании приема «Принцип универсальности» введен новый механизм – Роли (типажи) [27]. В роли можно промоделировать как поведение интерфейса, указав необходимые для этой роли методы (спецификацию), так и предоставить реализацию метода внутри роли, что позволяет многократно использовать один раз описанный код.

Роли синтаксически похожи на классы и объявляются с ключевым словом role. Как и классы, роли могут содержать методы и атрибуты (переменные класса), но основное назначение ролей – быть базовыми для других классов.

В минимальном варианте, когда роль содержит только объявления методов, роли являются интерфейсами, такими же, как в Java и других языках.

Методы, объявленные без тела (используется синтаксис с многоточием: method name {...}), должны быть реализованы в производном классе. Класс может унаследовать множество ролей. Пример использования роли:

```
role C_Person
{
    has $.Name;
    method GetName
    {
            return $.Name;
    }
}
class C_Student does C_Person
{
}
my $student = C_Student.new(Name => 'J.Dyllan');
say $student.GetName()
```

Как правило, в языках с использованием ролей (типажей) класс определяется только с атрибутами и параметрами, связанными с классом, а методы наследуются из ролей.

Также можно сказать, что роли являются новым видом абстракции, то есть являются механизмом группы «Абстракция», а наследование ролей является механизмом группы «Наследование». Таким образом, произошла следующая итерация ТРИЗ-эволюции (рис. 3.11).

Задание 2

Используя полученные знания о механизмах наследования, выполните задание 1 с использованием языка, механизмы которого, на Ваш взгляд, являются наиболее подходящими для решения задачи.

Оформите отчет о лабораторной работе, который должен иметь следующую структуру:

1) Постановка задачи (задание 1 на лабораторную работу 2).
2) Ход решения задачи.
3) Анализ полученного решения (выявление недостатков, формулировка противоречий, описание предполагаемого способа разрешения противоречий).
4) Решение исходной задачи механизмами, изученными в ходе разрешения противоречий (задание 2 на лабораторную работу 2).

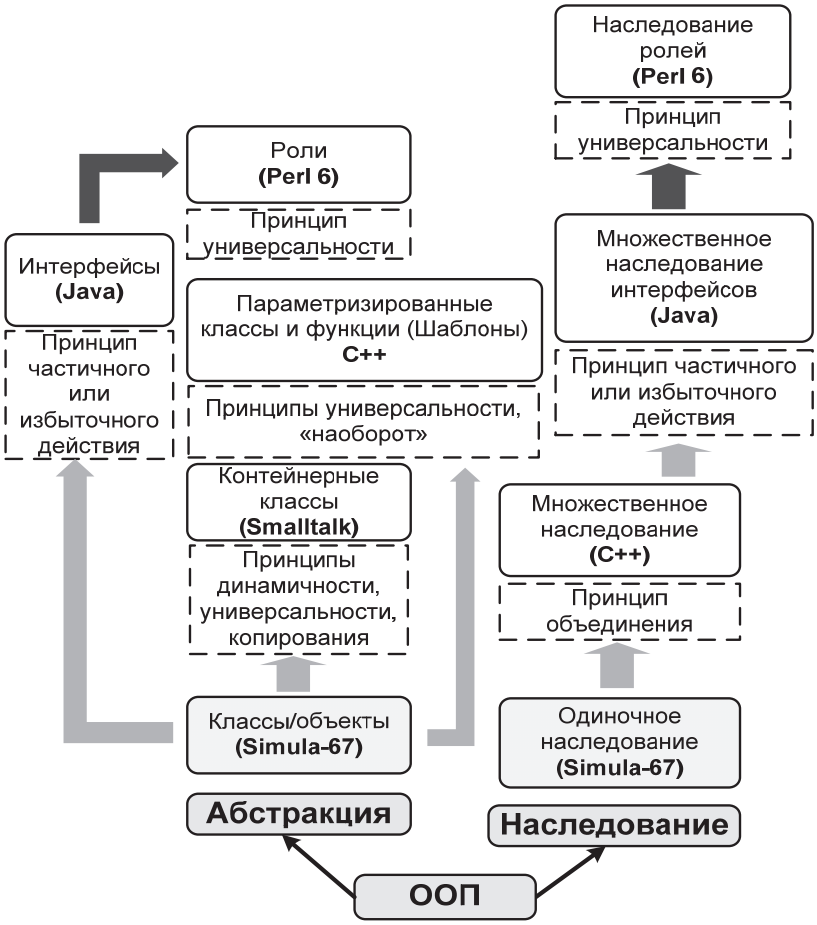

Рис. 3.11. Четвертая итерация ТРИЗ-эволюции группы механизмов ООП «Абстракция» и третья итерация ТРИЗ-эволюции группы механизмов ООП «Наследование»

4. ИНКАПСУЛЯЦИЯ И ПОЛИМОРФИЗМ

4.1. Инкапсуляция как группа механизмов объектно-ориентированного программирования

Инкапсуляция – это механизм ООП, который объединяет данные и методы, манипулирующие этими данными, и защищает их от внешнего вмешательства или неправильного использования [10]. Когда методы и данные объединяются таким способом, создается объект.

Применение инкапсуляции защищает данные, принадлежащие объекту, от возможных ошибок, которые могут возникнуть при прямом доступе к этим данным. Кроме того, применение этого принципа очень часто помогает локализовать возможные ошибки в коде программы, что упрощает процесс поиска и исправления ошибок.

Можно сказать, что инкапсуляция подразумевает под собой сокрытие данных, что позволяет защитить эти данные.

Основные принципы инкапсуляции:

1) переменные состояния объекта скрыты от внешнего мира;

2) изменение состояния объекта (его переменных) возможно только с помощью его методов.

Эти принципы позволяют защитить переменные состояния объекта от неправильного их использования и существенно ограничивают возможность введения объекта в недопустимое состояние и/или несанкционированное разрушение этого объекта.

Хорошим примером применения принципа инкапсуляции являются команды доступа к файлам. Обычно доступ к данным на диске можно осуществить только через специальные функции, то есть отсутствует прямой доступ к данным, размещенным на диске. Таким образом, данные, размещенные на диске, можно рассматривать скрытыми от прямого вмешательства. Доступ к ним можно получить с помощью специальных функций, которые по своей роли схожи с методами объектов.

Структура класса с учетом инкапсуляции упрощенно включает следующие блоки:

1) функции доступа к параметрам класса (интерфейс) – функции ввода/вывода, управления экземпляром класса;

2) методы, параметры класса, отвечающие за внутреннюю реализацию класса – это могут быть функции промежуточного подсчета значений, функции отслеживания состояний и т.д.

4.2. Механизмы инкапсуляции в языках программирования

В первом объектно-ориентированном языке Simula-67 инкапсуляция была слабовыраженным механизмом. У программиста не было средств управления инкапсуляцией. То есть, возникло *противоречие 4.1*: при улучшении качества работы с классами и объектами недопустимо снижается количество выразительных средств языка для его обеспечения.

Противоречие 4.1 было разрешено в языке C++ приемом «принцип посредника» путем введения операторов, которые бы позволили разделять реализацию класса на блоки.

Тело класса в C++ разбивается на три основные части, соответствующие трем атрибутам: private, protected, public [28].

```
class имя
{
          private:
          …
          protected:
          …
          public:
          …
};
```

Атрибут **private** имеют члены класса, доступные только для составных и дружественных функций этого класса. Эти члены класса называются **закрытыми**. Свойства и методы с модификатором «private» видны только внутри того класса, в котором они описаны. Ни в его наследниках, ни в объектах этого и наследующих классов приватные свойства и методы доступны не будут. При попытках подобного обращения программисту будет выдаваться сообщение об отсутствии данных свойств и методов.

Атрибут **protected** имеют члены класса, доступные для составных и дружественных функций классов, которые являются производными от этого класса или совпадают с ним. Эти члены класса называются з**ащищенными**. Как и приватные, защищенные конструкции не видны «снаружи» класса. Отличие проявляется при попытке расширить класс и его модифицировать. Внутри класса protected-конструкции видны и могут быть использованы.

Атрибут **public** имеют члены класса, обращение к которым осуществляется как к полям структуры. Эти члены называются **открытыми**.

Рассмотрим пример объявления класса.

```
class C_Person
     {
          char Name;
     public:
          double get_Name() {return Name;}
     };
```

В данном случае элемент класса Name по умолчанию является закрытым, и обращение к нему, как к открытому члену, приведет к ошибке. Этот элемент можно будет читать с помощью функции get_Name():

```
void main()
     {
     C_Person P1;
     char val;
     val=P1.Name;   //ошибка!
     val = P1.get_Name();     //верно
     }
```

Задание 4.1. *Как Вы думаете, какими недостатками обладает данный механизм? Опишите противоречия.*

Пусть существуют классы А, В и С. Класс А является базовым, а В и С – производными. В классе А прописан некоторый метод, который должен быть доступен классу А, но недоступен классу Б. Каким образом лучше поступить в данном случае? Есть два способа:

1) скрыть реализацию метода при помощи модификатора private от обоих классов и переписать метод в классе А, что снижает эффективность повторного использования кода в программе;

2) описать метод при помощи модификатора protected, тем самым предоставив доступ к методу всем производным классам. В данном случае надежность программы снижается.

Таким образом, возникает *противоречие 4.2*: при повышении надежности программы недопустимо снижается эффективность повторного использования кода.

Задание 4.2. *Как Вы думаете, какими способами можно разрешить указанное противоречие.*

Противоречие 4.2 может быть разрешено при помощи приема «Принцип динамичности» путем создания более гибкого механизма инкапсуляции, который бы позволил явно прописывать наследников класса при описании модификаторов доступа к свойствам и методам.

Такой механизм был сознан в языке Eiffel [29]. В языке разработчик класса также должен определить, какая часть класса будет свободно доступной (публичной), а какая – будет скрытой. Отличие состоит в том, что в Eiffel можно гибко управлять видимостью компонентов класса. Например:

```
class
     A
feature
     a is ... end
     b is ... end
feature {B}
     c is ... end
feature {C,D}
     d is ... end
feature {NONE}
     e is ... end
end
```

Методы *a* и *b* будут доступны всем, метод *c* – только классу *B*, метод *d* – только классам *C* и *D*, метод *e* – только классу *A* и его наследникам.

При этом если класс A делает метод *f* видимым классу *B*, то говорят, что класс *A* экспортирует *f* классу *B*.

Eiffel также позволяет в производном классе изменить описанные в базовом классе параметры экспорта компонентов. Производный класс посредством секции export может полностью изменить картину экспорта:

```
class
  X
inherit
  A
    export
      {NONE}a -- a уже никому не доступна.
      {B,C} b -- b уже доступна и B, и C.
```

```
      {ANY} c -- c уже доступна всем.
    end
...
end
```

При множественном наследовании может возникнуть необходимость скрыть от клиентов все методы, унаследованные от какого-то базового класса. В этом случае используется ключевое слово *all*:

```
class
     X
inherit
     A
     B
          export
          -- Никому не будут доступны унаследованные методы B.
          {NONE} all
          end
...
end
```

Таким образом, произошла эволюция группы механизмов ООП «Инкапсуляция» (рис. 4.1).

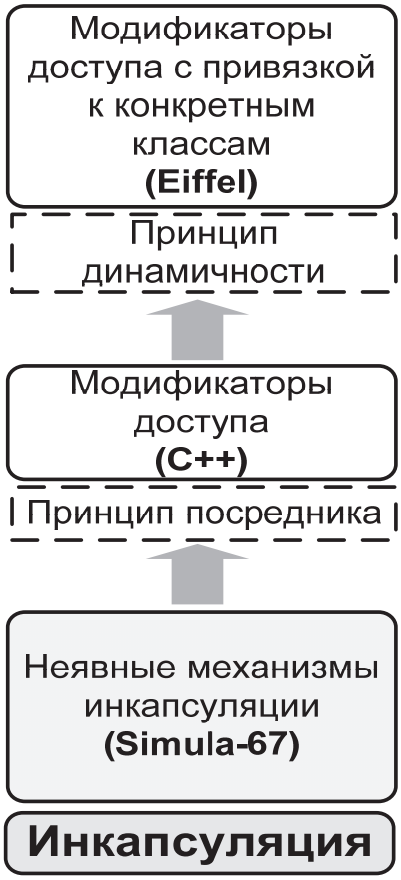

Рис. 4.1. ТРИЗ-эволюция группы механизмов ООП «Инкапсуляция»

4.3. Полиморфизм как группа механизмов объектно-ориентированного программирования

Полиморфизм позволяет одно и то же имя использовать для решения двух или более схожих, но технически разных задач [10].

Целью полиморфизма является использование одного имени для задания общих для класса действий. Выполнение каждого конкретного действия будет определяться типом данных. Тип данных, который используется при вызове функции, определяет, какая конкретная версия функции действительно выполняется. В более общем смысле, концепцией полиморфизма является идея «один интерфейс, множество методов». Это означает, что можно создать общий интерфейс для группы близких по смыслу действий.

Преимуществом полиморфизма является то, что он помогает снижать сложность программ, разрешая использование того же интерфейса для задания единого класса действий. Выбор же конкретного действия, в зависимости от ситуации, возлагается на компилятор.

Полиморфизм может применяться также и к операторам. Фактически во всех языках программирования ограниченно применяется полиморфизм, например, в арифметических операторах. Так, в Си, символ + используется для складывания целых, длинных целых, символьных переменных и чисел с плавающей точкой. В этом случае компилятор автоматически определяет, какой тип арифметики требуется.

Ключевым в понимании полиморфизма является то, что он позволяет манипулировать объектами различной степени сложности путём создания общего для них стандартного интерфейса для реализации похожих действий.

Simula-67 поддерживает полиморфизм: если класс B производный от класса A, то присваивание a1 :- b1 корректно для a1 типа A и b1 типа B. Если во время выполнения объекты имеют правильное отношение соответствия, то источник является потомком цели. Если соответствия нет, то результатом будет ошибка во время выполнения, а не специальная величина, обнаруживаемая и обрабатываемая ПО (как при попытке присваивания).

В Simula-67 при наследовании можно переопределять функции родительского класса и затем производить динамическое связывание.

Если в базовом классе определена составная функция, которая должна различным образом выполняться для объектов различных производных классов, то она в этих производных классах должна быть определена заново. Такая функция называется **переопределенной**. Рассмотрим пример.

```
!Тело главной функции
Begin
    ! Объявление переменных (экземпляров классов)
    Ref(C_Student) P0;
```

```
    Ref(C_Employee) P1;
!Объявление класса «Человек», входные параметры – Имя, Возраст
Class C_Person(PName,age); Text Pname; Integer age;
Begin
   ! Метод класса – вывод данных об экземпляре класса на экран
    Procedure Show;
    Begin
        OutText("  ");OutImage;
        OutText(Pname);
        OutFIX(Age,0,6);
    End of Show;
End of C_Person;
!Объявление класса «Студент», входной параметр – Год обучения
!Класс наследуется от класса «Человек»
C_Person Class C_Student(Grade); Integer Grade;
Begin
   ! Метод класса – вывод данных об экземпляре класса на экран
    Procedure Show;
    Begin
        OutFIX(Grade,0,6);
    End of Show;
End of C_Student;
!Создание экземпляра класса «Студент»
    p0:-New C_Student("J.Dyllan",22,4);
        !Вызов родительского метода «Show()»
        P0 Qua C_Person.Show;
        !Вызов собственного метода «Show()»
        P0.Show;
!Завершение программы
End of programm;
```

В данном примере класс C_Student является наследником класса C_Person и переопределяет метод Show() родительского класса. Можно при вызове задать динамическое связывание через конструкцию qua:

```
P0 Qua C_Person.Show;
```

Конечно, теряется автоматическая адаптация операции к ее целевому объекту. Однако можно получить желаемое поведение динамического связывания (его можно считать изобретением Simula-67), объявляя полиморфные подпрограммы.

Задание 4.3. *Проанализируйте приведенный программный код. Какими недостатками он обладает с точки зрения реализации полиморфизма? Сформулируйте противоречия.*

В Simula-67 для выполнения однотипных операций с различными типами данных используются функции с разными названиями:

```
OutText(Pname);  !Вывод на экран строки
OutFIX(Age,0,6); !Вывод на экран числа
```

То есть с увеличением количества типов данных недопустимо увеличивается количество функций, обозначающих одно и то же действие (*противоречие 4.3*).

При вызове методов класса происходит статическое связывание, то есть если в классе не переопределен родительский метод, его все равно придется вызывать командой `P0 Qua C_Person.Show`. Это неудобно и снижает читабельность кода. Возникает *противоречие 4.4*: при увеличении эффективности использования наследования методов из базовых классов недопустимо снижается наглядность кода.

Задание 4.4. *Каким способом можно разрешить данные противоречия?*

Лабораторная работа 3

ПОЛИМОРФИЗМ

Задание 1

Классы, созданные при выполнении лабораторной работы 2, дополнить методами, реализующими операции сравнения и сложения. Алгоритм работы операций сравнения и сложения необходимо разработать самостоятельно. Продемонстрировать работу программы, иллюстрируя действия пользователя соответствующими сообщениями.

Проанализируйте полученное решение. Насколько эффективно оно решает поставленную задачу? Насколько оно удобно и трудоемко? Сформулируйте противоречия.

4.4. Перегрузка операторов

Противоречие 4.3 может быть разрешено приемом «Принцип универсальности» путем добавления возможности одновременного существования в одной области видимости нескольких различных вариантов применения оператора, имеющих одно и то же имя, но различающихся типами параметров, к которым они применяются.

Такой подход был применен в языке Smalltalk. Так, например, вывод на экран происходит при использовании метода display объекта Transcript:

```
Transcript display:  Name
Transcript display:  Age
```

В C++ данный механизм называется **перегрузкой операторов**.

Любая операция, определенная в C++, может быть перегружена для созданного класса. Это делается с помощью функций специального вида, называемых функциями-операциями (операторными функциями) [25]. Общий вид такой функции:

```
[возвращаемый тип] operator #(список параметров){тело функции}
```

Вместо знака # ставится знак перегружаемой операции. Например:

```
Some_Class operator+(const Some_Class&);     // сложение
Some_Class operator-(const Some_Class&);     // вычитание
```

Перегрузка операций позволяет определить для классов значения любых операций, исключая “.”, ”::”, ”.*”, ”?:”, sizeof [24]. Перегрузка операторов позволяет не только переопределять существующие операторы, но и вводить новые, например: «\=\».

Рассмотрим пример перегрузки операции сложения для класса Point:

```
class Point {
          double X. y:
public:
          //. . .
          Point Point::operator +(Point& p) {
               return Point(x + p.x. y + p.y):
}
```

Несколько отличается перегрузка операций инкремента и декремента. Операция инкремента имеет две формы: префиксную и постфиксную. Для первой формы сначала изменяется состояние объекта в соответствии с данной операцией, а затем он (объект) используется в том или ином выражении. Для второй формы объект используется в том состоянии, которое у него было до начала операции, а потом уже его состояние изменяется.

Чтобы компилятор смог различить эти две формы операции инкремента, для них используются разные сигнатуры, например:

```
Point & operator ++():  // префиксный инкремент
Point operator ++(int); // постфиксный инкремент
```

Таким образом, произошла первая итерация ТРИЗ-эволюции группы механизмов ООП «Полиморфизм» (рис. 4.2).

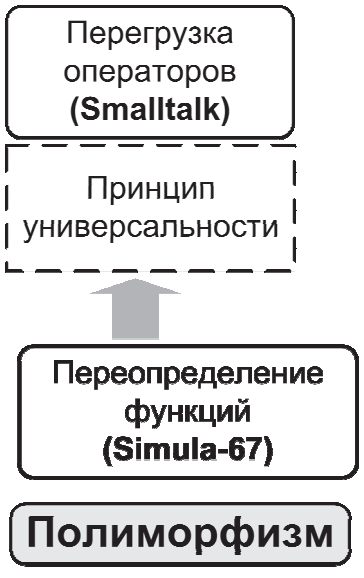

Рис. 4.2. Первая итерация ТРИЗ-эволюции группы механизмов ООП «Полиморфизм»

4.5. Виртуальные функции

Противоречие 4.4 может быть разрешено при помощи приема «принцип предварительного действия». Предположим, что задан массив объектов базового класса. Если его элементы являются объектами производных классов, то функции базового класса не могут быть переопределены. Такое решение используется в языке C++. Рассмотрим пример**:**

```
class C_Person
{
//описание полей класса
protected: string Name;
           int Age;
//описание методов класса
public:
        // вывод информации об объекте на экран
        void Show()
           {
              cout<<"\n"<<Name<<", "<<Age;
           }
};
class C_Student : virtual public C_Person
{
protected: int Year;

public:
           void Show() // Вывод информации об объекте на экран
           {
              // Вызов метода Show() из базового класса
              C_Person::Show();
              cout<<", "<<Year;
           }
};
void main()
     {
     clrscr(); // Очистка экрана
     // Создаём объекты
     C_Person *P0 = (C_Person *)new C_Student;
     P0 -> show();  // Выводим сообщения
     getch();  // Ожидание нажатия клавиши
     }
```

В результате работы программы командой P0 -> show() будет вызвана функция базового класса. Для того чтобы решить проблему переопределения функций в производных классах, объекты которых заданы с помощью указателей на объекты базовых классов, применяются виртуальные функции. Они определяются в базовом классе следующим образом:

```
virtual [тип возвращаемого значения]  [имя]([параметры])
```

Виртуальные составные функции позволяют выбирать члены класса с одним и тем же именем через указатель функции в зависимости от типа указателя.

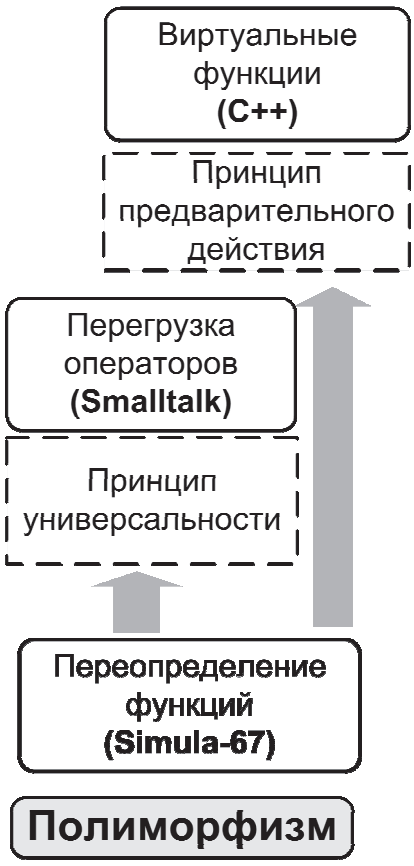

Рис. 4.3. Вторая итерация ТРИЗ-эволюции группы механизмов ООП «Полиморфизм»

В частности, если в нашем примере в базовом классе (C_Person) указать `virtual void show()`, а остальной текст оставить без изменения, то программа вызовет метод Show() из класса C_Student.

Таким образом, функция, определенная как виртуальная в базовом классе и переопределенная с таким же списком аргументов и типом возвращаемого значения в производном классе, становится виртуальной для объектов производного класса. Если она не переопределена в производном классе, то при ее вызове для объектов производного класса будет вызываться соответствующая функция из ближайшего по иерархии базового класса. Виртуальные функции не могут быть статическими.

Таким образом, произошла первая итерация ТРИЗ-эволюции группы механизмов ООП «Полиморфизм» (рис. 4.3).

Задание 2

Выполните задание 1, оптимизировав программный код, используя механизмы перегрузки операций и виртуальных функций.

Оформите отчет о лабораторной работе, который должен иметь следующую структуру:

1) Постановка задачи (задание 1 на лабораторную работу 3).
2) Ход решения задачи.
3) Анализ полученного решения (выявление недостатков, формулировка противоречий, описание предполагаемого способа разрешения противоречий).
4) Решение исходной задачи механизмами, изученными в ходе разрешения противоречий (задание 2 на лабораторную работу 3).

5. ЭВОЛЮЦИЯ ПРОЧИХ МЕХАНИЗМОВ В ОБЪЕКТНО-ОРИЕНТИРОВАННЫХ ЯЗЫКАХ ПРОГРАММИРОВАНИЯ

5.1. Синтаксис

От языка к языку механизмы синтаксиса менялись и на данный момент по-разному используются в языках программирования. У каждого из способов есть свои достоинства и недостатки, многие из которых достаточно субъективны. Рассмотрим, как происходила эволюция механизмов синтаксиса в объектно-ориентированных языках программирования.

Рассмотрим пример реализации консольного приложения на языке Simula-67.

Постановка задачи: Пусть существует некоторое учебное заведение. Необходимо разработать приложение для вывода на экран информации о студентах и сотрудниках учреждения.

Формализация задачи: Студенты и сотрудники имеют общие параметры: «Имя» и «Возраст», а также уникальные параметры: студенты – «Год обучения», сотрудники – «Размер заработной платы» и «Стаж работы». Опишем три класса: класс «Человек», который будет описывать общие параметры; классы «Студент» и «Сотрудник», которые будут хранить уникальные параметры. Далее представлен код получившейся программы.

```
!Тело главной функции
Begin
    ! Объявление переменных (экземпляров классов)
    Ref(C_Student) P0;
    Ref(C_Employee) P1;
!Объявление класса «Человек», входные параметры – Имя, Возраст
Class C_Person(PName,age); Text Pname; Integer age;
Begin
    ! Метод класса – вывод данных об экземпляре класса на
экран
    Procedure Show;
    Begin
        OutText("  ");OutImage;
        OutText(Pname);
        OutFIX(Age,0,6);
    End of Show;
End of C_Person;
!Объявление класса «Студент», входной параметр – Год обучения
!Класс наследуется от класса «Человек»
C_Person Class C_Student(Grade); Integer Grade;
Begin
    !Метод класса – вывод данных об экземпляре класса на экран
    Procedure Show;
```

```
    Begin
        OutFIX(Grade,0,6);
    End of Show;
End of C_Student;
!Объявление класса «Сотрудник», входной параметр – Размер
!заработной платы (считаем целым числом), Стаж работы
!Класс наследуется от класса «Человек»
C_Person Class C_Employee(Salary,Experience);
Integer Salary;Integer Experience;
Begin
    ! Метод класса – вывод данных об экземпляре класса на
экран
    Procedure Show;
    Begin
        OutFIX(Salary,0,6);
        OutFIX(Experience,0,6);
    End of Show;
End of C_Employee;
    !Создание экземпляра класса «Студент»
    p0:-New C_Student("J.Dyllan",22,4);
    !Создание экземпляра класса «Сотрудник»
    P1:-New C_Employee("D.Grant",39,6,2000);
        !Вывод на экран текстовой информации
        OutText("Employees:");OutImage;
        !Вызов родительского метода «Show()»
        P1 Qua C_Person.Show;
        !Вызов собственного метода «Show()»
        P1.Show;
            OutText("Students:");OutImage;
            P0 Qua C_Person.Show;
            P0.Show;
!Завершение программы
End of programm;
```

Задание 5.1. *Проанализируйте полученное решение с точки зрения использованного синтаксиса. Какие есть недостатки? Сформулируйте противоречия.*

Во-первых, из-за сложности синтаксических конструкций наглядность кода ухудшается, т.к. его трудно воспринимать и читать. Таким образом, при увеличении объема кода недопустимо снижается его наглядность за счет сложности синтаксических конструкций (*противоречие 5.1*).

Это также снижает наглядность кода и увеличивает время на компиляцию программы. Другими словами: с ростом количества однотипных операций недопустимо увеличивается количество вызываемых функций (*противоречие 5.2*).

Во-вторых, по тексту программы видно, что иногда для выполнения простых операций требуется вызов нескольких функций:

```
OutText("Employees:"); OutImage;
```

Задание 5.2. *Подумайте, как можно разрешить указанные противоречия.*

Противоречие 5.1 можно разрешить при использовании приема «Принцип местного качества» путем замены синтаксических конструкций на условные знаки. Такое решение было использовано, например, в языке C++. На рис. 5.1 приведено сравнение описание класса «Человек» на языке C++ и на языке Simula-67.

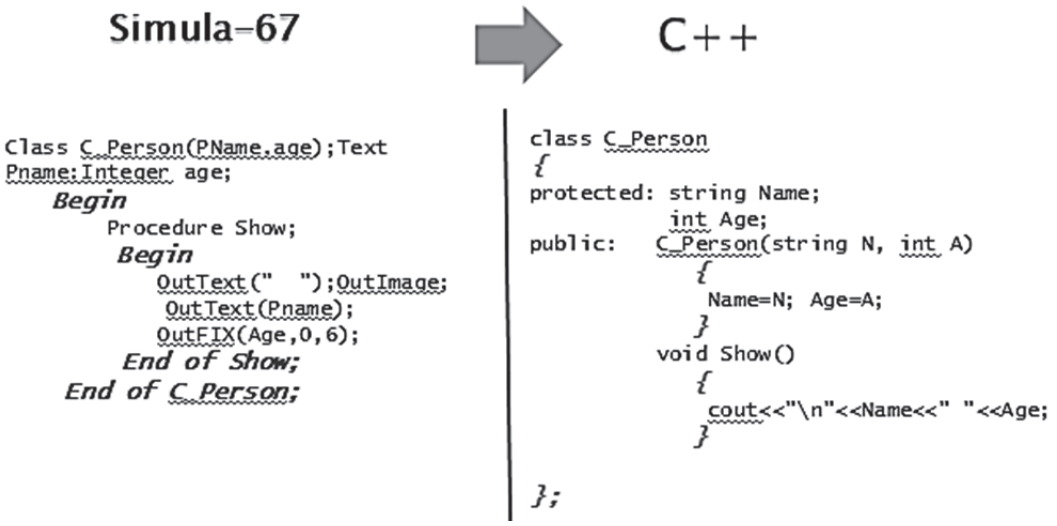

Рис. 5.1. Описание класса «Человек» на языках Simula-67, C++

Как показано на рис. 5.1, конструкции, обозначающие начало и конец процедуры, заменены на фигурные скобки. Подобное решение увеличивает читабельность языка, так как нельзя спутать данные конструкции с переменными или процедурами в коде программы, и визуально код легче воспринимаем.

Противоречие 5.1 также может быть разрешено в языке при использовании приема «Принцип самообслуживания», путем разработки механизмов автоматической документируемости языка. Такой механизм был создан в языке Eiffel. Среда разработки Eiffel является самодокументируемой. При написании кода автоматически расставляется табуляция, метки для написания комментариев и оператор завершения процедуры (рис. 5.2). Таким образом, читабельность и легкость создания программ повысились.

Противоречие 5.2 может быть разрешено при использовании приема «Принцип объединения» путем объединения функций, направленных на выполнение одной операции, в единую функцию. Такое решение было использовано в языке C++.

Например, вывод на экран сообщения «Employees:» имеет вид: `cout<<"Employees:";`

Та же операция на языке Simula-67 имеет вид:

```
OutText("Employees:"); OutImage;
```

```
feature
    make is
        1 - --Initialization without parameters
    do
 2 - <-> Precursor{EMPLOYEE}
        io.put_string ("Enter Year:")
        io.put_new_line
        io.read_integer_32
        Year:=io.last_integer_32
3 - end
```

Рис. 5.2. Пример оформления кода на языке Eiffel

Очевидно, что вывод на экран сообщения при помощи C++ более практичен, нет необходимости запоминать множество операторов для одного и того же действия. При этом увеличивается легкость создания программ и снижается вероятность появления ошибок.

Таким образом, произошла первая итерация ТРИЗ-эволюции группы механизмов «Синтаксис» (рис. 5.3).

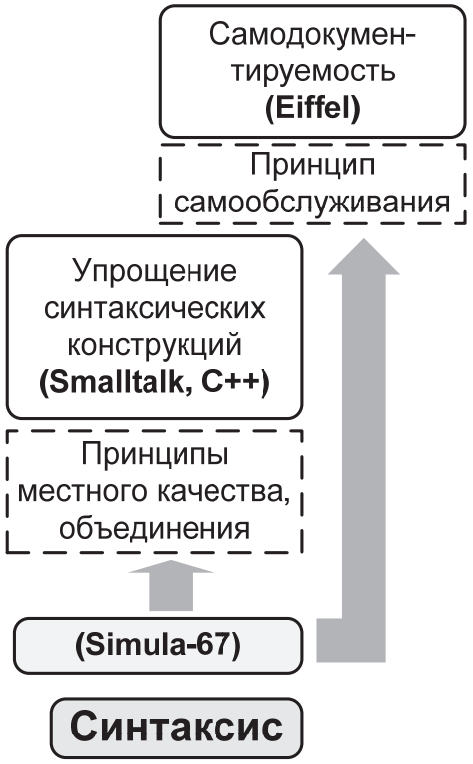

Рис. 5.3. Первая итерация ТРИЗ-эволюции
группы механизмов «Синтаксис»

Следует упомянуть, что вариант разрешения противоречия путем замены синтаксических конструкций на условные знаки, который был применен в языке C++, не всегда рассматривается разработчиками как идеальный. При большом объеме кода и отсутствии документируемости сложно ориентироваться в коде, снижается его наглядность. То есть появляется *противоречие 5.3*: с увеличением объема кода недопустимо ухудшается читабельность языка из-за отсутствия идентификации конструкций начала/завершения процедур.

Задание 5.3. *Подумайте, как можно разрешить указанное противоречие.*

При использовании приемов «Принцип вынесения» и «Принцип однородности» можно упразднить синтаксические конструкции для обозначения начала/конца процедуры и ввести разграничение процедур путем выравнивания кода по имени процедуры. Такое решение было использовано в языке Python:

```
#Объявление класса "Человек"
class C_Person(object):
     # Объявление свойств класса при помощи кортежа
     __slots__ = ("Pname", "Age")

     # Метод возврата строкового представления объекта
     def __repr__(self):
          return ", ".join(map(lambda key: "%s: %s" % (key,
                          getattr(self, key)), self.__slots__))

     # Конструктор класса, переключается в зависимости
     # от переданных параметров. Если они есть, устанавливает
     # их у класса, а если нет, то запрашивает их у пользователя
     def __init__(self, **kwargs):
          if len(kwargs.keys()):
               for key in kwargs:
                    if key in self.__slots__:
                         self.__setattr__(key, kwargs[key])
          else:
               for attr in self.__slots__:
                    self.__setattr__(attr,raw_input("%s:"attr))
```

Таким образом, произошла вторая итерация ТРИЗ-эволюции механизма «Синтаксис» (рис. 5.4).

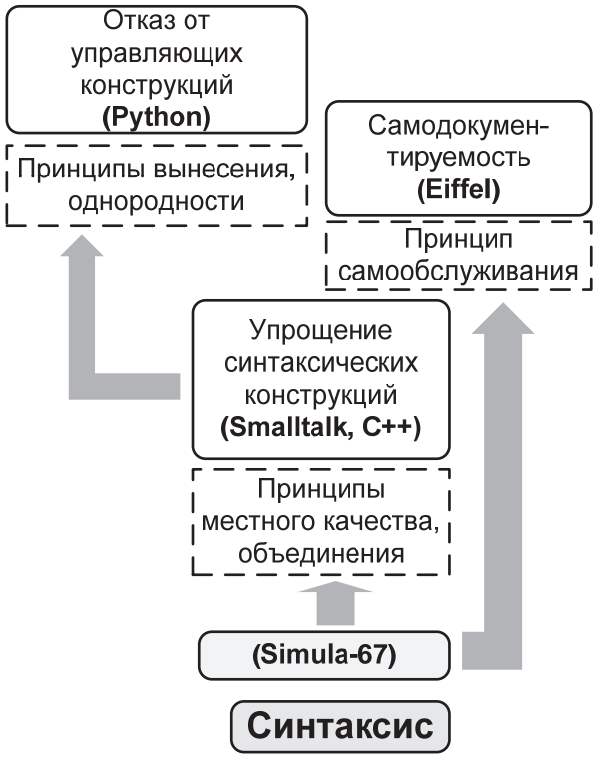

Рис. 5.4. Вторая итерация ТРИЗ-эволюции группы механизмов «Синтаксис»

5.2. Структура программ

Способы организации программного кода также имеют разные вариации и появились в результате эволюции механизма «Структура программы».

В первом объектно-ориентированном языке Simula-67 программный код хранился в одном файле, что было достаточно неудобно. При разработке крупномасштабных систем с такой организацией кода его наглядность и читабельность снижаются. То есть возникает *противоречие 5.4*: с увеличением сложности программируемой системы недопустимо уменьшается читабельность языка.

***Задание 5.4**. Подумайте, как можно разрешить указанное противоречие.*

Противоречие 5.4 может быть разрешено приемом «Принцип матрешки» совместно с приемом «Принцип дробления». Так, например, в программе на языке Smalltalk все является объектом. Классы хранятся в дереве классов и связаны отношением наследования. Любой класс является объектом класса более высшего порядка. Описания классов хранятся отдельно от кода главной рабочей функции программы, которая представляет собой набор сообщений, посылаемых объектам. Это повышает читабельность языка и легкость создания программ.

Еще один вариант разрешения противоречия – прием «Принцип вынесения». Часть процедур может быть вынесена во внешние библиотеки, так что программист может сам решить, где их использовать. Такое решение было реализовано в C++.

В данном решении многие процедуры, в том числе, процедуры управления памятью, выносятся в библиотеки, которые программист по желанию может использовать. Благодаря этому производительность приложений, написанных на языке C++, стала выше производительности приложений, написанных на языке Simula-67. Кроме того, у программиста появляется возможность самостоятельно создавать библиотеки классов и процедур, тем самым значительно облегчая код программы и повышая его читабельность и надежность.

Таким образом, произошла ТРИЗ-эволюция группы механизмов «Структура программы» (рис. 5.5).

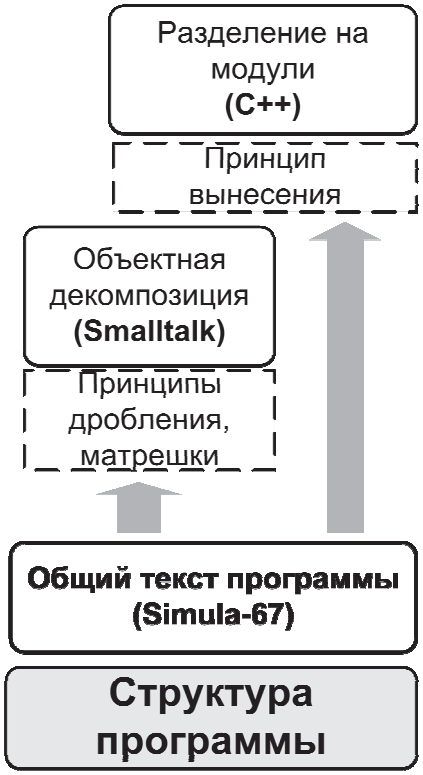

Рис. 5.5. ТРИЗ-эволюция группы механизмов «Структура программы»

5.3. Средства отладки приложений

Далее рассмотрим, как изменялись средства отладки приложений в объектно-ориентированных языках программирования.

Рассмотрим задачу: определить класс «Человек» с параметрами «Имя» и «Возраст» и методом ввода информации об объекте. Определение класса на языке Simula-67 выглядит следующим образом:

```
!Объявление класса C_Person
Class C_Person(PName,age); Text Pname; Integer age;
Begin
    !Процедура, запрашивающая ввод информации пользователем
    Procedure Enter_Info;
    Begin
        OutText("  ");OutImage;
        !Запрос ввода имени
        OutText("Enter Name: ");OutImage;
        Pname:-blanks(8);
        !Чтение данных, введенных с клавиатуры,
```

```
        ! запись значения свойства «Имя»
        Pname:=InText(8);
        !Запрос ввода возраста
        OutText("Enter Age: ");OutImage;
        !Чтение данных, введенных с клавиатуры,
        !запись значения свойства «Возраст»
        age:=InInt;
    End of Enter_Info;
End of C_Person;
```

Заведомо совершим ошибку. Изменим в текстах программ параметр класса C_Person PName на Name и попробуем отладить программу. На рис. 5.6 представлен пример работы с компилятором Simula-67.

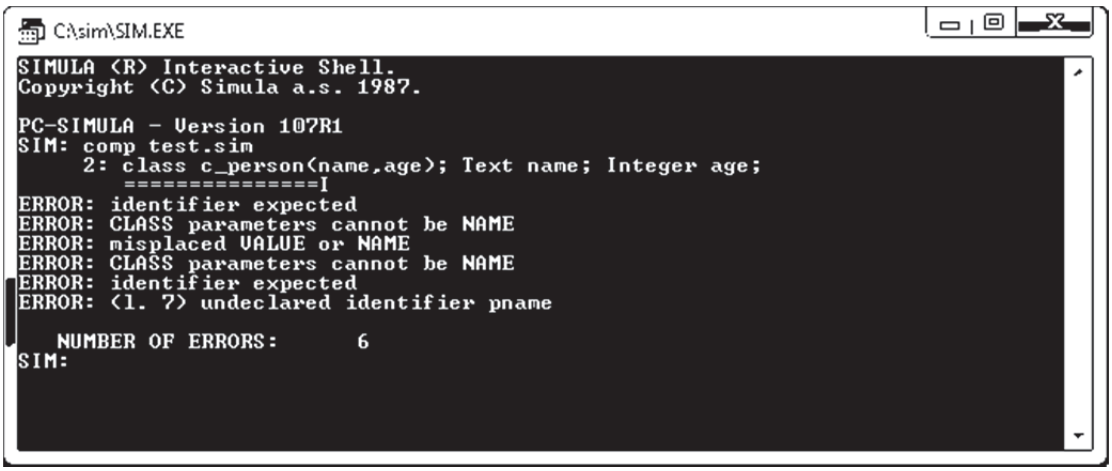

Рис. 5.6. Отладка программы на языке Simula-67

Компилятор Simula-67 отображает строку, в которой возможно совершена ошибка и список ошибок, которые она порождает в ходе компиляции программы. Для того чтобы устранить ошибку, необходимо открыть код программы, который расположен в файле «TEST.sim», внести изменения, сохранить их и попробовать скомпилировать программу заново. После того как ошибки, выявленные компилятором, будут устранены, можно будет запустить программу. Если же в программе есть ошибки, не выявленные компилятором, то необходимо будет каждый раз перекомпилировать программу после внесения изменений.

При отладке крупных приложений на выполнение этого алгоритма тратится много времени, т.е. с увеличением сложности разрабатываемого ПО недопустимо увеличивается время на отладку программы (*противоречие 5.5*).

В программировании существует аксиома [30]: на 1000 строк кода в среднем приходится неизменное количество ошибок. Компилятор не может выявить всех ошибок на этапе написания программы. Естественно, что некоторые виды ошибок проявляются только во время выполнения программ и часто такие ошибки приводят к аварийному завершению программы. Возникает *противоречие 5.6*: с увеличением объема кода недопустимо снижается надежность программы.

Задание 5.5. *Подумайте, как можно разрешить указанные противоречия.*

Противоречие 5.5 может быть разрешено путем перехода в надсистему, которой является среда разработки, обладающая пользовательским интерфейсом. Визуализация значительно упрощает отладку программ. Такое решение было использовано в языке Smalltalk.

Далее представлено определение класса «Человек» на языке Smalltalk.

```
"Инициализация класса C_Person"
Object subclass: #C_Person
     "Объявление переменных, доступных классу"
     instanceVariableNames: 'PName Age'
     classVariableNames: ''
     poolDictionaries: ''
     classInstanceVariableNames: ''
"Объявление методов, используемых классом"
"для ответа на сообщения"
!C_Person methods!
"Метод ввода информации о параметрах объекта"
Enter_Info: Str and:Val
"Устанавливает значение PName объекта-получателя равным Str"
PName:=Str.
"Устанавливает значение Age объекта-получателя равным Val"
Age:=Val.
"Метод вывода информации об объекте"
Show
"Возвращает значение PName получателя"
^PName
```

Заведомо совершим ошибку. На рис. 5.7 представлен пример работы со средой разработки Dolphin Smalltalk.

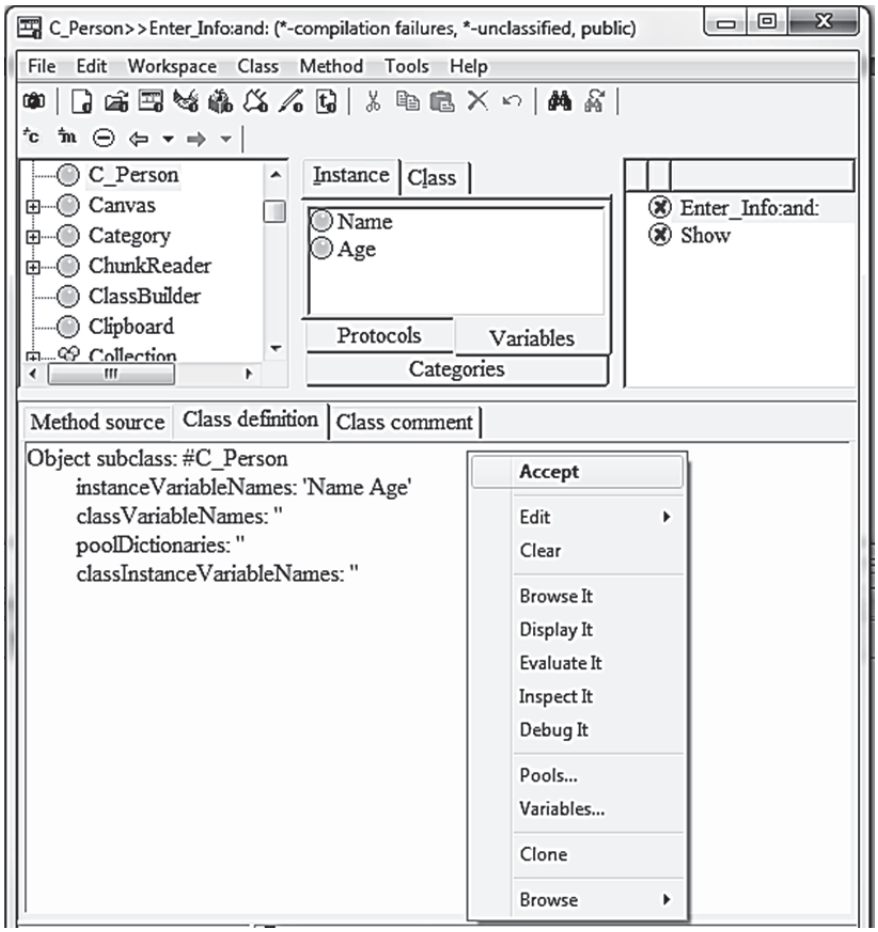

Рис. 5.7. Отладка программы в среде Dolphin Smalltalk

На рис. 5.7 изображено окно работы с конструктором класса. В среде Smalltalk есть возможность без компиляции всей программы проверить правильность написанного кода, всего или отдельной строки. Для этого необходимо правой кнопкой мыши щелкнуть по строке кода и выбрать пункт «Accept». В случае возникновения ошибки программа сообщит об этом. После проверки строки с ошибкой методы класса сменили значок с  на  . Это означает, что внесенные изменения повлекли за собой ошибку. Попробуем провести отладку метода класса (рис. 5.8).

При нажатии кнопки «Accept» была выделена строка, в которой обнаружена ошибка, и в нижней части окна приведено прозрачное описание данной ошибки. Подобная визуализация значительно упрощает отладку программы.

Кроме того, написание программы на языке Smalltalk заключается в последовательном изменении состояния ее объектов/классов, в любой момент времени в систему можно внести практически любое изменение, отсутствует привычный цикл разработки edit-compile-run-debug. Программист не тратит время на низкоуровневые детали, вся разработка состоит только лишь из стадии выполнения/инспектирования.

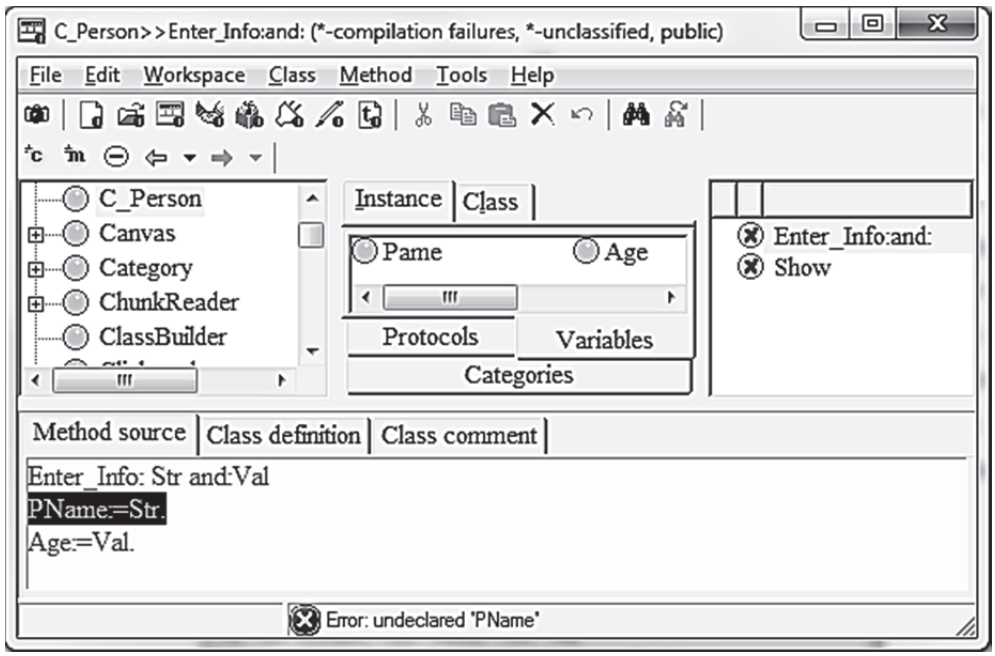

Рис. 5.8. Отладка метода Enter_Info

Среда диктует общую структуру приложения. Она описывает, как распределены обязанности между различными компонентами и как последние должны взаимодействовать между собой. Преимущество среды состоит в том, что разработчику нового приложения нужно всего лишь сконцентрироваться на специфике решаемой в данный момент проблемы. Более ранние разработки в той же среде не требуют длительной подготовки к повторному использованию, а их код не нуждается в переписывании. Подобный подход значительно повышает легкость создания программ.

Однако даже при использовании среды разработки противоречие 5.6 не разрешается.

Противоречие 5.6 может быть разрешено при помощи приема «Принцип самообслуживания» путем добавления к языку возможности самостоятельно обрабатывать ошибки во время выполнения программы.

Такое решение было использовано в языке C++ путем добавления к языку возможности обработки исключительных ситуаций [24]. Данный механизм предназначен для описания реакции программы на ошибки времени выполнения и другие возможные проблемы (исключения), которые могут возникнуть при выполнении программы и приводят к невозможности (бессмысленности) дальнейшей отработки программой её базового алгоритма. Для работы с исключениями в C++ используют три ключевых оператора:

- try (пытаться) – начало блока исключений;
- catch (поймать) – начало блока, "ловящего" исключение;

– throw (бросить) – ключевое слово, "создающее" ("возбуждающее") исключение.

Рассмотрим пример. Опишем класс «Человек», который содержит конструктор с параметрами. Параметр «Возраст» не может быть меньше 0. Будем считать ситуацию, когда параметр «Возраст» объявлен меньше 0, исключительной ситуацией. Произведем обработку этой ситуации в программе.

```
class C_Person //объявление класса
{
protected: string Name;
           int Age;
public:    C_Person(string N,int A)//конструктор с параметрами
           {
           if (A<0)//проверка параметра «Возраст»
              throw 1;//Выбрасывание исключения
           Name=N; Age=A;
           }
};
void main ()
{
 try
 {//создание объекта с ложным параметром
    C_Person P0("J.Bin",-5);
 }
  catch (int unAge)
  {//обработка исключения «Недопустимое значение»
   cout<<"\n Caught exception 1:  Unacceptable Age";
  }
  getch();
}
```

После выполнения данного кода появится сообщение «Caught exception 1: Unacceptable Age».

Подобный подход значительно упрощает отладку приложений, увеличивает их надежность и позволяет описывать самотестирующийся код.

Кроме того, в языке С++ при использовании приемов «Принцип универсальности» и «Принцип наоборот» был реализован механизм описания новой абстракции данных – шаблонов. При помощи шаблонов в С++ можно описывать функции, классы и т.д. Особенность их в том, что при описании шаблон принимает в качестве аргумента тип данных. Это позволяет один раз написанный код использовать на множестве типов данных. Компилятор сам генерирует код функции или класса для нужного типа данных, а следовательно, и вероятность совершения ошибки становится меньше.

Шаблоны и исключения – это две стороны одной медали. Первые позволяют уменьшить число ошибок во время выполнения, расширяя спектр задач, с которыми может справиться статическая система типов. В свою очередь, исключения дают механизм для обработки оставшихся ошибок [19].

Противоречия 5.5 и 5.6 могут быть также разрешены приемом «Принцип предварительного действия». Так, например, для упрощения отладки программ и повышения надежности в Eiffel разработан механизм «Проектирование по контракту» (Design By Contract). Механизм позволяет задавать различные типы условий (контракты), проверяемых во время работы программы. Как и в реальной жизни, при нарушении одного из пунктов контракта наступает заранее обговоренная и согласованная мера – происходит выбрасывание исключения, т.е. нормальная работа прерывается и, если данное исключение должным образом не обрабатывается, приложение завершается. В любом случае, будет известно, что произошло нарушение условий контракта и есть ошибка.

Проектирование по контракту является довольно простой, но в то же время мощной методикой, основанной на документировании прав и обязанностей программных модулей для обеспечения корректности программы. Механизм базируется на трех основных конструкциях: предусловиях, постусловиях и инвариантах.

Предусловия проверяются перед началом выполнения тела метода, т.е. о выполнении предусловий должна позаботиться вызывающая сторона, и метод не должен отрабатывать, если его предусловия не выполняются.

Постусловия проверяются после завершения тела метода, т.е. о выполнении постусловий должна позаботиться реализация метода. Инварианты проверяются перед и после завершения работы публичных методов класса.

Использование данного механизма помогает обеспечить автоматическое тестирование кода, следовательно, упрощает отладку, сокращает время на разработку программы и повышает ее надежность.

Таким образом, произошла ТРИЗ-эволюция группы механизмов «Отладка» (рис. 5.9).

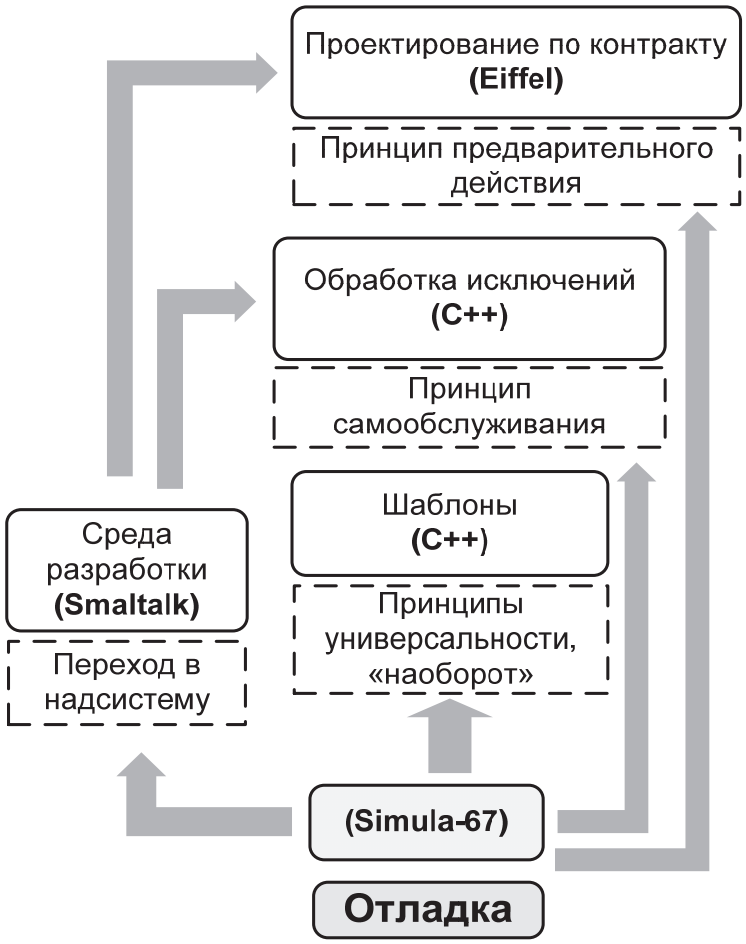

Рис. 5.9. ТРИЗ-эволюция группы механизмов «Отладка»

6. ИСПОЛЬЗОВАНИЕ ТРИЗ-ЭВОЛЮЦИОННОГО ПОДХОДА ПРИ САМОСТОЯТЕЛЬНОМ ИЗУЧЕНИИ PYTHON/DJANGO

6.1. Общие сведения

ТРИЗ-эволюционный подход может применяться как при обучении в группе, так и в рамках индивидуального обучения. Таким образом, был проведен «эксперимент» как использование подхода влияет на скорость и эффективность самостоятельного обучения программированию. В рамках реализации «эксперимента» можно выделить следующие этапы:

1) постановка и формализация реальной прикладной задачи;
2) разбиение задачи на подзадачи;
3) выбор средств решения поставленных подзадач (по возможности – неизвестных исполнителю);
4) изучение средств решения подзадач с использованием методологии обучения, основанной на ТРИЗ-эволюционном подходе;
5) решение подзадач по мере изучения новых средств.

6.2. Постановка и формализация задачи

В качестве задачи для рассмотрения была выбрана задача разработки информационного сайта для студентов и преподавателей КнАГТУ, основным назначением которого является информационная поддержка проводимых занятий по ТРИЗ-дисциплинам.

Согласно требованиям к системе, определенным в техническом задании, был проведен анализ возможных средств реализации системы. При анализе был использован метод анализа иерархий [1]. Выбор производился из пяти основных альтернатив: JavaScript, PHP, Perl, Python, Ruby. Результат проведенного анализа показал, что наиболее оптимальным средством разработки системы является язык Python.

«Python – язык универсальный, он широко используется во всем мире для самых разных целей – базы данных и обработка текстов, встраивание интерпретатора в игры, программирование GUI и быстрое создание прототипов (RAD). И, конечно же, Python используется для программирования Internet и Web приложений – серверных (CGI), клиентских (роботы), Web-серверов и серверов приложений. Python обладает богатой стандартной библиотекой и еще более богатым набором модулей, написанных третьими лицами. Python и приложения, написанные на нем, используют самые известные и крупные фирмы – IBM, Yahoo!, Google.com, Hewlett Packard, Infoseek, NASA, Red Hat, CBS MarketWatch, Microsoft [2]».

Для начала работы над разработкой системы, описанной в техническом задании, также необходимо следующее программное обеспечение:

Веб-фреймворк для языка Python. Для разработки системы был выбран фреймворк Django, так как на сегодняшний день именно этот фреймворк является наиболее популярным среди разработчиков (как новичков, так и профессионалов) [3].

Unix-подобная операционная система. Использование Unix-подобной операционной системы позволит избежать возможных проблем совместимости, которые возникают при работе с фреймворком Django в операционной системе Windows. Для разработки была выбрана операционная система Ubuntu.

Веб-сервер. Для нормального функционирования Django обязательно наличие одного из определённых веб-серверов: веб-сервер Apache с возможностью использования одного из модулей mod_fastcgi, mod_fcgid или mod_wsgi; веб-сервер Nginx с возможностью использования модуля ngx_http_fastcgi_module; веб-сервер Lighttpd с возможностью использования модуля ModFastCGI [4]. Для разработки системы был выбран веб-сервер Apache и модуль mod_wsgi.

СУБД. Для разработки системы была выбрана СУБД MySQL. Для того чтобы использовать данную СУБД при разработке системы, необходимо дополнительно установить библиотеку Python python-mysql.

Работа с изображениями. Для работы с изображениями необходима установка библиотеки Python – PIL.

Для хранения проекта системы и управления версиями было решено использовать *репозиторий*. Использование репозитория позволит упростить загрузку изменений системы на сервер, а также предоставит возможность осуществлять контроль над версиями системы и резервное копирование. Для хранения проекта системы был выбран репозиторий BitBucket.org, так как он позволяет бесплатно хранить закрытые проекты. Для работы с репозиторием необходима установка утилиты git.

Необходимо, чтобы состав программного обеспечения на сервере и компьютере разработчика был идентичным во избежание проблем интеграции при выгрузке проекта системы на сервер.

Всего было выделено 3 этапа работы над проектом:

1) *подготовительный этап*, включающий в себя установку и настройку программного обеспечения, необходимого для разработки;

2) *этап реализации*, включающий в себя разработку системы согласно требованиям, описанным в техническом задании;

3) *этап тестирования и модификации системы*, который включает в себя тестирование системы группой пользователей и модификацию системы согласно определенным во время тестирования требованиям.

Далее мы рассмотрим только этап реализации, то есть разработки системы, а именно примеры решения промежуточных задач и изучения средств разработки системы с использованием ТРИЗ-эволюционного подхода.

6.3. Примеры использования ТРИЗ-эволюционного подхода

6.3.1. Интегрированная среда разработки

После того как была выполнена установка и настройка описанного выше программного обеспечения, выяснилось, что для написания кода можно использовать только встроенный в систему текстовый редактор (в графическом или консольном интерфейсе), а для компиляции написанного кода необходимо использовать команду в консоли (рис. 6.1) `python manage.py runserver`.

Данная команда перезапускает встроенный веб-сервер Django и исполняет написанный код. В случае возникновения ошибки, необходимо вновь открывать текст программы, производить поиск ошибки, редактировать текст программы, сохранять изменения и опять перезапускать веб-сервер. Данный механизм разработки крайне неэффективен, так как отнимает много времени на отладку и компиляцию кода программы.

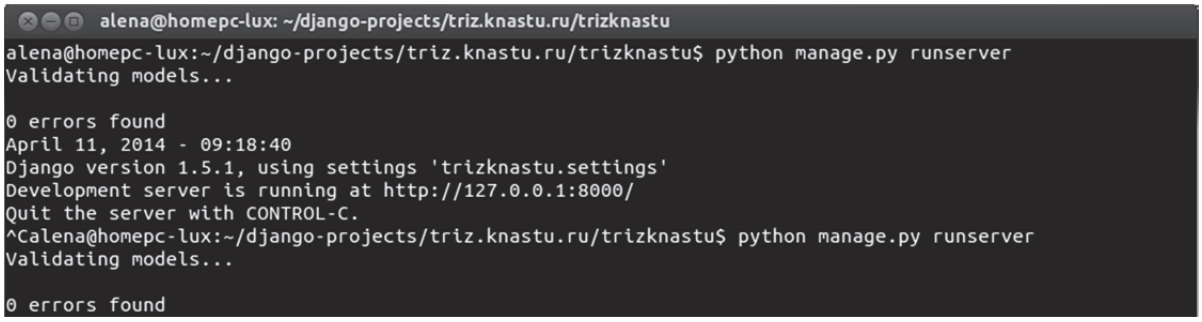

Рис. 6.1. Компиляция кода и запуск веб-сервера из консоли

Возникает *противоречие 1*: с увеличением объема кода недопустимо увеличивается время на отладку программы.

При необходимости одновременной отладки программы, работы с базой данных, выполнения в терминале системных процессов, запуске веб-сервера и т. д. необходимо каждый из процессов попеременно запускать/останавливать либо вести работу в нескольких окнах командной строки (терминала), что порождает *противоречие 2*: при увеличении количества одновременно выполняемых процессов недопустимо увеличивается количество окон интерфейса, навигация между которыми затруднена в виду их однообразного оформления.

Данные противоречия были разрешены путем использования закона перехода в надсистему и принципа универсальности. Была установлена интегрированная среда разработки PyCharm, которая объединяет в себе возможности по написанию кода с автозаполнением, подсказками, ссылками на смежные файлы; отладки текста программы; компиляции «на лету»; работы с консолью, репозиторием, оболочкой python’а, фоновым запуском веб-сервера и т.д. (рис. 6.2).

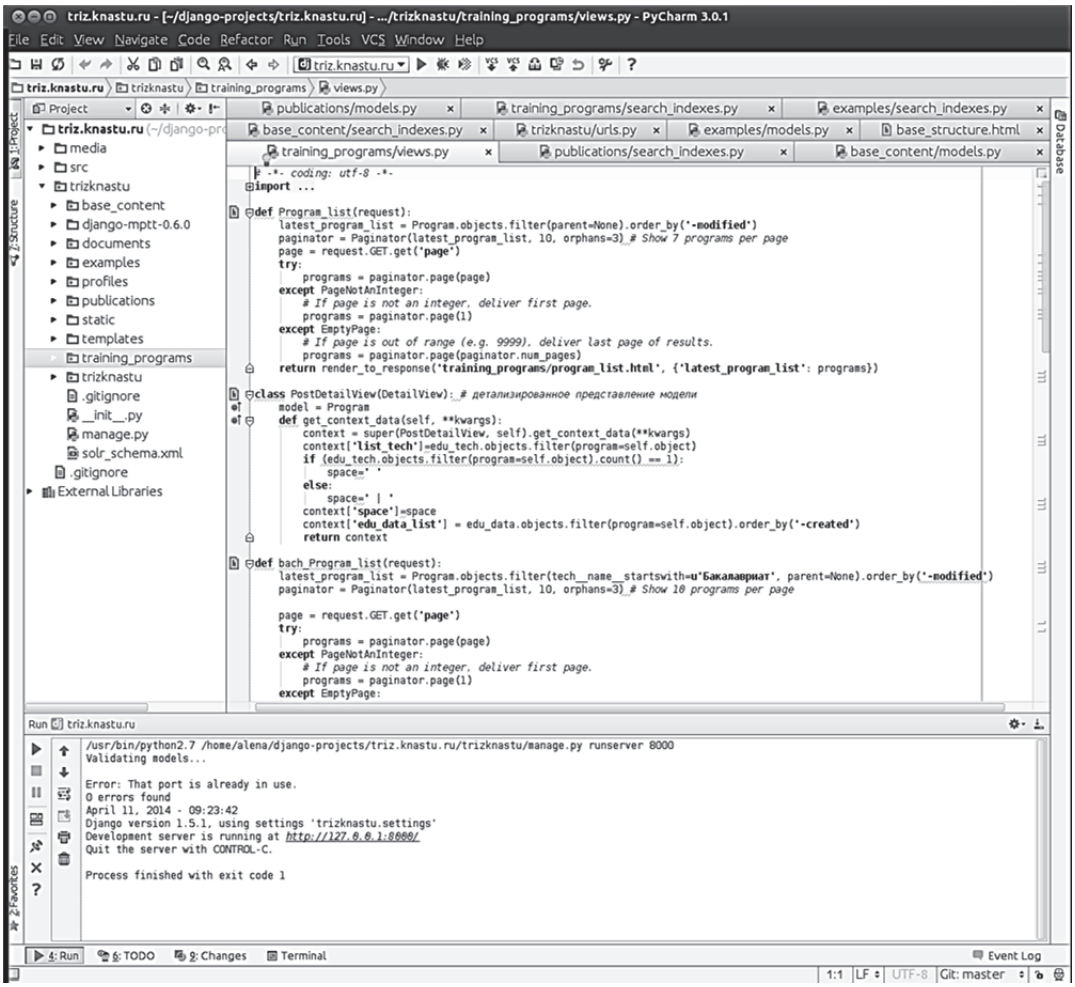

Рис. 6.2. Интегрированная среда разработки PyCharm

Таким образом, произошла первая итерация ТРИЗ-эволюции средств разработки сайта (рис. 6.3).

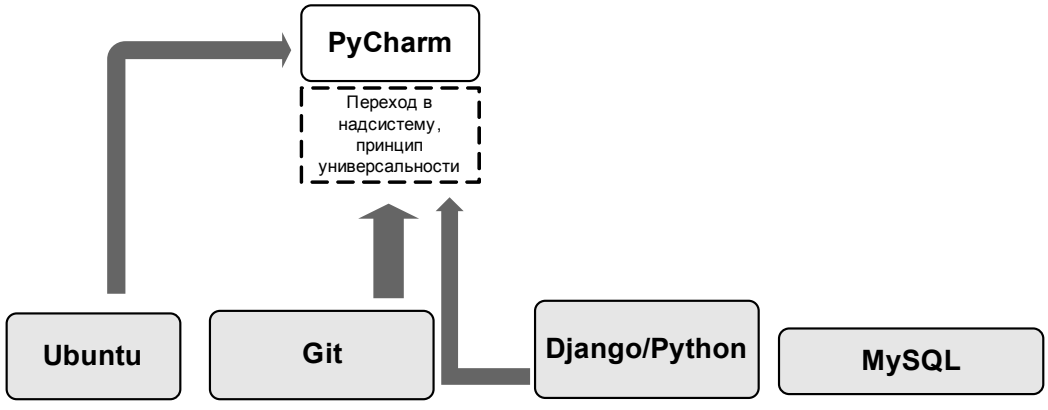

Рис. 6.3. Первая итерация ТРИЗ-эволюции

6.3.2. Разбиение программы на модули

Согласно техническому заданию сайт должен содержать информацию о трех основных разделах в базе данных – образовательных программах, примерах и публикациях. Также должна быть предусмотрена система наполнения сайта информационным контентом.

Кроме того, должны решаться задачи, не связанные напрямую со сбором, хранением и обработкой информации об указанных сущностях. Каждая сущность в структуре программы должна содержать описание модели в базе данных (файл models.py), алгоритм отображения на сайте (файл views.py), ссылку на отображение (файл urls.py). При создании проекта (команда `python manage.py startproject [Имя проекта]`) данная структура файлов создается в корне проекта (рис. 6.4) автоматически вместе с файлами настройки и является общей для всего проекта.

Создание сайта требует описания в указанных файлах множества сущностей, связанных между собой общей принадлежностью к основным разделам базы данных. При этом, если код всего проекта будет храниться в общих файлах, то наглядность кода и его читабельность недопустимо ухудшатся, что потребует также больших временных затрат на отладку программы. Таким образом, возникает *противоречие 3*: с увеличением сложности программируемой системы недопустимо уменьшается читабельность кода программы.

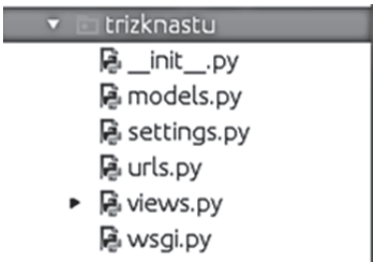

Рис. 6.4. Структура файлов в корне проекта

Данное противоречие может быть разрешено с использованием приемов «Принцип дробления» и «Принцип посредника». Используя принцип дробления, можно «раздробить» проект на более маленькие подпроекты, каждый из которых будет иметь структуру файлов, позволяющих описывать их как независимые приложения. Общие файлы настройки всего проекта можно использовать в качестве «посредника» для интеграции всех подпроектов в общий проект. Данная возможность реализована в Django, что позволяет дробить проект на любое количество подпроектов, каждый из которых подключается в готовый проект через файл настроек settings.py (рис. 6.5).

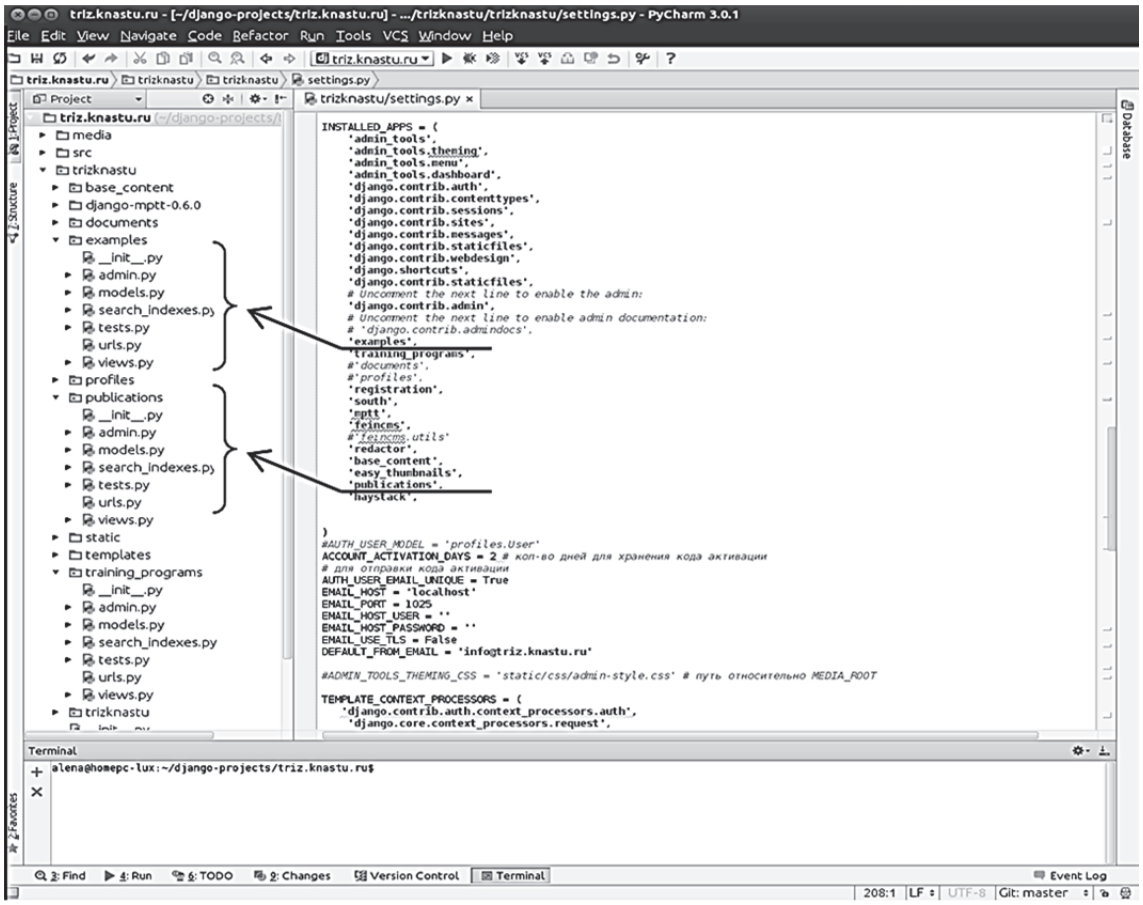

Рис. 6.5. Структура проекта и файл настроек settings.py

Таким образом, произошла вторая итерация ТРИЗ-эволюции средств разработки сайта (рис. 6.6).

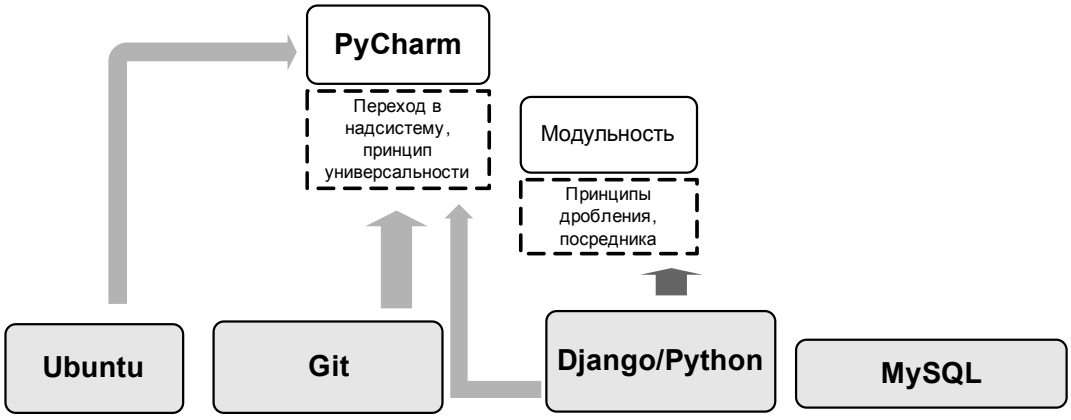

Рис. 6.6. Вторая итерация ТРИЗ-эволюции

6.3.3. Механизм авторизации

Одно из ключевых требований к системе, согласно техническому заданию, разграничение доступа к контенту на сайте. Часть контента будет доступна всем без исключения пользователям, а часть только авторизованным пользователям. Такие ограничения могут быть введены как на отдельные страницы, так и на разделы сайта.

Можно решить данную задачу описав дополнительную функцию в каждом представлении, которая будет проверять, авторизован ли пользователь и только после этого открывать доступ к контенту. Однако подобную функцию придется прописывать для каждого представления, для которого необходимо выставить настройку авторизации. Это ведет к избыточности кода, снижает его наглядность и усложняет процесс отладки программы. Даже если запрограммировать процедуру проверки пользователя через представление в виде отдельно подключаемого модуля, в каждом конкретном представлении придется вызывать данную функцию проверки, что сократит объем кода, но, в сущности, не решит проблему.

Таким образом, возникает *противоречие 4*: с увеличением гибкости настройки механизма доступа к контенту на сайте недопустимо снижается наглядность кода.

Данное противоречие разрешено при помощи приемов «Принцип универсальности», «Принцип посредника» и «Принцип вынесения» путем использования декоратора функций login_required.

Декораторы используются для оборачивания функций в целях изменения их поведения. Ключевой особенностью декоратора является возможность принимать функции и возвращать функции. Функция, возвращенная декоратором, будет вызвана в момент вызова декорируемой функции.

В данном случае декоратор [5] – это функция, реализованная в отдельном модуле, которая проверяет, авторизован пользователь или нет, и может быть вызвана в любом месте программы. Если пользователь авторизован, то он получает доступ к странице с необходимым ему контентом, если нет, то ему предлагается войти на сайт при помощи логина/пароля. Использовать данный декоратор можно в файле конфигурирования url-адресов (urls.py) за пределами описания самих функций представления, что значительно повышает наглядность кода и упрощает отладку. Далее представлен пример настройки доступа к главной странице сайта с использованием описанного декоратора.

```
# Импорт функции представления главной страницы
from views import MainView
# Импорт функции-декоратора
from django.contrib.auth.decorators import login_required
# Описание url-шаблона
```

```
urlpatterns = patterns('',
     url(r'^$',login_required(MainView.as_view(),login_url='log
     in/'),name='home'),)
```

Таким образом, произошла третья итерация ТРИЗ-эволюции (рис. 6.7).

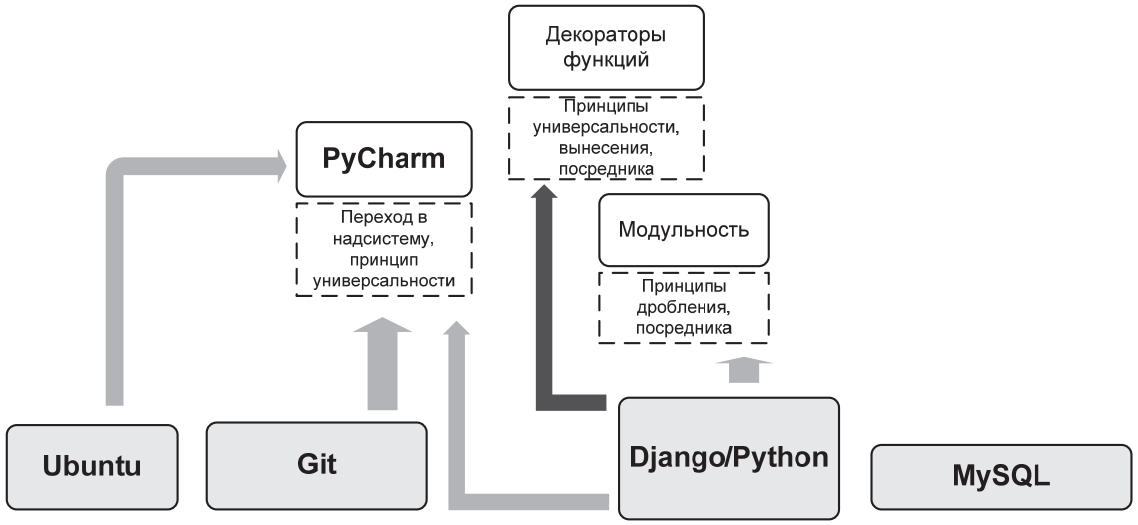

Рис. 6.7. Третья итерация ТРИЗ-эволюции

6.3.4. Интерфейс администратора

При проектировании любого сайта возникает необходимость в разработке системы управления контентом. Как правило, это системы прямого доступа к базе данных с использованием графического интерфейса для создания, изменения и удаления записей в базе данных и многих других. В случае отсутствия системы управления контентом пользователю придется потратить огромное количество времени для внесения изменений в базу данных. Кроме того, во много раз повышается риск занесения ошибочной информации. То есть возникают противоречия 5 и 6.

Противоречие 5: При увеличении количества записей в базе данных недопустимо увеличивается время на заполнение и редактирование базы данных.

Противоречие 6: При увеличении количества записей в базе данных недопустимо увеличивается вероятность занесения ошибочной информации в базу.

Данные противоречия были разрешены приемом «Принцип посредника» путем подключения к проекту модуля django.contrib.admin, предоставляющего возможности графического управления контентом сайта.

Стандартная настройка интерфейса администрирования при помощи django-admin позволяет редактировать любые модели базы данных, зареги-

стрированные в файле admin.py. Далее приведен пример регистрации модели в интерфейсе администрирования.

```
# Класс описания модели в интерфейсе администратора
class FldAdmin(admin.ModelAdmin):
# Поля, которые отображаются в общем списке таблицы
    list_display = ('name', 'created', 'modified', 'remove')
# Поле, по которому можно произвести быстрый поиск
    search_fields = ['name']
# Количество объектов, выводимых на одну страницу
    list_per_page = 40
# Регистрация модели с учетом класса описания
admin.site.register(appModels.author_types,FldAdmin)
```

На рис. 6.8 представлен внешний вид модели «Области знаний» в интерфейсе администратора.

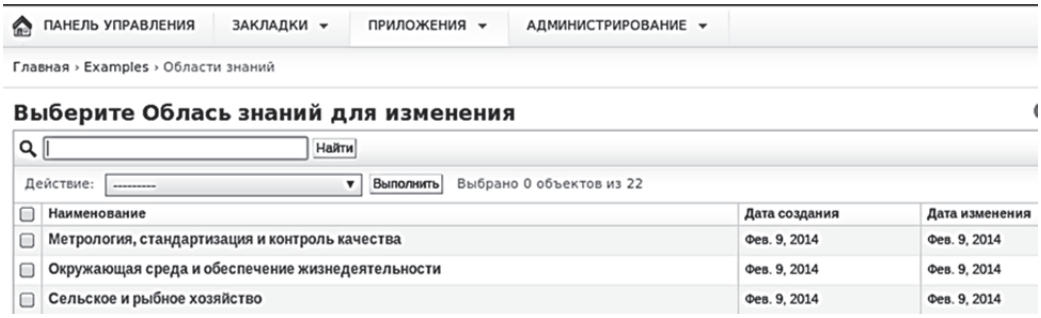

Рис. 6.8. Окно редактирования объектов модели «Области знаний»

Однако данный интерфейс не является полностью оптимизированным. Например, чтобы удалить объект базы данных, необходимо отметить удаляемый объект, выбрать «Удалить выбранные объекты» в графе «Действие» и нажать «Выполнить», что неудобно при наличии большого числа записей в базе. Возникает *противоречие 7*: при увеличении количества записей в базе данных недопустимо снижается удобство использования функции удаления.

Данное противоречие может быть разрешено с использованием приемов «Принцип предварительного действия» и «Принцип копирования». Реализуя данные приемы, можно добавить к списку моделей дополнительную колонку «Удалить» и разместить напротив каждого объекта кнопку удаления, скопировав описание функции удаления в описание реакции на кнопку «Удалить». Далее представлен код, иллюстрирующий данное решение.

```
    def remove(self, entry):
        from django.core.urlresolvers import reverse
        name = 'admin:%s_%s_delete' % (
```

```
                self.model._meta.app_label,
                self.model._meta.module_name
        )
        return '<a href="%s" class="deletelink">Удалить</a>'\
                % reverse(name, args=(entry.pk,))
    remove.allow_tags = True
    remove.short_description='Удалить?'
```

На рис. 6.9 представлен вид окна администрирования после внесения изменений.

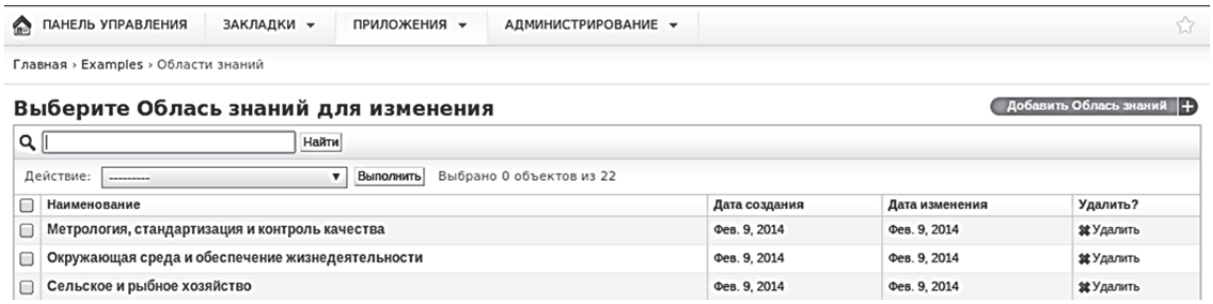

Рис. 6.9. Окно редактирования объектов модели «Области знаний»

Еще одно противоречие возникает при редактировании объектов, связанных в базе данных связью «многие к одному». Рассмотрим пример. Существует модель «Образовательная программа» и «Документ». Каждый документ может быть привязан только к одной образовательной программе, в то время как к программе может быть привязано несколько документов.

Интерфейс администратора позволяет редактировать зарегистрированные модели в отдельных окнах, что неудобно, так как отдельно от образовательной программы документы не представляют интереса. А связывать их с программами каждый в отдельности неудобно и нерационально по времени.

Таким образом, возникает *противоречие 8*: при повышении эффективности использования интерфейса администратора для объектов, связанных по принципу «многие к одному», недопустимо увеличивается время для описания связей в базе данных.

Данное противоречие было решено при использовании приема «принцип предварительного действия» путем добавления механизма редактирования связанных объектов в одном окне.

Интерфейс администратора Django позволяет редактировать связанные объекты на одной странице с родительским объектом. Это называется «Inlines».

Пример реализации данного механизма представлен на рис. 6.10.

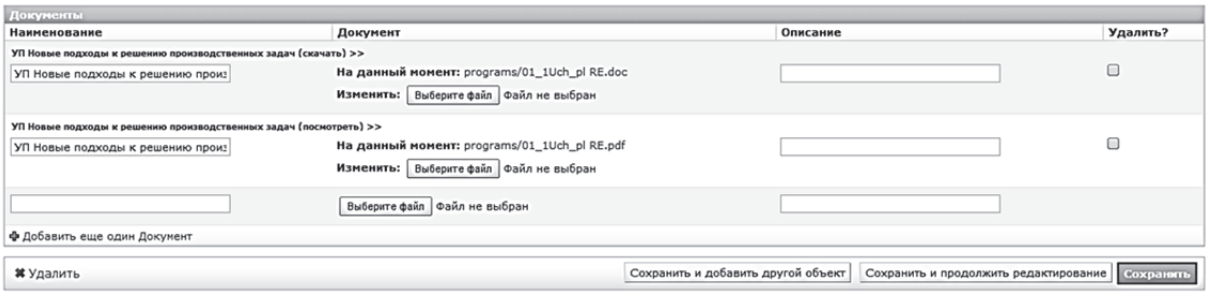

Рис. 6.10. Редактирование документов в окне изменения образовательной программы

Таким образом, произошли четвертая, пятая, шестая итерации ТРИЗ-эволюции средств разработки сайта (рис. 6.11).

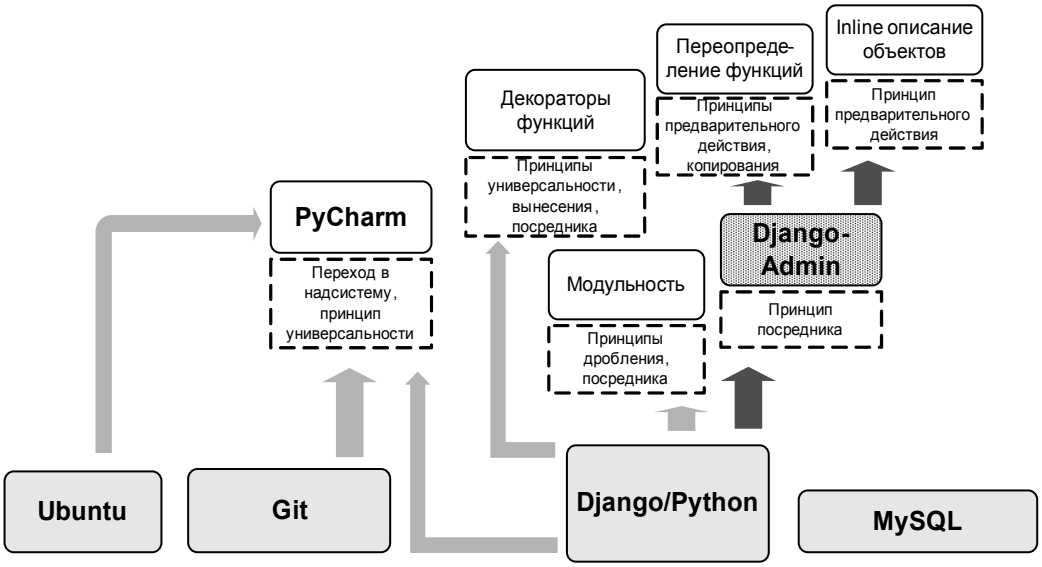

Рис. 6.11. Четвертая, пятая, шестая итерации ТРИЗ-эволюции

6.3.5. Иерархия в базе данных

Согласно техническому заданию в базе данных должна храниться информация об объектах, представляющих собой иерархически организованные последовательности. Кроме того, при представлении информации об объектах на сайте необходимо: отображать иерархические последовательности в виде блоков, отображать отдельные «ветви иерархии», отображать родительские и дочерние элементы отдельных узлов иерархии, оценивать общее количество узлов иерархии, выводить последовательность узлов без учета неузловых ответвлений и т.д. На разработку всего пакета описанных функций может уйти большое количество времени, неизмеримое с важностью подзадачи в целом.

Таким образом, возникает *противоречие 9*: при увеличении эффективности работы с иерархически организованными данными недопустимо увеличивается объем времени на программирования пакета необходимых для этого функций.

Данное противоречие можно решить при помощи приема «принцип посредника» путем использования готового модуля, включающего в себя перечисленные пакеты функций. Один из таких модулей является модуль «MPTT».

Несмотря на то, что модуль решает проблемы организации сбора, хранения и доступа к данным в базе, существуют проблемы с отображением иерархических структур в интерфейсе администратора. Необходимо, чтобы данные отображались в виде раскрывающихся списков. MPTT не предоставляет такой возможности. Возникает *противоречие 10*: при увеличении эффективности работы с иерархическими данными недопустимо снижается наглядность представления иерархических данных в интерфейсе администратора.

Данное противоречие было разрешено приемами «Принцип посредника» и «Принцип частичного или избыточного действия» путем частичного использования компонентов системы управления контентом FeinCMS. Установив пакет FeinCMS, воспользуемся только функциями, отвечающими за отображение иерархии в интерфейсе администратора. Внешний вид окна редактирования иерархически организованных объектов модели «Информационные разделы сайта» представлен на рис. 6.12.

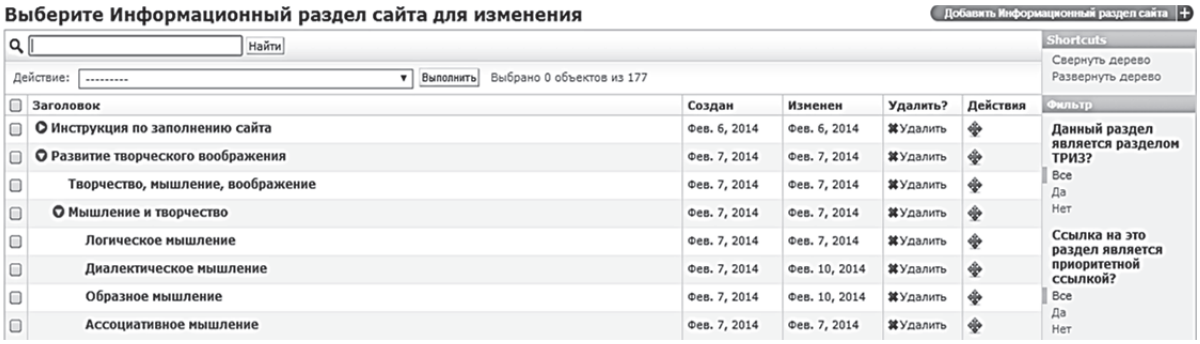

Рис. 6.12. Окно редактирования объектов модели «Информационные разделы сайта»

Таким образом, произошли седьмая и восьмая итерации ТРИЗ-эволюции средств разработки сайта (рис. 6.13).

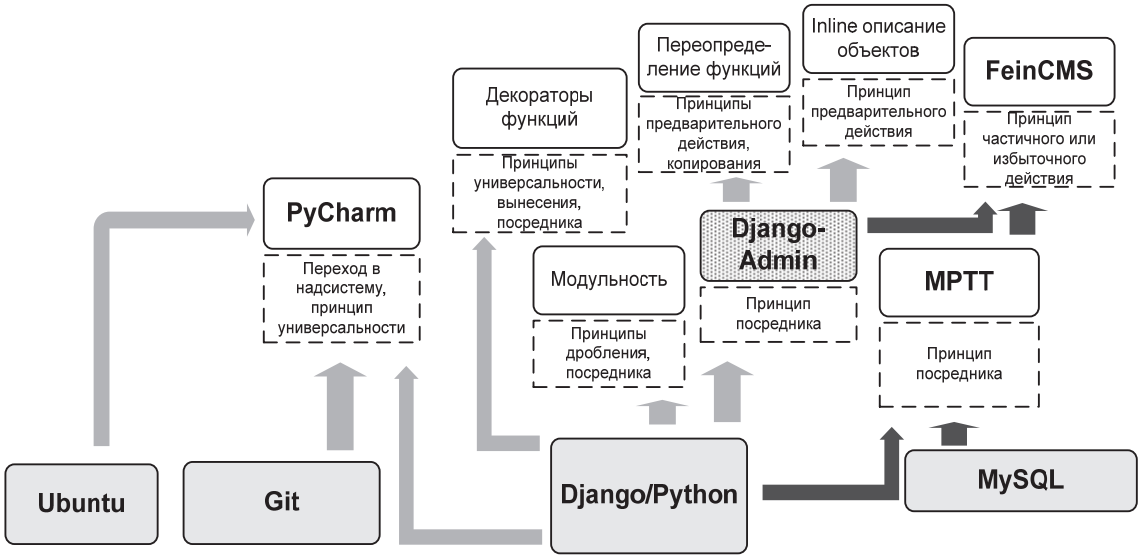

Рис. 6.13. Седьмая и восьмая итерации ТРИЗ-эволюции

6.3.6. Загрузка файлов на сервер

Проект базы данных, описанной в техническом задании, предполагает сохранение на сервере различных пользовательских файлов. Встроеный веб-сервер Django использует кодировку ascii, что приводит к проблемам, связанным с загрузкой на сервер файлов с русскоязычными названиями. Возникает *противоречие 11*: с повышением удобства пользовательского интерфейса недопустимо снижается работоспособность программы.

Данное противоречие можно разрешить несколькими способами.

Используя приемы «Принцип предварительного действия» и «Принцип частичного или избыточного действия», можно подготовить конструкцию для пользователя по загрузке файлов на сайт, в которой указать, что для загрузки на сайт все файлы должны иметь англоязычное название. При этом в случае невыполнения пользователем указанных требований, загрузка файла не будет произведена, что позволит программе продолжить выполнение без прерываний, вызванных критической ошибкой сервера.

Используя прием «Принцип заранее подложенной подушки», можно запрограммировать функцию трансляции русскоязычного текста в англоязычный. При этом, используя прием «Принцип посредника», можно запрограммировать функцию, которая будет считывать название загруженного файла, транслировать его при помощи функции трансляции названия и передавать в базу данных.

Оба способа были применены при разработке сайта. В инструкции для пользователя указано, что при работе с текстом через WYSIWYG редактор (рис. 6.14) необходимо загружать рисунки с англоязычными названиями, в противном случае они не загружаются на сервер.

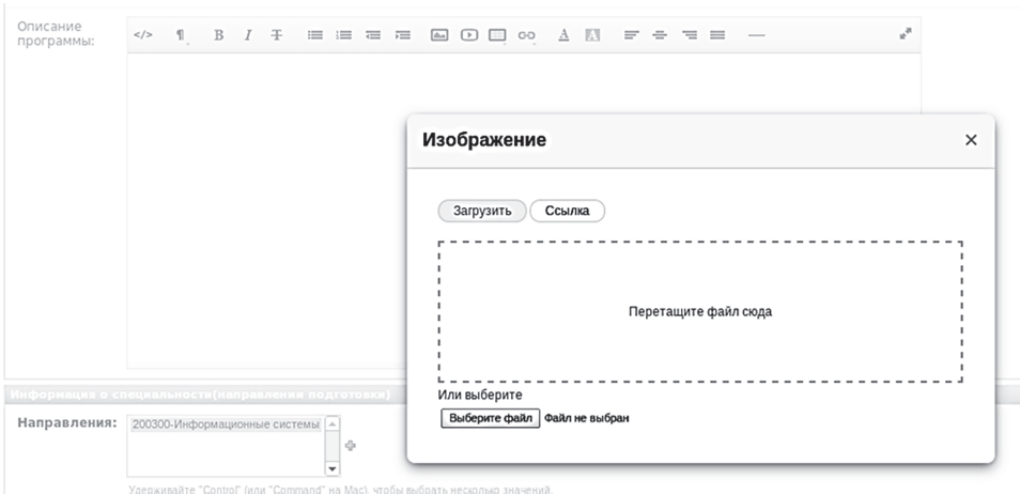

Рис. 6.14. Добавление файла через WYSIWYG редактор

При работе с файлами образовательных программ, примеров и публикаций, для разрешения противоречия используется второй способ.

Рассмотрим реализацию данного способа на примере добавления файла образовательной программы. Сначала опишем функцию транслитерации текста с русского языка на английский.

Таким образом, при загрузке файла с названием «Отчет о реализации.doc» файл будет сохранен под названием «Otchet o realizatsii.doc» (рис. 6.15).

Рис. 6.15. Транслитерация названия файла при загрузке на сервер

Таким образом, произошла девятая итерация ТРИЗ-эволюции средств разработки сайта (рис. 6.16).

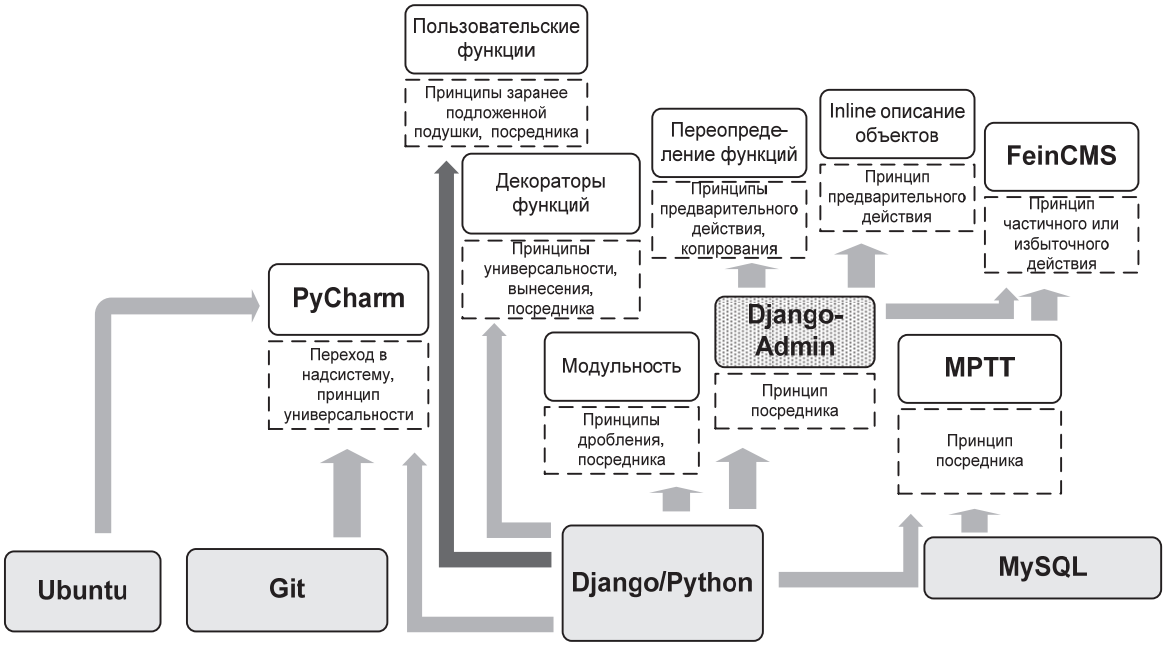

Рис. 6.16. Девятая итерация ТРИЗ-эволюции

6.3.7. Изменение структуры базы данных

В процессе разработке сайта часто случается, что необходимо скорректировать структуру базы данных. После описания модели базы данных в файле models.py необходимо выполнить команду `python manage.py syncdb`, чтобы создать или удалить таблицы. Однако, если необходимо изменить название поля, добавить новое поле в уже существующую таблицу – возникают проблемы. Необходимо вручную корректировать структуру нужных таблиц через терминал командной строки и клиент mysql. Возникает *противоречие 12*: при повышении идеальности структуры таблиц базы данных время на внесение изменений недопустимо увеличивается.

Данное противоречие можно разрешить, используя приемы «Принцип заранее подложенной подушки» и «Принцип посредника» путем реализации отдельного модуля, который бы хранил информацию об исходной структуре базы данных в отдельном файле и по необходимости применял внесенные пользователем изменения, сравнивая предыдущую версию структуры с последней сохраненной. Таким образом, появится возможность не только изменять структуру базы «на лету», но и в случае необходимости возвращаться к предыдущим версиям. Данный механизм называется «Миграции» и реализован в модуле South.

Чтобы использовать миграции, необходимо подключить приложение к проекту командой `python manage.py schemamigration [имя приложения] –initial`. Данная команда создаст исходную структуру базы, с которой потом будут сравниваться последующие изменения.

После создания/изменения моделей нужно создавать новые миграции: `python manage.py schemamigration [имя приложения] –auto`. А затем применять миграции: `python manage.py migrate [имя приложения]`.

Таким образом, произошла десятая итерация ТРИЗ-эволюции средств разработки сайта (рис. 6.17).

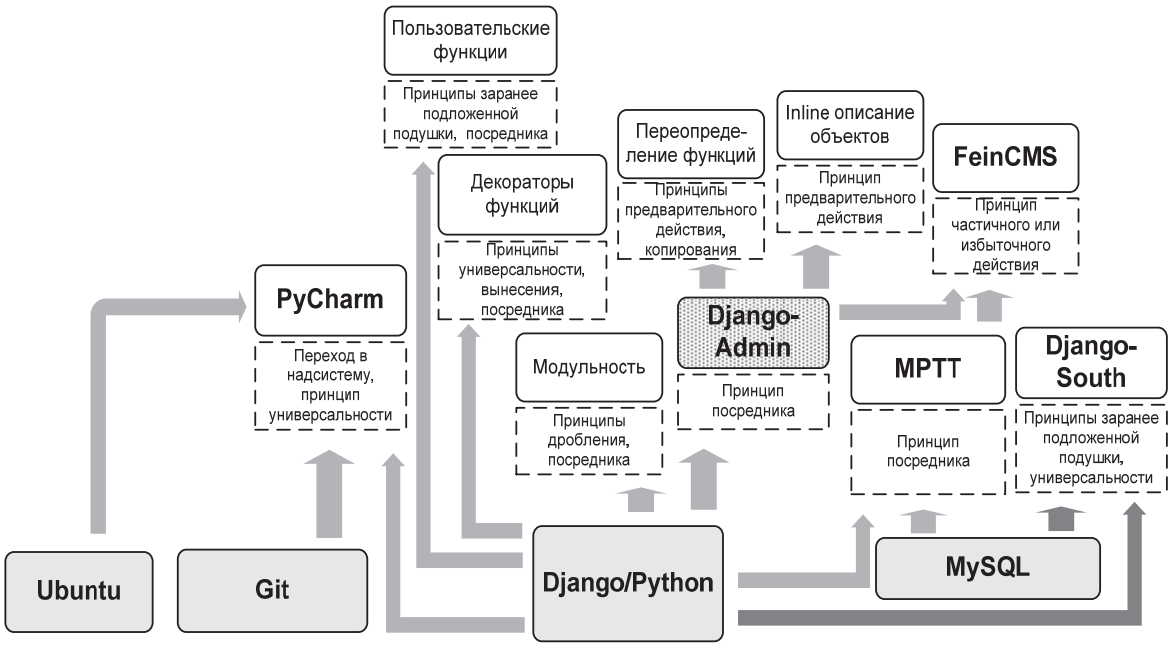

Рис. 6.17. Десятая итерация ТРИЗ-эволюции

Аналогичным образом, путем «от противоречия к противоречию» были «открыты» последующие элементы ТРИЗ-эволюционной карты средств разработки сайта.

6.3.8. Прочие итерации ТРИЗ-эволюции

Далее приведен перечень противоречий и приемов, при помощи которых эти противоречия были разрешены.

Противоречие 13: с увеличением эффективности управления массивами объектов недопустимо увеличивается объем разрабатываемого кода (разрешено при помощи приема «Принцип универсальности» путем добавления возможности описания шаблонов вывода динамического содержимого базы данных).

Противоречие 14: с увеличением количества однотипных элементов оформления страниц и навигации недопустимо увеличивается объем кода (разрешено при помощи приемов «Принцип объединения» и «Принцип матрешки» при объединении дублируемых параметров в одном из шаблонов и введения возможности объекту наследовать блоки другого шаблона с неограниченной глубиной наследования).

Противоречие 15: при увеличении сложности операций над переменными, выводимыми в шаблонах страниц, недопустимо увеличивается время на разработку шаблона (разрешено при помощи приемов «Принцип однородности» и «Принцип частичного или избыточного действия» путем реализации механизма использования тегов, которые повторяют некоторые структуры языка программирования – тег *if* для проверки на истинность, тег *for* для циклов, и др., но они не выполняются непосредственно как код Python, и система шаблонов не будет выполнять произвольное выражение Python. Только встроенные теги и синтаксис поддерживаются по умолчанию).

Противоречие 16: с увеличением количества однотипных элементов вывода данных из базы недопустимо увеличивается объем дублируемого кода (разрешено приемами «Принцип объединения» и «Принцип динамичности» путем реализации возможности создания пользовательских тегов, которые могут быть вызваны в любом шаблоне приложения).

Противоречие 17: с увеличением количества способов отображения данных из базы недопустимо увеличивается время на разработку алгоритмов, реализующих данные способы (разрешено при помощи приемов «Принцип посредника», «Принцип заранее подложенной подушки» путем реализации порядка 30 стандартных задач по обработке выводимого из базы содержимого страниц в виде шаблонных фильтров).

Противоречие 18: при увеличении эффективности работы с изображениями в шаблонах недопустимо увеличивается время на написание кода (разрешено при помощи приема «принцип посредника» путем использования модуля Easy-thumbnail, который предоставляет набор функций для работы с изображениями в шаблонах).

Противоречие 19: при улучшении качества оформления веб-страниц наглядность кода в шаблонах недопустимо ухудшается (разрешено при помощи приемов «Принцип посредника», «Принцип объединения» и «Принцип вынесения» путем создания возможности прописывать стилистические особенности шаблона не в нем самом, а в отдельном подключаемом CSS-файле, при этом для каждого тега *html* может быть прописано единое оформление).

Противоречие 20: при увеличении количества интерактивных элементов веб-страницы недопустимо увеличивается объем разрабатываемого кода (разрешено при помощи приемов «Принцип посредника», «Принцип заранее подложенной подушки» путем использования многофункциональной JavaScript библиотеки jQuery, которая предоставляет огромное количество готовых решений по описанию интерактивных элементов веб-страниц).

Противоречие 21: при увеличении объема используемых встроенных возможностей Django по работе с базой данных недопустимо снижается качество поиска (разрешено при помощи приема «Принцип посредника» путем использования поискового движка Solr через Django-haystack для организации поиска на сайте).

Противоречие 22: при улучшении качества поиска путем использования сервера Solr недопустимо ухудшается непрерывность работоспособности поиска (разрешено при помощи приемов «Принцип посредника» и «Принцип вынесения» путем написания скрипта, который позволит запускать/останавливать/перезапускать выполнение Solr в фоновом режиме).

Противоречие 23: при улучшении качества работы серверного аппаратно-программного обеспечения недопустимо увеличивается время на перезапуск Solr (разрешено при помощи приемов «Принцип предварительного действия» и «Принцип самообслуживания» путем добавления скрипта запуска Solr в автозагрузку операционной системы).

Противоречие 24: при увеличении количества форм поискового запроса недопустимо снижается качество поиска (разрешено приемами «Принцип посредника» и «Принцип предварительного действия» путем настройки встроенного компонента Solr, позволяющего производить морфологический поиск, и подбора специально организованного морфологического словаря).

Противоречие 25: при увеличении количества вариаций значений слов поискового запроса недопустимо снижается качество поиска (разрешено при помощи приемов «Принцип местного качества» и «Принцип посредника» путем настройки встроенного компонента Solr, позволяющего производить поиск по словам-синонимам, и подготовки файла, в котором хранятся цепочки условно принятых синонимов).

Противоречие 26: при увеличении скорости переноса проекта на рабочий сервер недопустимо увеличивается время на изменение файлов настройки, где в явном виде прописаны пути к директориям проекта в файловой системе (разрешено при помощи приемов «Принцип посредника», «Принцип предварительного действия» путем подготовки ла .gitignore, в котором указываются файлы, изменения которых не отображаются на рабочем сервере).

Общая ТРИЗ-эволюционная карта средств разработки сайта представлена на рис. 6.18.

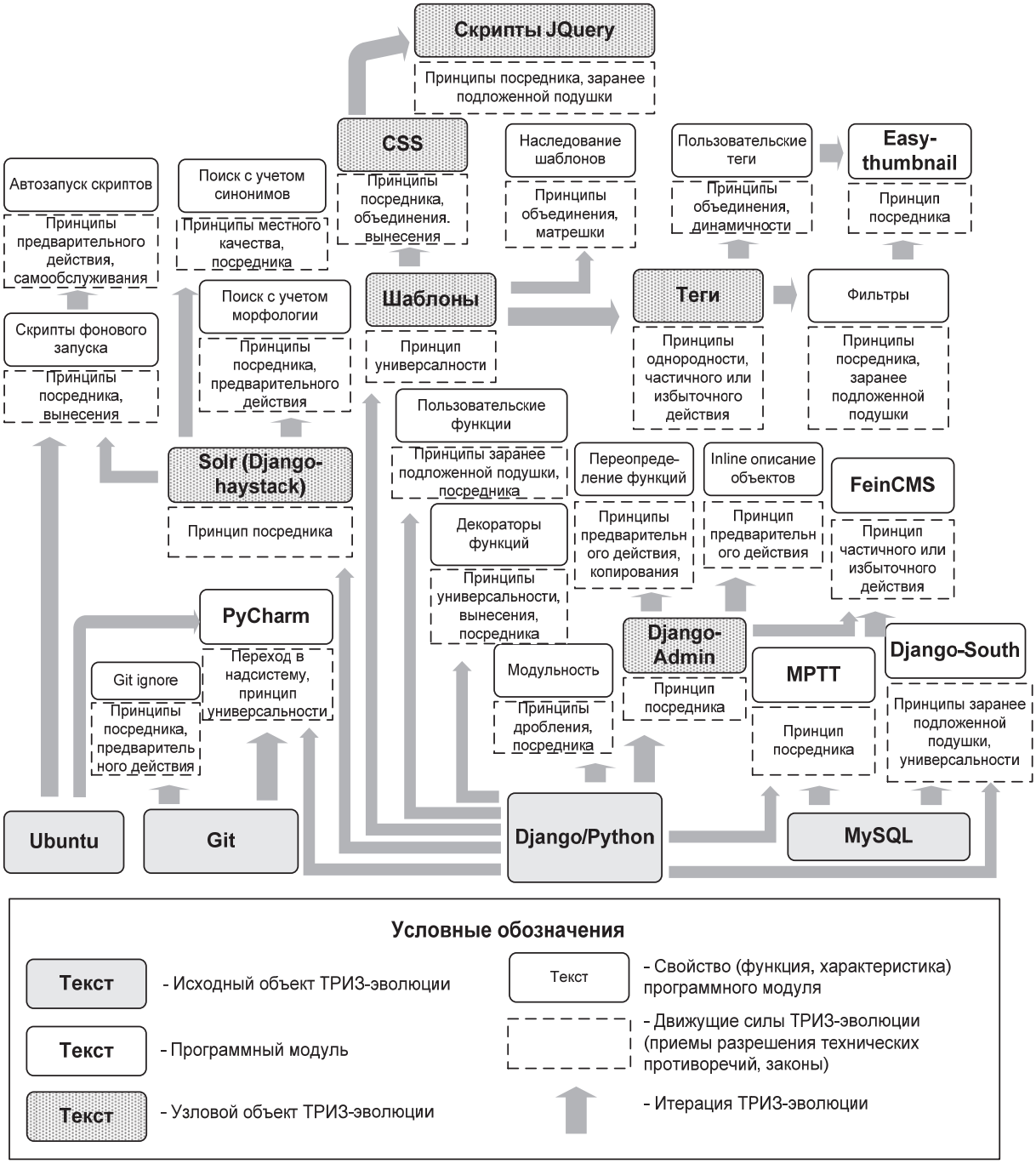

Рис. 6.18. ТРИЗ-эволюционная карта инструментов разработки сайта

6.4. Результаты применения ТРИЗ-эволюционного подхода при индивидуальном обучении

В целом, можно сказать, что использование ТРИЗ-эволюционного подхода позволяет существенно интенсифицировать изучение выбранной области знаний за счет систематизации знаний не только на уровне разрешения противоречий, но и на уровне формулировки противоречий.

Способ мышления «от противоречия к противоречию» позволяет систематизировать информацию о возникшей проблеме, формализовать ее, выделить ухудшаемые и улучшаемые параметры и предельно четко ее сформулировать, что ускоряет поиск решения, ведь иногда правильно заданный вопрос – уже ответ.

Построенная ТРИЗ-эволюционная карта инструментов разработки сайта впоследствии систематизирует знания о наиболее востребованных и актуальных инструментах разработки сайта и может быть использована начинающими Django-разработчиками, так как позволит сократить время на поиск информации и решение ряда стандартных задач, которые недостаточно освещены в литературе, посвященной Django.

Следует отметить, что данная ТРИЗ-эволюционная карта может быть расширена и детализирована на уровне как исходных, так и узловых объектов ТРИЗ-эволюции.

7. КУРСОВАЯ РАБОТА

7.1. Задание на курсовую работу

В рамках данной дисциплины предусмотрено выполнение одной курсовой работы. Курсовую работу студент должен защитить. Во время защиты курсовой работы преподавателем проверяется правильность и самостоятельность его выполнения. Студент поясняет ход выполнения заданий, алгоритмы, параметры используемых функций.

Задание на курсовую работу включает в себя следующие части:

1) описание шаблона контейнерного класса заданной абстрактной структуры данных;

2) решение задачи, которая включает в себя описание класса или иерархии классов.

Каждая часть курсовой работы должна содержать следующие подразделы:

1) постановка задачи;

2) теоретические основы решения поставленной задачи, ее формализация;

3) ход решения;

4) тестирование программы;

5) анализ решения, выводы, заключение;

6) список использованных источников.

Образцы заданий для выполнения приведены ниже.

Варианты заданий на курсовую работу могут быть определены по табл. 7.1.

Таблица 7.1

Варианты заданий на курсовую работу

Варианты заданий на курсовую работу		Номер задания по п. 6.1.2					
		1	2	3	4	5	6
Номер задания по п. 6.1.1	1	В.1	В. 7	В. 13	В. 19	В. 25	В. 31
	2	В. 2	В. 8	В. 14	В. 20	В. 26	В. 32
	3	В. 3	В. 9	В. 15	В. 21	В. 27	В. 33
	4	В. 4	В. 10	В. 16	В. 22	В. 28	В. 34
	5	В. 5	В. 11	В. 17	В. 23	В. 29	В. 35
	6	В. 6	В. 12	В. 18	В. 24	В. 30	В. 36

7.1.1. Описание шаблона контейнерного класса

Необходимо описать шаблонный класс одной из предложенных абстрактной структуры данных:

1) **Стек**. Эта структура данных представляет собой линейный список, доступ к элементам которого осуществляется по принципу LIFO («последним вошел – первым вышел»). Две основные операции − извлечение и добавление элемента в список, также должны быть определены конструкторы списка и деструктор.

2) **Очередь**. Эта структура данных представляет собой линейный список, доступ к элементам которого осуществляется по принципу FIFO («первым вошел – первым вышел»). Две основные функции обслуживания очереди: функция, которая помещает элемент в конец очереди, и функция, которая извлекает из очереди первый элемент и возвращает его значение. Также должны быть определены конструкторы списка и деструктор.

3) **Бинарное дерево**. По определению бинарного дерева каждый его элемент содержит собственную информацию и ссылки на левое и правое поддеревья, растущие из этого элемента. Корнем (root) называется первый элемент дерева. Элементы данных называются узлами (node). Узлы, не имеющие наследников, называются листьями (leaf), фрагмент дерева называется поддеревом (subtree) или ветвью. Высота дерева (height) определяется количеством уровней, на которых располагаются его узлы. Необходимо описать конструкторы, деструктор, функции добавления/удаления узла дерева, функцию симметричного обхода дерева с целью поиска элемента.

4) **Множество** − массив элементов, среди которых нет одинаковых. Если один массив получается из другого массива перестановкой элементов, то множества, которые представляются этими массивами, считаются тождественными. Для множеств должны быть определены операции объединения, пересечения, разности, а также конструкторы и деструктор.

5) **Ассоциативный массив** – эта структура данных позволяет хранить пары вида «ключ, значение» и поддерживает операции добавления пары, а также поиска и удаления пары по ключу: добавить (ключ, значение), найти (ключ), удалить (ключ). Предполагается, что ассоциативный массив не может хранить две пары с одинаковыми ключами. Необходимо также описать конструкторы и деструктор.

6) **Кольцевой двухсвязный список**. Подобный список имеет поля с данными и два указателя: один указатель хранит адрес предшествующего элемента списка, второй – адрес последующего элемента. При этом последний элемент связан с первым. Вполне естественно для работы с двухсвязным списком использовать два указателя, хранящие адреса начала и конца такого списка. Необходимо описать конструкторы, деструктор, функции добавления, удаления элемента списка, сортировки списка.

7.1.2. Общая задача на разработку классов

Разработать указанный в варианте класс или иерархию классов и написать программу, которая будет демонстрировать решение поставленной задачи. Класс должен содержать не менее трех конструкторов (конструктор без параметров, конструктор копирования и один или несколько конструкторов с параметрами), деструктор, не менее пяти методов и десяти перегруженных операций, должен быть реализован механизм инкапсуляции, продемонстрирована работа механизмов отладки приложения.

Варианты заданий:

1) Разработать класс «Граф» в виде списка смежности. Определить конструкторы и деструктор. Переопределить операции ввода-вывода. Написать методы проверки связности графа, проверки полноты графа, проверки двудольности графа, получения дополнения графа, нахождения источника графа, нахождения стока графа. Наследовать от этого класса класс «Взвешенный граф». Написать методы получения кратчайшего пути между двумя вершинами, получения каркаса минимального веса.

2) Создать иерархию классов «Вагоны пассажирского поезда» с разделением на купейные, плацкартные, СВ. Каждый класс вагона должен содержать информацию о количестве мест разных типов (нижнее, верхнее, нижнее боковое, верхнее боковое), о наличии дополнительных услуг и ценах на них. С помощью виртуальных функций получить полный доход от эксплуатации вагона. Создать класс «Пассажирский поезд», который хранит список вагонов. Подсчитать выручку от одного рейса поезда.

3) Описать класс «Каталог библиотеки». Каждая запись каталога содержит информацию о книге – название, автор, количество экземпляров, количество экземпляров «на руках». Предусмотреть возможность формирования каталога с клавиатуры и из файла, печати каталога, сохранения в

файл, поиска книги по какому-либо признаку (например, автору или названию), добавления книг в библиотеку, удаления книг из нее, операции получения или возврата книги читателем.

4) Описать класс «Расписание приема пациентов». Каждая запись содержит дату, время, фамилию пациента. Время приема одного пациента должно быть равно одному часу. Предусмотреть возможность формирования расписания с клавиатуры и из файла, печати всего расписания, или расписания в конкретный день, добавления и удаления записей, сохранения в файл. При добавлении записи следует учитывать, что время записи должно быть свободно (не существует уже созданной записи с этим же временем).

5) Разработать класс «Многоугольник», который хранится в виде массива его вершин. Определить конструктор, методы для организации ввода, вывода и переопределить операции сравнения многоугольников по площади. Написать методы вычисления площади многоугольника, определения, принадлежит ли точка многоугольнику, определения, является ли многоугольник выпуклым.

6) Разработать класс «Полином», в котором информация о коэффициентах хранится в виде списка. Реализовать для класса методы ввода-вывода, сложения и умножения полиномов, умножения полинома на число, интегрирования и дифференцирования полинома.

7.2. Пример выполнения курсовой работы

Задача 1. Необходимо определить параметризованный контейнерный класс – массив. При записи и чтении его элементов необходимо контролировать выход за границы массива. Определим конструктор, деструктор и функцию вывода на экран элемента массива.

Решение

```
#include <conio.h>
#include <iostream.h>    //библиотека потокового ввода-вывода
#include <stdlib.h>      //стандартная библиотека

template <class Atype> class array
{
     Atype *a;   // элементы массива
     int length; // число элементов
public:
          array(int size);         //конструктор
          ~array() {delete [] a;}  //деструктор
     Atype& operator[] (int i);    //получение элемента массива
};

template <class Atype>
```

```
array <Atype>:: array (int size) // конструктор
{
     int i; length = size;
     a = new Atype[size];          // выделение памяти
     if(!a)   { cout << "\nнет памяти для массива";
            exit(1);
}
     for(i=0; i<size; i++) a[i] = 0; // запись нулей
}
template <class Atype>

Atype& array <Atype>:: operator[](int i)
{
     if(i<0 ||  i > length-1)
     {
          cout << "\nзначение с индексом " << i;
          cout << " выходит за пределы массива";
          exit(1);
     }
return a[i];
}
```

Задача 2. Разработать класс объектов «Квадратная матрица». Класс должен содержать не менее трех конструкторов (конструктор без параметров, конструктор копирования и один или несколько конструкторов с параметрами), деструктор, не менее пяти методов и десяти перегруженных операций. Класс также должен содержать статические компоненты и функции. Составить демонстрационную программу.

Решение

Исходя из задания, класс должен иметь следующие поля:

1) Поле, хранящее значение размерности матрицы.
2) Указатель на массив элементов матрицы.
3) Статический элемент – размерность матрицы по умолчанию.

Для указателя был выбран тип float, чтобы не ограничивать элементы матрицы только целочисленными значениями и иметь возможность нахождения обратной матрицы.

Были разработаны конструкторы и деструктор класса. Деструктор представляет собой стандартную конструкцию. Конструктор по умолчанию обращается к статическому полу и создает матрицу размерности, указанной в статическом поле. Конструктор с параметром имеет входной параметр – целочисленное значение размерности матрицы.

Далее были определены операции, производимые над классом:

1) операция присваивания;
2) операция сложения;
3) операция вычитания;
4) операция произведения;

5) операция произведения матрицы на число;
6) операция идентификации элемента матрицы;
7) операция транспонирования матрицы;
8) операция нахождения норм матрицы;
9) операция обращения матрицы в нулевую или единичную;
10) операция нахождения определителя матрицы;
11) операция нахождения обратной матрицы.

Операция присваивания выполняет действия, аналогичные конструктору копирования. Отличие состоит в том, что данная операция не создает новую матрицу, а производит действия внутри уже инициализированной матрицы.

Операция идентификации элемента матрицы позволяет обращаться к элементам напрямую, а не через указатель на массив элементов.

Остальные операции являются математическими и для их реализации использовались описанные в математике алгоритмы и определения. Далее приведены основные из них.

Существуют три нормы матрицы:

1) первая норма матрицы представляет собой максимальное из чисел, полученных при сложении всех элементов каждого столбца, взятых по модулю;

2) вторая норма матрицы представляет собой квадратный корень из суммы квадратов всех элементов матрицы;

3) третья норма матрицы представляет собой максимальное из чисел, полученных при сложении всех элементов каждой строки, взятых по модулю.

Наиболее сложными операциями класса являются операции нахождения определителя и обратной матрицы.

Определитель (или детерминант) − одно из основных понятий линейной алгебры. Определитель матрицы является многочленом от элементов квадратной матрицы (то есть такой, у которой количество строк и столбцов равны). Определитель матрицы A обозначается как: det(A), |A| или Δ(A).

В разработанном классе перегруженная операция, вычисляющая определитель матрицы, обращается к рекурсивной функции, выполняющей основные действия, так как наиболее эффективный алгоритм вычисления определителя является рекурсивным и требует наличия двух входных параметров – массива элементов матрицы и ее размерности. В то же время перегружаемая для нахождения определителя операция «!» является унарной и не предполагает наличие входных параметров.

Квадратная матрица B называется обратной для квадратной матрицы A того же порядка, если их произведение A× B = B, A = E, где E − единичная матрица того же порядка, что и матрицы A и B. Алгоритм нахождения обратной матрицы представлен на рис. 7.1.

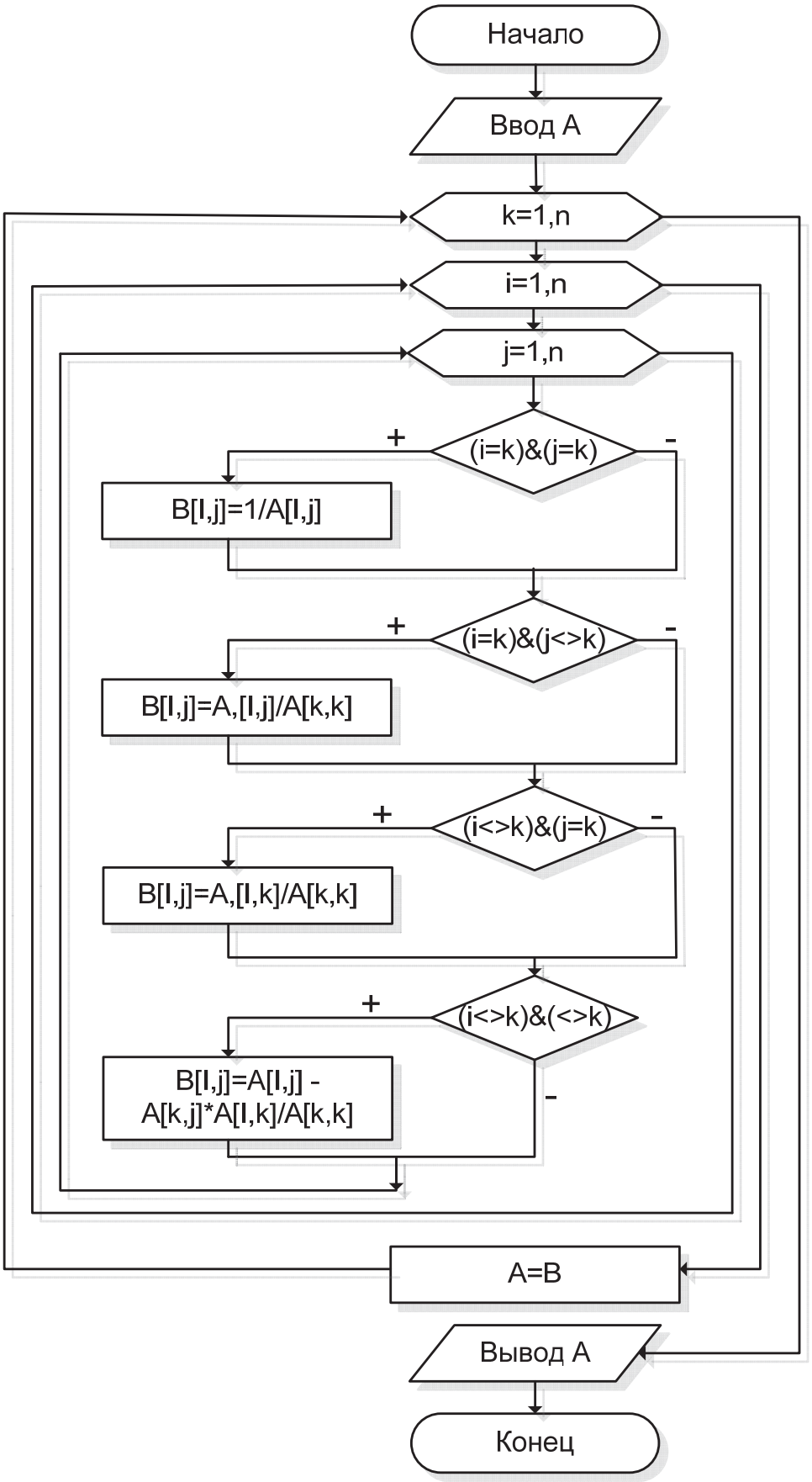

Рис. 7.1. Алгоритм нахождения обратной матрицы

На следующем этапе разработки класса были определены основные методы класса:

1) нахождение максимального элемента матрицы;
2) нахождение минимального элемента матрицы;
3) вывод на экран диагонали;
4) заполнение матрицы;
5) вывод матрицы на экран.

В реализации данные методы просты и не требуют детального описания. Также в классе были описаны функции для работы со статическим значением – это функции чтения и записи значения размерности матрицы по умолчанию.

В программе объявление класса выглядит следующим образом:

```
class Q_Matrix
{
  int n;
  float**ar;
public:
  Q_Matrix();//конструктор по умолчанию
  Q_Matrix(int);//конструктор с параметром
  Q_Matrix(Q_Matrix &); //конструктор копирования
  ~Q_Matrix(); //дестрктор
     void operator=(Q_Matrix);//операция присваивания
     Q_Matrix operator+(Q_Matrix &);// сумма матриц
     Q_Matrix operator-(Q_Matrix &);//разность матриц
     float *operator[](int);//операция идентификации элемента
     Q_Matrix operator*(Q_Matrix &);//перемножение матриц
     void operator*(float); // умножение матрицы на число
     void operator~(); //Транспонирование
     float operator()(int);//нормы матрицы
     void operator^(int);// Обращение в нулевую или единичную
     float operator!();//det(A)
     Q_Matrix operator-();//обратная матрица
  float Max();
  float Min();//нахождение минимального элемента матрицы
  void show_diag(); //вывод на экран диагонали матрицы
  void add(); //заполнение матрицы
  void show();//вывод на экран матрицы
friend float det(float **, int);
static int count; //количество элементов матрицы, для которых
вызывается конструктор по умолчанию
static int get_count(); // функция чтения количества элементов
матрицы по умолчанию
static void change_count(int);// функция замены количества
элементов матрицы по умолчанию
};
```

В ходе работы было создано консольное приложение, позволяющее пользователю работать с классом квадратичных матриц.

Для удобства пользователя при помощи функций ввода/вывода должен быть организован диалоговый пользовательский интерфейс.

В приложении может быть реализовано меню, состоящее из следующих пунктов:

1) сумма матриц;
2) разность матриц;
3) произведение матриц;
4) умножение матрицы на число;
5) транспонирование;
6) нахождение норм матрицы;
7) обращение матрицы;
8) найти минимальный элемент;
9) найти максимальный элемент;
10) вывести на экран диагональ;
11) вывести на экран элемент массива;
12) найти обратную матрицу;
13) найти определитель матрицы;
14) изменить значение ‘count’;
15) другое.

Каждый из пунктов меню в диалоге с пользователем демонстрирует работу функций и методов класса. В пункте «Другое» должна быть наглядно продемонстрирована работа конструктора по умолчанию, конструктора копирования и деструктора. После выполнения каждой операции программа предлагает пользователю выйти из программы либо продолжить работу.

Далее представлен общий код описания класса:

```
#include <vcl.h>
#pragma hdrstop
#include <conio.h>
#include <stdio.h>
#include <math.h>
#include <iostream.h>
class Q_Matrix
{
  int n;
  float**ar;
public:
  Q_Matrix();//конструктор по умолчанию
  Q_Matrix(int);//конструктор с параметром
  Q_Matrix(Q_Matrix &); //конструктор копирования
  ~Q_Matrix(); //дестрктор
     void operator=(Q_Matrix);//операция присваивания
```

```
     Q_Matrix operator+(Q_Matrix &);// сумма матриц
     Q_Matrix operator-(Q_Matrix &);//разность матриц
     float *operator[](int);//операция идентификации элемента
     Q_Matrix operator*(Q_Matrix &);//перемножение матриц
     void operator*(float); // умножение матрицы на число
     void operator~(); //Транспонирование
     float operator()(int);//нормы матрицы
     void operator^(int);// Обращение в нулевую или единичную
     float operator!();//det(A)
     Q_Matrix operator-();//обратная матрица
  float Max();
  float Min();//нахождение минимального элемента матрицы
  void show_diag(); //вывод на экран диагонали матрицы
  void add(); //заполнение матрицы
  void show();//вывод на экран матрицы
friend float det(float **, int);
static int count; //количество элементов матрицы, для которых
вызывается конструктор по умолчанию
static int get_count(); // функция чтения количества элементов
матрицы по умолчанию
static void change_count(int);// функция замены количества
элементов матрицы по умолчанию
};
  int Q_Matrix::count=5;

void Q_Matrix::change_count(int x)
  {
   count=x;
  }

int Q_Matrix::get_count()
  {
    return count;
   }

Q_Matrix::Q_Matrix()
 {
     int i;
     n=count;
     ar=new float*[n];
     for(i=0;i<n;i++) ar[i]=new float[n];
  }
Q_Matrix::Q_Matrix(int k)
  {
    int i;
    if (k>0)
    {
     n=k;
     ar=new float*[n];
        for(i=0;i<n;i++) ar[i]=new float[n];
```

```
    }
  }
Q_Matrix::Q_Matrix (Q_Matrix &t)
  {
   int i,j;
   n=t.n;
   ar=new float*[n];
   for(i=0;i<n;i++) ar[i]=new float[n];
      for(i=0;i<n;i++)
         for(j=0; j<n; j++)
         {
           ar[i][j]=t.ar[i][j];
         }
   }

  Q_Matrix::~Q_Matrix()
  {
   int i;
   for(i=0;i<n;i++)
      delete[]ar[i];
   delete[]ar;
  }
 void Q_Matrix::add()
  {
    int i,j;
    for(i=0;i<n;i++)
      {
       cout<<"\n";
       for(j=0;j<n;j++)
       {
         cout<<"  "<<"["<<i<<"]["<<j<<"]"<<" = ";
         cin>>ar[i][j];
       }
      }
  }
 void Q_Matrix::show()
  {
   int i,j;
   for (i=0;i<n;i++)
    {
      cout<<"\n";
      for(j=0;j<n;j++)
      {
        cout<<" "<<ar[i][j];
      }
    }
  }
void Q_Matrix::operator=(Q_Matrix t)
  {
    int i,j;
```

```
    for(i=0;i<n;i++)
        delete ar[i];
    delete[]ar;
    ar=new float*[t.n];
    n=t.n;
    for(i=0;i<n;i++)
       {
       ar[i]=new float[n];
       for(j=0;j<n;j++)
          ar[i][j]=t.ar[i][j];
       }
  }
Q_Matrix Q_Matrix::operator+(Q_Matrix &t)
  {
    int i,j;
    Q_Matrix C(t.n);
      for(i=0;i<n;i++)
         for(j=0;j<n;j++)
             C.ar[i][j]=ar[i][j]+t.ar[i][j];
    return C;
  }
Q_Matrix Q_Matrix::operator-(Q_Matrix &t)
  {
    int i,j;
    Q_Matrix C(t.n);
      for(i=0;i<n;i++)
         for(j=0;j<n;j++)
             C.ar[i][j]=ar[i][j]-t.ar[i][j];
    return C;
  }
float* Q_Matrix::operator[](int k)
  {
   return ar[k];
  }
Q_Matrix Q_Matrix::operator*(Q_Matrix &t)
  {
   int i,j,k,sum;
   Q_Matrix C(n);
     for(i=0;i<n;i++)
        for(j=0;j<n;j++)
        {
          sum=0;
            for(k=0;k<n;k++)
            sum=sum+ar[i][k]*t.ar[k][j];
          C[i][j]=sum;
        }
    return C;
  }
void Q_Matrix::operator~()
  {
```

```
    int i,j;
    Q_Matrix C(n);
    for (i=0; i<n; i++)
        for (j=0; j<n; j++)
            C.ar[i][j]=ar[j][i];
    for (i=0; i<n; i++)
        for (j=0; j<n; j++)
            ar[i][j]=C.ar[i][j];
   }
void Q_Matrix::operator*(float k)
  {
    int i,j;
    for (i=0; i<n; i++)
        for (j=0; j<n; j++)
            ar[i][j]=ar[i][j]*k;
   }
float Q_Matrix::operator()(int k)
  {
    int i,j, norm;
    float sum, p_norm=0;
    norm=k;
    if(norm==1)
    {
       for (j=0; j<n; j++)
       {
         sum=0;
            for (i=0; i<n; i++)
                sum=sum+fabs(ar[i][j]);
         if (p_norm<sum) p_norm=sum;
       }
    }
    else if(norm==2)
    {
      sum=0;
      for (j=0; j<n; j++)
            for (i=0; i<n; i++)
                sum=sum+pow(ar[i][j],2);
       p_norm=sqrt(sum);
    }
    else if(norm==3)
    {
       for (i=0; i<n; i++)
       {
         sum=0;
         for (j=0; j<n; j++)
                sum=sum+fabs(ar[i][j]);
         if (p_norm<sum) p_norm=sum;
       }
    }
    return p_norm;
```

```
  }
void Q_Matrix::operator^(int k)
  {
    int i,j;
    for (i=0; i<n; i++)
        for (j=0; j<n; j++)
            ar[i][j]=k;
  }
void Q_Matrix::show_diag()
  {
   int i,j;
   cout<<"\n";
   for (i=0; i<n; i++)
        for (j=0; j<n; j++)
           if(i==j) cout<<ar[i][j]<<" ";
  }
float Q_Matrix::Min()
  {
   int i,j;
   float min;
   min=ar[0][0];
   for (i=0; i<n; i++)
        for (j=0; j<n; j++)
           if(ar[i][j]<min) min=ar[i][j];
   return min;
  }
float Q_Matrix::Max()
  {
   int i,j;
   float max;
   max=ar[0][0];
   for (i=0; i<n; i++)
        for (j=0; j<n; j++)
           if(ar[i][j]>max) max=ar[i][j];
   return max;
  }
float det(float **ar, int n) //рекурсивная функция для вычис-
ления определителя
  {
   int i,j,k,m,m1;
   float **a, d=0;
     if(n==2) d=ar[0][0]*ar[1][1]-ar[0][1]*ar[1][0];
     else
     {
       a=new float*[n-1];
       for(i=0;i<n;i++)
       {
          for(j=0;j<n-1;j++)
          {
             if(j<i) a[j]=ar[j];
```

```
            else a[j]=ar[j+1];
          }
          d=d+pow(-1,(i+j))*det(a,n-1)*ar[i][n-1];
        }
        delete a;
      }
      return d;
  }
float Q_Matrix::operator!()
  {
    float determ;
    determ=det(ar,n);
    return determ;
  }

Q_Matrix Q_Matrix::operator-()
  {
    int i,j,k;
    float temp;
    Q_Matrix C(n);
    Q_Matrix E(n);
       for (int i=0;i<n;i++)
        for (int j=0;j<n;j++)
        {
            E[i][j]=0;
            C[i][j]=ar[i][j];
            if (i==j)
                E[i][j]=1;
        }
            for (int k=0;k<n;k++)
           {
             temp = C[k][k];
             for (int j=0;j<n; j++)
               {
                 C[k][j] /= temp;
                 E[k][j] /= temp;
               }
                  for (int i=k+1;i<n;i++)
                 {
                      temp = C[i][k];
                      for (int j=0;j<n;j++)
                      {
                         C[i][j]-=C[k][j]*temp;
                         E[i][j]-=E[k][j]*temp;
                      }
                  }
            }
          for(int k=n-1;k>0;k--)
          {
            for (int i=k-1;i>=0;i--)
```

```
            {
              temp = C[i][k];
              for (int j=0;j<n;j++)
              {
                C[i][j]-=C[k][j]*temp;
                E[i][j]-=E[k][j]*temp;
              }
             }
           }
        for (int i=0;i<n;i++)
        for (int j=0;j<n;j++)
             C[i][j]=E[i][j];
  return C;
}
```

8. КОНТРОЛЬНЫЕ ВОПРОСЫ И ТЕСТЫ

1) Укажите языки программирования, которые в той или иной степени были охвачены во время изучения курса.

2) Укажите средства, механизмы, функции, технологии, которые Вы изучили в ходе прохождения данного курса?

3) Назовите базовые принципы объектно-ориентированного программирования.

4) Через какие средства в объектно-ориентированных языках программирования выражена абстракция?

5) Через какие средства в объектно-ориентированных языках программирования выражена инкапсуляция?

6) Через какие средства в объектно-ориентированных языках программирования выражен полиморфизм?

7) Через какие средства в объектно-ориентированных языках программирования выражено наследование?

8) Укажите недостатки одиночного/множественного наследования.

9) Укажите недостатки механизма интерфейсов.

10) Какими средствами можно воспользоваться для того, чтобы избежать проблем с множественным наследованием? В каких языках реализованы такие средства?

11) Дайте определение понятию «абстрактный тип данных».

12) Что такое обработка исключений? Какую проблему решает использование данного механизма?

13) Перечислите языковые средства отладки программы.

14) Что такое контейнерные классы?

15) Шаблоны классов и функций – для чего они применяются? Какие проблемы позволяют разрешить?

16) Что такое роли (миксины, типажи)? Где они применяются?

17) Перечислите виды реализации инкапсуляции.

18) Что такое перегрузка операций? Для чего она применяется?

19) Какие роли выполняет наследование?

20) Какие виды наследования возможны в C++?

21) Чем отличается модификатор доступа protected от модификаторов private и public?

22) Можно ли в производном классе объявлять новые поля? А методы?

23) Если имя нового поля совпадает с именем унаследованного, то каким образом можно разрешить конфликт имен?

24) Что происходит, если имя метода-наследника совпадает с именем базового метода?

25) Объясните, зачем нужны виртуальные функции.

26) Как реализован полиморфизм в C++?

27) Дайте определение абстрактного класса. Для чего применяются абстрактные классы?

28) Метод это:

а) функция, получающая в качестве обязательного параметра указатель на объект;

б) структура, хранящая указатели this, parrent, supper;

в) структурная переменная, содержащая всю информацию о некотором физическом предмете или реализуемом в программе понятии;

г) определенный программистом абстрактный тип данных.

29) Объект это:

а) функция, получающая в качестве обязательного параметра указатель на объект;

б) структура, хранящая указатели this, parrent, supper;

в) структурная переменная, содержащая всю информацию о некотором физическом предмете или реализуемом в программе понятии;

г) определенный программистом абстрактный тип данных.

30) Класс это:

а) структурная переменная, содержащая всю информацию о некотором физическом предмете или реализуемом в программе понятии;

б) определенный программистом абстрактный тип данных;

в) переменная, описанная абстрактным типом данных.

31) Когда функция определяется независимо в каждом производном классе и имеет в этих классах общее имя?

а) При инкапсуляции.

б) При полиморфизме.

в) При наследовании и инкапсуляции.

32) Почему в некоторых языках программирования отказываются от поддержки множественного наследования (имеется в виду наследование реализации)?

а) Поддержка множественного наследования ведет к большим потерям производительности, так как для каждого класса необходимо держать сильноветвящуюся иерархию его предков.

б) Множественное наследование практически никогда не используется, в отличие от обычного наследования от одного класса.

в) Множественное наследование невозможно реализовать с помощью таблицы виртуальных функций, поэтому требуются другие намного более сложные алгоритмы.

г) Из-за неоднозначности выбора поведения, в случае если суперклассы некоторого класса содержат методы с одинаковыми сигнатурами.

33) Как называется способность объекта скрывать свои данные и реализацию от других объектов системы?

а) Полиморфизм.

б) Инкапсуляция.

в) Абстракция.

г) Наследование.

34) Драконы умеют летать (как, например, птицы) и ползать (как, например, ящерицы). С точки зрения ООП, примером чего является данная ситуация (выберите наиболее точный вариант)?

а) Инкапсуляция.

б) Композиция.

в) Наследование.

г) Множественное наследование.

д) Полиморфизм.

9. ЭКЗАМЕНАЦИОННЫЕ ВОПРОСЫ И ЗАДАЧИ

Экзаменационная оценка определяется результатом ответа на экзаменационный билет, который содержит два теоретических вопроса и практическую задачу. Перечни типовых вопросов и задач приведены ниже.

Теоретические вопросы

1) Место объектно-ориентированной парадигмы программирования в системе языков программирования.

2) История развития представлений о программировании.

3) Формирование объектной модели.

4) Парадигмы программирования. Машинное кодирование. Ассемблирование.

5) Парадигмы программирования. Процедурное и логическое программирование.

6) Парадигмы программирования. Структурная и функциональная парадигмы.

7) Объектно-ориентированная парадигма программирования.

8) Концептуальная база объектно-ориентированной парадигмы. Базовые принципы ООП.

9) Концептуальная база объектно-ориентированной парадигмы. Дополнительные принципы ООП.

10) Концептуальная база объектно-ориентированной парадигмы. Принципы ООП согласно Алану Кею.

11) Объектно-ориентированные языки программирования.

12) Классификация объектно-ориентированных языков программирования.

13) Абстракция как группа механизмов объектно-ориентированного программирования.

14) Классы и объекты.

15) Статические элементы класса.

16) Класс как структура. Тело класса и составные функции.

17) Конструктор и деструктор.

18) Виды конструктора. Примеры.

19) Контейнерные классы.

20) Абстрактные структуры данных.

21) Контейнеры в языках с динамической типизацией.

22) Шаблоны классов.

23) Шаблоны функций.

24) Наследование как группа механизмов объектно-ориентированного программирования.

25) Одиночное наследование.

26) Множественное наследование.

27) Недостатки множественного наследования.

28) Абстрактные классы и интерфейсы.

29) Роли (типажи).

30) Инкапсуляция как группа механизмов объектно-ориентированного программирования. Примеры инкапсуляции.

31) Модификаторы private, public, protected.

32) Механизмы инкапсуляции в языках программирования.

33) Полиморфизм как группа механизмов объектно-ориентированного программирования.

34) Перегрузка операторов.

35) Виртуальные функции.

36) Синтаксис. Структура программ.

37) Средства отладки приложений.
38) Обработка исключительных ситуаций.
39) Шаблоны как средство отладки приложения.
40) Проектирование по контракту.

Практические задачи

1) Разработать класс «Комплексное число». Определить в нем конструктор, перегрузить операции сложения и умножения, операции сравнения, статический метод получения комплексного числа из строки. Протестировать все возможности класса.

2) Разработать класс «Комплексное число в тригонометрической форме». Определить в нем конструктор, перегрузить операции вычитания и умножения, операции сравнения и статический метод получения комплексного числа из строки. Протестировать все возможности класса.

3) Разработать класс «Комплексное число», в котором данные хранятся в двух видах: алгебраической и тригонометрической формах. Определить в нем конструкторы и деструктор, операцию преобразования в строку и статический метод получения комплексного числа из строки, написать методы преобразования числа из одной формы в другую. Протестировать все возможности класса.

4) Разработать класс «Дата». Определить в нем конструкторы и деструктор, перегрузить операцию добавления к дате заданного количества дней, операцию вычитания двух дат, операции сравнения. Протестировать все возможности класса.

5) Разработать класс «Время». Определить в нем конструкторы и деструктор, перегрузить операцию добавления к времени заданного количества минут, операцию вычитания двух моментов времени. Протестировать все возможности класса.

6) Разработать класс «Множество». Определить конструкторы и деструктор. Переопределить операции объединения, пересечения и разности двух множеств, методы для организации ввода-вывода. Протестировать все возможности класса.

7) Разработать класс «Бинарное дерево сортировки». Написать конструкторы и деструктор, методы добавления нового узла, удаления узла по ключевому значению, вычисления глубины дерева, объединения двух деревьев. Протестировать все возможности класса.

8) Разработать класс «Граф» в виде списка смежности. Определить конструкторы и деструктор. Переопределить операции ввода-вывода. Наследовать от этого класса класс «Взвешенный граф». Протестировать все возможности класса.

9) Создать иерархию классов «Вагоны пассажирского поезда» с разделением на купейные, плацкартные, СВ. Каждый класс вагона должен

содержать информацию о количестве мест разных типов (нижнее, верхнее, нижнее боковое, верхнее боковое), о наличии дополнительных услуг и ценах на них. Протестировать все возможности класса.

10) Разработать класс «Полином», в котором информация о коэффициентах хранится в виде списка. Реализовать для класса методы ввода-вывода, умножения полинома на число, сложения полиномов.

11) Описать класс «Каталог библиотеки». Каждая запись каталога содержит информацию о книге – название, автор, количество экземпляров, количество экземпляров «на руках». Предусмотреть возможность добавления книг в библиотеку, удаления книг из нее, операции получения или возврата книги читателем.

12) Описать класс «Расписание занятий». Каждая запись содержит день недели, время, название учебной дисциплины, аудиторию. Предусмотреть возможность вывода расписания и расписания на конкретный день, добавления и удаления записей.

13) Описать класс «Расписание приема пациентов». Каждая запись содержит дату, время, фамилию пациента. Время приема одного пациента должно быть равно одному часу. Предусмотреть возможность формирования расписания. При добавлении записи следует учитывать, что время записи должно быть свободно (не существует уже созданной записи с этим же временем).

14) Разработать класс «Треугольник». Определить в нем конструкторы и деструктор, перегрузить операцию проверки включения точки в треугольник, операцию сравнения треугольников (по площади).

15) Разработать класс «Прямоугольник». Определить в нем конструкторы и деструктор, перегрузить операцию вычисления площади прямоугольника, операцию сравнения (по площади).

16) Разработать класс «Многоугольник», который хранится в виде массива его вершин. Определить конструктор, методы для организации ввода, вывода и переопределить операции сравнения многоугольников по площади.

17) Разработать класс «Фигура на плоскости». Определить для него все виды конструкторов, деструктор, перегрузить операцию определения площади.

18) Определить класс m x n – матрицы целых чисел. Конструктор инициирует коэффициенты нулями, по умолчанию одного из аргументов строит матрицу как состоящую из одного столбца. Перегрузить операции вывода на экран значения элемента с заданными индексами, умножения коэффициентов матрицы на целое число. Разработать подпрограмму (функцию), возвращающую количество объектов, находящихся в области видимости.

19) Определить класс, объектами которого являются n x n – матрицы, где n – статический элемент класса. Определить операции сложения,

умножения на число. Конструктор строит матрицу по указанным коэффициентам.

20) Создать иерархию классов-многоугольников: «Треугольник», «Четырехугольник», «Пятиугольник», «Шестиугольник». Создать класс «Фигура на плоскости», который задает фигуру как массив объектов-многоугольников. Определить в классе методы перемещения фигуры, определения, принадлежит ли точка фигуре и др.

Пример экзаменационного билета

1 Формирование объектной модели.

2 Класс как структура. Тело класса и составные функции.

Задача: разработать класс «Фигура на плоскости». Определить для него все виды конструкторов, деструктор, перегрузить операцию определения площади.

Примечание. При ответе на вопросы обязательно следует приводить примеры, подтверждающие ответ.

Пример решения экзаменационной задачи

Задача. Создать класс «Трехмерный вектор», инкапсулирующий свойства точки в двумерном пространстве (класс «Точка»). Класс должен содержать не менее двух полей. Класс должен содержать не менее одного конструктора с аргументами, а также стандартный конструктор и деструктор. Во всех конструкторах и деструкторе должны находиться индикационные сообщения. Предусмотреть в классе функцию вывода на экран полей класса. Предусмотреть в классе функцию вывода на экран полей класса, которая должна вызывать аналогичную функцию базового класса.

Решение

Для выполнения задания запустим программу Borland C++ 6.0 и напишем программу для работы с классами объектов «Точка» и «Трехмерный вектор».

Опишем базовый класс объектов «Точка»:

```
class point
{
protected:
  int x;
  int y;
public:
//конструктор по умолчанию
   point()
  {
    cout << "\n\nКонструктор по умолчанию";
    x=0; y=0;
```

```
    cout << "\nСоздана точка(" <<0<<", "<<0<<")";
  }
 //конструктор с параметром
point(int p, int q)
  {
   cout << "\n\nКонструктор с параметром";
      x=p;
      y=q;
      cout << "\nСоздана точка(" <<p<<", "<<q<<")";
  }
//деструктор
  ~point()
  {
    cout << "\n\nУдаление точки";
    cout << "\nТочка удалена\n";
  }
// функция изменения точки
void add()
    {
      cout << "Введите x ";
      cin>>x;
      cout << "Введите y ";
      cin>>y;
    }
// Функция вывода точки на экран
  void show()
  {
        cout<<" "<<x;
        cout<<" "<<y;
  }

};
```

В данном случае мы используем тип protected для полей класса, следовательно, они будут доступны для составных и дружественных функций классов, которые являются производными от этого класса.

Далее создадим производный класс «Трехмерный вектор»:

```
class vector3 : public point
 {
   float z;
 public:
 //конструктор по умолчанию
   vector3()
  {
    cout << "\n\nКонструктор по умолчанию";
    z=0;
    cout << "\nСоздан вектор(" <<x<<", "<<y<<", "<<z<<")";
  }
//конструктор с параметром
```

```
  vector3(int p, int q, float j): point(p,q)
  {
   cout << "\n\nКонструктор с параметром";
       z=j;
   cout << "\nСоздан вектор(" <<x<<", "<<y<<", "<<z<<")";
  }
//деструктор
  ~vector3()
  {
    cout << "\n\nУдаление вектора";
    cout << "\nВектор удален\n";
  }
// функция изменения вектора
 void add()
  {
      point::add();
      cout << "Введите z ";
      cin>>z;
    }
//Функция вывода на экран информации о векторе
  void show()
  {
        point::show();
        cout<<" "<<z;
  }
 };
```

Так как мы указали область видимости в производном классе как public, мы сможем использовать поля и функции базового класса в соответствии с областью видимости в базовом классе. Согласно заданию был создан конструктор по умолчанию, конструктор с параметром, деструктор и функции чтения/записи вектора.

Следует отметить, что при создании конструктора не обязательно указывать значения полей базового класса, они будут инициализированы при помощи конструктора по умолчанию базового класса.

Для того чтобы создать вектор по параметрам трех координат, мы используем следующую запись: vector3(int p, int q, float j): point(p,q). Здесь мы часть параметров передаем конструктору с параметром базового класса, и конструктор по умолчанию при этом не вызывается.

Для того чтобы при вызове функций производного класса автоматически вызывались и функции базового класса, используется конструкция <Имя базового класса>::<Функция базового класса>.

Проконтролируем работу конструкторов и функций класса, прописав в модуле main соответствующие функции для динамически созданных объектов производного класса:

```
void main ()
{
   cout << "***Программа для работы с координатами точек***";
 vector3 *v1, *v2;
 v1=new vector3;
 v2=new vector3(5,7,9);
 v1->add();
 v1->show();
 delete v2;
 delete v1;
 getch();
}
```

В программе сначала инициализируется вектор по умолчанию, затем вектор с параметрами, происходит изменение значений координат в первом векторе, выводятся на экран результаты изменения, затем происходит удаление объектов.

ЗАКЛЮЧЕНИЕ

В пособии изложены основные принципы объектно-ориентированного программирования, применяемые в различных языках программирования. Рассмотрены противоречия, возникающие по мере развития соответствующих языков программирования. Показано, как эти противоречия устраняются инструментами ТРИЗ. Такого рода информация, безусловно, окажется полезной не только для студентов, но и для преподавателей и специалистов, работающих в области программирования.

Пособие наглядно, в виде ТРИЗ эволюционных карт, иллюстрирует развитие объектно-ориентированных языков программирования, а также эволюцию их механизмов.

Следует отметить, что в пособии рассмотрены не все объектно-ориентированные языки программирования, известные в настоящее время, а только наиболее популярные. Читателям, которые захотят более серьезно освоить ТРИЗ эволюционный подход, авторы рекомендуют ознакомиться с дополнительной литературой, которая указана ниже.

БИБЛИОГРАФИЧЕСКИЙ СПИСОК

1. Покушалова, Л. В. Метод case-study как современная технология профессионально-ориентированного обучения студентов / Л. В. Покушалова // Молодой ученый. – 2011. – № 5., Т.2., с. 155-157.
2. Berdonosov, V. D. Fractality of knowledge and TRIZ / V. D. Berdonosov// Proceedings of the ETRIA TRIZ Future Conference, Kortrijk. – 2006.
3. Berdonosov, V. D. Concept of the TRIZ Evolutionary Approach in Education / V. D. Berdonosov // ETRIA TRIZ Future Conference / Arts Et Metiers ParicTech, 2013.
4. Mandelbrot, B. The Fractal Geometry of Nature / B. Mandelbrot. – New York : Freeman, 1983.
5. Berdonosov, V. D. TRIZ-Fractality of mathematics / V. D. Berdonosov, E. V. Redkolis // ELSEVIER: ScienceDirect international Journal. – 2011. – Vol. 09, pp. 461-472.
6. Berdonosov, V. D. TRIZ-fractality of computer-aided software engineering systems / V. D. Berdonosov, E. V. Redkolis // ELSEVIER: ScienceDirect international Journal. – 2011. – Vol. 09, pp. 199-213.
7. Berdonosov, V. D. TRIZ-evolution of Programming System / V. D. Berdonosov, T. Sycheva // ETRIA TRIZ Future Conference/ Institute of Technology Tallaght, 2011.
8. Фридман, А. Л. Основы объектно-ориентированной разработки программных систем / А. Л. Фридман. – М. : Финансы и статистика, 2000. – 97 с.
9. Вегнер, П. Программирование на языке Ада / П. Вегнер ; пер. с англ. Ю. Ю. Галимова, Э. М. Киуру – М. : Мир, 1983. – 239 с.
10. Буч, Г. Объектно-ориентированный анализ и проектирование с примерами приложений на C++ / Г. Буч ; пер. с англ. И. Романовский, Ф. Андреев. – М. : Бином, 1998. – 560 с.
11. Пратт, М. Языки программирования / М. Пратт, Б. Зелковитц. – 4-е изд. – М. : Питер, 2005. – 212 с.
12. Кун, Т. Структура научной революции / Т. Кун. – 3-е изд. – М. : Юниверсити Чикаго Пресс, 2001. – 346 с.
13. Себеста, Р. Основные концепции языков программирования / Р. Себеста. – М. : Вильямс, 2001.– 659 с.
14. Лайтфут, Д. Модульные языки программирования: 7-я конференция Модулярных языков / Д. Лайтфут. – М. : Лондон-Лимитед, 2010. – 30 с.
15. Фридман, П. Основы языков программирования / П. Фридман. – М. : МИТ, 2008 – 42 с.

16. Хоудак, П. Концепции, эволюция и применение функциональных языков программирования / П. Хоудак. – М. : АСМ Компьютинг, 1989. – 13 с.

17. Габриелли, М. Языки программирования: принципы и парадигмы / М. Габриелли, С. Мартини. – М. : Лондон Лимитед, 2010. – 450 с.

18. Бадд, Т. Объектно-ориентированное программирование в действии / Т. Бадд — СПб. : Питер, 1997. – 464 с.

19. Страуструп, Б. Дизайн и эволюция C++ / Б. Страуструп. – СПб. : ДМК Пресс, 2006. – 445 с.

20. Бобровский, С. История объектно-ориентированного программирования / С. Бобровский // PC Week/RE. – 2003. – № 28. – С. 19-23.

21. Подбельский, В. В. Язык Си++ / В. В. Подбельский. – М. : Финансы и статистика, 2002. – 560 с.

22. Топп, У. Структуры данных в C++ / У. Топп, У. Форд. – М. : ЗАО «Издательство БИНОМ», 1999. – 816 с.

23. Goldberg, A. Smalltalk: the language and its implementation / A. Goldberg. – Xerox Corporation, 1983. – 720 p.

24. Павловская, Т. А. C++. Объектно-ориентированное программирование: Практикум / Т. А. Павловская, Ю. А. Щупак. – СПб. : Питер, 2006. – 265 с.

25. Андрианова, А. А. Практикум по курсу «Объектно-ориентированное программирование» на языке C# : учеб. пособие / А. А. Андрианова, Л. Н. Исмагилов, Т. М. Мухтарова. – Казань : Казанский (Приволжский) федеральный университет, 2012. – 112 с.

26. Медведев, В. И. Особенности объектно-ориентированного программирования на C++/CLI, C# и Java / В. И. Медведев. – 2-е изд. – Казань : РИЦ «Школа», 2010. – 444 с.

27. Шитов, А. Знакомство с языком Perl 6 / А. Шитов // Workshop “Perl Today”, 2007. – 29 с.

28. Хусаинов, А. А. Организация ЭВМ и систем : учеб. пособие / А. А. Хусаинов. – Комсомольск-на-Амуре : ФГБОУ ВПО «КнАГТУ», 2002. – 89 с.

29. Мейер, Б. Объектно-ориентированное конструирование программных систем / Б. Мейер. – М. : Интернет-университет информационных технологий. – ИНТУИТ.ру", 2005. – 1232 с.

30. Охотников, Е. А. В поисках лучшего языка // EAO197.NAROD.RU – сайт Евгения Охотникова. 2007. URL : http://eao197.narod.ru/better_language/ index.html (дата обращения: 23.05.2012).

ПРИЛОЖЕНИЕ

СПИСОК УСЛОВНЫХ ОБОЗНАЧЕНИЙ И ТЕРМИНОВ

Далее приведен перечень основных определений и условных обозначений, принятых в тексте, в алфавитном порядке.

Абстрагирование – выделение существенных характеристик некоторого объекта, отличающих его от всех других видов объектов и, таким образом, четко определяет его концептуальные границы с точки зрения наблюдателя.

Абстрактный класс – класс, на основе которого не могут создаваться объекты.

Абстрактный тип данных – совокупность данных вместе с множеством операций, которые можно выполнять над этими данными.

Ассемблирование – парадигма, при которой происходит компиляция исходного текста программы, написанной на языке Ассемблер, в программу на машинном языке.

Ассоциированный массив – массив, доступ к элементам которого осуществляется не по номеру, а по ключу (т.е. это таблица, состоящая из пар «ключ-значение»).

Базовый класс – класс, на основе которого созданы другие классы (производные).

Бинарное дерево – динамическая структура данных, состоящая из узлов, каждый из которых содержит помимо данных не более двух ссылок на различные бинарные поддеревья.

Виртуальная функция – функция, объявленная с ключевым словом virtual в базовом классе и переопределенная в одном или в нескольких производных классах.

Группа механизмов ООП – набор выразительных средств языка или программных инструментов, которые объединены по признаку реализации одного из принципов ООП.

Дерево поиска – динамическая структура данных, организованная таким образом, что для каждого узла все ключи его левого поддерева меньше ключа этого узла, а все ключи его правого поддерева – ассоциативный массив.

Деструктор – составная функция класса, которая вызывается перед разрушением объекта

Задача программирования – обеспечение взаимодействия компьютера и человека.

Закрытые атрибуты класса – члены класса, доступные только для составных и дружественных функций этого класса.

Защищенные атрибуты класса – члены класса, доступные для составных и дружественных функций классов, которые являются производными от этого класса или совпадают с ним.

Императивный язык – язык программирования, который описывает процесс вычисления в виде инструкций, изменяющих состояние программы.

Инкапсуляция – процесс отделения друг от друга элементов объекта, определяющих его устройство и поведение.

Интерфейсы – конструкции, определяющие границу взаимодействия между классами или компонентами, специфицируя определенную абстракцию, которую осуществляет реализующая сторона.

Класс – абстрактное описание свойств и методов для совокупности похожих объектов, представители которой называются экземплярами класса.

Конструктор – составная функция класса, вызываемая при создании объекта.

Контейнерные классы – классы, в которых хранятся организованные данные.

Концепция программирования – это движущая сила для возникновения новой группы языков программирования или, другими словами, это противоречие или совокупность противоречий, разрешение которых привело к возникновению парадигмы программирования.

Линейный список – динамическая структура данных, в которой каждый элемент связан со следующим и, возможно, с предыдущим.

Логическое программирование – парадигма программирования, основанная на автоматическом доказательстве теорем.

Массив – конечная совокупность однотипных величин. Занимает непрерывную область памяти и предоставляет прямой (произвольный) доступ к элементам по индексу.

Машинное кодирование – система команд вычислительной машины, которая интерпретируется для конкретного микропроцессора.

Метод объекта – объявленная в классе процедура, описывающая поведение объекта.

Механизм – некоторый инструмент или выразительное средство языка, которое может включать в себя модели, алгоритмы, синтаксические единицы, принципы построения программы и т.д.

Множественное наследование – механизм наследования, при котором класс может быть образован от нескольких базовых классов.

Модульность – свойство системы, которая была разложена на внутренне связанные, но слабо связанные между собой модули.

Наследование – механизм, позволяющий определять новые типы данных на основе существующих таким образом, что данные и функции существующих классов становятся членами наследуемых классов.

Обработка исключений – механизм предназначен для описания реакции программы на ошибки времени выполнения и другие возможные проблемы (исключения), которые могут возникнуть при выполнении программы и приводят к невозможности (бессмысленности) дальнейшей отработки программой её базового алгоритма.

Объект – это модель или абстракция реальной сущности в программной системе (экземпляр класса).

Объектно-ориентированное программирование (ООП) – парадигма программирования, в которой основными концепциями являются понятия объектов и классов.

Объектно-ориентированный язык программирования – язык программирования, в качестве базовых элементов которого выступают объекты, имеющие собственные параметры и именованные операции и образующие иерархически организованные классы объектов.

Одиночное наследование – механизм наследования, при котором класс может быть образован только от одного базового класса.

Открытые атрибуты класса – члены класса, обращение к которым осуществляется как к полям структуры.

Отладка – механизм, упрощающий процесс выявления и устранения ошибок в программе

Парадигма программирования – совокупность идей и понятий, сформулированных на основе концепции программирования и определяющих стиль программирования.

Параллелизм – возможность различным объектам действовать одновременно.

Перегрузка операторов – возможности одновременного существования в одной области видимости нескольких различных вариантов применения оператора, имеющих одно и то же имя, но различающихся типами параметров, к которым они применяются.

Переопределенная функция – если в базовом классе определена составная функция, которая должна различным образом выполняться для объектов различных производных классов, то она в этих производных классах должна быть определена заново. Такая функция называется **переопределенной.**

Полиморфизм – обозначение различных действий одним именем и свойство объекта отвечать на направленный к нему запрос сообразно своему типу.

Программирование – процесс описания программной идеи с помощью языка программирования, понятного приемнику и передатчику.

Программная идея – направление деятельности, на основе которого мы описываем задачу программирования.

Проектирование по контракту – механизм, который позволяет задавать различные типы условий (контракты), проверяемых во время работы программы.

Производный класс – класс, созданный на основе другого класса (базового).

Противоречие – это проявление несоответствия между разными требованиями, предъявляемыми к системе, и ограничениями, налагаемыми на нее.

Процедурное программирование – программирование на императивном языке, при котором последовательно выполняемые операторы можно собрать в подпрограммы, то есть более крупные целостные единицы кода, с помощью механизмов самого языка.

Роль (типаж) – абстракция, в которой можно промоделировать как поведение интерфейса, указав необходимые для этой роли методы (спецификацию), так и предоставить реализацию метода внутри роли, что позволяет многократно использовать один раз описанный код.

Свойство объекта – объявленный в классе параметр, характеризующий объект.

Синтаксис – механизм языка программирования, который описывает структуру программ в виде наборов символов, слов, операторов.

Статические элементы класса – элементы класса, которые являются общими для всех объектов этого класса.

Структура программы – механизм, определяющий строение программного кода.

Структурное программирование – парадигма программирования, в основе которой лежит представление программы в виде иерархической структуры блоков.

Типизация – правила использования объектов, не допускающие или ограничивающие взаимную замену объектов разных классов.

Устойчивость – способность объекта существовать во времени, переживая породивший его процесс, и (или) в пространстве, перемещаясь из своего первоначального адресного пространства.

Функциональное программирование – парадигма программирования, в которой процесс вычисления трактуется как вычисление значений функций в математическом понимании.

Шаблон функции – обобщенное описание поведения функций, которые могут вызываться для объектов разных типов; другими словами, шаблон функции представляет семейство функций.

Шаблоны – средство языка, предназначенное для кодирования обобщённых алгоритмов, без привязки к некоторым параметрам.

Язык программирования - частный случай понятия «язык», которое является средством взаимодействия между передатчиком сообщения и приемником с целью реализации программной идеи.

Учебное издание

Бердоносов Виктор Дмитриевич
Животова Алена Анатольевна

ОБЪЕКТНО-ОРИЕНТИРОВАННОЕ ПРОГРАММИРОВАНИЕ

Учебное пособие

Научный редактор – А. В. Еськова,
кандидат технических наук, доцент

Редактор Е. В. Безолукова

Подписано в печать 16.06.2015.
Формат 60 × 84 1/16. Бумага 80 г/м^2. Ризограф EZ570E.
Усл. печ. л. 8,12. Уч.-изд. л. 8,00. Тираж 50 экз. Заказ 27171.

Редакционно-издательский отдел
Федерального государственного бюджетного образовательного
учреждения высшего профессионального образования
«Комсомольский-на-Амуре государственный технический университет»
681013, г. Комсомольск-на-Амуре, пр. Ленина, 27.

Полиграфическая лаборатория
Федерального государственного бюджетного образовательного
учреждения высшего профессионального образования
«Комсомольский-на-Амуре государственный технический университет»
681013, г. Комсомольск-на-Амуре, пр. Ленина, 27.