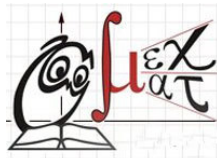




САРАТОВСКИЙ ГОСУНИВЕРСИТЕТ



МЕХАНИКО-МАТЕМАТИЧЕСКИЙ ФАКУЛЬТЕТ

Основы объектно-ориентированного программирования

Составители: Блинков Ю. А., Месянжин В. В.

Оглавление

Установочный модуль

Введение

Практическое задание

- 0.1 Оценка
- 0.2 Пример решения задачи «Телефонный справочник»
 - 0.2.1 Описание предметной области
 - 0.2.2 Первоначальная постановка задачи
 - 0.2.3 Развитие постановки задачи
- 0.3 Набор заданий
 - 0.3.1 Страховая компания
 - 0.3.2 Гостиница
 - 0.3.3 Ломбард
 - 0.3.4 Реализация готовой продукции
 - 0.3.5 Ведение заказов

- 0.3.6 Бюро по трудоустройству
- 0.3.7 Нотариальная контора
- 0.3.8 Курсы по повышению квалификации
- 0.3.9 Определение факультативов для студентов
- 0.3.10 Распределение учебной нагрузки
- 0.3.11 Распределение дополнительных обязанностей
- 0.3.12 Техническое обслуживание станков
- 0.3.13 Туристическая фирма
- 0.3.14 Грузовые перевозки
- 0.3.15 Учет телефонных переговоров
- 0.3.16 Учет внутриофисных расходов
- 0.3.17 Библиотека
- 0.3.18 Прокат автомобилей
- 0.3.19 Выдача банком кредитов
- 0.3.20 Инвестирование свободных средств
- 0.3.21 Занятость актеров театра
- 0.3.22 Платная поликлиника
- 0.3.23 Анализ динамики показателей финансовой отчетности различных предприятий
- 0.3.24 Учет телекомпанией стоимости прошедшей в эфире рекламы
- 0.3.25 Интернет-магазин
- 0.3.26 Ювелирная мастерская
- 0.3.27 Парикмахерская
- 0.3.28 Химчистка
- 0.3.29 Сдача в аренду торговых площадей

Модуль

1 Качество ПО

- 1.1 Внешние и внутренние факторы
- 1.2 Обзор внешних факторов
- 1.3 О программном сопровождении

Контрольные вопросы

2 Критерии объектной ориентации

- 2.1 О критериях
- 2.2 Метод и язык
- 2.3 Реализация и среда

Контрольные вопросы

3 Модульность

- 3.1 Пять критериев
- 3.2 Пять правил
- 3.3 Пять принципов
- 3.4 Ключевые концепции

Контрольные вопросы

4 Подходы к повторному использованию

- 4.1 Цели повторного использования
- 4.2 Что следует повторно использовать?
- 4.3 Повторяемость при разработке ПО
- 4.4 Пять требований к модульным структурам

- 4.5 Традиционные модульные структуры
- 4.6 Перегрузка и универсальность
- 4.7 Ключевые концепции
- Контрольные вопросы

Модуль 2

5 К объектной технологии

- 5.1 Функциональная декомпозиция
- 5.2 Декомпозиция, основанная на объектах
- 5.3 Ключевые концепции
- Контрольные вопросы

6 Абстрактные типы данных (АТД)

- 6.1 Критерии
- 6.2 К абстрактному взгляду на объекты
- 6.3 От абстрактных типов данных к классам
- 6.4 Ключевые концепции
- Контрольные вопросы

7 Статические структуры

- 7.1 Классы, а не объекты - предмет обсуждения
- 7.2 Роль классов
- 7.3 Унифицированная система типов

- 7.4 Простой класс
- 7.5 Ключевые концепции
- Контрольные вопросы

8 Динамические структуры: объекты

- 8.1 Объекты
- 8.2 Объекты как средство моделирования
- 8.3 Работа с объектами и ссылками
- 8.4 Ключевые концепции
- Контрольные вопросы

Модуль 3

9 Проектирование по контракту: построение надежного ПО

- 9.1 Базисные механизмы надежности
- 9.2 О корректности ПО
- 9.3 Выражение спецификаций
- 9.4 Введение утверждений в программные тексты
- 9.5 пример Класс стек
- 9.6 Контракты и надежность ПО
- Контрольные вопросы

10 Когда контракт нарушается: обработка исключений

- 10.1 Базисные концепции обработки исключений

Контрольные вопросы

11 Введение в наследование

11.1 Основные соглашения и терминология

11.2 Ключевые концепции

Контрольные вопросы

12 Множественное наследование

12.1 Примеры множественного наследования

12.2 Структурное наследование

12.3 Наследование функциональных возможностей

12.4 Ключевые концепции

Контрольные вопросы

Модуль 4

13 Техника наследования

13.1 Наследование и утверждения

13.2 Наследование и скрытие информации

Контрольные вопросы

14 Типизация

14.1 Проблема типизации

Контрольные вопросы

15 Глобальные объекты и константы

15.1 Константы базовых типов

Контрольные вопросы

Литература

Список иллюстраций

Список таблиц

Предметный указатель

Установочный модуль

Введение

В пособии подробно излагаются основные понятия объектной технологии – классы, объекты, управление памятью, типизация, наследование, универсализация. Большое внимание уделяется проектированию по контракту и обработке исключений, как механизмам, обеспечивающим корректность и устойчивость программных систем.

Рассматриваются основы объектно-ориентированного программирования. Изложение начинается с рассмотрения критериев качества программных систем и обоснования того, как объектная технология разработки может обеспечить требуемое качество. Основные понятия объектной технологии и соответствующая нотация появляются как результат тщательного анализа и обсуждений. Подробно рассматривается понятие класса - центральное понятие объектной технологии. Рассматривается абстрактный тип данных, лежащий в основе класса, совмещение классом роли типа данных и модуля и другие аспекты построения класса. Столь же подробно рассматриваются объекты и проблемы управления памятью. Большая часть уделена отношениям между классами – наследованию, универсализации и их роли в построении программных систем. Важную часть составляет введение понятия контракта, описание технологии проектирования по контракту, как механизма, обеспечивающего корректность создаваемых программ. Не обойдены вниманием и другие важные темы объектного программирования – скрытие информации, статическая типизация, динамическое связывание и обработка исключений.

Цель. Курс предназначен для изучения инженерии разработки программных систем на основе

объектной технологии. Хотя в изложении и используется нотация языка программирования *Python*, данный курс не является учебником по программированию на *Python* или каком либо конкретном языке программирования. Он дает фундаментальное описание объектной технологии разработки и в равной степени полезна всем, кто создает программные системы в объектном стиле независимо от того, в какой рабочей среде и на каком языке программирования эти системы создаются. Курс лекций в первую очередь адресуется будущим профессионалам, создающим качественный программный продукт.

Практическое задание содержит варианты заданий и пример решения.

В первом разделе рассматривается качество ПО. Качество - это цель инженерной деятельности; построение качественного ПО (*software*) - цель программной инженерии (*software engineering*). В данном пособии рассматриваются средства и технические приемы, позволяющие значительно улучшить качество ПО. Прежде чем приступить к изучению этих средств и приемов, следует хорошо представлять цель. Качество ПО лучше всего описывается комбинацией ряда факторов.

Критериям объектной ориентации посвящена следующая глава. Здесь будет дано лаконичное пояснение того, что делает систему объектно-ориентированной.

В четвертом разделе будут рассмотрены требования к разработке программного продукта. Буде построен переход от понятия модуля к объектной технологии.

В пятом разделе будут рассмотрены некоторые из проблем, направленных на широкомасштабное внедрение повторного использования программных компонентов.

Расширяемость, возможность повторного использования и надежность требуют выполнения ряда условий, определенных в предыдущих разделах. Для их достижения требуется систематический метод декомпозиции системы на модули. В этой разделе представлены основные элементы такого метода, основанного на простой идее: строить каждый модуль на базе некоторого типа объектов. Здесь эта идея объясняется, логически обосновывается и из нее выводятся некоторые следствия.

Чтобы объекты играли лидирующую роль в архитектуре ПО, нужно их адекватно описывать. В

седьмом разделе показывается, как это делать на примере абстрактных типов данных.

В восьмом разделе делается переход от теоретической основы ОО-подхода - абстрактного типы данных к классам.

Следующий раздел посвящен экземплярам классов, которые называются объектами. Объекты формируют объекты модель ОО-вычислений времени выполнения. Будут также рассмотрены некоторые аспектам реализации.

Факторы качества - повторное использование, расширяемость, совместимость - не должны достигаться ценой надежности (корректность и устойчивость). В десятом разделе рассматривается концепция надежности.

В случае нарушения контракта стандартным средством ОО-технологии является обработка исключений. В этом разделе представлена технология исключений и ее применение.

В двенадцатом разделе рассмотрено одна из главных ОО-технологий – наследование. Почти всегда новые программы являются расширениями предыдущих разработок, лучший способ создания нового - это подражание старым образцам, их уточнение и комбинирование.

Полноценное применение наследования требует важного расширения этого механизма. Множественное наследование рассматривается в тринадцатом разделе.

В следующей главе рассматривается техника наследования. Наследование - ключевая составляющая ОО-подхода к повторному использованию и расширяемости.

Эффективное применение объектной технологии требует четкого описания в тексте системы типов всех объектов, с которыми она работает на этапе выполнения. Это правило, известное как статическая типизация (*static typing*), делает наше ПО: более надежным, позволяя компилятору и другим инструментальным средствам устранять несоответствия прежде, чем они смогут нанести вред; более понятным, обеспечивая точной информацией читателей: авторов клиентских систем и тех, кто будет сопровождать систему; более эффективным, поскольку информация о типах данных позволит компилятору сгенерировать оптимальный код. Хотя вопросами типизации данных активно занимались и

вне объектной среды, да и сама статическая типизация применяется в языках, не поддерживающих ООП, особенно ярко эти идеи проявили себя именно при объектном подходе, во многом основанном на понятии типа, которое, сливаясь с понятием модуля, образует базовую ОО-конструкцию - класс. Этому посвящен пятнадцатый раздел.

Локальных знаний не достаточно - компонентам ПО необходима глобальная информация: разделяемые данные, общее окно для вывода ошибок, шлюз для подключения к базе данных или сети. В классическом подходе достаточно объявить такой объект глобальной переменной главной программы. В ОО-системах нет ни главной программы, ни глобальных переменных. Но разделяемые (*shared*) объекты по-прежнему нужны. Этим разделом заканчивается пособие.

Практическое задание

0.1 Оценка

Для получения зачета по практике (оценка '3') необходимо полностью владеть кодом примера решения задачи «Телефонный справочник» и для своей предметной области разработать и описать

- Первоначальная постановка задачи
 - Диаграмму прецедентов.
 - Структуру классов.
- Развитие постановки задачи
 - Диаграмму прецедентов.
 - Структуру классов.

Для получения по практике оценки '4', в дополнении к зачету по практике, необходимо сделать реализацию первоначальной постановки задачи и развития постановки задачи в объеме примера «Телефонный справочник».

Для получения по практике оценки '5' необходимо реализовать набор запросов предоставляемых преподавателем.

В следующем семестре такие же «Описания предметной области» будут использованы на практических занятиях за курсом “Базы данных”.

0.2 Пример решения задачи «Телефонный справочник»

0.2.1 Описание предметной области

Вашей задачей является создание телефонного справочника организации.

Организация имеет различные подразделения. Каждое из них может иметь собственные подотделы. Один сотрудник может иметь несколько телефонных номеров и, наоборот, один телефон могут иметь несколько сотрудников. Необходимо создать справочник для поиска по подразделениям (подотделам), сотрудникам и телефонам.

Классы объектов

Сотрудники (Фамилия, Имя, Отчество).

Подразделения (Наименование, Сотрудники, Подотделы).

Типы телефонов (Наименование).

Телефоны (Телефон, Типа телефона, Сотрудник).

Развитие постановки задачи

Нужно учесть, что один сотрудник может работать в разных подразделениях. Например сотрудники в подразделении «ответственные за пожарную безопасность» работают и в других подразделениях (по основному месту работы).

0.2.2 Первоначальная постановка задачи

Диаграмма прецедентов содержит только один прецедент “Поиск в телефонном справочнике”. Поиск должен осуществляться по подразделениям (подотделам), сотрудникам и телефонам.

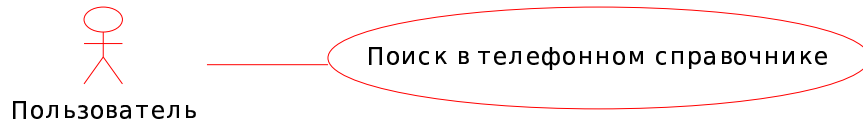
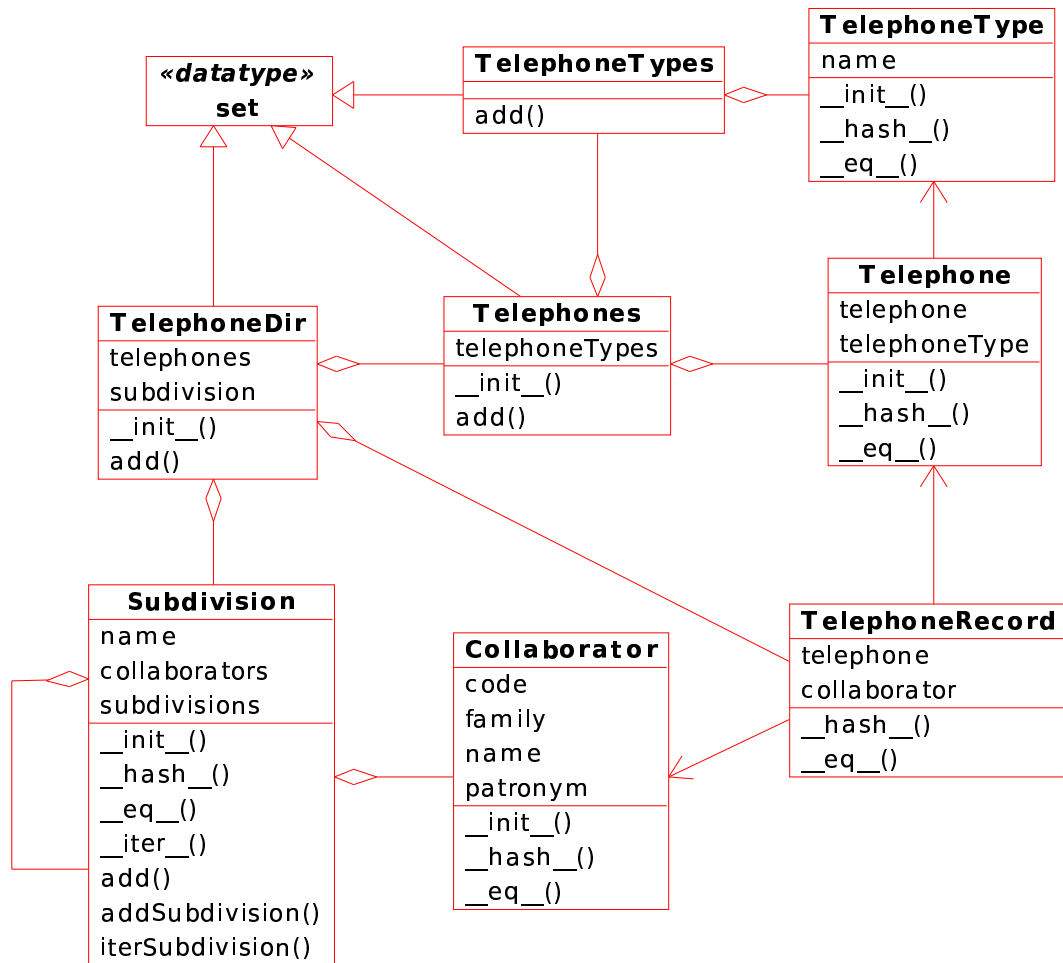


Диаграмма классов значительно сложнее.



Класс *Collaborator* имеет атрибуты:

code — код сотрудника, это необходимо при одинаковых “ФИО” сотрудников или, например, при изменении фамилии сотрудника;

family — фамилия;

name — имя;

patronym — отчество.

Операции класса это:

__init__ — конструктор класса, в данном случае он необходим поскольку его атрибуты нужно инициализировать;

__hash__ — определение этой функции позволяет использовать объект класса в качестве элемента словарей и множеств;

__eq__ — поскольку хэш представляет отображение элементов некоторого множества в конечное множество чисел, обычно это числа уместяющиеся в машинное слово, то необходимо разрешать так называемые *коллизии*. В случае коллизии два разных объекта имеют одинаковое значение функции **__hash__** и **__eq__** позволяет различить эти объекты.

Подразделения представлены классом *Subdivision*, который имеет следующие атрибуты:

name — название подразделения;

collaborators — множество его сотрудников;

subdivisions — множество его собственных подразделений.

Операции класса это также состоят из конструктора, операций `__hash__` и `__eq__` поскольку подразделение может быть в качестве элемента множества *subdivisions*, а также других операций:

`__iter__` — итератор для просмотра всех сотрудников подразделения, включая и сотрудников его собственных подразделений;

add — для добавления сотрудника к подразделению;

addSubdivision — для добавления к подразделению другого подразделения в качестве его собственного подразделения;

iterSubdivision — итератор для просмотра его собственных подразделений.

Оставшиеся элементы диаграммы классов могут быть описаны кратко. *TelephoneType* определяется своим названием. *TelephoneTypes* представляет собой множество из объектов класса *TelephoneType*. Класс *Telephone* состоит из номера и его типа. Телефоны содержатся в множестве задаваемым классом *Telephones*. Телефонный справочник содержит *TelephoneRecord* ссылающийся на телефоны и подразделения.

Рассмотрим реализацию телефонного справочника . Для усвоения использованного ниже материала необходимо усвоить 1-4 лекции и часть лекции 7 курса “Учебный курс - Язык программирования Python” [3] Сузи Романа Арвиевича.

```
1 # -*- coding: utf-8 -*-
2
3 """
```

```

4 Телефонная книга организации
5 """
6
7 import itertools
8
9 class Collaborator:
10     """
11     Сотрудник
12     """
13     def __init__(self, code, family, name, patronym):
14         self.code = code
15         self.family = family
16         self.name = name
17         self.patronym = patronym
18
19     def __hash__(self):
20         return hash(self.code)
21
22     def __eq__(self, other):
23         return self.code == other.code

```

Строка 1 определяет используемую кодировку. Затем, на строках 3-5, определяем документацию нашей программы. На 7 строке импортируем модуль *itertools* для организации итераторов для обхода подразделения по его сотрудникам и по его собственным подразделениям. Далее следует описание класса *Collaborator*. При вычислении хэша используется стандартная *hash* функция языка *Python*.

```

25 class Subdivision:

```

```
26  """
27  Подразделение
28  """
29  def __init__(self, name):
30      self.name = name
31      self.collaborators = set()
32      self.subdivisions = set()
33
34  def __hash__(self):
35      return hash(self.name)
36
37  def __eq__(self, other):
38      return self.name == other.name
39
40  def __iter__(self):
41      i = iter(self.collaborators)
42      for s in self.subdivisions:
43          i = itertools.chain(i, iter(s))
44      return i
45
46  def add(self, collaborator):
47      assert collaborator not in self
48      self.collaborators.add(collaborator)
49
50  def addSubdivision(self, subdivision):
51      assert subdivision not in self.subdivisions
52      assert not set(self).intersection(set(subdivision))
```

```

53     self.subdivisions.add(subdivision)
54
55     def iterSubdivision(self):
56         i = iter(self.subdivisions)
57         for s in self.subdivisions:
58             i = itertools.chain(i, s.iterSubdivision())
59     return i

```

В *Subdivision* для организации итераторов `__iter__` и *iterSubdivision* используется так называемое зацепление итераторов. Операция *add* на строчке 47 запрещает добавлять сотрудника к подразделению если он есть в нем или в его собственном подразделении. Это достигается за счет переопределения стандартного метода `__iter__` для контейнера языка *Python*. Строчка 51 запрещает добавлять подразделение если там оно уже есть, а строчка 52 если в нем есть хотя бы один сотрудник уже имеющиеся в нашем подразделении.

```

61 class TelephoneType:
62     """
63     Тип телефона
64     """
65     def __init__(self, name):
66         self.name = name
67
68     def __hash__(self):
69         return hash(self.name)
70
71     def __eq__(self, other):
72         return self.name == other.name

```

Тип телефона определяется его именем.

```
74 class TelephoneTypes(set):
75     """
76     Типы телефонов
77     """
78     def add(self, telephoneType):
79         assert telephoneType not in self
80         set.add(self, telephoneType)
```

На строке 79 стоит запрет на добавление к множеству типов телефонов элемента если он там уже есть.

```
82 class Telephone:
83     """
84     Телефон
85     """
86     def __init__(self, telephone, telephoneType):
87         self.number = telephone
88         self.type = telephoneType
89
90     def __hash__(self):
91         return hash(self.number)
92
93     def __eq__(self, other):
94         return self.number == other.number
```

Телефон определяется его типом и номером.


```

96 class Telephones(set):
97     """
98     Телефоны
99     """
100    def __init__(self, telephoneTypes):
101        set.__init__(self)
102        self.telephoneTypes = telephoneTypes
103
104    def add(self, telephone):
105        assert telephone not in self
106        assert telephone.type in self.telephoneTypes
107        set.add(self, telephone)

```

На строке 105 стоит запрет на добавление к множеству телефонов элемента с уже существующим номером и на строке 106 если его типа нет в *telephoneTypes*.

```

109 class TelephoneRecord:
110     """
111     Запись в телефонном справочнике
112     """
113    def __init__(self, telephone, collaborator):
114        self.telephone = telephone
115        self.collaborator = collaborator
116
117    def __hash__(self):
118        return hash((self.telephone, self.collaborator))
119
120    def __eq__(self, other):

```

```
121     return self.telephone == other.telephone and \  
122         self.collaborator == other.collaborator
```

Запись в телефонном справочнике должна ссылаться на телефон и сотрудника.

```
124 class TelephoneDir(set):  
125     """  
126     Телефонный справочник  
127     """  
128     def __init__(self, telephones, subdivision):  
129         set.__init__(self)  
130         self.telephones = telephones  
131         self.subdivision = subdivision  
132  
133     def add(self, telephoneRecord):  
134         assert telephoneRecord.telephone in self.telephones  
135         assert telephoneRecord.collaborator in self.subdivision  
136         assert telephoneRecord not in self  
137         set.add(self, telephoneRecord)
```

На строке 134 стоит запрет на добавление к множеству телефона, если его нет среди телефонов справочника. Аналогично, на строке 135, запрет для подразделений. Строка 136 запрещает добавлять записи в телефонный справочник, если они уже там есть.

```
139 if __name__ == '__main__':  
140     import os, csv  
141  
142     subdivision = {}
```

```

143     for rec in csv.reader(open(os.path.join(os.curdir, 'subdivision1.csv'),
'rb'), delimiter=';'):
144         subdivision[rec[0]] = Subdivision(rec[0])
145         if rec[1]:
146             subdivision[rec[1]].addSubdivision(subdivision[rec[0]])
147         else:
148             telephoneDir = TelephoneDir(Telephones(TelephoneTypes()),
subdivision[rec[0]])
149
150     telephones, telephoneTypes, collaborators = {}, {}, {}
151     for rec in csv.reader(open(os.path.join(os.curdir, 'ssu1.csv'), 'rb'),
delimiter=';'):
152         if rec[6] not in telephoneTypes:
153             telephoneTypes[rec[6]] = TelephoneType(rec[6])
154             telephoneDir.telephones.telephoneTypes.add(telephoneTypes[rec[6]])
155         if rec[0] not in telephones:
156             telephones[rec[0]] = Telephone(rec[0], telephoneTypes[rec[6]])
157             telephoneDir.telephones.add(telephones[rec[0]])
158         key = int(rec[1])
159         if key not in collaborators:
160             collaborators[key] = Collaborator(key, rec[2], rec[3], rec[4])
161             subdivision[rec[5]].add(collaborators[key])
162         telephoneDir.add(TelephoneRecord(telephones[rec[0]],
collaborators[key]))
163
164     for s in telephoneDir.subdivision.iterSubdivision():
165         if s.name == 'помощник проректора':


```

```

166         for r in telephoneDir:
167             if r.collaborator in s and r.collaborator.family.find('сх') >= 0:
168                 print r.telephone.number, "%s %s. %s." % \
169                     (r.collaborator.family, r.collaborator.name[:2],
r.collaborator.patronym[:2])
170             break
171
172     for s in telephoneDir.subdivision.iterSubdivision():
173         if s.name == 'зав. кафедрой':
174             for r in telephoneDir:
175                 if r.collaborator in s and r.collaborator.family.find('сх') >= 0:
176                     print r.telephone.number, "%s %s. %s." % \
177                         (r.collaborator.family, r.collaborator.name[:2],
r.collaborator.patronym[:2])
178             break

```

Каждый модуль языка *Python* может быть использован в качестве головного, в этом случае значение переменной `__name__` будет иметь значение `'__main__'` (см. строку 139).

Словарь *subdivision* будем использовать для хранения объектов, читаемых из файла `'subdivision1.csv'` . Содержимое файла показано ниже.


```

"Саратовский госуниверситет";
"ректорат"; "Саратовский госуниверситет"
"ректор"; "ректорат"
"проректор"; "ректорат"
"помощник ректора"; "ректорат"
"зам. проректора"; "ректорат"
"помощник проректора"; "ректорат"

```

"мех/мат факультет"; "Саратовский госуниверситет"
"деканат"; "мех/мат факультет"
"декан"; "деканат"
"зам. декана"; "деканат"
"зав. кафедрой"; "мех/мат факультет"

Файл имеет структуру *csv* (она описана в части 7 лекции). Первая часть записи название подразделения, а вторая его владельца. Естественно "Саратовский госуниверситет" не имеет владельца. В строчках 143-148 происходит чтение из файла и построение структуры подразделений.

С помощью аналогичного подхода происходит разбор файла 'ssu1.csv'  с представленной ниже структурой и заполнение телефонного справочника.

"26-16-96"; 1; "Коссович"; "Леонид"; "Юрьевич"; "ректор"; "раб"
"26-16-96"; 2; "Павлова"; "Юлия"; "Анатольевна"; "помощник ректора"; "раб"
"51-16-35"; 3; "Абрамейцева"; "Валерия"; "Владиславовна "; "помощник ректора"; "раб"
"51-18-84"; 4; "Первушов"; "Евгений"; "Михайлович"; "проректор"; "раб"
"51-18-84"; 5; "Лосатинская"; "Алла"; "Сергеевна"; "помощник проректора"; "раб"
.....

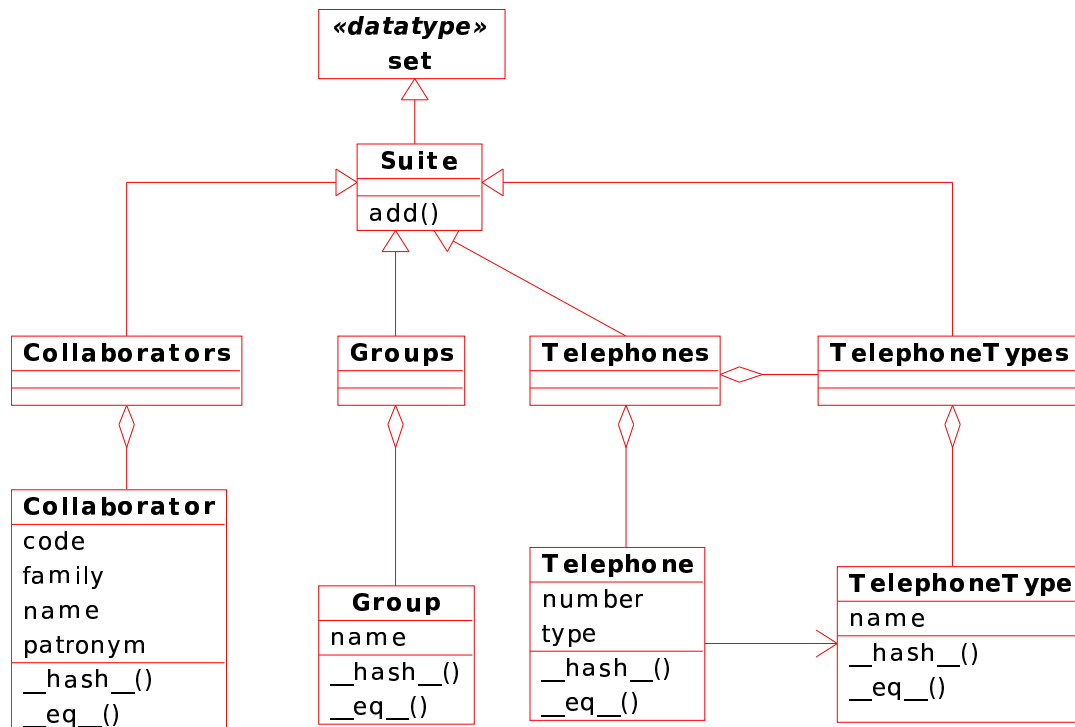
В первом запросе, строки 164-170, среди сотрудников подразделения 'помощник проректора' найти телефоны сотрудников содержащих в фамилии словосочетание 'ск'. Второй запрос представлен на строках 172-178. Результат работы программы представлен ниже.

51-57-39 Виноградский С. Г.
51-18-84 Лосатинская А. С.

В результате первый запрос вернул два телефона, а второй не одного.

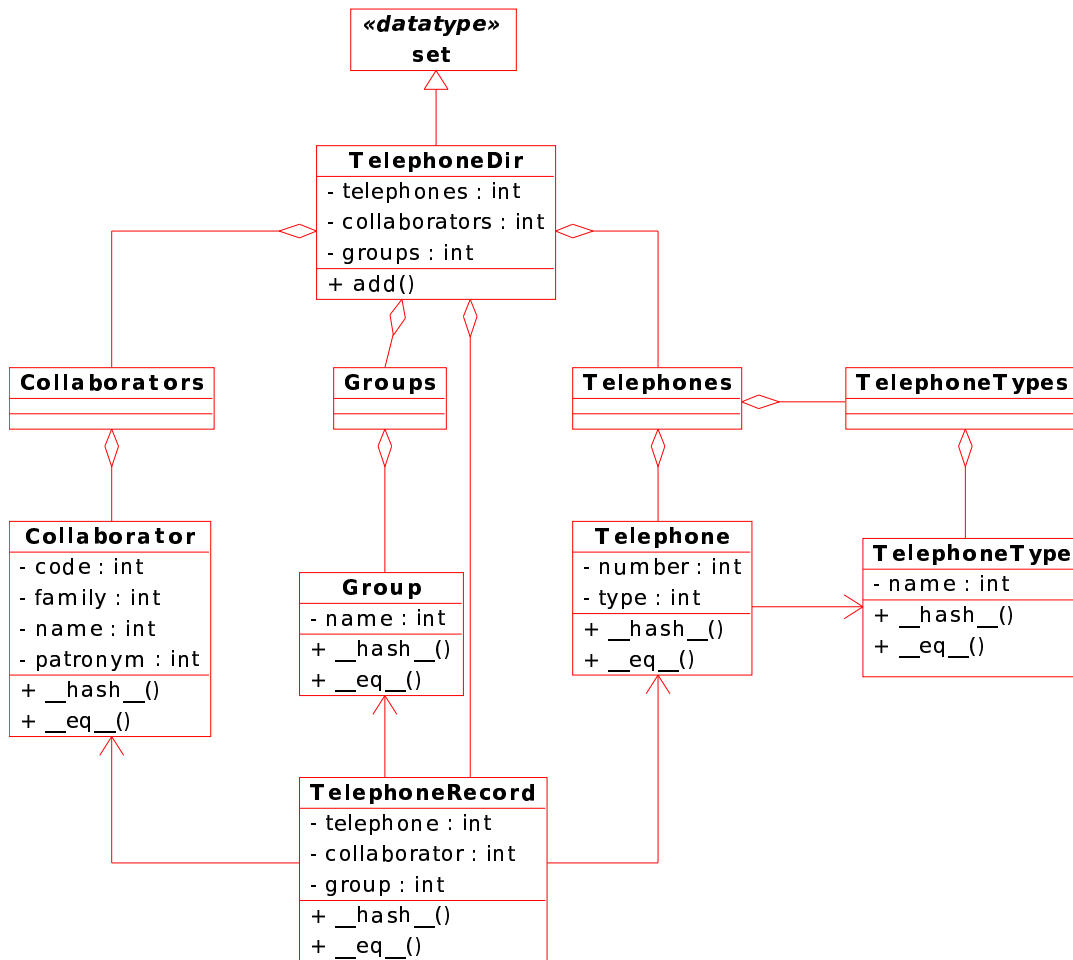
0.2.3 Развитие постановки задачи


Диаграмма прецедентов остается без изменений. Диаграмма классов лучше разбить на две.



Поскольку сотрудники могут одновременно входить в разные подразделения, заменим *Subdivision* на *Group*. Также *Subdivision* образовывали иерархическую структуру, а *Group* нет, поэтому для

организации хранения введем множество групп *Groups*. Из предыдущего решения задачи видно, что *Collaborators*, *TelephoneTypes*, *Telephones* и теперь и *Groups* обладают общими свойствами и поведением, а именно все они множества с запретом на добавление элемента если он уже там есть. Поэтому они могут быть порождены от общего класса *Suite* (Набор), который в свою очередь порожден от стандартного типа языка *Python set*. Атрибуты и операции классов уже описаны в предыдущей задаче.



Изменения, по сравнению с предыдущей задачей , коснулись в основном классов *TelephoneRecord*, *TelephoneDir*. Это связано с тем, что теперь запись в телефонном справочнике представляет собой набор из телефона, сотрудника и группы. Соответственно *TelephoneDir* содержит *Telephones*, *Collaborators* и *Groups*.

```
1  #-*- coding: utf-8 -*-
2
3  """
4  Телефонная книга организации
5  """
6
7  class Suite(set):
8      """
9      Множество с запретом на добавление элементов
10     уже содержащихся в нём
11     """
12     def add(self, elt):
13         assert elt not in self
14         set.add(self, elt)
15
16 class Collaborator:
17     """
18     Сотрудник
19     """
20     def __init__(self, code, family, name, patronym):
21         self.code = code
22         self.family = family
```

```
23     self.name = name
24     self.patronym = patronym
25
26     def __hash__(self):
27         return hash(self.code)
28
29     def __eq__(self, other):
30         return self.code == other.code
31
32 class Collaborators(Suite):
33     """
34     Сотрудники
35     """
36     pass
37
38 class Group:
39     """
40     Группа сотрудников
41     """
42     def __init__(self, name):
43         self.name = name
44
45     def __hash__(self):
46         return hash(self.name)
47
48     def __eq__(self, other):
49         return self.name == other.name
```

```
50
51 class Groups(Suite):
52     """
53     Группы
54     """
55     pass
56
57 class TelephoneType:
58     """
59     Тип телефона
60     """
61     def __init__(self, name):
62         self.name = name
63
64     def __hash__(self):
65         return hash(self.name)
66
67     def __eq__(self, other):
68         return self.name == other.name
69
70 class TelephoneTypes(Suite):
71     """
72     Типы телефонов
73     """
74     pass
75
76 class Telephone:
```

```
77     """
78     Телефон
79     """
80     def __init__(self, telephone, telephoneType):
81         self.number = telephone
82         self.type = telephoneType
83
84     def __hash__(self):
85         return hash(self.number)
86
87     def __eq__(self, other):
88         return self.number == other.number
89
90 class Telephones(set):
91     """
92     Телефоны
93     """
94     def __init__(self, telephoneTypes):
95         set.__init__(self)
96         self.telephoneTypes = telephoneTypes
97
98     def add(self, telephone):
99         assert telephone not in self
100         assert telephone.type in self.telephoneTypes
101         set.add(self, telephone)
102
103 class TelephoneRecord:
```

```
104     """
105     Запись в телефонном справочнике
106     """
107     def __init__(self, telephone, collaborator, group):
108         self.telephone = telephone
109         self.collaborator = collaborator
110         self.group = group
111
112     def __hash__(self):
113         return hash((self.telephone, self.collaborator, self.group))
114
115     def __eq__(self, other):
116         return self.telephone == other.telephone and \
117             self.collaborator == other.collaborator and \
118             self.group == other.group
119
120 class TelephoneDir(set):
121     """
122     Телефонный справочник
123     """
124     def __init__(self, telephones, collaborators, groups):
125         set.__init__(self)
126         self.telephones = telephones
127         self.collaborators = collaborators
128         self.groups = groups
129
130     def add(self, telephoneRecord):
```

```

131     assert telephoneRecord.telephone in self.telephones
132     assert telephoneRecord.collaborator in self.collaborators
133     assert telephoneRecord.group in self.groups
134     assert telephoneRecord not in self
135     set.add(self, telephoneRecord)

```

Хотя реализация в многом повторяет предыдущую, она заметно упростилась, как и объеме, так и с точки зрения простоты кода.

```


137 if __name__ == '__main__':
138     import os, csv
139
140     telephoneDir = TelephoneDir(Telephones(TelephoneTypes()), Collaborators(),
Groups())
141     telephones, telephoneTypes, collaborators, groups = {}, {}, {}, {}
142     for rec in csv.reader(open(os.path.join(os.curdir, 'ssu2.csv'), 'rb'),
delimiter=';'):
143         if rec[6] not in telephoneTypes:
144             telephoneTypes[rec[6]] = TelephoneType(rec[6])
145             telephoneDir.telephones.telephoneTypes.add(telephoneTypes[rec[6]])
146         if rec[0] not in telephones:
147             telephones[rec[0]] = Telephone(rec[0], telephoneTypes[rec[6]])
148             telephoneDir.telephones.add(telephones[rec[0]])
149         if rec[5] not in groups:
150             groups[rec[5]] = Group(rec[5])
151             telephoneDir.groups.add(groups[rec[5]])
152         key = int(rec[1])
153         if key not in collaborators:

```

```

154         collaborators[key] = Collaborator(key, rec[2], rec[3], rec[4])
155         telephoneDir.collaborators.add(collaborators[key])
156     telephoneDir.add(TelephoneRecord(telephones[rec[0]], collaborators[key],
groups[rec[5]]))
157
158     for r in telephoneDir:
159         if r.group.name == 'помощник проректора' and
r.collaborator.family.find('ск') >= 0:
160             print r.telephone.number, "%s %s. %s."% \
161                 (r.collaborator.family, r.collaborator.name[:2],
r.collaborator.patronym[:2])
162
163     for r in telephoneDir:
164         if r.group.name == 'зав. кафедрой' and r.collaborator.family.find('cc')
>= 0:
165             print r.telephone.number, "%s %s. %s."% \
166                 (r.collaborator.family, r.collaborator.name[:2],
r.collaborator.patronym[:2])

```

Чтение теперь производится только из одного файла 'ssu2.csv'  с представленной ниже структурой.

```

"26-16-96";1;"Коссович";"Леонид";"Юрьевич";"ректорат";"раб"
"26-16-96";1;"Коссович";"Леонид";"Юрьевич";"ректор";"раб"
"26-16-96";1;"Коссович";"Леонид";"Юрьевич";"зав. кафедрой";"раб"
"26-16-96";2;"Павлова";"Юлия";"Анатольевна";"помощник ректора";"раб"
"26-16-96";2;"Павлова";"Юлия";"Анатольевна";"ректорат";"раб"
.....

```

Заметно, что один и тот же сотрудник может быть членом различных групп. Поэтому работа запросов дала немного другой результат.

51-57-39 Виноградский С. Г.

51-18-84 Лосатинская А. С.

26-16-96 Коссович Л. Ю.

Сработал и второй запрос.

0.3 Набор заданий

0.3.1 Страховая компания

Описание предметной области

Вы работаете в страховой компании. Вашей задачей является отслеживание финансовой деятельности компании.

Компания имеет различные филиалы по всей стране. Каждый филиал характеризуется названием, адресом и телефоном. Деятельность компании организована следующим образом: к Вам обращаются различные лица с целью заключения договора о страховании. В зависимости от принимаемых на страхование объектов и страхуемых рисков, договор заключается по определенному виду страхования (например, страхование автотранспорта от угона, страхование домашнего имущества, добровольное медицинское страхование). При заключении договора Вы фиксируете дату заключения, страховую сумму, вид страхования, тарифную ставку и филиал, в котором заключался договор.

Классы объектов

Договоры (Номер договора, Дата заключения, Страховая сумма, Тарифная ставка, Филиал, Вид страхования).

Вид страхования (Вид страхования, Наименование).

Филиал (Филиал, Наименование филиала, Адрес, Телефон).

Развитие постановки задачи

Нужно учесть, что договоры заключают страховые агенты. Помимо информации об агентах (фамилия, имя, отчество, адрес, телефон), нужно еще хранить филиал, в котором работают агенты. Кроме того, исходя из базы данных, нужно иметь возможность рассчитывать заработную плату агентам. Заработная плата составляет некоторый процент от страхового платежа (страховой платеж это страховая сумма, умноженная на тарифную ставку). Процент зависит от вида страхования, по которому заключен договор.

0.3.2 Гостиница

Описание предметной области

Вы работаете в гостинице. Вашей задачей является отслеживание финансовой стороны работы гостиницы.

Ваша деятельность организована следующим образом: гостиница предоставляет номера клиентам на определенный срок. Каждый номер характеризуется вместимостью, комфортностью (люкс, полулюкс, обычный) и ценой. Вашими клиентами являются различные лица, о которых Вы собираете определенную информацию (фамилия, имя, отчество и некоторый комментарий). Сдача номера клиенту производится при наличии свободных мест в номерах, подходящих клиенту по указанным выше параметрам. При поселении фиксируется дата поселения. При выезде из гостиницы для каждого места запоминается дата освобождения.

Классы объектов

Клиенты (Клиент, Фамилия, Имя, Отчество, Паспортные данные, Комментарий).

Номера (Номер, Количество человек, Комфортность, Цена).

Поселение (Клиент, Номер, Дата поселения, Дата освобождения, Примечание).

Развитие постановки задачи

Необходимо хранить информацию не только по факту сдачи номера клиенту, но и осуществлять бронирование номеров. Кроме того, для постоянных клиентов, а также для определенных категорий клиентов, предусмотрена система скидок. Скидки могут суммироваться.

Внести в структуру сущностей изменения, учитывающие этот факт, и изменить существующие запросы. Добавить новые запросы.

0.3.3 Ломбард

Описание предметной области

Вы работаете в ломбарде. Вашей задачей является отслеживание финансовой стороны работы ломбарда.

Деятельность Вашей компании организована следующим образом: к Вам обращаются различные лица с целью получения денежных средств под залог определенных товаров. У каждого из приходящих к Вам клиентов Вы запрашиваете фамилию, имя, отчество и другие паспортные данные. После оценивания стоимости принесенного в качестве залога товара Вы определяете сумму, которую готовы выдать на руки клиенту, а также свои комиссионные. Кроме того, определяете срок возврата денег. Если клиент согласен, то Ваши договоренности фиксируются в виде документа, деньги выдаются клиенту, а товар остается у Вас. В случае если в указанный срок не происходит возврата денег, товар переходит в Вашу собственность.

Классы объектов

Клиенты (Клиент, Фамилия, Имя, Отчество, Номер паспорта, Серия паспорта, Дата выдачи паспорта).

Категории товаров (Категория товаров, Название, Примечание).

Сдача в ломбард (Категория товаров, Клиент, Описание товара, Дата сдачи, Дата возврата, Сумма, Комиссионные).

Развитие постановки задачи

После перехода прав собственности на товар, ломбард может продавать товары по цене, меньшей или большей, чем была заявлена при сдаче. Цена может меняться несколько раз, в зависимости от ситуации на рынке. (Например, владелец ломбарда может устроить распродажу зимних вещей в конце зимы). Помимо текущей цены, нужно хранить все возможные значения цены для данного товара.

0.3.4 Реализация готовой продукции

Описание предметной области

Вы работаете в компании, занимающейся оптово-розничной продажей различных товаров. Вашей задачей является отслеживание финансовой стороны работы компании.

Деятельность Вашей компании организована следующим образом: Ваша компания торгует товарами из определенного спектра. Каждый из этих товаров характеризуется наименованием, оптовой ценой, розничной ценой и справочной информацией. В Вашу компанию обращаются покупатели. Для каждого из них Вы запоминаете в базе данных стандартные данные (наименование, адрес, телефон, контактное лицо) и составляете по каждой сделке документ, запоминая наряду с покупателем количество купленного им товара и дату покупки.

Классы объектов

Товары (Наименование, Оптовая цена, Розничная цена, Описание).

Покупатели (Телефон, Контактное лицо, Адрес).

Сделки (Дата сделки, Товар, Количество, Покупатель, Признак оптовой продажи).

Развитие постановки задачи

Теперь ситуация изменилась. Выяснилось, что обычно покупатели в рамках одной сделки покупают не один товар, а сразу несколько. Также компания решила предоставлять скидки в зависимости от количества закупленных товаров и их общей стоимости.

0.3.5 Ведение заказов

Описание предметной области

Вы работаете в компании, занимающейся оптовой продажей различных товаров. Вашей задачей является отслеживание финансовой стороны работы компании.

Деятельность Вашей компании организована следующим образом: Ваша компания торгует товарами из определенного спектра. Каждый из этих товаров характеризуется ценой, справочной информацией и признаком наличия или отсутствия доставки. В Вашу компанию обращаются заказчики. Для каждого из них Вы запоминаете в базе данных стандартные данные (наименование, адрес, телефон, контактное лицо) и составляете по каждой сделке документ, запоминая наряду с заказчиком количество купленного им товара и дату покупки.

Классы объектов

Заказчики (Наименование, Адрес, Телефон, Контактное лицо).

Товары (Цена, Доставка, Описание).

Заказы (Заказчик, Товар, Количество, Дата).

Развитие постановки задачи

Теперь ситуация изменилась. Выяснилось, что доставка разных товаров может производиться разными способами, различными по цене и скорости. Нужно хранить информацию по тому, какими способами может осуществляться доставка каждого товара и информацию о том, какой вид доставки (а, соответственно, и какую стоимость доставки) выбрал клиент при заключении сделки.

0.3.6 Бюро по трудоустройству

Описание предметной области

Вы работаете в бюро по трудоустройству. Вашей задачей является отслеживание финансовой стороны работы компании.

Деятельность Вашего бюро организована следующим образом: Ваше бюро готово искать работников для различных работодателей и вакансии для ищущих работу специалистов различного профиля. При обращении к Вам клиента-работодателя, его стандартные данные (название, вид деятельности, адрес, телефон) фиксируются в базе данных. При обращении к Вам клиента-соискателя, его стандартные данные (фамилия, имя, отчество, квалификация, профессия, иные данные) также фиксируются в базе данных. По каждому факту удовлетворения интересов обеих сторон составляется документ. В документе указываются соискатель, работодатель, должность и комиссионные (доход бюро).

Классы объектов

Работодатели (Название, Вид деятельности, Адрес, Телефон). *Сделки* (Работодатель, Должность, Комиссионные). *Соискатели* (Фамилия, Имя, Отчество, Квалификация, Вид деятельности, Иные данные, Предполагаемый размер заработной платы).

Развитие постановки задачи

Оказалось, что база данных не совсем точно описывает работу бюро. В базе фиксируется только сделка, а информация по открытым вакансиям не храниться. Кроме того, для автоматического поиска вариантов, необходимо вести справочник «виды деятельности».

0.3.7 Нотариальная контора

Описание предметной области

Вы работаете в нотариальной конторе. Вашей задачей является отслеживание финансовой стороны работы компании.

Деятельность Вашей нотариальной конторы организована следующим образом: Ваша фирма готова предоставить клиенту определенный комплекс услуг. Для наведения порядка Вы формализовали эти услуги, составив их список с описанием каждой услуги. При обращении к Вам клиента, его стандартные данные (название, вид деятельности, адрес, телефон) фиксируются в базе данных. По каждому факту оказания услуги клиенту составляется документ. В документе указываются услуга, сумма сделки, комиссионные (доход конторы), описание сделки.

Классы объектов

Клиенты (Название, Вид деятельности, Адрес, Телефон).

Сделки (Клиент, Услуга, Сумма, Комиссионные, Описание).

Услуги (Название, Описание).

Развитие постановки задачи

Теперь ситуация изменилась. В рамках одной сделки клиенту может быть оказано несколько услуг. Стоимость каждой услуги фиксирована. Кроме того, компания предоставляет в рамках одной сделки различные виды скидок. Скидки могут суммироваться.

Фирма по продаже запчастей

Описание предметной области

Вы работаете в фирме, занимающейся продажей запасных частей для автомобилей. Вашей задачей является отслеживание финансовой стороны работы компании.

Основная часть деятельности, находящейся в Вашем ведении, связана с работой с поставщиками. Фирма имеет определенный набор поставщиков, по каждому из которых известны название, адрес и телефон. У этих поставщиков Вы приобретаете детали. Каждая деталь наряду с названием характеризуется артикулом и ценой (считаем цену постоянной). Некоторые из поставщиков могут поставлять одинаковые детали (один и тот же артикул). Каждый факт покупки запчастей у поставщика фиксируется в базе данных, причем обязательными для запоминания являются дата покупки и количество приобретенных деталей.

Классы объектов

Поставщики (Поставщик, Название, Адрес, Телефон).

Детали (Название, Артикул, Цена, Примечание).

Поставки (Поставщик, Деталь, Количество, Дата).

Развитие постановки задачи

Теперь ситуация изменилась. Выяснилось, что цена детали может меняться от поставки к поставке. Поставщики заранее ставят Вас в известность о дате изменения цены и о его новом значении. Нужно хранить не только текущее значение цены, но и всю историю изменения цен.

0.3.8 Курсы по повышению квалификации

Описание предметной области

Вы работаете в учебном заведении и занимаетесь организацией курсов повышения квалификации.

В Вашем распоряжении имеются сведения о сформированных группах студентов. Группы формируются в зависимости от специальности и отделения. В каждой из них включено определенное количество студентов. Проведение занятий обеспечивает штат преподавателей. Для каждого из них у Вас в базе данных зарегистрированы стандартные анкетные данные (фамилия, имя, отчество, телефон) и стаж работы. В результате распределения нагрузки Вы получаете информацию о том, сколько часов занятий проводит каждый преподаватель с соответствующими группами. Кроме того, хранятся также сведения о виде проводимых занятий (лекции, практика), предмете и оплате за 1 час.

Классы объектов

Группы (Специальность, Отделение, Количество студентов).

Преподаватели (Фамилия, Имя, Отчество, Телефон, Стаж).

Нагрузка (Преподаватель, Группа, Количество часов, Предмет, Тип занятия, Оплата).

Развитие постановки задачи

В результате работы с базой данных выяснилось, что размер почасовой оплаты зависит от предмета и типа занятия. Кроме того, каждый преподаватель может вести не все предметы, а только некоторые.

0.3.9 Определение факультативов для студентов

Описание предметной области

Вы работаете в высшем учебном заведении и занимаетесь организацией факультативов.

В Вашем распоряжении имеются сведения о студентах, включающие стандартные анкетные данные (фамилия, имя, отчество, адрес, телефон). Преподаватели Вашей кафедры должны обеспечить проведение факультативных занятий по некоторым предметам. По каждому факультативу существует определенное количество часов и вид проводимых занятий (лекции, практика, лабораторные работы). В результате работы со студентами у Вас появляется информация о том, кто из них записался на какие факультативы. Существует некоторый минимальный объем факультативных предметов, которые должен прослушать каждый студент. По окончанию семестра Вы заносите информацию об оценках, полученных студентами на экзаменах.

Классы объектов

Студенты (Фамилия, Имя, Отчество, Адрес, Телефон).

Предметы (Название, Объем лекций, Объем практик, Объем лабораторных работ).

Учебный план (Студент, Предмет, Оценка).

Развитие постановки задачи

Теперь ситуация изменилась. Выяснилось, что некоторые из факультативов могут длиться более одного семестра. В каждом семестре для предмета устанавливается объем лекций, практик и лабораторных работ в часах. В качестве итоговой оценки за предмет берется последняя оценка, полученная студентом.

0.3.10 Распределение учебной нагрузки

Описание предметной области

Вы работаете в высшем учебном заведении и занимаетесь распределением нагрузки между преподавателями кафедры.

В Вашем распоряжении имеются сведения о преподавателях кафедры, включающие наряду с анкетными данными сведения об их ученой степени, занимаемой административной должности и стаже работы. Преподаватели Вашей кафедры должны обеспечить проведение занятий по некоторым предметам. По каждому из них существует определенное количество часов. В результате распределения нагрузки у Вас должна получиться информация следующего рода: «Такой-то преподаватель проводит занятия по такому-то предмету с такой-то группой».

Классы объектов

Преподаватели (Фамилия, Имя, Отчество, Ученая степень, Должность, Стаж).

Предметы (Название, Количество часов).

Нагрузка (Преподаватель, Предмет, Номер группы).

Развитие постановки задачи

Теперь ситуация изменилась. Выяснилось, что все проводимые занятия делятся на лекционные и практические. По каждому виду занятий устанавливается свое количество часов. Кроме того, данные по нагрузке нужно хранить несколько лет.

0.3.11 Распределение дополнительных обязанностей

Описание предметной области

Вы работаете в коммерческой компании и занимаетесь распределением дополнительных разовых работ. Вашей задачей является отслеживание хода выполнения дополнительных работ.

Компания имеет определенный штат сотрудников, каждый из которых получает определенный оклад. Время от времени, возникает потребность в выполнении некоторой дополнительной работы, не входящей в круг основных должностных обязанностей сотрудников. Для наведения порядка в этой сфере деятельности Вы проклассифицировали все виды дополнительных работ, определившись с суммой оплаты по факту их выполнения. При возникновении дополнительной работы определенного вида Вы назначаете ответственного, фиксируя дату начала. По факту окончания Вы фиксируете дату и выплачиваете дополнительную сумму к зарплате с учетом Вашей классификации.

Классы объектов

Сотрудники (Фамилия, Имя, Отчество, Оклад).

Виды работ (Описание, Оплата за день).

Работы (Сотрудник, Вид работ, Дата начала, Дата окончания).

Развитие постановки задачи

Теперь ситуация изменилась. Выяснилось, что некоторые из дополнительных работ являются достаточно трудоемкими и, в то же время, срочными, что требует привлечения к их выполнению нескольких сотрудников. Также оказалось, что длительность работ в каждом конкретном случае

составляет разную величину. Соответственно, нужно заранее планировать длительность работы и количество сотрудников, занятых для выполнения работы.

0.3.12 Техническое обслуживание станков

Описание предметной области

Ваше предприятие занимается ремонтом станков и другого промышленного оборудования. Вашей задачей является отслеживание финансовой стороны деятельности предприятия.

Клиентами Вашей компании являются промышленные предприятия, оснащенные различным сложным оборудованием. В случае поломок оборудования они обращаются к Вам.

Ремонтные работы в Вашей компании организованы следующим образом: все станки проклассифицированы по странам-производителям, годам выпуска и маркам. Все виды ремонта отличаются названием, продолжительностью в днях, стоимостью. Исходя из этих данных, по каждому факту ремонта Вы фиксируете вид станка и дату начала ремонта.

Классы объектов

Виды станков (Страна, Год выпуска, Марка).

Виды ремонта (Название, Продолжительность, Стоимость, Примечания).

Ремонт (Вид станка, Ремонт, Дата начала, Примечания).

Развитие постановки задачи

Теперь ситуация изменилась. Несложный анализ показал, что нужно не просто подразделять станки по типам, а иметь информацию о том, сколько раз ремонтировался тот или иной конкретный станок.

0.3.13 Туристическая фирма

Описание предметной области

Вы работаете в туристической компании. Ваша компания работает с клиентами, продавая им путевки. Вашей задачей является отслеживание финансовой стороны деятельности фирмы.

Работа с клиентами в Вашей компании организована следующим образом: у каждого клиента, пришедшего к Вам, собираются некоторые стандартные данные – фамилия, имя, отчество, адрес, телефон. После этого Ваши сотрудники выясняют у клиента, куда он хотел бы поехать отдыхать. При этом ему демонстрируются различные варианты, включающие страну проживания, особенности местного климата, имеющиеся отели разного класса. Наряду с этим, обсуждается возможная длительность пребывания и стоимость путевки. В случае если удалось договориться, и найти для клиента приемлемый вариант, Вы регистрируете факт продажи путевки (или путевок, если клиент покупает сразу несколько путевок), фиксируя дату отправления. Иногда Вы решаете предоставить клиенту некоторую скидку.

Классы объектов

Маршруты (Страна, Климат, Длительность, Отель, Стоимость).

Путевки (Маршрут, Клиент, Дата отправления, Количество, Скидка).

Клиенты (Фамилия, Имя, Отчество, Адрес, Телефон).

Развитие постановки задачи

Теперь ситуация изменилась. Фирма работает с несколькими отелями в нескольких странах. Путевки продаются на одну, две или четыре недели. Стоимость путевки зависит от длительности тура и

отеля. Скидки, которые предоставляет фирма, фиксированы. Например, при покупке более 1 путевки, предоставляется скидка 5

0.3.14 Грузовые перевозки

Описание предметной области

Вы работаете в компании, занимающейся перевозками грузов. Вашей задачей является отслеживание стоимости перевозок с учетом заработной платы водителей.

Ваша компания осуществляет перевозки по различным маршрутам. Для каждого маршрута Вы определили некоторое название, вычислили примерное расстояние и установили некоторую оплату для водителя. Информация о водителях включает фамилию, имя, отчество и стаж. Для проведения расчетов Вы храните полную информацию о перевозках (маршрут, водитель, даты отправки и прибытия). По факту некоторых перевозок водителям выплачивается премия.

Классы объектов

Маршруты (Название, Дальность, Количество дней в пути, Оплата).

Водители (Фамилия, Имя, Отчество, Стаж).

Проделанная работа (Маршрут, Водитель, Дата отправки, Дата возвращения, Премия).

Развитие постановки задачи

Теперь ситуация изменилась. Ваша фирма решила ввести гибкую систему оплаты. Так, оплата водителям должна теперь зависеть не только от маршрута, но и от стажа водителя. Кроме того, нужно учесть, что перевозки могут осуществлять два водителя.

0.3.15 Учет телефонных переговоров

Описание предметной области

Вы работаете в коммерческой службе телефонной компании. Компания предоставляет абонентам телефонные линии для междугородних переговоров. Вашей задачей является отслеживание стоимости междугородних телефонных переговоров.

Абонентами компании являются юридические лица, имеющие телефонную точку, ИНН, расчетный счет в банке. Стоимость переговоров зависит от города, в который осуществляется звонок, и времени суток (день, ночь). Каждый звонок абонента автоматически фиксируется в базе данных. При этом запоминаются город, дата, длительность разговора и время суток.

Классы объектов

Абоненты (Номер телефона, ИНН, Адрес).

Города (Название, Тариф дневной, Тариф ночной).

Переговоры (Абонент, Город, Дата, Количество минут, Время суток).

Развитие постановки задачи

Теперь ситуация изменилась. Ваша фирма решила ввести гибкую систему скидок. Так, стоимость минуты теперь уменьшается в зависимости от длительности разговора. Размер скидки для каждого города разный.

0.3.16 Учет внутриофисных расходов

Описание предметной области

Вы работаете в бухгалтерии частной фирмы. Сотрудники фирмы имеют возможность осуществлять мелкие покупки для нужд фирмы, предоставляя в бухгалтерию товарный чек. Вашей задачей является отслеживание внутриофисных расходов.

Ваша фирма состоит из отделов. Каждый отдел имеет название. В каждом отделе работает определенное количество сотрудников. Сотрудники могут осуществлять покупки в соответствии с видами расходов. Каждый вид расходов имеет название, некоторое описание и предельную сумму средств, которые могут быть потрачены по данному виду расходов в месяц. При каждой покупке сотрудник оформляет документ, где указывает вид расхода, дату, сумму и отдел.

Классы объектов

Отделы (Название, Количество сотрудников).

Виды расходов (Название, Описание, Предельная норма).

Расходы (Вид расходов, Отдел, Сумма, Дата).

Развитие постановки задачи

Теперь ситуация изменилась. Оказалось, что нужно хранить данные о расходах не только в целом по отделу, но и по отдельным сотрудникам. Нормативы по расходованию средств устанавливаются не в целом, а по каждому отделу за каждый месяц. Неиспользованные в текущем месяце деньги могут быть использованы позже.

0.3.17 Библиотека

Описание предметной области

Вы являетесь руководителем библиотеки. Ваша библиотека решила зарабатывать деньги, выдавая напрокат некоторые книги, имеющиеся в небольшом количестве экземпляров. Вашей задачей является отслеживание финансовых показателей работы библиотеки.

У каждой книги, выдаваемой в прокат, есть название, автор, жанр. В зависимости от ценности книги Вы определили для каждой из них залоговую стоимость (сумма, вносимая клиентом при взятии книги напрокат) и стоимость проката (сумма, которую клиент платит при возврате книги, получая назад залог). В библиотеку обращаются читатели. Все читатели регистрируются в картотеке, которая содержит стандартные анкетные данные (фамилия, имя, отчество, адрес, телефон). Каждый читатель может обращаться в библиотеку несколько раз. Все обращения читателей фиксируются, при этом по каждому факту выдачи книги запоминаются дата выдачи и ожидаемая дата возврата.

Классы объектов

Книги (Название, Автор, Залоговая стоимость, Стоимость проката, Жанр).

Читатели (Фамилия, Имя, Отчество, Адрес, Телефон).

Выданные книги (Книга, Читатель, Дата выдачи, Дата возврата).

Развитие постановки задачи

Теперь ситуация изменилась. Несложный анализ показал, что стоимость проката книги должна зависеть не только от самой книги, но и от срока ее проката. Кроме того, необходимо добавить систему штрафов за вред, нанесенный книге и систему скидок для некоторых категорий читателей.

0.3.18 Прокат автомобилей

Описание предметной области

Вы являетесь руководителем коммерческой службы в фирме, занимающейся прокатом автомобилей. Вашей задачей является отслеживание финансовых показателей работы пункта проката.

В Ваш автопарк входит некоторое количество автомобилей различных марок, стоимостей и типов. Каждый автомобиль имеет свою стоимость проката. В пункт проката обращаются клиенты. Все клиенты проходят обязательную регистрацию, при которой о них собирается стандартная информация (фамилия, имя, отчество, адрес, телефон). Каждый клиент может обращаться в пункт проката несколько раз. Все обращения клиентов фиксируются, при этом по каждой сделке запоминаются дата выдачи и ожидаемая дата возврата.

Классы объектов

Автомобили (Марка, Стоимость, Стоимость проката, Тип).

Клиенты (Фамилия, Имя, Отчество, Адрес, Телефон).

Выданные автомобили (Автомобиль, Клиент, Дата выдачи, Дата возврата).

Развитие постановки задачи

Теперь ситуация изменилась. Несложный анализ показал, что стоимость проката автомобиля должна зависеть не только от самого автомобиля, но и от срока его проката, а также от года выпуска. Также нужно ввести систему штрафов за возвращение автомобиля в ненадлежащем виде и систему скидок для постоянных клиентов.

0.3.19 Выдача банком кредитов

Описание предметной области

Вы являетесь руководителем информационно-аналитического центра коммерческого банка. Одним из существенных видов деятельности Вашего банка является выдача кредитов юридическим лицам. Вашей задачей является отслеживание динамики работы кредитного отдела.

В зависимости от условий получения кредита, процентной ставки и срока возврата все кредитные операции делятся на несколько основных видов. Каждый из этих видов имеет свое название. Кредит может получить юридическое лицо (клиент), при регистрации предоставивший следующие сведения: название, вид собственности, адрес, телефон, контактное лицо. Каждый факт выдачи кредита регистрируется банком, при этом фиксируются сумма кредита, клиент и дата выдачи.

Классы объектов

Виды кредитов (Название, Условия получения, Ставка, Срок).

Клиенты (Название, Вид собственности, Адрес, Телефон, Контактное лицо).

Кредиты (Вид кредитов, Клиент, Сумма, Дата выдачи).

Развитие постановки задачи

Теперь ситуация изменилась. После проведения различных исследований выяснилось, что используемая система не позволяет отслеживать динамику возврата кредитов. Для устранения этого недостатка Вы приняли решение учитывать в системе еще и дату фактического возврата денег. Нужно еще учесть, что кредит может гаситься частями, и за задержку возврата кредита начисляются штрафы.

0.3.20 Инвестирование свободных средств

Описание предметной области

Вы являетесь руководителем аналитического центра инвестиционной компании. Ваша компания занимается вложением денежных средств в ценные бумаги.

Ваши клиенты – предприятия, которые доверяют Вам управлять их свободными денежными средствами на определенный период. Вам необходимо выбрать вид ценных бумаг, которые позволят получить прибыль и Вам и Вашему клиенту. При работе с клиентом для Вас весьма существенной является информация о предприятии – название, вид собственности, адрес и телефон.

Классы объектов

Ценные бумаги (Код ценной бумаги, Минимальная сумма сделки, Рейтинг, Доходность за прошлый год, Дополнительная информация).

Инвестиции (Ценная бумага, Клиент, Котировка, Дата покупки, Дата продажи).

Клиенты (Клиент, Название, Вид собственности, Адрес, Телефон).

Развитие постановки задачи

При эксплуатации базы данных стало понятно, что необходимо хранить историю котировок каждой ценной бумаги. Кроме того, помимо вложений в ценные бумаги, существует возможность вкладывать деньги в банковские депозиты.

0.3.21 Занятость актеров театра

Описание предметной области

Вы являетесь коммерческим директором театра, и в Ваши обязанности входит вся организационно-финансовая работа, связанная с привлечением актеров и заключением контрактов.

Вы поставили дело следующим образом: каждый год театр осуществляет постановку различных спектаклей. Каждый спектакль имеет определенный бюджет. Для участия в конкретных постановках в определенных ролях Вы привлекаете актеров. С каждым из актеров Вы заключаете персональный контракт на определенную сумму. Каждый из актеров имеет некоторый стаж работы, некоторые из них удостоены различных наград и званий.

Классы объектов

Актеры (Фамилия, Имя, Отчество, Звание, Стаж).

Спектакли (Название, Год постановки, Бюджет).

Занятость актеров в спектакле (Актер, Спектакль, Роль, Стоимость годового контракта).

Развитие постановки задачи

В результате эксплуатации базы данных выяснилось, что в рамках одного спектакля на одну и ту же роль привлекается несколько актеров. Контракт определяет базовую зарплату актера, а по итогам реально отыгранных спектаклей актеру назначается премия. Кроме того, в базе данных нужно хранить информацию за несколько лет.

0.3.22 Платная поликлиника

Описание предметной области

Вы являетесь руководителем службы планирования платной поликлиники. Вашей задачей является отслеживание финансовых показателей работы поликлиники.

В поликлинике работают врачи различных специальностей, имеющие разную квалификацию. Каждый день в поликлинику обращаются больные. Все больные проходят обязательную регистрацию, при которой в базу данных заносятся стандартные анкетные данные (фамилия, имя, отчество, год рождения). Каждый больной может обращаться в поликлинику несколько раз, нуждаясь в различной медицинской помощи. Все обращения больных фиксируются, при этом устанавливается диагноз, определяется стоимость лечения, запоминается дата обращения.

Классы объектов

Врачи (Фамилия, Имя, Отчество, Специальность, Категория).

Пациенты (Фамилия, Имя, Отчество, Год рождения).

Обращения (Врач, Пациент, Дата обращения, Диагноз, Стоимость лечения).

Развитие постановки задачи

В результате эксплуатации базы данных выяснилось, что при обращении в поликлинику пациент обследуется и проходит лечение у разных специалистов. Общая стоимость лечения зависит от стоимости тех консультаций и процедур, которые назначены пациенту. Кроме того, для определенных категорий граждан предусмотрены скидки.

0.3.23 Анализ динамики показателей финансовой отчетности различных предприятий

Описание предметной области

Вы являетесь руководителем информационно-аналитического центра крупного холдинга. Вашей задачей является отслеживание динамики показателей для предприятий Вашего холдинга.

В структуру холдинга входят несколько предприятий. Каждое предприятие имеет стандартные характеристики (название, реквизиты, телефон, контактное лицо). Работа предприятия может быть оценена следующим образом: в начале каждого отчетного периода на основе финансовой отчетности вычисляется по неким формулам определенный набор показателей. Принять, что важность показателей характеризуется некоторыми числовыми константами. Значение каждого показателя измеряется в некоторой системе единиц.

Классы объектов

Показатели (Название, Важность, Единица измерения).

Предприятия (Название, Банковские реквизиты, Телефон, Контактное лицо).

Динамика показателей (Показатель, Предприятие, Дата, Значение).

Развитие постановки задачи

В результате эксплуатации базы данных выяснилось, что некоторые показатели считаются в рублях, некоторые в долларах, некоторые в евро. Для удобства работы с показателями нужно хранить изменения курсов валют относительно друг друга.

0.3.24 Учет телекомпанией стоимости прошедшей в эфире рекламы

Описание предметной области

Вы являетесь руководителем коммерческой службы телевизионной компании. Вашей задачей является отслеживание расчетов, связанных с прохождением рекламы в телеэфире.

Работа построена следующим образом: заказчики просят поместить свою рекламу в определенной передаче в определенный день. Каждый рекламный ролик имеет определенную продолжительность. Для каждой организации-заказчика известны банковские реквизиты, телефон и контактное лицо для проведения переговоров. Передачи имеют определенный рейтинг. Стоимость минуты рекламы в каждой конкретной передаче известна (определяется коммерческой службой, исходя из рейтинга передачи и прочих соображений).

Классы объектов

Передачи (Название, Рейтинг, Стоимость минуты).

Реклама (Передача, Заказчик, Дата, Длительность в минутах).

Заказчики (Название, Банковские реквизиты, Телефон, Контактное лицо).

Развитие постановки задачи

В результате эксплуатации базы данных выяснилось, что необходимо также хранить информацию об агентах, заключивших договоры на рекламу. Зарплата рекламных агентов составляет некоторый процент от общей стоимости рекламы, прошедшей в эфире.

0.3.25 Интернет-магазин

Описание предметной области

Вы являетесь сотрудником коммерческого отдела компании, продающей различные товары через Интернет. Вашей задачей является отслеживание финансовой составляющей работы компании.

Работа Вашей компании организована следующим образом: на Интернет-сайте компании представлены (выставлены на продажу) некоторые товары. Каждый из них имеет некоторое название, цену и единицу измерения (штуки, килограммы, литры). Для проведения исследований и оптимизации работы магазина Вы пытаетесь собирать данные с Ваших клиентов. При этом для Вас определяющее значение имеют стандартные анкетные данные, а также телефон и адрес электронной почты для связи. В случае приобретения товаров на сумму свыше 5000р. клиент переходит в категорию «постоянных клиентов» и получает скидку на каждую покупку в размере 2%. По каждому факту продажи Вы автоматически фиксируете клиента, товары, количество, дату продажи, дату доставки.

Классы объектов

Товары (Название, Цена, Единица измерения).

Клиенты (Фамилия, Имя, Отчество, Адрес, Телефон, email, Признак постоянного клиента).

Продажи (Товар, Клиент, Дата продажи, Дата доставки, Количество).

Развитие постановки задачи

В результате эксплуатации базы данных выяснилось, что иногда возникают проблемы, связанные с нехваткой информации о наличии нужных товаров на складе в нужном количестве. Кроме того,

обычно клиенты в рамках одного заказа покупают не один вид товара, а несколько видов. Исходя из суммарной стоимости заказа, компания предоставляет дополнительные скидки.

0.3.26 Ювелирная мастерская

Описание предметной области

Вы работаете в ювелирной мастерской. Ваша мастерская осуществляет изготовление ювелирных изделий для частных лиц на заказ. Вы работаете с определенными материалами (платина, золото, серебро, различные драгоценные камни и т.д.). При обращении к Вам потенциального клиента Вы определяетесь с тем, какое именно изделие ему необходимо. Все изготавливаемые Вами изделия принадлежат к некоторому типу (серьги, кольца, броши, браслеты), бывают выполнены из определенного материала, имеют некоторый вес и цену (включающую стоимость материалов и работы).

Классы объектов

Изделия (Название, Тип, Материал, Вес, Цена).

Материалы (Название, Цена за грамм).

Продажи (Изделие, Дата продажи, Фамилия покупателя, Имя покупателя, Отчество покупателя).

Развитие постановки задачи

В процессе опытной эксплуатации базы данных выяснилось, что ювелирное изделие может состоять из нескольких материалов. Кроме того, постоянным клиентам мастерская предоставляет скидки.

0.3.27 Парикмахерская

Описание предметной области

Вы работаете в парикмахерской.

Ваша парикмахерская стрижет клиентов в соответствии с их пожеланиями и некоторым каталогом различных видов стрижки. Так, для каждой стрижки определены название, принадлежность полу (мужская, женская), стоимость работы. Для наведения порядка Вы, по мере возможности, составляете базу данных клиентов, запоминая их анкетные данные (фамилия, имя, отчество). Начиная с 5-ой стрижки, клиент переходит в категорию постоянных и получает скидку в 3% при каждой последующей стрижке. После того, как закончена очередная работа, в кассе фиксируются стрижка, клиент и дата производства работ.

Классы объектов

Стрижки (Название, Пол, Стоимость).

Клиенты (Фамилия, Имя, Отчество, Пол, Признак постоянного клиента).

Работа (Стрижка, Клиент, Дата).

Развитие постановки задачи

Теперь ситуация изменилась. У Вашей парикмахерской появился филиал, и Вы хотели бы видеть, в том числе, и отдельную статистику по филиалам. Кроме того, стоимость стрижки может меняться с течением времени. Нужно хранить не только последнюю цену, но и все данные по изменению цены стрижки.

0.3.28 Химчистка

Описание предметной области

Вы работаете в химчистке.

Ваша химчистка осуществляет прием у населения вещей для выведения пятен. Для наведения порядка Вы, по мере возможности, составляете базу данных клиентов, запоминая их анкетные данные (фамилия, имя, отчество). Начиная с 3-го обращения, клиент переходит в категорию постоянных клиентов и получает скидку в 3% при чистке каждой последующей вещи. Все оказываемые Вами услуги подразделяются на виды, имеющие название, тип и стоимость, зависящую от сложности работ. Работа с клиентом первоначально состоит в определении объема работ, вида услуги и, соответственно, ее стоимости. Если клиент согласен, он оставляет вещь (при этом фиксируется услуга, клиент и дата приема) и забирает ее после обработки (при этом фиксируется дата возврата).

Классы объектов

Виды услуг (Название, Тип, Стоимость).

Клиенты (Фамилия, Имя, Отчество, Признак постоянного клиента).

Услуги (Вид услуги, Клиент, Дата приема, Дата возврата).

Развитие постановки задачи

Теперь ситуация изменилась. У Вашей химчистки появился филиал, и Вы хотели бы видеть, в том числе, и отдельную статистику по филиалам. Кроме того, вы решили делать надбавки за срочность и сложность работ.

0.3.29 Сдача в аренду торговых площадей

Описание предметной области

Вы работаете в крупном торговом центре, сдающим в аренду коммерсантам свои торговые площади.

Вашей задачей является наведение порядка в финансовой стороне работы торгового центра.

Работы Вашего торгового центра построена следующим образом: в результате планирования Вы определили некоторое количество торговых точек в пределах Вашего здания, которые могут сдаваться в аренду. Для каждой из торговых точек важными данными являются этаж, площадь, наличие кондиционера и стоимость аренды в день. Со всех потенциальных клиентов Вы собираете стандартные данные (название, адрес, телефон, реквизиты, контактное лицо). При появлении потенциального клиента Вы показываете ему имеющиеся свободные площади. При достижении соглашения Вы оформляете договор, фиксируя в базе данных торговую точку, клиента, период (срок) аренды.

Классы объектов

Торговые точки (Этаж, Площадь, Наличие кондиционера, Стоимость аренды в день).

Клиенты (Название, Реквизиты, Адрес, Телефон, Контактное лицо).

Аренда (Торговая точка, Клиент, Дата начала, Дата окончания).

Развитие постановки задачи

В результате эксплуатации базы данных выяснилось, что некоторые клиенты арендуют сразу несколько торговых точек. Помимо этого, Вам необходимо собирать информацию об ежемесячных платежах, поступающих Вам от арендаторов.

Модуль 1

1 Качество ПО

Качество - это цель инженерной деятельности; построение качественного программного обеспечения (ПО) (software) - цель программной инженерии (software engineering).

Определение 1. *ПО* это набор программ и соответствующая им документация.

В данном пособии рассматриваются средства и технические приемы, позволяющие значительно улучшить качество ПО. Прежде чем приступить к изучению этих средств и приемов, следует хорошо представлять нашу цель. Качество ПО лучше всего описывается комбинацией ряда факторов. В этой лекции мы постараемся проанализировать некоторые из них, и покажем, где необходимы улучшения.

1.1 Внешние и внутренние факторы

Все мы хотим, чтобы наше ПО было быстродействующим, надежным, легким в использовании, читаемым, модульным, структурным и т.д. Но эти определения описывают два разных типа качества. Наличие или отсутствие таких качеств, как скорость и простота использования ПО, может быть обнаружено его пользователями. Эти качества можно назвать *внешними* факторами качества.

Под словом “пользователи” нужно понимать не только людей, взаимодействующих с конечным продуктом, но и тех, кто их закупает, занимается администрированием. Такое свойство, например, как легкость адаптации продуктов к изменениям спецификаций - далее определенная в нашей

дискуссии как расширяемость - попадает в категорию внешних факторов, поскольку она может представлять интерес для администраторов, закупающих продукт, хотя и не важна для “конечных пользователей”, непосредственно работающих с продуктом.

Такие характеристики ПО, как модульность или читаемость, являются *внутренними* факторами, понятными только для профессионалов, имеющих доступ к тексту ПО.

В конечном счете, только внешние факторы имеют значение. Но ключ к достижению внешних факторов скрыт во внутренних факторах: для того, чтобы достичь видимого качества, проектировщики и конструкторы должны иметь внутренние приемы, позволяющие улучшать скрытые от пользователя качества.

1.2 Обзор внешних факторов

Рассмотрим самые важные внешние факторы качества, стремление к которым есть центральная задача ОО-построения ПО.

Корректность (Correctness)

Определение 2. *Корректность* - это способность ПО выполнять точные задачи так, как они определены их спецификацией.

Корректность является важнейшим качеством. Если система не делает того, что она должна делать, то все остальное - ее быстродействие, хороший пользовательский интерфейс - не имеет особого значения.

Методы обеспечения корректности обычно условны. Серьезная система ПО, даже небольшая по нынешним меркам, использует столь многое, что невозможно гарантировать ее корректность, рабо-



Рис. 1.1: Слои в разработке ПО

тая со всеми компонентами на одном уровне. Необходим многоуровневый подход: В условном подходе к корректности мы заботимся только о том, чтобы обеспечить корректность каждого уровня, основываясь на предположении, что нижележащие уровни корректны. Это единственно реалистичный подход, поскольку он позволяет разделить проблему и на каждой ступени сконцентрироваться на ограниченном круге задач. Нельзя проверить, что программа на языке высокого уровня корректна, если не предположить, что используемый компилятор корректно реализует язык. Это не слепое доверие компилятору, а разделение проблемы на две: проверка корректности компилятора и проверка корректности программы относительно семантики языка.

Здесь также применим условный подход: следует обеспечить корректность библиотек и корректность приложения при условии, что библиотеки корректны.

Многие практики полагают, что достижение корректности ПО связано с тестированием и исправлением ошибок. В дальнейших лекциях исследуется ряд технических приемов, в частности типизация и метод утверждений, направленных на построение ПО, корректного с самого начала. Исправление ошибок и тестирование, конечно, остаются необходимыми как средства дополнительной проверки результата. Можно было бы пойти дальше и принять совсем формальный подход к построению ПО. Это не является целью наших лекций, как ясно из несколько “робких” терминов

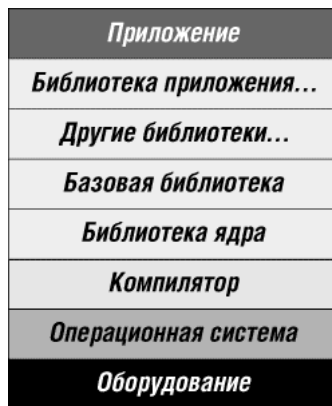


Рис. 1.2: Уровни в процессе разработки, включающем повторное использование



Рис. 1.3: Устойчивость против корректности

- “проверять”, “гарантировать”, “обеспечивать”, используемых выше вместо слова “доказывать”. Все же многие из описанных ниже технических приемов происходят непосредственно от математических методов формальной спецификации и верификации программ, проходя длинный путь к обеспечению идеала корректности.

Устойчивость (Robustness)

Определение 3. *Устойчивость* - это способность ПО соответствующим образом реагировать на аварийные ситуации.

Устойчивость дополняет корректность. Корректность относится к поведению системы в случаях, определенных спецификацией; устойчивость характеризует то, что происходит за пределами этой спецификации. Как видно из определения, устойчивость по своей природе более нечеткое понятие, чем корректность. Невозможно сказать, как в случае с корректностью, что в аварийных ситуациях система должна “выполнять свои задачи”, поскольку ситуации выходят за пределы спецификации. Если бы эти задачи были известны, аварийный случай стал бы частью спецификации, и мы бы снова

вернулись в область корректности.

Это определение “аварийной ситуации” нам еще понадобится при изучении обработки исключений. Оно подразумевает, что понятия нормальной и аварийной ситуации всегда относительно по отношению к заданной спецификации; ситуация аварийна, если она выходит за рамки спецификации. Если расширить спецификацию, аварийные случаи становятся нормальными - даже если они соответствуют таким нежелательным событиям, как, например, ошибочный ввод пользователя.

Всегда будут существовать случаи, на которые спецификация явно не распространяется. Роль требования устойчивости - удостовериться, что и в таких случаях система не приводит к непоправимой ситуации; она должна выдать соответствующее сообщение об ошибке, гладко завершить работу или войти в так называемый режим “постепенного вывода из работы”.

Расширяемость (Extendibility)

Определение 4. *Расширяемость* - это легкость адаптации ПО к изменениям спецификации.

Проблема расширяемости это проблема масштаба. Для маленьких программ изменение не является обычно большой проблемой, но по мере увеличения ПО адаптация становится все труднее. Большая программная система часто видится как огромный картонный дом, удаление одного элемента может привести к разрушению всего построения.

Традиционные подходы к построению ПО не уделяли должного внимания изменениям. Они скорее исходили из идеального взгляда на жизненный цикл ПО, где требования замораживаются после завершения первоначальной ступени анализа. Последующий процесс посвящался проектированию и построению решения при фиксированных требованиях. Это вполне понятно: на том этапе развития дисциплины задача состояла в разработке надежных технических приемов для постановки и решения фиксированных проблем. Но сейчас стало возможным признать и рассмотреть центральный

вопрос - что делать, если проблема изменяется в ходе ее решения. Изменения характерны для процесса разработки ПО: меняются требования, наше понимание требований, алгоритмы, представление данных, приемы реализации. Поддержка изменений является основной целью объектной технологии и постоянной темой наших лекций.

Хотя многие из технических приемов, улучшающих расширяемость, можно объяснить во вводных курсах и на небольших примерах, их значимость становится явной только для больших проектов. Для улучшения расширяемости важны два принципа:

- *Простота построения*: простая архитектура легче адаптируется к изменениям, чем сложная.
- *Децентрализация*: чем более автономны модули, тем выше вероятность того, что простое изменение затронет только один или небольшое количество модулей и не вызовет цепную реакцию изменений во всей системе.

Повторное использование (Reusability)

Определение 5. *Повторное использование* есть способность элементов ПО служить для построения многих различных приложений.

Необходимость и возможность повторного использования возникает из наблюдений сходства систем - системы ПО часто имеют похожую схему. Следует использовать это сходство и не изобретать велосипед заново. Понимание этой схемы даст возможность повторно применять созданный элемент ПО во многих других разработках.

Повторное использование влияет на все остальные аспекты качества ПО. Поскольку решение проблемы повторного использования в сущности означает, что нужно писать меньше программ, следовательно, можно прилагать больше усилий (при той же общей стоимости) к улучшению других факторов, таких как, например, корректность и устойчивость.

Совместимость (Compatibility)

Определение 6. *Совместимость* - это легкость сочетания одних элементов ПО с другими.

Совместимость важна, поскольку мы не разрабатываем элементы ПО в вакууме: им необходимо взаимодействовать друг с другом. Но при этом слишком часто возникают проблемы, поскольку суждения разных элементов об остальном мире противоречивы. Простейшим примером может служить широкое разнообразие несовместимых файловых форматов, из-за чего, например, одна программа не может непосредственно использовать результат работы другой программы.

Ключ к совместимости находится в однородности построения и в стандартных соглашениях на коммуникации между программами. Эти подходы включают:

- Стандартные форматы файлов, как в системе Unix, где каждый текстовый файл - это просто последовательность символов.
- Стандартные структуры данных, как в системе Lisp, где все данные, а также программы, представлены бинарными деревьями (называемыми списками).
- Стандартные пользовательские интерфейсы, как в различных версиях Windows, Unix/Linux и MacOS, где все инструменты опираются на единую парадигму для коммуникации с пользователем, основанную на стандартных компонентах, таких как окна, значки, меню и т. д.

Большая общность достигается при определении стандартных протоколов доступа ко всем важным элементам, управляемым программами. Такова идея, лежащая в основе абстрактных типов данных и ОО-подхода, а также так называемого связующего программного обеспечения (middleware), например CORBA и Microsoft's OLE-COM (ActiveX).

Эффективность (Efficiency)

Определение 7. *Эффективность* - это способность ПО как можно меньше зависеть от ресурсов оборудования: процессорного времени, пространства, занимаемого во внутренней и внешней памяти, пропускной способности, используемой в устройствах связи.

Почти синонимом эффективности является слово “производительность” (performance). В программистском сообществе есть два типичных отношения к эффективности:

- Некоторые разработчики одержимы проблемами производительности, что заставляет их прилагать много усилий к предполагаемой оптимизации.
- Существует общая тенденция недооценки вопросов эффективности, вытекающая из справедливых убеждений, существующих в промышленности: “сделай правильно, прежде чем сделать быстро” и “модель компьютера будущего года все равно будет на 50% быстрее”.

Постоянное увеличение компьютерной мощности, каким бы оно ни было впечатляющим, не может заменить эффективность, по крайней мере, по трем причинам:

- Тот, кто покупает большой и более быстрый компьютер, хочет видеть действительные выгоды от дополнительной мощности - решать новые задачи, более быстро работать со старыми задачами, решать более важные версии старых задач за то же время. Если новый компьютер решает старые задачи за то же самое время - это нехорошо!
- Явный эффект повышения мощности компьютера сказывается тогда, когда велика доля “хороших” алгоритмов по отношению к плохим. Предположим, что новая машина работает в два раза быстрее, чем старая. Пусть n - размер решаемой задачи, а N - максимальный размер,

при котором удастся решить задачу на старом компьютере за приемлемое время. Если используется линейный алгоритм, временная сложность которого $O(n)$, то новый компьютер даст возможность решить задачу вдвое большего размера - $2 * N$. Для квадратичного алгоритма со сложностью $O(n^2)$ увеличение N составит только 41%. Переборный алгоритм со сложностью $O(2^n)$ добавит к N только единицу - небольшое улучшение за такие деньги.

- В некоторых случаях эффективность может влиять на корректность. Спецификация может устанавливать, что ответ компьютера на определенное событие должен произойти не позже, чем за определенное время, например, бортовой компьютер должен быть готов определить и обработать сообщение с сенсора рычага управления двигателя достаточно быстро, чтобы сработало корректирующее действие. Эта связь между эффективностью и корректностью не ограничивается приложениями, работающими “в реальном времени”; немногие люди заинтересуются моделью предсказания погоды, которой требуется 24 часа, чтобы предсказать погоду на завтра.

Переносимость (Portability)

Определение 8. *Переносимость* - это легкость переноса ПО в различные программные и аппаратные среды.

Переносимость имеет дело с разнообразием не только физического оборудования, но чаще аппаратно-программного механизма, того, который мы действительно программируем, включающего операционную систему, систему окон, если она применяется, и другие основные инструменты.

Простота использования (Easy of Use)

Определение 9. *Простота использования* - это легкость, с которой люди с различными знаниями и квалификацией могут научиться использовать ПО и применять его для решения задач. Сюда также относится простота установки, работы и текущего контроля.

Определение подчеркивает наличие различных уровней опытности потенциальных пользователей. Это требование ставит одну из важных проблем перед проектировщиками ПО, занимающимися простотой использования: как обеспечить подробное руководство и объяснения начинающим пользователям, не мешая умелым пользователям, которые сразу хотят приняться за работу?

Желательно, чтобы проектировщики ПО, озабоченные простотой использования, с некоторым недоверием рассматривали принцип “знай пользователя”. Подразумевается, что хороший проектировщик должен приложить усилие для понимания того, для каких пользователей предназначена система. Этот взгляд игнорирует одно из свойств успешной системы: она всегда выходит за пределы предполагаемого круга пользователей. Напомню два старых известных примера - язык Fortran разрабатывался как инструмент для решения задачи небольшого сообщества инженеров и ученых, программирующих на IBM 704, операционная система Unix предназначалась для внутреннего использования в Bell Laboratories. Система, изначально спроектированная для особой группы людей, исходит из предположений, которые просто не будут работать для более широкой группы.

Хорошие проектировщики пользовательского интерфейса придерживаются более осмотрительной политики. Они делают как можно меньше предположений относительно своих пользователей. При проектировании интерактивной системы можно считать, что пользователи просто люди и что они умеют читать, двигать мышью, нажимать кнопки и набирать текст (медленно), и не более. Если ПО создается для специализированной области приложения, вероятно, можно, предположить, что пользователи знакомы с ее основными концепциями. Но даже это рискованно.

Принцип построения пользовательского интерфейса — Не делайте вид, что вы знаете пользо-

вателя - это не так.

Функциональность (Functionality)

Определение 10. *Функциональность* - это степень возможностей, обеспечиваемых системой.

Одна из самых трудных проблем, с которой сталкивается руководитель проекта, - определение достаточной функциональности. Всегда существует желание добавлять в систему все новые и новые свойства. Желание, известное на языке индустрии как *фичеризм* (featurism) , часто *ползучий фичеризм* (creeping featurism) . Его последствия плачевны для внутренних проектов, где давление исходит от разных групп пользователей внутри одной и той же компании. Они еще хуже для коммерческих продуктов, испытывающих давление, например от журналистских сравнительных обзоров, представляющих чаще всего таблицу, включающую одновременно свойства разных конкурирующих продуктов.

Расширение свойств системы приводит к двум проблемам, одна сложнее другой. Более простая проблема - потеря непротиворечивости, которая может возникнуть при добавлении новых свойств, затрагивающих простоту использования. Известно, что пользователи жалуются, что все украшения новой версии продукта делают его ужасно сложным. Однако таким комментариям не стоит слишком доверять. Новые свойства не возникают из ничего - в основном они возникают из спроса пользователей, других пользователей. Что для меня выглядит ненужной безделушкой, может для вас быть необходимым свойством.

Каково же решение проблемы? Необходимо снова и снова работать над состоянием всего продукта, пытаясь привести его в соответствие с общим замыслом. Хорошее ПО основывается на небольшом количестве сильных идей. У него может быть много специальных свойств - все они должны быть следствиями основных положений. “Великий план” должен быть виден, и в нем всему должно отводиться свое место.

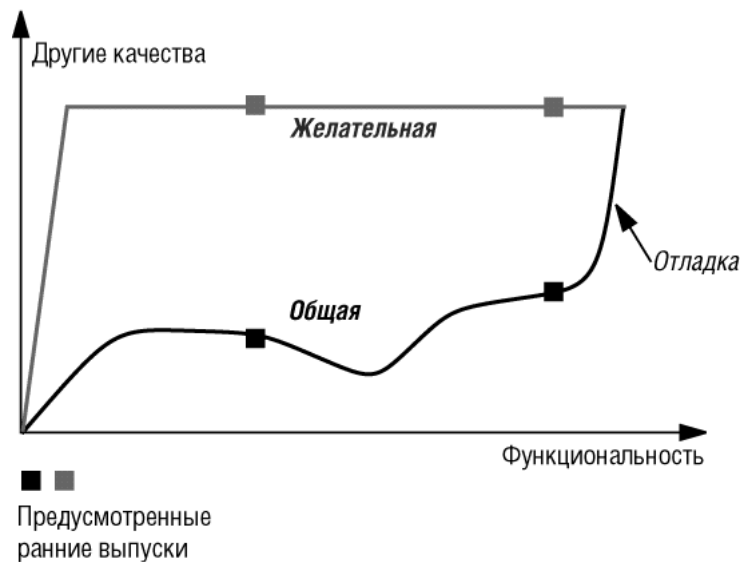


Рис. 1.4: Кривые Осмонда

Более сложная проблема - слишком большое внимание к одним свойствам в ущерб другим качествам системы. В проектах часто встречается ошибка, ситуацию, которую описал Роджер Осмонд в виде двух возможных путей работы над проектом: Нижняя кривая описывает фичеризм: в лихорадочной погоне за дополнительными свойствами теряется нить общего качества. Завершающая фаза такого проекта, предполагающая общую корректировку всех свойств, может быть долгой и напряженной. Если под давлением пользователей или конкурентов вы вынуждены выпустить продукт

достаточно быстро - на стадиях, отмеченных на рисунке квадратами, - результат может повредить вашей репутации.

Осмонд предлагает (верхняя кривая) во время создания проекта поддерживать на высоком постоянном уровне качество всех факторов, кроме функциональности. Никаких компромиссов по надежности, расширяемости и прочим факторам: вы просто отказываетесь от добавления новых свойств до тех пор, пока вас удовлетворяют существующие.

Своевременность (Timeliness)

Определение 11. *Своевременность* - это выпуск ПО в нужный момент, то есть тогда или незадолго до того, как у пользователей появилась соответствующая потребность в подобной системе.

Несвоевременность - одно из больших разочарований нашей промышленности. Прекрасное ПО, появляющееся слишком поздно, может совсем не достичь своей цели. Так обстоит дело и в других отраслях промышленности, разница в том, что немногие продукты появляются так же быстро как программные.

Другие качества

Другие качества, кроме тех, которые до сих пор обсуждались, затрагивают пользователей систем ПО и людей, покупающих эти системы или заказывающих их разработки. В частности:

- *Верифицируемость (Verifiability)* - это легкость подготовки процедур приемки, особенно тестовых данных, процедур обнаружения неполадок и трассировки ошибок на этапах заключительной проверки и введения проекта в действие.

- *Целостность* (Integrity) - это способность ПО защищать свои различные компоненты (программы, данные) от несанкционированного доступа и модификации.
- *Восстанавливаемость* (Repairability) - это способность облегчать устранение дефектов.
- *Экономичность* (Economy) сочетается с своевременностью - это способность системы завершиться, не превысив выделенного бюджета или даже не истратив его.

О документации

Казалось бы, наличие хорошей документации это тоже один из факторов качества ПО. Но это не так - напротив, необходимость документации является следствием других факторов качества, рассмотренных выше. Выделим три вида документации:

- Внешнюю, дающую пользователям возможность понять сильные стороны системы и удобство их использования. Необходимость в ней является следствием простоты использования системы.
- Внутреннюю, дающую разработчикам ПО возможность понять структуру и реализацию системы, - следствие требования расширяемости.
- Описывающую интерфейс модулей. Она дает возможность разработчикам понять функции, реализованные модулем, без изучения его реализации. Этот вид документации является следствием требования повторного использования и расширяемости, поскольку документация позволяет определить, будет ли данное изменение влиять на определенный модуль.

Документацию не следует считать независимой частью проекта. Предпочтительнее в максимально возможной степени создавать самодокументируемое ПО. Это справедливо для всех трех видов документации:

- Включение возможности получения справки проясняет соглашения пользовательского интерфейса. Тем самым облегчается задача авторов руководств пользователей и других форм внешней документации.
- Хороший язык реализации устраняет необходимость большей части внешней документации. Это будет одним из главных требований ОО-нотации, разработанной в этой книге.
- Нотация будет поддерживать сокрытие информации и другие технические приемы (такие как утверждения), позволяющие отделить интерфейс модуля от его реализации. При этом становится возможным из текстов автоматически извлекать документацию интерфейса модулей. Эта тема подробно изучается в лекциях книги. Все эти приемы уменьшают роль традиционной документации, хотя, конечно, не следует ожидать, что они полностью ее заменят.

Компромиссы

В данном обзоре внешних факторов качества ПО мы встретились с требованиями, которые могут конфликтовать друг с другом.

Как можно достичь *целостности*, если не вводить защиты различного рода, что неизбежно затруднит *простоту использования*? Экономичность часто конфликтует с функциональностью.

Оптимальная *эффективность* требует полной адаптации к определенному оборудованию и программной среде, что является противоположностью *переносимости*. Повторное использование требует решения общих задач, что расширяет границы, заданные спецификацией. Давление *своевременности* может склонить нас к технике RAD - быстрой разработки приложения (Rapid Application Development), что может повредить *расширяемости*. Хотя во многих случаях удастся найти решение, примиряющее явно конфликтующие факторы, иногда приходится идти на компромисс.

Разработчики слишком часто и без колебаний идут на компромисс, не давая себе труда рассмотреть соответствующие вопросы и имеющиеся варианты. В таких молчаливых решениях доминирующим фактором обычно является эффективность. По-настоящему инженерный подход к созданию ПО подразумевает работу по ясной формулировке критериев и осознанного выбора вариантов.

Как бы ни были необходимы компромиссы между факторами качества, один из факторов стоит в стороне от остальных - корректность. Нет никакого оправдания тому, что корректность подвергается опасности ради других факторов, таких как эффективность. Если ПО не выполняет свою функцию, все остальное не имеет смысла.

Ключевые вопросы

Все описанные выше факторы важны. Но при современном состоянии индустрии ПО четыре фактора имеют особую важность:

- *Корректность и устойчивость*: все еще слишком трудно создавать ПО без ошибок (bugs), и слишком сложно исправлять ошибки, когда они появляются. Разновидности технических приемов для улучшения корректности и устойчивости одни и те же: более систематические подходы к построению ПО; более формальные спецификации; встроенный контроль в течение всего процесса построения ПО (не просто испытания и отладка после создания); более совершенные языковые механизмы, такие как статическая типизация, утверждения, автоматическое управление памятью и упорядоченное управление исключительными ситуациями, обеспечение возможности разработчикам устанавливать требования корректности и устойчивости в сочетании с возможностью инструментов обнаруживать случаи несостоятельности до того, как они приведут к ошибкам. Близость вопросов корректности и устойчивости делает удобным введение общего термина для обозначения обоих факторов - надежность (reliability).

- *Расширяемость и повторное использование*: ПО должно быть легко изменяемым; компоненты создаваемого ПО должны быть широко применимы, и должен существовать большой перечень общецелевых компонентов, которые можно повторно использовать при разработке новой системы. Здесь также одни и те же идеи полезны для улучшения обоих качеств: любая идея, помогающая производить продукт с более децентрализованной архитектурой, компоненты которой автономны и взаимодействуют только через ограниченные и ясно определенные каналы, будет полезной. Термин *модульность* (modularity) включает повторное использование и расширяемость.

ОО-метод может значительно улучшить четыре основных фактора качества, вот почему он так привлекателен. Он также может внести значительный вклад в другие аспекты, в частности:

- *Совместимость*: метод обеспечивает общий стиль проектирования и стандартизацию интерфейсов модулей и систем, что помогает совместно работать разным системам.
- *Переносимость*: уделяя особое внимание абстракции и скрытию информации, объектная технология способствует тому, что проектировщики начинают отделять спецификацию от особенностей реализации, что и облегчает перенос. Полиморфизм и динамическое связывание делает возможным создание системы, автоматически адаптируемой к аппаратно-программному механизму, например, различным системам окон или различным системам управления базами данных.
- *Простота использования*: вклад ОО-инструментов в современные интерактивные системы, и особенно их пользовательские интерфейсы, так хорошо известен, что иногда он затмевает другие аспекты (люди, создающие рекламу - не единственные, кто называет “объектно-ориентированной” любую систему, использующую значки, окна и ввод с помощью мыши).

- *Эффективность*: как отмечалось выше, повторное использование компонентов профессионального качества часто может значительно улучшить производительность.
- *Своевременность, экономичность и функциональность*: ОО-техника дает возможность тем, кто ее освоил, производить ПО быстрее и по более низкой стоимости; она облегчает добавление функций и даже сама может предложить новые функции.

Несмотря на все эти успехи, мы должны помнить, что ОО-метод - это не панацея, и что многие обычные вопросы проектирования ПО остаются нерешенными. Помощь в решении проблемы - это не то же самое, что ее решение.

1.3 О программном сопровождении

Приведенный список факторов не включил обычно приводимое качество: возможность сопровождения (maintainability). Чтобы понять почему, мы должны поближе взглянуть на лежащее в его основе понятие: *сопровождение* (maintenance) . Сопровождение начинается с момента поставки ПО пользователям.

Обсуждения методологии создания ПО обычно сосредоточивается на фазе разработки; то же находим и во вводных курсах по программированию. Но широко известно, что 70% стоимости ПО приходится на его сопровождение. Никакое изучение качества ПО не может быть удовлетворительным, если оно игнорирует этот аспект. Вышеприведенная диаграмма, взятая из ключевого исследования Лиенца и Свонсона, проливает некоторый свет на то, что на самом деле значит включающий разнообразные понятия термин “сопровождение”. Исследование рассмотрело 487 систем, разрабатывающих ПО разного рода; возможно, оно немного устарело, но более поздние публикации подтверждают те



Рис. 1.5: Распределение расходов на сопровождение

же общие результаты. Оно показывает долю стоимости, приходящуюся на каждый идентифицированный авторами вид работ по сопровождению.

Ключевые концепции

- Целью программной инженерии является нахождение путей построения ПО высокого качества.
- Качество ПО лучше всего видится как компромисс между целым рядом различных целей, а не как единый фактор.
- Внешние факторы, понятные пользователям и клиентам, следует отличать от внутренних факторов, понятных проектировщикам и конструкторам.
- Действительное значение имеют внешние факторы, но управление системой возможно только через внутренние факторы, благодаря которым достигается нужный эффект.
- Список основных внешних факторов качества приведен выше. ОО-метод направлен на улучшение качества тех факторов, которые прежде всего нуждаются в лучших подходах. К ним относятся факторы корректности и устойчивости, связанные с безопасностью, вместе известные как надежность, и факторы, требующие децентрализованной архитектуры ПО, - повторное использование и расширяемость, вместе известные как модульность.
- Сопровождение ПО, потребляющее большую долю его стоимости, находится в невыгодном положении из-за трудности реализации изменений в ПО и из-за слишком большой зависимости программ от физической структуры данных, которыми они манипулируют.

Контрольные вопросы

1. К внешним факторам, влияющим на качество ПО относятся

- расширяемость
- модульность
- язык разработки
- корректность

2. К внутренним факторам, влияющим на качество ПО относятся

- расширяемость
- модульность
- объектная ориентированность
- переносимость
- совместимость

3. Кривые Осмонда характеризуют

- вклад различных факторов на сопровождение ПО
- корректность разработки ПО
- эффективность разработки ПО
- различные подходы к проектированию ПО, основанные на приоритете различных факторов

4. Под корректностью ПО понимается?

способность отвечать на все вопросы пользователя
безошибочная работа ПО во всех ситуациях
способность ПО реагировать на изменения спецификаций
способность ПО работать в точном соответствии со спецификацией

5. Под расширяемостью ПО понимается?

автоматическое развертывание кода модулей
возможность добавления новых функций
возможность сборки ПО из готовых компонентов
легкость адаптации ПО к изменениям спецификации

6. Документация

может быть внешней, ориентированной на пользователя
может быть внутренней, ориентированной на разработчиков
является самостоятельным фактором качества ПО
должна допускать автоматическое извлечение
должна быть неотъемлемой частью ПО

7. При разработке сложного ПО основная доля затрат приходится на?

отладку
создание спецификаций
сопровождение
разработку

8. Повторное использование означает

использование библиотек компонентов

создание компонентов ПО, способных служить для построения различных приложений

многократный вызов процедур и функций в пределах одного модуля

многократный вызов процедур и функций в пределах одного приложения

9. При сопровождении два основных фактора, определяющих затраты, связаны с?

построением документации

изменением в формате данных

отладкой оставшихся ошибок

улучшением эффективности

изменением требований пользователя

10. При разработке ПО в первую очередь следует заботиться о?

интерфейсе пользователя

простоте использования

корректности

функциональности

11. Условная корректность ПО означает?

задание условий, при которых допустимо нарушение спецификаций

возможность нарушения спецификаций в небольшом проценте ситуаций

корректность при условии корректности спецификации

корректность при условии, что все нижележащее ПО – библиотеки, компиляторы, среда разработки - корректны

12. Выделите четыре основных фактора, влияющих на качество ПО

расширяемость

повторное использование

устойчивость

простота использования

самодокументируемость

функциональность

корректность

своевременность

13. Отметьте истинные высказывания

эффективность является одним из важнейших факторов

линейный алгоритм со сложностью $O(n)$ всегда работает быстрее квадратичного алгоритма со сложностью $O(n^2)$

нужно всегда стремиться к построению наиболее эффективного ПО

оптимизация ПО может оказаться излишней и повредить расширяемости

14. Отметьте истинные высказывания

своевременность – это выпуск ПО в тот момент, когда появляется в нем необходимость

децентрализация модулей улучшает расширяемость ПО

система должна выполнять свои задачи и в аварийных ситуациях, выходящих за пределы спецификаций

после того, как спецификации к системе сформулированы, они замораживаются до момента завершения процесса разработки системы

15. Отметьте истинные высказывания

разработчики ПО не должны думать о его сопровождении

совместимость – это легкость сочетания одних элементов ПО с другими

под надежностью ПО понимают его корректность и устойчивость

под модульностью ПО понимают расширяемость и повторное использование

2 Критерии объектной ориентации

2.1 О критериях

Рассмотрим выбор критериев, позволяющих оценить объектную ориентированность системы (objectness).

Набор критериев делится на три части:

- *Метод и язык* (Method and Language): эти два почти не различимые аспекта охватывают мыслительные процессы и нотацию, используемую для анализа, проектирования и программирования ПО. Заметьте, что (особенно в объектной технологии) термин “язык” относится не только к языку программирования в строгом смысле, но также и к языкам анализа и проектирования и используемой в них нотации, текстовой или графической.
- *Реализация* (Implementation) и *Среда* (Environment): критерии в этой категории описывают основные свойства инструментария, позволяющего разработчикам применять ОО-идеи.
- *Библиотеки* (Libraries): объектная технология основана на повторном использовании компонентов ПО. Критерии в этой категории описывают как наличие базовых библиотек, так и механизмы, необходимые для их использования и создания новых библиотек.

Такое деление удобно, но не абсолютно, поскольку некоторые критерии относятся к двум или трем категориям. Например критерий, помеченный “управление памятью”, относится к категории языка,

поскольку язык может поддерживать или не допускать автоматическую «сборку мусора». Этот же критерий относится к категории реализации и среды.

2.2 Метод и язык

Первый набор критериев относится к методу и поддерживающей его нотации.

Бесшовность (seamlessness)

ОО-подход включает весь жизненный цикл ПО. При рассмотрении ОО-решений следует проверить, что метод, язык и поддерживающие их инструменты, применимы к анализу и проектированию, а также к реализации и сопровождению. Язык, в частности, должен служить средством мышления, помогающим на всех стадиях работы.

В результате получается бесшовный процесс разработки, где общность концепций и нотации помогает сгладить переходы между последовательными ступенями жизненного цикла.

Классы

ОО-метод основан на понятии класса. Неформально, *класс* - элемент ПО, описывающий абстрактный тип данных и его частичную или полную реализацию. *Абстрактный тип данных* (интерфейс) - множество объектов, определяемое списком операций, применимых к этим объектам, и их свойствам.

Понятие класса должно быть центральной концепцией метода и языка.

Утверждения (Assertions)

Компоненты абстрактного типа данных имеют формально специфицированные свойства, отражаемые в соответствующих классах.

Утверждения - предусловия и постусловия программ класса и инварианты классов - играют эту роль.

Утверждения имеют три основных применения: помогают создать надежное ПО, обеспечивают систематическую документацию и являются инструментом тестирования и отладки ПО.

Язык должен давать возможность: поставлять класс и его компоненты вместе с утверждениями (предусловиями, постусловиями и инвариантами); включать инструментарий для получения документации из этих утверждений; осуществлять мониторинг утверждений во время выполнения программы.

Классы как модули

Объектная ориентация - в первую очередь архитектурная техника: она в основном затрагивает модульную структуру системы.

Классы должны быть единственным видом модулей.

В частности, исчезает понятие главной программы, а подпрограммы не существуют как независимые модульные единицы (они могут появляться только как часть классов).

Классы как типы

Понятие класса достаточно мощное, чтобы избежать необходимости любого другого механизма типизации.

Каждый тип должен быть основан на классе.

Даже базовые типы, такие как *integer* и *float*, можно рассматривать как классы; обычно такие классы являются встроенными.

Скрытие информации (information hiding)

Механизм, делающий определенные компоненты недоступными для клиентов, называется скрытием информации.

Создатели классов должны также иметь возможность избирательно экспортировать компоненты для избранных клиентов.

Автор класса должен иметь возможность указать, что компонент доступен: всем клиентам, ни одному клиенту или избранным клиентам.

Обработка исключений (Exception handling)

В процессе выполнения программ могут встречаться различные аномалии. В ОО-вычислениях они соответствуют вызовам, которые не могут быть выполнены надлежащим образом: например в результате сбоя в оборудовании, переполнения при выполнении арифметических операций или ошибок ПО.

Для создания надежного ПО необходимо иметь возможность восстановления нормального хода вычислений. Это является целью механизма обработки исключений.

Язык должен обеспечивать механизм восстановления в неожиданных аварийных ситуациях.

Статическая типизация (static typing)

Чтобы гарантировать корректное выполнение, язык должен быть типизирован. Это означает, что он отвечает нескольким правилам совместимости:

- Каждая сущность (entity) объявляется явным образом с указанием определенного типа, порожденного классом. Под сущностью понимается имя переменной, используемое в тексте ПО для ссылки на объекты времени выполнения.
- Каждый вызов компонента - это вызов доступного компонента соответствующего класса.
- Присваивание и передача аргументов подчиняются правилам согласования, требующим совместимости исходного типа и целевого типа.

В языке, включающем такую политику, возможен статический контроль типов. Тогда еще на этапе компиляции подобные ошибки будут обнаружены, и во время выполнения гарантируется отсутствие ошибок типа: “компонент недоступен объекту”. В языке *Python* используется динамическая типизация.

Универсальность (genericity)

Для того чтобы типизация была практичной, необходимо иметь возможность определять классы с параметрами, задающими тип. Такие классы известны как родовые. Родовой класс (параметрический класс) `List<G>` описывает списки элементов произвольного типа `G` - “формальным родовым параметром”.

Единичное наследование (single inheritance)

Разработка ПО включает создание большого числа классов, многие из которых являются вариантами ранее созданных классов. Для управления потенциальной сложностью такой системы необходим механизм классификации, известный как наследование. Класс `A` будет наследником (heir) класса `B`,

если он встраивает (наследует) компоненты класса В в дополнение к своим собственным. Потомок (descendant)- это прямой или непрямой наследник; обратное понятие - предок (ancestor).

Должно быть возможным объявить класс наследником другого класса.

Наследование - одно из центральных понятий ОО-метода; оно оказывает большое влияние на процесс разработки ПО.

Множественное наследование (Multiple inheritance)

Часто необходимо сочетать различные абстракции.

Класс должен иметь возможность быть наследником нескольких классов.

Конфликты имен при наследовании разрешаются адекватным механизмом.

Дублируемое наследование (Repeated inheritance)

При множественном наследовании возникает ситуация дублируемого наследования (repeated inheritance), когда некоторый класс многократно становится наследником одного и того же класса, проходя по разным ветвям наследования:

В этом случае язык должен обеспечить точные правила, определяющие, что происходит с компонентами, наследованными повторно от общего предка (на рисунке - это А). В некоторых случаях желательно, чтобы компонент из А создавал только один компонент в D (разделение), а в других - нужно, чтобы он создавал два (дублирование). Разработчики должны обладать гибкими средствами, позволяющими предписывать одну из возможностей независимо для каждого компонента.

При дублируемом наследовании судьбой компонентов должны управлять точно определенные правила, позволяющие разработчикам выбирать для каждого такого компонента разделение, либо дублирование.



Рис. 2.1: Дублируемое наследование

Ограниченная универсальность (Constrained genericity)

Сочетание универсальности и наследования дает полезную технику - ограниченную универсальность (constrained genericity). Теперь вы можете определить класс с родовым параметром, представляющим не произвольный тип, а лишь тип, являющийся потомком некоторого класса.

Родовой класс `SortedList` описывает списки; он содержит компонент `sort`, сортирующий элементы списка в соответствии с заданным отношением порядка. Параметр этого родового класса задает тип элементов списка. Но этот тип не может быть произвольным: он должен поддерживать отношение порядка. Фактический родовой параметр должен быть потомком библиотечного класса `Comparable`, описывающего объекты, снабженные отношением порядка. Ограниченная универсальность позволяет объявить наш родовой класс следующим образом: `SortedList<G, Comparable>`.

Механизм универсальности должен поддерживать форму ограниченной универсальности.

Переопределение (redefinition)

Когда класс является наследником другого класса, может потребоваться изменить реализацию или другие свойства некоторых наследованных компонент. Класс `Session`, управляющий сеансами пользователей в операционной системе, может иметь компонент `terminate`, выполняющий чистку в конце сеанса. Его наследником может быть класс `RemoteSession`, управляющий сеансом удаленного компьютера в сети.

Если завершение удаленного сеанса требует дополнительных действий, таких как, например, уведомление удаленного компьютера, класс `RemoteSession` переопределит компонент `terminate`.

Переопределение может повлиять на реализацию компонента, его сигнатуру (тип аргументов и результата) и спецификацию.

Должно быть возможным переопределить спецификацию, сигнатуру и реализацию наследованного компонента.

Полиморфизм

При наследовании, требование статической типизации, о котором говорилось выше, становится ограничивающим, если бы оно означало, что каждая сущность типа `C` может быть связана только с объектом точно такого же типа `C`.

Как уже отмечалось, “сущность” - это имя, к которому во время выполнения могут присоединяться различные значения. Сущность - это обобщение традиционного понятия переменной.

Полиморфизм (polymorphism) - способность присоединять к сущности объекты различных возможных типов. В статически типизированной среде полиморфизм не будет произвольным, а будет контролироваться наследованием.

Должна иметься возможность в период выполнения присоединять к сущности объекты различных возможных типов под управлением наследования.

Динамическое связывание

Сочетание последних двух механизмов, переопределения и полиморфизма, непосредственно предполагает следующий механизм. Вызов сущностью компонента всегда должен запускать тот компонент, который соответствует типу присоединенного объекта, а не типу сущности. При различных выполнениях одного и того же вызова могут запускаться разные компоненты.

Выяснение типа объекта в период выполнения

Разработчики ОО-ПО вскоре вырабатывают здоровую неприязнь к любому стилю вычислений, основанному на явном выборе между различными типами объекта. Полиморфизм и динамическое связывание намного предпочтительнее. Однако в некоторых случаях объект приходит извне, так что автор ПО не имеет возможности с определенностью предсказать его тип. В частности, это случается, если объект извлекается из внешних хранилищ, получен по сети или передан некоторой другой системой.

Тогда ПО нуждается в механизме, обеспечивающем безопасный способ доступа к объекту без нарушения ограничений статической типизации. Такой механизм должен проектироваться с большой аккуратностью, так чтобы не утратить пользы от полиморфизма и динамического связывания.

Абстрактные (abstract) свойства и классы

В некоторых случаях, для которых динамическое связывание дает элегантное решение, устраняя необходимость явных проверок, не существует начальной версии компонента, подлежащего переопределению. Необходимо иметь возможность написания класса или компонента как отложенного, то есть специфицированного, но не полностью реализованного.

Абстрактные классы особенно важны для ОО-анализа и высокоуровневого проектирования, поскольку они делают возможным задать основные аспекты системы, оставляя детали до более поздней стадии.

Управление памятью (memory management) и сборка мусора (garbage collection)

Может показаться, что этот критерий метода и языка должен принадлежать к следующей категории - реализации и среде. На самом деле он принадлежит к обеим категориям. Важнейшие требования предъявляются к языку, остальное - это вопрос хорошей инженерии.

ОО-системы даже в большей степени, чем традиционные системы, за исключением, быть может, Lisp, имеют тенденцию создания большого числа объектов, иногда со сложными взаимозависимостями. Политика, возлагающая на разработчиков ответственность за управление памятью, вредит и эффективности процесса разработки, и безопасности полученной системы. Трудно утилизировать память, занятую более не нужными объектами, усложняются программы, все это требует времени разработчиков, увеличивается риск некорректной обработки областей памяти. В хорошей ОО-среде управление памятью будет автоматическим, под контролем сборщика мусора (garbage collector) - компонента системы периода выполнения (runtime system).

Автоматическая сборка мусора - это проблема языка, так же как и реализации. Если язык явно не спроектирован для автоматического управления памятью, то зачастую реализация становится невозможной. Это справедливо для языков, где, например, указатель на объект определенного типа может быть преобразован (используя кастинг - cast) в указатель другого типа или даже в целое число, - такие средства делают невозможным создание надежного сборщика мусора.

Язык должен давать возможность надежного автоматического управления памятью, а реализация должна обеспечить наличие автоматического менеджера, управляющего памятью, в функцию которого входит сборка мусора.

2.3 Реализация и среда

Мы подошли к важным свойствам среды разработки, поддерживающей создание ОО-ПО.

Автоматическое обновление (automatic update)

Разработка ПО - процесс нарастающий. Разработчики обычно не пишут тысячи строк за один раз; они работают, добавляя и модифицируя, начиная чаще всего с системы, уже имеющей значительный размер.

При выполнении такого обновления важно иметь гарантию, что полученная в результате система будет согласованной. Традиционные подходы к этой проблеме предполагают работу вручную, заставляя разработчиков записывать все зависимости и прослеживать их изменения, используя специальные механизмы, известные как “создавать файлы” и “включать файлы”. Это неприемлемо в современных разработках программных продуктов, особенно в ОО-мире, где взаимозависимости между классами, вытекающие из отношений наследования, часто сложны, но могут быть выведены из систематического рассмотрения текста ПО.

Обновление системы после изменения должно быть автоматическим, а анализ межклассовых зависимостей выполняться инструментарием, а не вручную разработчиками.

Это требование можно удовлетворить в компилируемой среде (где компилятор будет работать вместе с инструментарием, выполняющим анализ зависимостей), в интерпретируемой среде или в среде, сочетающей обе эти техники реализации языка.

Быстрое обновление (fast update)

На практике механизм обновления системы должен быть не только автоматическим, но и быстрым. Более точно, он должен быть пропорциональным размеру изменений, а не размеру системы в це-

лом. Без этого свойства метод и среда могут быть применимыми только к небольшим системам, а применять их нужно к большим.

Время обработки ряда изменений в системе, создающих обновленную версию, должно быть функцией размера измененных компонентов и не зависит от размера системы в целом.

Сериализация (serialization)

Многие приложения, вероятно, большинство, требуют сохранения объектов от одного сеанса до следующего. Среда должна обеспечивать механизм выполнения этого простым способом.

Объект часто содержит ссылки на другие объекты, тоже содержащие, в свою очередь, ссылки на объекты. Поэтому каждый объект может иметь большое количество зависимых объектов с возможно сложным графом зависимости (который может содержать циклы). Обычно не имеет смысла сохранять или восстанавливать объект без всех его прямых и непрямых зависимых объектов. Механизм сериализации должен автоматически сохранять зависимые объекты наряду с самим объектом.

Документация

Разработчики классов и систем должны обеспечивать руководство, заказчиков и других разработчиков ясными высокоуровневыми описаниями создаваемого ПО. Им необходим инструментарий, помогающий в этой работе. Большая часть документации должна автоматически создаваться на основе текстов ПО. Утверждения, как уже отмечено, помогают сделать такие документы, извлекаемые из ПО, точными и информативными.

Должны быть в наличии инструментальные средства для автоматического получения документации о классах и системах.

Быстрый просмотр (browsing)

При работе с классом часто необходимо получить информацию о других классах; в частности, компоненты данного класса часто могут определяться не в самом классе, а в его различных предках. Среда должна обеспечить разработчиков инструментами для исследования текста класса, нахождения зависимых классов и быстрого переключения с текста одного класса на другой.

В этом и состоит задача просмотра. Типичные хорошие возможности просмотра включают: поиск классов - клиентов, поставщиков, потомков, предков; поиск всех переопределений компонента; поиск исходного объявления переопределенного компонента.

Средства интерактивного просмотра должны давать возможность разработчикам ПО быстро и удобно проследить зависимости между классами и компонентами.

Библиотеки

Один из характерных аспектов разработки ПО ОО-способом - возможность создавать его на основе существующих библиотек. ОО-среда должна обеспечивать хорошие библиотеки и механизмы создания новых библиотек.

Базовые библиотеки

Изначально в информатике изучаются фундаментальные структуры данных - множества, списки, деревья, стеки; связанные с ними алгоритмы - сортировки, поиска, обхода, сопоставления с образцом. Эти структуры и алгоритмы вездесущи в разработках ПО. Нередко, когда в своей системе очередной разработчик повторно их реализует. Это не только расточительно, но и пагубно отражается на качестве ПО, поскольку вряд ли отдельный разработчик, реализующий структуру данных не как цель

саму по себе, а в качестве компонента некоторого приложения, достигнет оптимальной надежности и производительности.

ОО-среда разработки должна обеспечить повторно используемые классы, удовлетворяющие общим потребностям.

Должны быть доступны повторно используемые классы, реализующие фундаментальные структуры данных и алгоритмы.

Графика и пользовательские интерфейсы

Многие современные системы ПО интерактивны. При взаимодействии с пользователем широко используется графика и удобный, чаще всего графический интерфейс. Это одна из областей, где ОО-модель оказалась наиболее впечатляющей и полезной. Разработчики должны иметь возможность использовать графические библиотеки для быстрого и эффективного построения интерактивных приложений.

Должны быть доступны повторно используемые классы для разработки приложений, обеспечивающих пользователей приятными графическими пользовательскими интерфейсами.

Механизмы эволюции библиотек

Разработка высококачественных библиотек - долгая и трудная задача. Невозможно гарантировать, что построенные библиотеки сразу будут совершенными. Следовательно, важной проблемой является обеспечение разработчиков библиотеки возможностью обновлять и модифицировать их проекты, не нанося вреда существующим системам, основанным на библиотеках. Этот важный критерий эволюции мы отнесли к категории библиотек, но он относится также и к категории метода и языка.

Должны быть доступны механизмы, облегчающие эволюцию библиотек с минимальными нарушениями работы ПО клиентов.

Механизмы индексации в библиотеках

Еще одна насущная проблема библиотек - это необходимость механизмов идентификации классов для удовлетворения определенных нужд. Этот критерий затрагивает все три категории: библиотеки, язык (поскольку должен быть способ вводить индексирующую информацию в текст каждого класса) и инструментарий (для обработки запросов для классов, удовлетворяющих определенным условиям).

Библиотечные классы должны быть снабжены индексирующей информацией, допускающей поиск, основанный на свойствах.

Контрольные вопросы

1. Для ОО-метода разработки ПО

- класс всегда является типом данных

- встроенные типы данных классами не являются

- некоторые классы представляют собой модули, другие типы данных

- класс всегда является модулем

- тип данных всегда является классом

2. Универсальный класс

- класс, способный выполнять операции над данными разных типов

- это класс с родовыми параметрами, представляющими собой типы

синоним абстрактного класса

это класс, способный решать широкий спектр задач

3. Живучесть - это?

возможность сохранения состояния объекта с одновременным сохранением всех связанных объектов

способность ПО работать в условиях, когда нарушаются спецификации

свойство класса, позволяющее сохранять его методы и свойства

возможность сохранения состояния объекта

4. Статическая типизация означает?

для каждой сущности при ее объявлении задается тип

после того, как сущность связана с объектом, другие объекты не могут присоединяться к сущности

тип объекта, связываемого с сущностью, должен совпадать с типом сущности

связывание объекта и сущности выполняется еще на этапе трансляции и эта связь не может изменяться динамически в процессе выполнения приложения

5. Наследование может быть?

многоязычное

единичное

повторное

множественное

6. Сборка мусора – это?

удаление методов класса сразу после их вызова

удаление объекта сразу после того, как с ним разорвана связь

удаление модулей, не вызываемых в текущей сессии работы ПО

автоматическое удаление неиспользуемых объектов

обязанность программиста периодически освобождать память, отводимую объектам

7. Отложенный класс

класс, для которого не заданы спецификации

понятие эквивалентное понятию абстрактный класс

это класс, который временно не включается в состав приложения

класс, для которого не задана реализация

класс, для которого реализация задана частично

8. Под скрытием информации понимается?

скрытие спецификаций от пользователей

механизм, делающий определённые компоненты недоступными для клиентов

скрытие реализации от пользователей

недокументированные возможности ПО

9. Полиморфизм - это?

способность присоединять к сущности объекты различных возможных типов

возможность управлять объектами, тип которых не известен
способность присоединять к сущности значение void
существование нескольких реализаций метода класса

10. Обработка исключений

позволяет отключать некоторые модули приложения
является механизмом восстановления в аварийных ситуациях
предназначена для обработки специальных случаев, предусмотренных спецификацией
обеспечивает устойчивость ПО

11. Под бесшовностью понимается?

использование одного языка программирования для разработки всех компонентов ПО
процесс разработки ПО из готовых компонентов
использование одних и тех же ОО-механизмов на всех этапах жизненного цикла создания ПО - проектирования, разработки, сопровождения
использование специальной технологии "NoShow"

12. При ОО-вычислениях

допустим единственный механизм – вызов объектом своего метода и свойства
разрешается использовать глобальные сущности
разрешается вызывать глобальные методы, не принадлежащие конкретному классу
методу могут передаваться аргументы в момент его вызова объектом
допустим широкий спектр организации вычислений

13. Отметьте истинные высказывания

клиентом класса А является класс В, вызывающий компоненты класса А
утверждения должны быть частью ПО и автоматически включаться в документацию
повторное наследование – это ситуация, при которой класс динамически получает нового родителя
родовые классы – это классы, для которых задан их род

14. Отметьте истинные высказывания

потомок не может изменять реализацию наследованного им компонента
для того чтобы типизация была практичной, необходимо иметь возможность определять классы с параметрами, задающими тип
предки и потомки могут обмениваться компонентами при наследовании
компонент может быть доступным одним клиентам и скрыт для других

15. Отметьте истинные высказывания

сборщик мусора - это компонент системы периода выполнения
ОО-модель не пригодна для задачи создания графического интерфейса
в ситуациях, где допустима операция присваивания, допустима и операция попытка присваивания
если класс является отложенным, – он считается эффективным

Набрано баллов

3 Модульность

Чтобы обеспечить расширяемость (extendibility) и повторное использование (reusability), двух основных факторов качества, необходима система с гибкой архитектурой, состоящая из автономных программных компонент. Именно поэтому был введен термин модульность (modularity), сочетающий оба фактора. Модульное программирование ранее понималось как сборка программ из небольших частей, обычно подпрограмм. Но такой подход не может обеспечить реальную расширяемость и повторное использование программного продукта, если не гарантировать, что элементы сборки - модули - являются самодостаточными и образуют устойчивые структуры. Любое достаточно полное определение модульности должно обеспечивать реализацию этих свойств. Таким образом, метод проектирования программного продукта является модульным, если он помогает проектировщикам создать систему, состоящую из автономных элементов с простыми и согласованными структурными связями между ними.

3.1 Пять критериев

Метод проектирования, который можно называть “модульным”, должен удовлетворять пяти основным требованиям:

1. Декомпозиции (decomposability).

2. Композиции (composability).
3. Понятности (understandability).
4. Непрерывности (continuity).
5. Защищенности (protection).

Декомпозиция

Определение 12. Метод проектирования удовлетворяет критерию *декомпозиции*, если он помогает разложить задачу на несколько менее сложных подзадач, объединяемых простой структурой, и настолько независимых, что в дальнейшем можно отдельно продолжить работу над каждой из них.

Такой процесс часто будет рекурсивным, поскольку каждая подзадача может оказаться достаточно сложной и потребует дальнейшего разложения. Следствием требования декомпозиции является *разделение труда* (division of labor): как только система будет разложена на подсистемы, работу над ними следует распределить между разными разработчиками или группами разработчиков. Это трудная задача, так как необходимо ограничить возможные взаимозависимости между подсистемами:

- Необходимо свести такие взаимозависимости к минимуму; в противном случае разработка каждой из подсистем будет ограничиваться темпами работы над другими подсистемами.
- Эти взаимозависимости должны быть известны: если не удастся составить перечень всех связей между подсистемами, то после завершения разработки проекта будет получен набор элементов программы, которые, возможно, будут работать каждая в отдельности, но не смогут быть

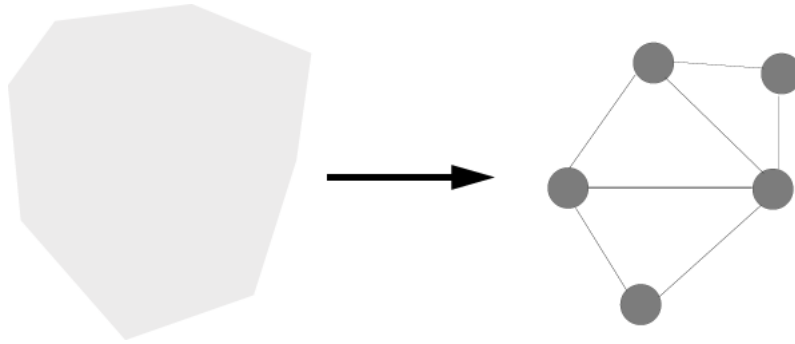


Рис. 3.1: Декомпозиция

собраны вместе в завершенную систему, удовлетворяющую общим требованиям к исходной задаче.

Наиболее очевидным примером обсуждаемого метода, удовлетворяющим критерию декомпозиции, является метод *нисходящего (сверху вниз) проектирования* (top-down design). В соответствии с этим методом разработчик должен начать с наиболее абстрактного описания функции, выполняемой системой. Затем последовательными шагами детализировать это представление, разбивая на каждом шаге каждую подсистему на небольшое число более простых подсистем до тех пор, пока не будут получены элементы с настолько низким уровнем абстракции, что становится возможной их непосредственная реализация. Этот процесс можно представить в виде дерева.

Типичным контрпримером (counter-example) является любой метод, предусматривающий включение в разрабатываемую систему модуля глобальной инициализации. Многие модули системы нуждаются в инициализации - открытии файлов или инициализации переменных.

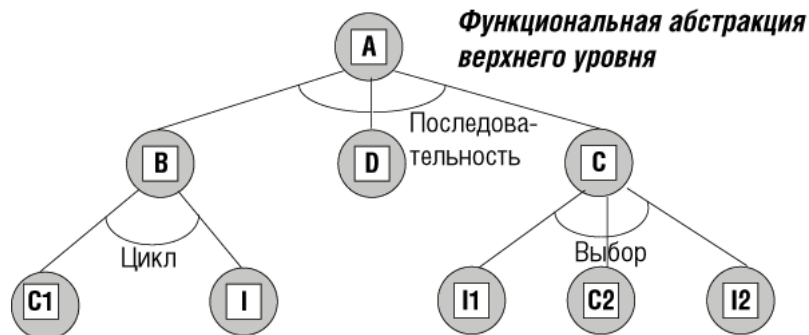


Рис. 3.2: Иерархия нисходящего проектирования

Каждый модуль должен произвести эту инициализацию до начала выполнения непосредственно возложенных на него операций.

В объектно-ориентированном методе каждый модуль должен самостоятельно инициализировать свои структуры данных.

Модульная композиция

Определение 13. Метод удовлетворяет критерию *модульной композиции*, если он обеспечивает разработку элементов программного продукта, свободно объединяемых между собой для получения новых систем, быть может, в среде, отличающейся от той, для которой эти элементы первоначально разрабатывались.

Композиция определяет процесс, обратный декомпозиции: элементы программного продукта извле-

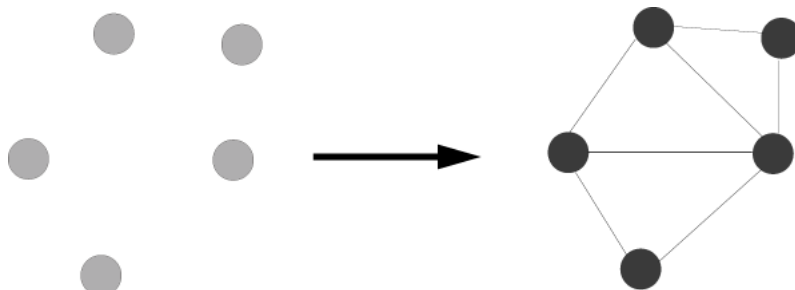


Рис. 3.3: Композиция

каются из того контекста, для которого они были первоначально предназначены, для использования их вновь в ином контексте.

Метод модульного проектирования облегчает этот процесс, создавая автономные элементы программного продукта достаточно независимыми от первоначально поставленной задачи, что делает такое извлечение возможным.

Композиция непосредственно связана с повторным использованием. Этот критерий отражает старую мечту - превратить процесс конструирования программного продукта в работу по складыванию кубиков так, чтобы строить программы из фабрично изготовленных элементов.

- Пример 1: Библиотеки подпрограмм. Библиотеки подпрограмм создаются как наборы компонентов. Одной из областей, где они успешно используются, являются численные вычисления, основанные на тщательно подготовленных библиотеках подпрограмм для решения задач линейной алгебры, метода конечных элементов, дифференциальных уравнений и др.
- Пример 2: Соглашения, принятые в командном языке Shell операционной системы UNIX. Ос-

новные команды системы UNIX оперируют с входным потоком последовательных символов и выдают результат, имеющий такую же стандартную структуру. Потенциальная возможность композиции поддерживается оператором | командного языка “Shell”. Запись A | B означает композицию программ. Вначале запускается программа A, ее результаты поступают на вход программы B, начинающей свою работу по завершении работы программы A. Такое системное соглашение благоприятствует композиции программных средств.

- Контрпример: Препроцессоры. Общепринятым способом расширения языка программирования, а иногда и преодоления его недостатков, является использование “препроцессора”, принимающего входные данные в расширенном синтаксисе и отображающего их в стандартной для этого языка форме. Типичные препроцессоры для Fortran’a и C поддерживают графические примитивы, расширенные управляющие структуры или операции над базами данных. Однако обычно такие расширения не являются взаимно совместимыми; что не позволяет сочетать два таких препроцессора, и приходится выбирать между, например, графикой или базой данных.

Как композиция, так и декомпозиция являются частью требований к модульному методу проектирования. Неизбежна смесь двух подходов к проектированию: сверху-вниз и снизу-вверх.

Модульная понятность

Определение 14. Метод удовлетворяет критерию *модульной понятности*, если он помогает получить такую программу, читая которую можно понять содержание каждого модуля, не зная текста остальных, или, в худшем случае, ознакомившись лишь с некоторыми из них.

Важность этого критерия следует из его влияния на процесс сопровождения программного продукта. Почти все действия по сопровождению программы, как неизбежные, так и не столь неизбежные,

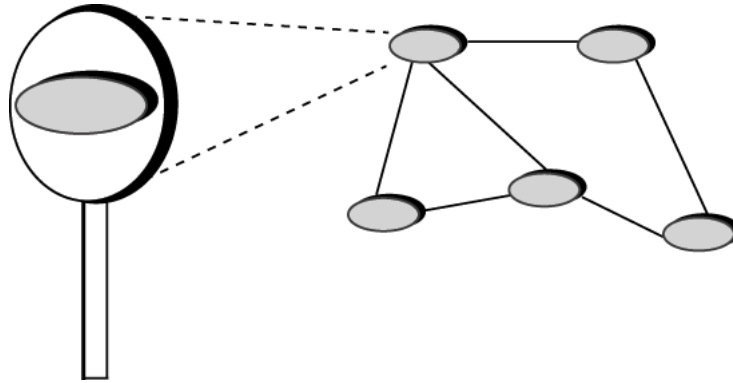


Рис. 3.4: Понятность

связаны с глубоким пониманием ее элементов. Метод едва ли может называться модульным, если тот, кто читает программный текст, не в состоянии понять его смысл.

Этот критерий, подобно четырем остальным, применим к модулям при описании системы на любом уровне: анализа, проектирования, реализации.

Модульная непрерывность

Определение 15. Метод удовлетворяет критерию *модульной непрерывности*, если незначительное изменение спецификаций разработанной системы приведет к изменению одного или небольшого числа модулей.

Этот критерий непосредственно связан с критерием расширяемости. Как подчеркивалось в преды-

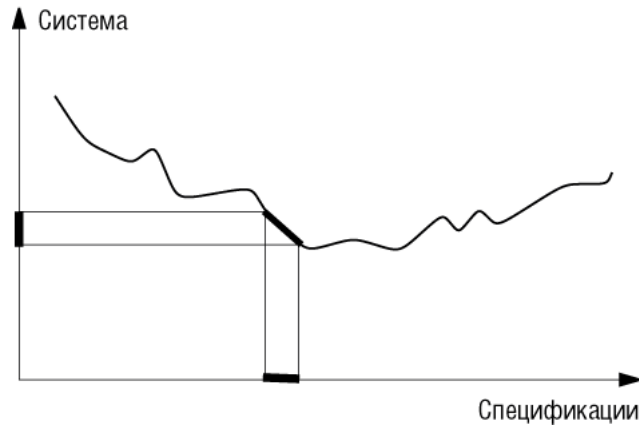


Рис. 3.5: Непрерывность

душей лекции, внесение изменений является неотъемлемой частью процесса разработки программного продукта. Соответствующие требования к программе будут неминуемо изменяться в ходе разработки. Непрерывность означает, что небольшие изменения будут воздействовать только на отдельные модули в структуре системы, а не на всю систему. Этот математический термин введен здесь лишь по аналогии, поскольку не существует формального понятия размера спецификации и программы. Можно было бы ввести приемлемую меру для определения “небольших” или “больших” изменений программы, но дать подобное определение для спецификаций к программе это уже настоящая проблема.

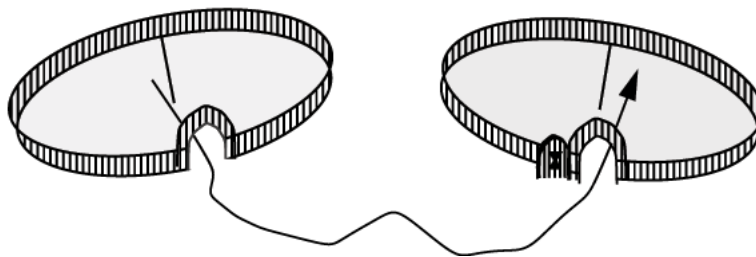


Рис. 3.6: Нарушение защищенности

Модульная защищенность

Определение 16. Метод удовлетворяет критерию *модульной защищенности*, если он приводит к архитектуре системы, в которой аварийная ситуация, возникшая во время выполнения модуля, ограничится только этим модулем, или, в худшем случае, распространится лишь на несколько соседних модулей.

Вопрос об отказах и ошибках является основным в программной инженерии. Сейчас речь идет об ошибках периода исполнения программы, связанных с аппаратными прерываниями, ошибочными входными данными или исчерпанием необходимых ресурсов (например, из-за недостаточного объема памяти). Критерий защищенности направлен не на предотвращение или исправление ошибок, а на проблему, непосредственно связанную с модульностью - распространением ошибок в модульной системе.

3.2 Пять правил

Из рассмотренных критериев следуют пять правил, которые должны соблюдаться, чтобы обеспечить модульность:

1. Прямое отображение (Direct Mapping).
2. Минимум интерфейсов (Few Interfaces).
3. Слабая связность интерфейсов (Small interfaces - weak coupling).
4. Явные интерфейсы (Explicit Interfaces).
5. Скрытие информации (инкапсуляция) (Information Hiding).

Первое правило касается отношения между внешней системой и ПО. Следующие четыре правила касаются общей проблемы - как модули общаются между собой. Для получения хорошей модульной архитектуры необходим управляемый и строгий метод обеспечения межмодульных связей.

Прямое отображение

Любая прикладная система стремится удовлетворить потребности некоторой проблемной области. Если имеется хорошая модель для описания этой проблемной области, то желательно обеспечить четкое отображение структуры проблемы, описываемой моделью, на структуру системы. Из этого следует первое правило:

Модульная структура, создаваемая в процессе конструирования ПО, должна оставаться совместимой с модульной структурой, создаваемой в процессе моделирования проблемной области.

Эта рекомендация следует, в частности, из двух критериев модульности:

- Непрерывность: отслеживание модульной структуры проблемы в структуре решения облегчит оценку и ограничит последствия изменений.
- Декомпозиция: если уже была проделана некоторая работа по анализу модульной структуры проблемной области, то это может явиться хорошей отправной точкой для разбиения программы на модули.

Минимум интерфейсов

Правило *минимума интерфейсов* ограничивает общее число информационных каналов, связывающих модули системы:

Каждый модуль должен поддерживать связь с возможно меньшим числом других модулей.

Связь между модулями может осуществляться различными способами. Модули могут вызывать друг друга (если они являются процедурами), совместно использовать структуры данных и так далее. Правило Минимума Интерфейсов ограничивает число таких связей.

В системе, составленной из n модулей, число межмодульных связей должно быть намного ближе к минимальному значению $n - 1$, как показано на рисунке (А), чем к максимальному $n(n - 1)/2$, как показано на рисунке (В).

Это правило следует, в частности, из критериев непрерывности и защищенности: если между модулями имеется слишком много взаимосвязей, то влияние изменения или ошибки может распространиться на большое число модулей. Оно также имеет отношение к критериям композиции (чтобы модуль мог использоваться в новой программной среде, он не должен зависеть от слишком большого числа других модулей), понятности и декомпозиции.

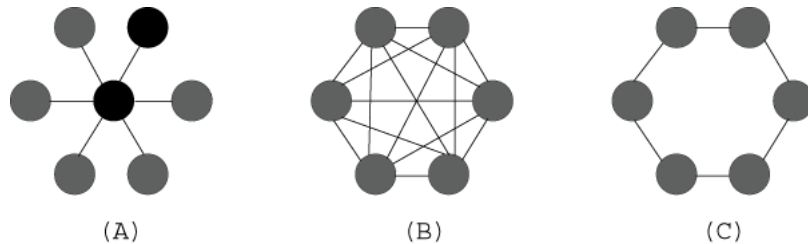


Рис. 3.7: Виды структур межмодульных связей

Вариант (А) на последнем рисунке показывает, как добиться минимального числа связей, $n - 1$, с помощью весьма централизованной структуры: один основной модуль, а все остальные общаются только с ним. Но имеются намного более “демократические” структуры, такие как (С), содержащие почти такое же число связей. В этой схеме каждый модуль непосредственно общается с двумя ближайшими соседями, центральной власти здесь нет. Такой подход к конструированию программы кажется сначала немного неожиданным, поскольку он не согласуется с традиционной моделью нисходящего проектирования. Но он может приводить к надежным, расширяемым решениям. Это именно такой вид структуры, к созданию которой будет стремиться ОО-метод при его разумном применении.

Слабая связность интерфейсов

Правило *слабой связности интерфейсов* относится к размеру передаваемой информации, а не к числу связей:

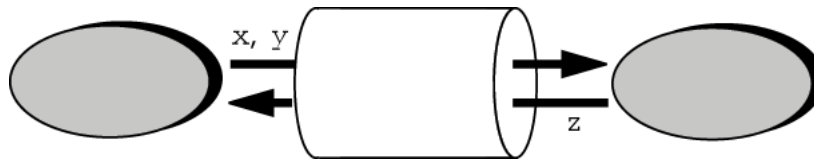


Рис. 3.8: Канал связи между модулями

Если два модуля общаются между собой, то они должны обмениваться как можно меньшим объемом информации.

Инженер-электрик сказал бы, что каналы связи между модулями должны иметь ограниченную полосу пропускания:

Требование Слабой связности интерфейсов следует, в частности, из критериев непрерывности и защищенности.

Явные интерфейсы

Всякое общение двух модулей А и В между собой должно быть очевидным и отражаться в тексте А и/или В.

За этим правилом стоят критерии:

- Декомпозиции и композиции. Если нужно разложить модуль на несколько подмодулей или компоновать его с другими модулями, то любая внешняя связь должна быть ясно видна.

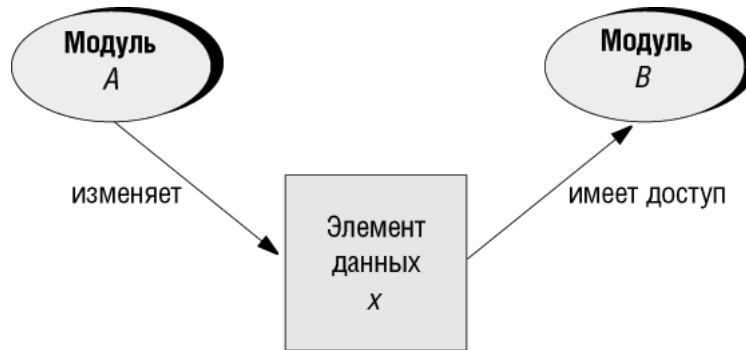


Рис. 3.9: Совместное использование данных

- Непрерывности. Должно быть очевидно, какие элементы могут быть затронуты возможным изменением.
- Понятности. Как можно истолковывать действие модуля А, если на его поведение может косвенным образом влиять модуль В?

Одной из проблем, возникающих при применении правила *явных интерфейсов*, является то, что межмодульная связь может осуществляться не только через вызов процедуры; источником косвенной связи может быть, например, совместное использование данных (data sharing):

Скрытие информации

Правило *скрытия информации* можно сформулировать следующим образом:

Разработчик каждого модуля должен выбрать некоторое подмножество свойств модуля в качестве официальной информации о модуле, доступной авторам клиентских модулей.

Применение этого правила означает, что каждый модуль известен всем остальным (то есть разработчикам других модулей) через некоторое официальное описание, или так называемые общедоступные (public) свойства.

В основе правила скрытия информации лежит критерий непрерывности. Предположим, что в некотором модуле происходят изменения, касающиеся лишь его скрытых элементов и не затрагивающие общедоступных свойств; тогда на другие обращающиеся к нему модули, называемые его клиентами, эти изменения не действуют. Чем меньше общедоступная часть, тем больше шансов на то, что изменения в модуле будут содержаться в его скрытой части.

Правило скрытия информации придает особое значение отделению описания функции от ее реализации, - что делает функция и как она это делает - разные вещи. Помимо критерия непрерывности, это правило связано также с критериями декомпозиции, композиции и понятности. Нельзя независимо разрабатывать модули системы, комбинировать существующие модули или понимать действие отдельных модулей, если неизвестно в точности, что каждый из них может (или не может) ожидать от других модулей.

Какие же из свойств модуля должны быть общедоступными, а какие - скрытыми? Как правило, в общедоступную часть следует включать функциональность, заданную спецификацией модуля, а все, что связано с реализацией этих функциональных возможностей, должно быть скрыто, предохраняя другие модули от последующих изменений реализации программы.

Однако эта рекомендация является нечеткой, так как не дано определение спецификации (specification) и реализации (implementation). Действительно, можно поддаться искушению, изменив определение на прямо противоположное, и утверждать, что спецификация состоит из общедоступных свойств модуля, а реализация - из его скрытых свойств! ОО-подход обеспечит намного более точные рекомендации на основе теории абстрактных типов данных.



Рис. 3.10: Модуль в условиях скрытия информации

3.3 Пять принципов

Из предыдущих правил и, косвенным образом, из критериев следуют пять принципов конструирования ПО:

- Лингвистических модульных единиц (Linguistic Modular Units).
- Самодокументирования (Self-Documentation).
- Унифицированного доступа (Uniform Access).
- Открыт-закрыт (Open-Closed).
- Единственного выбора (Single Choice).

Лингвистические модульные единицы

Принцип *лингвистических модульных единиц* утверждает, что формализм описания ПО на различных уровнях (спецификации, проектирования, реализации) должен поддерживать модульность:

Модули должны соответствовать синтаксическим единицам используемого языка.

Упомянутым выше языком может быть язык программирования, язык проектирования, язык оформления технических требований и т. д. В случае языка программирования модули должны независимо компилироваться.

Самодокументирование

Подобно правилу скрытия информации, принцип *самодокументирования* определяет, как следует документировать модули:

Разработчик модуля должен стремиться к тому, чтобы вся информация о модуле содержалась в самом модуле.

Обычно реализации этого принципа мешает общепринятое положение, согласно которому информацию о модуле помещают в отдельные проектные документы.

Унифицированный доступ

В общем виде принцип можно сформулировать так:

Все службы, предоставляемые модулем, должны быть доступны в унифицированной нотации, которая не подведет вне зависимости от реализации, использующей память или вычисления.

Хотя вначале может показаться, что принцип *унифицированного доступа* направлен лишь на решение проблем, связанных с принятой нотацией, в действительности он задает правило проектирования, влияющее на многие аспекты ОО-разработки ПО. Принцип следует из критерия непрерывности; его можно рассматривать и как частный случай правила скрытия информации.

Открыт-Закрит

Любой метод модульной декомпозиции должен удовлетворять принципу семафора *открыт-закрит*:

Модули должны иметь возможность быть как открытыми, так и закрытыми.

Противоречие является лишь кажущимся, поскольку термины соответствуют разным целевым установкам:

- Модуль называют открытым, если он еще доступен для расширения. Например, имеется возможность расширить множество операций в нем или добавить поля к его структурам данных.
- Модуль называют закрытым, если он доступен для использования другими модулями. Это означает, что модуль (его интерфейс - с точки зрения скрытия информации) уже имеет строго определенное окончательное описание. На уровне реализации закрытое состояние модуля означает, что модуль можно компилировать, сохранять в библиотеке и делать его доступным для использования другими модулями (его клиентами). На этапе проектирования или спецификации закрытие модуля означает, что он одобрен руководством, внесен в официальный репозиторий утвержденных программных элементов проекта - базу проекта (project baseline), и его интерфейс опубликован в интересах авторов других модулей.

Единственный выбор

Последний из пяти принципов модульности можно считать следствием как принципа открыт-закрыт, так и правила скрытия информации.

В любом таком случае необходимо допускать возможность того, что список вариантов, заданных и известных на некотором этапе разработки программы, может в последующем быть изменен путем добавления или удаления вариантов. Чтобы обеспечить реализацию такого подхода к процессу разработки программного обеспечения, нужно найти способ защитить структуру программы от воздействия подобных изменений. Отсюда следует *принцип единственного выбора*:

Всякий раз, когда система программного обеспечения должна поддерживать множество альтернатив, их полный список должен быть известен только одному модулю системы.

Требование того, чтобы список выбора был известен лишь одному модулю, обеспечивает подготовку к последующим изменениям: при добавлении вариантов понадобится произвести обновление только того модуля, в котором содержится эта информация - такова сущность единственного выбора. А все остальные модули, в частности - его клиенты, смогут продолжать свою работу как обычно.

3.4 Ключевые концепции

- Выбор надлежащей структуры модуля является ключом к достижению целей его возможного повторного использования и расширяемости.
- Модули служат как для декомпозиции программного обеспечения (проектирование сверху вниз), так и для его композиции (снизу-вверх).
- Принципы модульности применимы как к спецификации и проектированию, так и к реализации ПО.
- Всеобъемлющее определение модульности должно объединять различные точки зрения; разные требования иногда оказываются взаимно противоречивыми, например декомпозиция (стимулирующая методы проектирования сверху-вниз) и композиция (способствующая использованию метода снизу-вверх).
- Управление количеством и формой связей между модулями является основой разработки хорошей модульной архитектуры.

- Для долгосрочной целостности структур модульной системы требуется скрывание информации, что приводит к необходимости строгого разделения интерфейса и реализации.
- Унифицированный доступ освобождает клиентов от знания выбора внутренних представлений, реализованных в модулях-поставщиках.
- Закрытым является такой модуль, который может использоваться, благодаря знанию его интерфейса, модулями-клиентами.
- Открытым является такой модуль, который еще можно расширять.
- Для эффективного руководства проектом следует поддерживать модули, являющиеся одновременно как открытыми, так и закрытыми. Но традиционные подходы к разработке и программированию не дают такой возможности.
- Принцип Единственного Выбора предписывает ограничивать распространение полной информации обо всех вариантах некоторого понятия.

Контрольные вопросы

1. Верно ли, что различия между правилами, критериями и принципами модульности состоят в том, что
разрабатываемая система должна удовлетворять критериям модульности
между правилами, критериями и принципами нет различий – это синонимичные понятия

правила модульности следует выполнять при разработке ПО
механизмы ОО следует проектировать в соответствии с принципами

2. Верно ли, что различия между правилами, критериями и принципами модульности состоят в том, что

критерии следуют из принципов
принципы следуют из правил
правила следуют из критериев
критерии являются взаимно независимыми

3. К принципам модульности относятся

декомпозиция
принцип лингвистических единиц
единственный выбор
слабая связность

4. К критериям модульности относятся

непрерывность
унифицированный доступ
единственный выбор
слабая связность интерфейсов

5. К правилам для обеспечения модульности относятся

непрерывность
явные интерфейсы
композиция
скрытие информации

6. Принцип Открыт-Закрит предполагает, что

модуль является либо открытым, либо закрытым
модуль всегда должен быть открытым для изменений
уже работающий модуль всегда должен быть закрытым
ОО-механизмы (наследование) позволяет построить систему, удовлетворяющему этому принципу

7. Правило прямого отображения требует, чтобы

модульная структура ПО непосредственно отображала структуру модели предметной области

модульная структура ПО непосредственно отображала структуру предметной области
модульная структура ПО непосредственно отображала структуру спецификаций

8. Система, удовлетворяющая критерию декомпозиции

допускает многократное применение приема декомпозиции
может быть разбита на относительно независимые подсистемы, допускающие самостоятельную разработку
может быть разбита на полностью независимые подсистемы, допускающие самостоятельную разработку

может быть разработана способом проектирования "сверху-вниз"

9. Критерий непрерывности требует, чтобы

разработка классов системы велась непрерывно

модули системы можно было непрерывно открывать и закрывать

малым изменениям спецификации системы соответствовало малое изменение ее реализации

если объем текста изменений в спецификации имеет размер в N слов, то объем кода внесенных изменений, выраженный в байтах, должен иметь порядок $O(N)$

10. Отметьте истинные высказывания?

правило скрывтия интерфейсов утверждает, что описание модуля должно содержать лишь описание некоторых свойств и методов модуля

минимальное число связей системы, состоящей из n модулей равно n

разработчик модуля должен стремиться к тому, чтобы вся информация о модуле содержалась в самом модуле

наличие общего блока данных – это свидетельство выполнения правила слабой связности интерфейсов

11. Отметьте истинные высказывания?

источником межмодульной связи может быть совместное использование данных

всякое общение двух модулей А и В между собой должно быть очевидным и отражаться в тексте А и/или В

на этапах проектирования и разработки должны использоваться различные языки

программное обеспечение и документация к нему должны разрабатываться независимо
принцип Открыт-Закрыт противоречив и удовлетворить ему реально невозможно

12. Отметьте истинные высказывания?

выбор надлежащей структуры модуля является ключом к достижению целей его возможного повторного использования и расширяемости

скрытие информации приводит к необходимости строгого разделения интерфейса и реализации

для эффективного руководства проектом следует поддерживать модули, являющиеся одновременно как открытыми, так и закрытыми

принципы модульности применимы на этапах спецификации и проектирования, но не применимы к реализации ПО

Набрано баллов

4 Подходы к повторному использованию

В этой лекции будут рассмотрены некоторые из проблем, направленных на широкомасштабное внедрение повторного использования программных компонентов.

4.1 Цели повторного использования

Прежде всего, следует понять, почему так важно улучшать возможности повторного использования ПО.

Ожидаемые преимущества

Повторное использование может обеспечить прогресс на следующих направлениях:

- *Своевременность* (timeliness) (в том смысле, который определен при обсуждении показателей качества: быстрота доведения проектов до завершения и продукции до рынка). При использовании уже существующих компонентов нужно меньше разрабатывать, а, следовательно, ПО создается быстрее.

- *Сокращение объема работ по сопровождению ПО* (decreased maintenance effort). Если кто-то разработал ПО, то он же отвечает и за его последующее развитие. Известен парадокс компетентного разработчика ПО: “чем больше вы работаете, тем больше работы вы себе создаете”. Довольные пользователи вашей продукции начнут просить добавления новых функциональных возможностей, переноса на новые платформы. Если не надеяться “на дядю”, то единственное решение парадокса - стать некомпетентным разработчиком, - чтобы никто больше не был заинтересован в вашей продукции. В этой книге подобное решение не поощряется.
- *Надежность*. Получая компоненты от поставщика с хорошей репутацией, вы имеете определенную гарантию, что разработчики предприняли все нужные меры, включая всестороннее тестирование и другие методы контроля качества. В большинстве случаев можно ожидать, что кто-то уже испытал эти компоненты до вас и обнаружил все возможно остававшиеся ошибки. Заметьте, вовсе не предполагается, что разработчики компонентов умнее вас. Для них создаваемые компоненты - будь то графические модули, интерфейсы баз данных, алгоритмы сортировки - это служебная обязанность, цель работы. Для вас это лишь второстепенная, рутинная работа, поскольку вашей целью является создание некоторой прикладной системы в вашей собственной области деятельности.
- *Эффективность*. Факторы, способствующие возможности повторного использования ПО, побуждают разработчиков компонентов пользоваться наилучшими алгоритмами и структурами данных, известными в их конкретной сфере деятельности. Однако в команде, разрабатывающей большой прикладной проект, трудно ожидать наличия специалистов по каждой проблеме, затрагиваемой в этом проекте. При разработке большого проекта невозможно оптимизировать все его детали. Следует стремиться к достижению наилучших решений в своей области знаний, а в остальном использовать профессиональные разработки.

- *Совместимость.* Если использовать хорошую современную ОО-библиотеку, то ее стиль повлияет, за счет естественного “процесса диффузии”, на стиль разработки всего ПО. Это существенно помогает повысить качество программного продукта.
- *Инвестирование.* Создание повторно используемого ПО позволяет сберечь плоды знаний и открытий лучших разработчиков, превращая временные ресурсы в постоянные.

Потребители и производители повторно используемых программ

В приведенном выше списке преимуществ можно выделить две ситуации - использование профессиональных или собственных компонентов. Первые четыре элемента списка описывают ситуацию использования существующих, профессионально разработанных компонентов. Последний элемент списка характеризует повторное использование собственного программного продукта. Элемент списка - совместимость - относится к обоим случаям.

Такое разграничение достоинств отражает два аспекта повторного использования: точку зрения потребителя, пользующегося продукцией разработчиков компонент, и точку зрения производителя, обеспечивающего возможность повторного использования своих разработок.

Для разработчиков ПО, еще не имеющих большого опыта, следует быть потребителями компонентов. Принципиально невозможно сразу приступить к производству повторно используемых программ. Единственно возможный путь стать производителем - состоит в изучении и копировании уже существующих хороших образцов. Такой подход сразу принесет свои полезные плоды, поскольку в своих разработках вы воспользуетесь достоинствами этих компонентов.

4.2 Что следует повторно использовать?

Первый возникающий вопрос - на каком уровне следует осуществлять повторное использование: персонала, спецификаций, проектов, их образцов, исходного кода, компонентов или абстрактных модулей.

Повторное использование персонала

Наиболее просто повторно использовать разработчиков, что широко практикуется в промышленности. Переводя разработчиков ПО с одного проекта на другой, фирмы избегают потери накопленного ими ранее опыта и обеспечивают его достойное применение в новых разработках.

Ввиду высокой текучести программистских кадров возможности такого подхода ограничены.

Повторное использование проектов и спецификаций

Этот подход является, по существу, более организованной версией предыдущего - повторного использования знаний, умений и опыта. Как показало обсуждение вопроса о документации, само представление проекта как независимого программного продукта, имеющего собственный жизненный цикл, независимый от соответствующей реализации, кажется сомнительным, поскольку трудно гарантировать, что проект и его реализация будут оставаться совместимыми в процессе изменения системы ПО.

Таким образом, если повторно использовать только проект, то возникает риск повторного использования неправильно работающих или уже вышедших из употребления элементов.

Эти замечания можно отнести и к другому смежному виду повторного использования: повторному использованию спецификаций.

Образцы проектов (design patterns)

Определение 17. *Образец* - это архитектурный принцип, применимый во многих прикладных областях; следуя образцу можно построить решение некоторой проблемы.

Повторное использование исходного текста

Несмотря на полезность повторного использования персонала, проектов и спецификаций, здесь не реализуется ключевая цель повторного использования. Если мы хотели бы найти программистский эквивалент повторно используемых деталей из других технических дисциплин, то это означало бы необходимость повторно использовать тот “хлам”, из которого фактически состоит наша программная продукция: исполняемые программы.

Существуют экономические и психологические препятствия на пути к распространению исходных кодов. Более серьезными ограничениям являются:

- Отождествление повторно используемого ПО с повторно используемым исходным текстом (source) исключает возможность скрытия информации. Следует иметь в виду, что повторное использование действительно больших проектов невозможно, если не предпринять систематических усилий по защите повторных пользователей от необходимости знания бесчисленных деталей.
- В сложных системах многие ее части могут не очевидным образом зависеть от других. Это часто затрудняет повторное использование отдельных элементов, приводя к необходимости повторно использовать и все остальное.

Удовлетворяющая требованиям модульности форма повторного использования должна устранить эти ограничения, поддерживая абстракцию и обеспечивая “мелкоструктурную” реализацию повторного использования.

Повторное использование абстрактных модулей

Все предыдущие подходы, несмотря на их ограниченную применимость, осветили важные аспекты проблемы повторного использования:

- Повторное использование персонала необходимо, но недостаточно. Наилучшие повторно используемые компоненты бесполезны при отсутствии хорошо подготовленных разработчиков, которые обладают достаточным опытом, чтобы распознать ситуацию, в которой может помочь использование уже существующих компонентов.
- Для повторного использования проектов необходимы не только готовые решения конкретных задач, но и достаточно высокий концептуальный уровень и универсальность повторно используемых компонентов. Классы, с которыми мы встретимся при обсуждении ОО-технологии, могут рассматриваться и как модули-проекты, так и как модули-реализации.
- Возможность повторного использования исходного кода служит напоминанием о том, что ПО в конечном счете определяется текстами программ. Разумная политика в области повторного использования должна приводить к созданию повторно используемых программных элементов.

Обсуждение позволило сузить область поиска подходящих единиц повторного использования. Такой единицей должен быть программный элемент (коллекция элементов). Он должен быть модулем приемлемого размера, удовлетворяющим требованиям модульности из предыдущей лекции. В частности, его связи должны быть строго ограничены, чтобы облегчить возможность независимого повторного использования. Информация, характеризующая возможности модуля, и составляющая первичную документацию для программистов, повторно его использующих (reusers), должна быть абстрактной: в соответствии с принципом скрытия информации она должна освещать лишь свойства, существенные для клиентов, а не описывать все детали модуля (как это делается в исходном коде).

Термин абстрактный модуль будет применяться к таким повторно используемым единицам (units of reuse), входящим в состав непосредственно применяемых ПО, доступ из внешнего мира к которым может осуществляться через описание, содержащее лишь подмножество свойств каждой единицы.

4.3 Повторяемость при разработке ПО

В поиске идеала абстрактного модуля следует рассмотреть суть процесса конструирования ПО. Наблюдая за разработкой, нельзя не обратить внимания на периодически повторяющиеся действия в этом процессе. Вновь и вновь программисты “сплетают” программу из множества стандартных элементов: сортировка, поиск, считывание, запись, сравнение, обход по дереву, - все повторяется.

Психологическим препятствием повторного использования является известный синдром: “Придумано Не Нами” (Not Invented Here или “NIH”). Говорят, что разработчики ПО являются индивидуалистами, предпочитающими все выполнять сами, не полагаясь на чужую работу.

Но на практике это не подтверждается. Разработчики ПО склонны к бесполезной работе не более других специалистов. Если имеется хорошее, широко известное и легкодоступное повторно используемое решение, то оно будет использовано.

Рассмотрим типичный случай лексического и синтаксического анализа. Намного проще создать программу грамматического анализа для командного языка или простого языка программирования, используя программные генераторы грамматического разбора (parser generators), например комбинацию известных программ Lex-Уасс, а не создавая все с нуля. Вывод очевиден: там, где инструментальные средства имеются, квалифицированные разработчики ПО повсеместно их используют.

Таким образом, обеспечить высокое качество при создании повторно используемых компонентов существенно важнее, чем для других видов ПО.

Обозначим через N стоимость уникального решения, R - решения, основанного на повторно ис-

пользуемых компонентах. Значение R никогда не будет равно нулю: сюда войдут затраты на обучение, затраты на включение компонентов систему, понадобится создать интерфейс вызова. Так что даже если экономия на повторном использовании и другие выгоды $r = (N - R)/N$ от повторного использования потенциально невелики, то придется все же убедить возможных “повторных пользователей” в том, что ради высокого качества повторно используемого решения стоит отказаться от желания полного контроля над всеми элементами системы.

Фирмы по разработке ПО и их стратегии

У фирмы по разработке ПО всегда существует искушение создавать решения, преднамеренно не удовлетворяющие критериям повторного использования, из опасения не получить следующий заказ, - поскольку если возможности уже приобретенного решения окажутся излишне широкими, то покупателю следующий заказ не потребуется!

Технологическая составляющая (engineering part) в разработке ПО не идентична такой же составляющей в индустрии массового производства; человеческий фактор будет, вероятно, по-прежнему играть ключевую роль в процессе конструирования ПО.

Цель повторного использования состоит не в том, чтобы заменить людей инструментальными средствами (а это часто, несмотря на всяческие утверждения, происходит с другими отраслями производства), а в изменении соотношения между тем, что следует поручить людям, а что - инструментальным средствам. Так что для фирмы, приобретшей известность за счет своих консультантов, эти нововведения не так уж плохи. В частности:

- Во многих случаях разработчики, применяющие повторно используемые компоненты, могут по-прежнему успешно пользоваться помощью специалистов, которые посоветуют, как наилучшим образом применять эти компоненты. Тем самым сохраняется существенная роль фирм по поставкам ПО и их консультантов.

- Как будет показано ниже, возможность повторного использования неотделима от расширяемости: хорошие повторно используемые компоненты будут оставаться открытыми для адаптации к конкретным обстоятельствам. Консультанты фирмы, разработавшей соответствующую библиотеку программ, имеют идеальную возможность выполнять настройку компонентов для отдельных заказчиков. Так что продажа компонентов и продажа услуг не обязательно являются взаимно исключающими видами деятельности; торговля компонентами может служить основой для торговли услугами.
- Хорошая повторно используемая библиотека может играть стратегическую роль в политике преуспевающей фирмы по производству ПО, даже если фирма продает решения, а не библиотеку, используя ее лишь для внутренних целей. Такая библиотека может дать фирме конкурентное преимущество в более быстрой и дешевой разработке нестандартных решений, удовлетворяющих требованиям заказчиков, чем могли бы сделать конкуренты, не опирающиеся на такую заранее заготовленную основу.

Организация доступа к компонентам

Вот что говорят скептики: прогресс в производстве повторно используемых ПО приведет к тому, что разработчики окажутся “заваленными” настолько большим количеством компонентов и это так усложнит их жизнь, что лучше бы этих компонентов не было.

Это высказывание следует рассматривать как предупреждение разработчикам повторно используемых ПО о том, что лучшие в мире повторно используемые компоненты бесполезны, если никто не знает об их существовании, или если для их получения придется затратить слишком много времени и усилий. Для практического успеха методов повторного использования требуется создание соответствующих баз данных, содержащих компоненты, запрос к которым позволял бы быстро выяснить, удовлетворяет ли нужным потребностям какой-либо из существующих компонентов.

Должны быть доступны и сетевые услуги, позволяющие осуществить заказ и немедленную доставку по сети выбранных компонентов.

Достижение этих целей требует решения технических и организационных проблем. Индексирование, поиск и доставка повторно используемых компонентов - это технические проблемы, решаемые известными средствами, в частности методами, основанными на использовании баз данных. Очевидно, справляться с программными компонентами ничуть не сложнее, чем с данными о заказчиках, информацией об авиарейсах или с библиотечными книгами.

Форматы для распространения повторно используемых компонентов

Еще одной задачей, охватывающей как технические, так и организационные проблемы, является выбор представления для распространения: исходный текст или двоичный формат? Это спорный вопрос, и мы ограничимся рассмотрением только нескольких доводов с обеих сторон.

Разработчики коммерческого ПО часто распространяют лишь описание интерфейса (соответствующая краткая форма (short form) рассматривается в одной из последующих лекций) и исполняемый код. Тем самым разработчики защищают секреты производства и свои инвестиции.

Двоичный код и в самом деле является предпочтительной формой распространения коммерческих прикладных программ, операционных систем и других инструментальных средств, в том числе компиляторов, интерпретаторов и сред разработки для ОО-языков.

Для изготовителя программного компонента польза от распространения исходного текста состоит в том, что это облегчает перенос программ (porting efforts). Можно избежать утомительной и малорентабельной деятельности по адаптации ПО к множеству несовместимых платформ, существующих в современном компьютерном мире, рассчитывая на то, что разработчики ОО-компиляторов и программных сред выполнят эту работу за вас.

4.4 Пять требований к модульным структурам

Как же найти такие модульные структуры, которые позволят создать компоненты, непосредственно готовые к повторному использованию, и, в то же время, допускающие возможность их адаптации?

Рассмотрим пример поиска данных в таблице сравнимых с образцом. Несомненно, алгоритм табличного поиска в общем виде всегда выглядит одинаково: начать с некоторой позиции в таблице t , затем приступить к последовательному просмотру таблицы, всякий раз проверяя, является ли искомым элемент в текущей позиции i , если это не так, то переходить к следующей позиции. Процесс завершается, если найден нужный элемент, либо проверка всех элементов оказалась безуспешной. Такая общая схема применима к многим возможным случаям представления данных и алгоритмам для табличного поиска, в том числе в массивах (отсортированных или не отсортированных), связанных списках (отсортированных или не отсортированных), последовательных файлах, двоичных деревьях, Б-деревьях и различных хеш-таблицах.

Это неформальное описание можно превратить в следующую функцию на языке Python:

```
def find(table, x):  
    for element in table:  
        if equal(element, x):  
            return element  
    return None
```

Пример применения данной функции:

```
>>> equal = lambda x, y: (x-y) % 5 == 0  
>>> find(range(15), 6)  
1
```

```
>>> find(range(0, 15, 5), 6)
>>>
```

Заметим при этом можно было использовать в качестве table, кроме списков, такие конструкции языка как списки, кортежи, файл, словари и другой любой итерируемый тип.

Задача табличного поиска и шаблон подпрограммы find иллюстрируют жесткие требования, предъявляемые к любому решению. Можно воспользоваться этим примером для выяснения, что же следует предпринять для перехода от обнаружения относительно нечеткой общности вариантов к реальному набору повторно используемых модулей. Такой анализ выявляет пять важных проблем:

- Изменчивость типов (Type Variation).
- Группирование подпрограмм (Routine Grouping).
- Изменчивость реализаций (Implementation Variation).
- Независимость представлений (Representation Independence).
- Факторизация общего поведения (Factoring Out Common Behaviors).

Изменчивость Типов (Type Variation)

Функция find предполагает, что таблица содержит объекты типа к которому применима функция equal. Повторно используемый модуль поиска должен быть применим ко многим различным типам элементов без того чтобы пользователи вынуждены были производить “вручную” изменения в тексте программы. Другими словами, необходимо средство для описания модулей, в которых типы выступают в роли параметров (type-parameterized), или короче - родовых (полиморфных) модулей.

Универсальность или полиморфность (genericity) (способность модулей быть родовыми) окажется важной частью ОО-метода.

Группирование Подпрограмм (Routine Grouping)

Шаблон подпрограммы find, даже если его полностью детализировать и ввести параметризацию типа, все еще не будет пригоден в качестве повторно используемого компонента. Поиск в таблице зависит от того, как таблица создавалась, как в нее включаются элементы, как они удаляются. Отдельно взятая программа поиска - это еще не модуль повторного использования. Самодостаточный, повторно используемый модуль должен включать множество подпрограмм, обеспечивающих каждую из упомянутых операций - создание, включение, удаление, поиск.

Эта идея лежит в основе формирования модуля как “пакета”.

Изменчивость Реализаций (Implementation Variation)

Шаблон find является весьма общим; и, как мы уже убедились, на практике имеется широкий выбор соответствующих структур данных и алгоритмов. Нельзя ожидать, что один модуль сможет обеспечить работу в столь разнообразных условиях, - он оказался бы просто огромным. Для охвата всех возможных реализаций требуется семейство модулей.

Общая методика создания и применения повторно используемых модулей должна поддерживать идею семейства модулей.

Независимость Представлений

Общая структура повторно используемого модуля должна позволять модулям-клиентам определять свои действия при отсутствии сведений о реализации модуля. Это требование называется Независи-

мостью Представлений.

Решение состоит в том, чтобы обеспечить автоматический выбор, осуществляемый системой исполнения. Такова будет роль динамического связывания (dynamic binding), ключевой составляющей ОО-подхода.

Факторизация Общего Поведения

Если требование Независимости Представлений отражает позицию клиента - игнорирование внутренних деталей и вариантов реализации - то последнее требование отражает позицию разработчиков повторно используемых классов. Их цель в получении преимуществ от любой общности (commonality), которая может существовать в семействе или подсемействе реализаций.

Многообразие реализаций, имеющее место в некоторых проблемных областях, требует, как уже отмечалось, решения, основанного на семействе модулей. Часто это семейство настолько велико, что естественно поискать соответствующие подсемейства. В случае табличного поиска первая попытка классификации может привести к трем обширным подсемействам:

- Таблицы, организуемые по некоторой схеме хеширования.
- Таблицы, организуемые как некоторая разновидность деревьев.
- Таблицы, организуемые последовательно.

Каждая из этих категорий охватывает много вариантов, но в большинстве случаев можно найти существенную общность между этими вариантами. Рассмотрим, например, семейство последовательных реализаций - таких, в которых элементы сохраняются и отыскиваются в порядке их первоначального включения в таблицу.

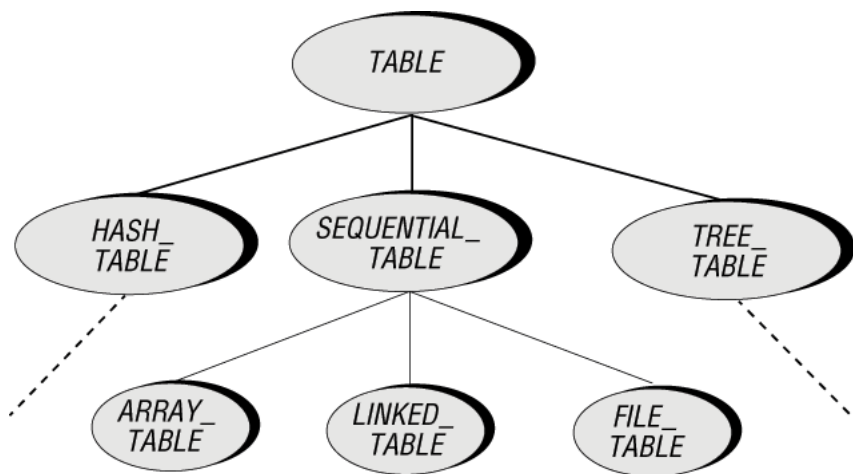


Рис. 4.1: Некоторые возможные реализации таблицы

4.5 Традиционные модульные структуры

Наряду с требованиями к модульности, изложенными в предыдущей лекции, пять требований изменчивости типов, группирования подпрограмм, изменчивости реализаций, независимости представлений и факторизации общего поведения определяют, чего следует ожидать от наших повторно используемых компонентов - абстрактных модулей.

Подпрограммы

Классический подход к повторному использованию состоит в том, чтобы создавать библиотеки подпрограмм. Здесь термин подпрограмма (routine) означает программный элемент, который может быть вызван другими элементами для выполнения некоторого алгоритма, используя некоторые входные данные, создавая некоторые выходные данные, и, возможно, модифицируя другие данные. Вызывающий элемент передает свои входные данные (а иногда - выходные данные и модифицируемые данные) в виде фактических аргументов (actual arguments) . Подпрограмма может также возвращать выходные данные в виде результата; в этом случае она называется функцией.

Библиотеки подпрограмм успешно использовались в различных прикладных областях, в частности, для численных расчетов, где применение отличных библиотек привело к первым сообщениям об успехах повторного использования. Декомпозицию систем на подпрограммы, функциональную декомпозицию, обеспечивает также метод нисходящего (сверху вниз) программирования. Подход, основанный на использовании библиотек подпрограмм, хорошо работает в случаях, когда можно определить множество (возможно - большое) отдельных задач, при наличии следующих ограничений:

- Каждая задача допускает простую спецификацию. Точнее, возможно охарактеризовать каждую отдельную задачу небольшим набором входных и выходных параметров.

- Задачи четко отличаются одна от другой, поскольку подход, основанный на подпрограммах, не позволяет воспользоваться возможной сколько-нибудь существенной их общностью - за исключением повторного использования некоторых конструкций.
- Отсутствуют сложные структуры данных, которые пришлось бы распределять между использующими их подпрограммами.

Пакеты

В семидесятые годы двадцатого века, в связи с развитием идей скрытия информации и абстракции данных, возникла необходимость в форме модуля, более совершенном, чем подпрограмма. Появилось несколько языков проектирования и программирования, наиболее известные из них: CLU, Modula-2 и Ada. В них предлагается сходная форма модуля, называемого в языке Ada пакетом, CLU - кластером, Modula - модулем.

Пакеты - это единицы программной декомпозиции, обладающие следующими свойствами:

1. В соответствии с принципом Лингвистических Модульных Единиц, "пакет" это конструкция языка, так что каждый пакет имеет имя и синтаксически четко определенную область.
2. Описание каждого пакета содержит ряд объявлений связанных с ним элементов, таких как подпрограммы и переменные, которые в дальнейшем будут называться компонентами (features) пакета.
3. Каждый пакет может точно определять права доступа, ограничивающие использование его компонентов другими пакетами. Другими словами, механизм пакетов поддерживает скрытие информации.

4. В компилируемом языке (таком, который может быть использован для реализации, а не только для спецификации и проектирования) поддерживается независимая компиляция пакетов.

Благодаря свойству 3, пакеты можно рассматривать как абстрактные модули. Их главным вкладом в программирование является свойство 2, удовлетворяющее требованию Группирования Подпрограмм. Пакет может содержать любое количество связанных с ним операций, таких как создание таблицы, включение, поиск и удаление элементов.

Пакеты: оценка

По сравнению с подпрограммами, механизм пакетов приводит к существенному совершенствованию разбиения системы ПО на абстрактные модули. Собрать нужные компоненты “под одной крышей” крайне полезно как для поставщиков, так и для клиентов:

- Автор модуля-поставщика может хранить в одном месте и совместно компилировать все элементы, относящиеся к некоторому заданному понятию. Это облегчает отладку и изменения. В отличие от этого, при использовании отдельных самостоятельных подпрограмм всегда есть опасность забыть произвести обновление некоторых подпрограмм при изменениях проекта или реализации; например, можно обновить new, put и has, но забыть обновить remove.
- Для авторов модулей-клиентов несомненно легче найти и использовать множество взаимосвязанных компонентов, если все они собраны в одном месте.

Преимущество пакетов по сравнению с подпрограммами особенно очевидно в таких случаях, как рассмотренный здесь пример с таблицей, где в пакете собраны все операции, применимые к конкретной структуре данных.

Однако пакеты все же не обеспечивают полного решения проблем повторного использования. Как уже отмечалось, они отвечают требованию Группирования Подпрограмм, но не удовлетворяют всем остальным требованиям. В частности, они не обеспечивают возможности факторизации общего поведения - “вынесения за скобки” общих компонентов. Конечно, благодаря скрытию информации, клиентам незачем интересоваться этим выбором. Но библиотека повторно используемых компонентов должна будет содержать модули для многих различных реализаций. Возникающую при этом ситуацию нетрудно предвидеть: типичная библиотека пакетов будет предлагать массу похожих, но вовсе не идентичных, модулей для заданной прикладной области, например, для работы с таблицами, но без какого-либо учета их общности. Обеспечивая возможность повторного использования для клиентов, такая методика приносит в жертву возможность повторного использования со стороны поставщиков.

4.6 Перегрузка и универсальность

Два технических приема - перегрузка (overloading) и универсальность (genericity) предлагают свои решения, направленные на достижение большей гибкости описанных выше механизмов. Рассмотрим, что же они могут дать.

Синтаксическая перегрузка

Перегрузка - это связывание с одним именем более одного содержания. Наиболее часто перегружаются имена переменных: почти во всех языках программирования различные по смыслу переменные могут иметь одно и то же имя, если они принадлежат различным модулям (различным блокам - в языке Algol и подобных ему).

Для этого обсуждения более существенной является перегрузка подпрограмм, частным случаем которой является перегрузка операторов, которая позволяет использовать одинаковые имена для нескольких подпрограмм. Такая возможность почти всегда имеет место для арифметических операторов: одна и та же запись, $a + b$, означает различные виды сложения, в зависимости от типов a и b (целые, вещественные с обычной точностью, вещественные с удвоенной точностью). Начиная с языка Algol 68, в котором допускалась перегрузка основных операторов, некоторые языки программирования распространили возможность перегрузки на операции, определяемые пользователем, и на обычные подпрограммы.

Семантическая перегрузка (предварительное представление)

Описанную форму перегрузки подпрограмм можно назвать синтаксической перегрузкой. В ОО-подходе будет предложена намного более интересная методика, динамическое связывание, отвечающая целям Независимости Представлений. Динамическое связывание можно назвать семантической перегрузкой.

4.7 Ключевые концепции

- Для разработки ПО характерна повторяющаяся деятельность, включающая частое использование общих образцов (common patterns). Но имеются существенные вариации того, как используются и комбинируются эти образцы, так примитивные попытки работать с компонентами, имеющимися в наличии, терпят неудачу.
- При практическом внедрении повторного использования возникают экономические, психологические и организационные проблемы. Последние связаны, в частности, с необходимостью

создания механизмов индексации, хранения и поиска большого числа повторно используемых компонентов. Более важными являются технические проблемы: общепринятые представления о модулях недостаточны для серьезной поддержки повторного использования.

- Основным затруднением при осуществлении повторного использования является необходимость сочетать повторное использование с расширяемостью. Дилемма - “повторно использовать или переделать” неприемлема. Хорошее решение должно обеспечить возможность сохранить одни свойства повторно используемого модуля и адаптировать другие.
- Простые подходы к решению проблемы: повторное использование персонала, повторное использование проектов, повторное использование исходного кода, библиотеки подпрограмм привели к некоторому успеху, но не позволили полностью реализовать потенциальные достоинства повторного использования.
- Компонентом программы, пригодным для повторного использования, является абстрактный модуль, обеспечивающий инкапсуляцию функциональных возможностей с помощью хорошо определенного интерфейса.
- Пакеты обеспечивают лучшую реализацию метода инкапсуляции, чем подпрограммы, поскольку в них объединяются структура данных и связанные с ней операции.
- Два метода позволяют повысить гибкость пакетов: перегрузка подпрограмм и универсальность.
- Перегрузка подпрограмм является синтаксическим средством, которое не решает важных проблем повторного использования, но затрудняет читабельность текстов программ.
- Универсальность способствует повторному использованию, но решает лишь проблему изменчивости типов.

- Что же нам требуется: техника, помогающая поставщику учесть общность в группах взаимосвязанных реализаций структур данных; и техника, избавляющая клиентов от необходимости знать о том, какой вариант реализации выбран поставщиком.

Контрольные вопросы

1. Повторное использование

уменьшает время разработки
снижает надежность приложения
повышает надежность приложения
позволяет отказаться от обработки исключительных ситуаций

2. Почему ПИК (Повторно Используемый Компонент) стоит использовать?

чаще всего, по экономическим причинам
для удовлетворения привычек пользователя
импортный товар всегда лучше
зачастую по причинам эффективности

3. Нужно ли создавать ПИК в процессе разработки системы?

только при наличии большого опыта и экономических предпосылок
всегда да, при наличии большого опыта

всегда нет, поскольку это удорожает и замедляет процесс ее создания
всегда да, поскольку это удешевляет и убыстряет процесс ее создания

4. Класс поведения - это?

абстрактный класс

отложенный класс

класс, описывающий поведение пользователей системы

класс, для которого задана частичная реализация, но некоторые особенности оставлены для реализации различными потомками

5. Что представляет собой наиболее приемлемый образец, полезный для повторного использования?

спецификацию проблемы

сценарий решения проблемы

компонент ПО

описание алгоритма

6. Что можно повторно использовать?

программный код

текст

спецификации

персонал

7. Повторно использовать или переделывать

следует использовать объектный инструментарий (наследование, обертывание), удовлетворяющий критерию Открыт-Закрыт, допускающий расширяемость и повторное использование

если компонент лишь частично удовлетворяет потребностям, его следует переделать
повторно использовать следует лишь тот компонент, который полностью соответствует потребностям

8. Перегрузка может быть?

экстраординарной
динамической
многократной
статической

9. Роль универсальности в повторном использовании в том, что?

позволяет справиться с проблемой изменчивости типов
взаимозависимости операций
позволяет справиться с проблемой независимости представлений
позволяет справиться с проблемой изменчивости реализаций

10. Проблема изменчивости реализаций для классов поведения состоит в том, что?

класс поведения может не фиксировать спецификацию, так что реализация зависит от дальнейшего уточнения спецификации
класс поведения может не фиксировать алгоритм, так что реализация зависит от дальнейшего выбора алгоритма

реализация с течением времени изменяется

класс поведения может не фиксировать структуру данных, так что реализация зависит от дальнейшего выбора структуры данных

11. Проблемы повторного использования связаны с?

непрерывностью представлений

вариацией представлений

изменчивостью типов

взаимозависимостью операций

12. Форматом распространения ПИК, допускающим расширения, является

двоичный код исполняемого модуля

исходный текст на языке программирования

двоичный код исполняемого модуля и описание интерфейса

код на промежуточном языке

13. Отметьте истинные высказывания?

решение проблемы "повторно использовать или переделать" должно позволять сохранять одни свойства повторно используемого модуля и адаптировать другие

универсальность решает все проблемы повторного использования

образцы, описанные в литературе, частично решают проблему повторного использования

перегрузка метода является синтаксическим средством; она не решает важных проблем повторного использования и затрудняет читабельность текстов программ

14. Отметьте истинные высказывания?

знаки операций могут быть перегруженными

синдромы NIN и HIN препятствуют повторному использованию

пакеты можно рассматривать как абстрактные модули

метод нисходящего проектирования обеспечивает декомпозицию системы на подпрограммы

15. Отметьте истинные высказывания?

пакеты удовлетворяют всем потребностям повторного использования

на создание ПИК требуются дополнительные затраты

подпрограммы удовлетворяют всем потребностям повторного использования

динамическое связывание является формой

Набрано баллов

Модуль 2

5 К объектной технологии

5.1 Функциональная декомпозиция

Вначале мы рассмотрим достоинства и ограничения традиционного подхода, использующего функции в качестве основы архитектуры программных систем.

Непрерывность

Ключевой проблемой при ответе на вопрос: “вокруг чего следует структурировать системы: вокруг функций или вокруг данных?” является проблема расширяемости, более точно - цель, названная непрерывностью в предшествующих лекциях. Как вы помните, метод проектирования удовлетворяет этому критерию, если он приводит к устойчивой архитектуре, обеспечивающей объем изменений в проекте, соразмерный объему изменений в спецификации.

Обеспечение непрерывности - это главная забота при рассмотрении реального жизненного цикла программных систем, включающего не только производство приемлемой первоначальной версии, но и эволюцию системы на протяжении долгого времени. Большинство систем подвергаются многочисленным изменениям после их первоначальной поставки. Поэтому всякая модель разработки ПО, которая рассматривает только период, предшествующий этой поставке, и игнорирующая последующую эру изменений и пересмотров, весьма далека от реальной жизни.

Чтобы оценить качество архитектуры (и породившего ее метода), нужно понять не только то, насколько просто было изначально получить эту архитектуру, не менее важно выяснить, насколько легко ее можно изменить.

Традиционным ответом на этот вопрос была функциональная декомпозиция “сверху вниз”, кратко определенная в одной из предыдущих лекций. Насколько хорошо разработка сверху вниз отвечает требованиям модульности?

Проектирование сверху вниз

Там был также весьма изобретательный архитектор, придумавший новый способ постройки домов. Постройка должна была начинаться с крыши и кончаться фундаментом. Он оправдывал мне этот способ ссылкой на приемы двух мудрых насекомых - пчелы и паука.

Джонатан Свифт, “Путешествия Гулливера”

При подходе сверху вниз система строится с помощью последовательных уточнений. Этот процесс начинается с самого общего утверждения об ее абстрактной функции, такого как

[C0]

"Оттранслировать СИ-программу в машинный код"

[P0]

'Обработать команду пользователя"

и продолжается путем последовательных шагов уточнения. На каждом шаге уровень абстракции получаемых элементов должен уменьшаться, каждая операция на нем разлагается на композицию одной или нескольких более простых операций. Например, следующий шаг в первом примере (транслятор с СИ) может привести к декомпозиции

[C1]

"Прочитать программу и породить последовательность лексем"
"Разобрать последовательность лексем и построить
абстрактное синтаксическое дерево"
"Снабдить дерево семантической информацией"
"Сгенерировать по полученному дереву код"

или, используя другую структуру (и сделав упрощающее предположение, что СИ-программа - это последовательность определений функций):

```
[C'1]
from
  "Инициализировать структуры данных"
until
  "Определения всех функций обработаны"
loop
  "Прочитать определение следующей функции"
  "Сгенерировать частичный код"
end
"Заполнить перекрестные ссылки"
```

В любом случае разработчик должен на каждом шаге проверять оставшиеся не полностью уточненными элементы (такие как "Читать программу..." и "Определения всех функций обработаны") и раскрывать их, используя тот же процесс уточнения до тех пор, пока все не окажется на достаточном низком уровне абстракции, допускающем непосредственную реализацию.

Процесс уточнения сверху вниз можно представить как построение дерева. Вершины представляют элементы декомпозиции, ветви показывают отношение "В есть уточнение А".

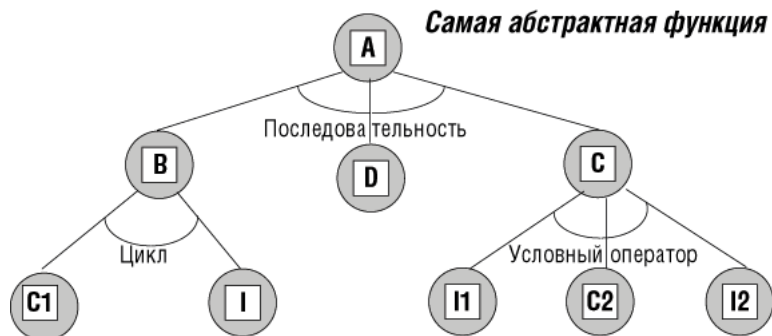


Рис. 5.1: Разработка сверху вниз: структура дерева

У метода проектирования сверху вниз имеется ряд достоинств. Он логичен, хорошо организует дисциплину мышления, поддается эффективному изучению, поощряет систематическое проектирование систем, помогает разработчику найти пути преодоления больших сложностей, возникающих обычно на начальной стадии разработки систем.

Нисходящий подход может быть весьма полезен при разработке отдельных алгоритмов. Однако у него есть ряд ограничений, которые делают сомнительным использование этого подхода при проектировании целых систем:

- Сомнительной является сама идея охарактеризовать всю систему посредством только одной функции.
- Используя в качестве основы декомпозиции системы на модули свойства, которые склонны подвергаться наибольшим изменениям, этот метод не способен учесть эволюционную природу



Рис. 5.2: Структура простой системы расчета зарплаты

программных систем.

Не только одна главная функция

При эволюции системы то, что вначале воспринималось как ее главная функция, с течением времени может стать менее важным.

Рассмотрим типичную систему расчета зарплаты. При формулировке начальных требований заказчик мог представить лишь то, что следует из ее названия: систему для генерации чеков на зарплату по соответствующим данным. Его представление системы, явное или неявное, могло оказаться версией следующей схемы, возможно, чуть более амбициозное:

Эта система получает некоторые входные данные (такие как часы работы служащего и некоторую информацию о нем) и производит некоторые выходные данные (чеки и т. п.). Это простая функциональная спецификация, в строгом смысле слова “функциональный”. Она определяет программу как механизм для выполнения одной функции - платить зарплату служащим. Функциональный метод проектирования сверху вниз предназначен как раз для таких строго очерченных проблем, когда задание состоит в вычислении одной функции - “вершины” конструируемой системы.

Предположим, однако, что разработка нашей платежной системы благополучно завершена и про-

грамма выполняет всю необходимую работу. Скорее всего, на этом разработка не прекратится. Хорошие системы имеют противную привычку возбуждать в своих пользователях множество идей о других вещах, которые они могут делать. Как разработчику системы вам было сказано вначале, что все, что вы должны сделать - это сгенерировать чеки и пару вспомогательных выходных данных. Но затем просьбы о расширениях начинают попадать на ваш стол одна за другой: “Может ли программа собирать некоторую дополнительную статистику?” “Я говорил вам, что в следующем квартале мы собираемся начать платить некоторым служащим ежемесячно, а некоторым - дважды в месяц, не так ли?” “И, между прочим, мне нужен ежемесячный суммарный отчет для администрации и еще один ежеквартальный для акционеров”. “Бухгалтерам требуется отдельный отчет для начисления налогов”. “Кстати, правильно ли вы храните информацию о зарплате? Очень хотелось бы предоставить персоналу интерактивный доступ к ней. Не понимаю, почему трудно добавить такую функцию?”

Процесс изменений происходит непрерывно. Новая система все еще является во многих отношениях “той же”, что и старая: все еще платежной системой, программой для ядерной физики, компилятором. Но исходная “главная функция”, которая вначале выглядела самой важной, часто становится просто одной из функций системы, а иногда и совсем исчезает, становясь ненужной.

Если при анализе и проектировании используется метод декомпозиции, основанный на функциях, то структура системы будет вытекать из исходного понимания разработчиками главной функции системы. При этом добавление всякой новой функции, даже если оно кажется заказчику простым, может разрушить всю структуру системы. Поэтому очень важно найти в качестве критерия декомпозиции свойства менее изменчивые, чем главная функция системы.

Интерфейсы и проектирование ПО

Архитектура системы должна основываться на содержании, а не на форме. Но проектирование сверху вниз стремится использовать в качестве основы для структуры самый поверхностный аспект

системы - ее внешний интерфейс.

Такой упор на внешний интерфейс неизбежен для метода, ключевой вопрос которого: “Что система будет делать для конечного пользователя?” Ответ на него обязательно будет акцентироваться на самых внешних аспектах.

Интерфейс пользователя, как правило, оказывается одним из наиболее изменчивых компонентов, поскольку трудно получить правильный интерфейс с первой попытки. Довольно часто удается построить интерфейс отдельно от других компонент системы, используя один из множества доступных сегодня инструментов реализации элегантных и дружелюбных интерфейсов, основанных на ОО-методах. В таких случаях интерфейс пользователя почти не оказывает влияния на проектирование всей системы.

Производство и описание

Одна из причин первоначальной привлекательности идей проектирования сверху вниз заключается в том, что этот стиль может быть удобен для объяснения каждого шага разработки. Но то, что хорошо для документации существующей разработки, не обязательно является наилучшим способом для ее проведения.

Сверху вниз - это разумный способ описания уже полностью понятых вещей. Но это неподходящий способ для проектирования, разработки или открытия чего-либо нового. Здесь имеется близкая параллель с математикой. В учебниках по математике ее отдельные дисциплины описываются в логическом порядке: каждая сформулированная и доказанная теорема используется при доказательстве последующих теорем. Но на самом деле эти теоремы не создавались или открывались указанными способами или в указанном порядке... Если у разработчика системы или программы в голове уже имеется ясное представление об окончательном результате, то он может применить метод сверху вниз, чтобы описать на бумаге то, что имеется у него в голове. Именно поэтому люди могут считать,

что они проектируют и разрабатывают сверху вниз и делают это весьма успешно: они смешивают способ описания с методом разработки. Когда начинается этап сверху вниз, задача уже решена и осталось уточнить лишь некоторые детали.

Проектирование сверху вниз: общая оценка

Проведенное обсуждение функционального проектирования сверху вниз показывает, что этот метод плохо приспособлен для разработки важных систем. Он остается полезной парадигмой для небольших программ и отдельных алгоритмов, он также полезен для описания хорошо понятных алгоритмов, особенно в учебниках по программированию. Но он не масштабируем и не годится для больших практических программных систем.

5.2 Декомпозиция, основанная на объектах

Использование объектов (или, более точно, как будет видно далее, - типов объектов) как ключа для разбиения системы на модули основано на содержательных целях, в частности, на расширяемости, возможности повторного использования и совместимости.

Доводы в пользу применения объектов будут довольно краткими, так как этот вопрос был уже ранее рассмотрен: многие из аргументов против основанного на функциях проектирования сверху вниз естественно превращаются в свидетельства в пользу основанного на объектах проектирования снизу вверх.

Эти свидетельства, тем не менее, не должны привести к полному отказу от функций. Как было отмечено в начале лекции, никакой подход к созданию ПО не может быть полным, если он не учитывает обе стороны - функции и объекты. Поэтому нам нужно и в ОО-методе сохранить надлежащее место для функций, даже если они в результирующей архитектуре системы будут подчинены

объектам. Понятие абстрактного типа данных предоставит нам определение объектов, в котором для функций зарезервировано подходящее место.

Расширяемость

Так как функции системы имеют тенденцию изменяться в течение ее жизни, то возникает вопрос о поиске более стабильной характеристики ее существенных свойств, которая могла бы руководить нашим выбором модулей и соответствовала бы цели непрерывности.

Типы объектов, с которыми работает система, являются более перспективными кандидатами. Что бы ни случилось с использованной в примере выше системой расчета зарплаты, она все равно будет манипулировать объектами, представляющими служащих, штатные расписания с зарплатами, инструкции компании, табель учета рабочего времени, чеки. Что бы ни случилось с компилятором или другим средством обработки языка, он все еще будет манипулировать исходными текстами, последовательностями лексем, деревьями разбора, абстрактными синтаксическими деревьями, целевым кодом. Что бы ни случилось с системой, реализующей метод конечных элементов, она по-прежнему будет манипулировать матрицами, конечными элементами и сетками.

Возможность повторного использования

Обсуждение возможности повторного использования показало, что процедура (элемент функциональной декомпозиции) обычно недостаточна как единица для повторного использования.

Отсюда появилась идея, что в этой задаче модулем, достаточно хорошо допускающим повторное использование, должна быть совокупность таких операций. Но если попытаться понять, какая концепция все эти операции объединяет, то мы обнаружим тип объектов, к которым они применяются - таблицы.

Такие примеры подсказывают, что типы объектов, полностью снабженные связанными с ними операциями, и будут стабильными единицами для повторного использования.

Совместимость

Другой показатель качества ПО, совместимость, был определен как легкость, с которой программные продукты (в данном обсуждении - модули) можно комбинировать между собой.

Если структуры данных не проектировались с этой целью, то имеющие к ним доступ действия комбинировать очень сложно. Почему бы тогда не попробовать комбинировать целиком структуры данных?

Объектно-ориентированное конструирование ПО

У нас уже накоплено достаточно оснований, чтобы попытаться определить ОО-конструирование ПО. Это будет лишь первый набросок, более конкретное определение последует в следующей лекции.

ОО-конструирование ПО - это метод разработки ПО, который строит архитектуру всякой программной системы на модулях, выведенных из типов объектов, с которыми система работает (а не на одной или нескольких функциях, которые она должна предоставлять).

Вопросы

Приведенное выше определение послужит отправной точкой для обсуждения ОО-метода. Оно не только дает ответ на некоторые относящиеся к ОО-проектированию вопросы, но и побуждает задать много новых вопросов таких, как:

- Как находить релевантные типы объектов?

- Как описывать типы объектов?
- Как описывать взаимоотношения типов объектов и их близость?
- Как использовать типы объектов для структурирования ПО?

5.3 Ключевые концепции

- Архитектуру системы можно получить исходя из функций или из типов объектов.
- Описание, основанное на типах объектов, с течением времени обеспечивает лучшую устойчивость и лучшие возможности для повторного использования, чем описание, основанное на анализе функций системы.
- Как правило, неестественно считать, что задача системы состоит в реализации только одной функции. У реальной системы обычно имеется не одна “вершина” и ее лучше описывать как систему, предоставляющую множество услуг.
- На ранних стадиях проектирования и разработки системы не нужно уделять много внимания ограничениям на порядок действий. Многие временные соотношения могут быть описаны более абстрактно в виде логических ограничений.
- Функциональное проектирование сверху вниз не подходит для программных систем с долгим жизненным циклом, включающим их изменения и повторное использование.
- При ОО-конструировании ПО структура системы основывается на типах объектов, с которыми она работает.

- При ОО-разработке первоначальный вопрос не в том, что система делает, а в том, с какими типами объектов она это делает. Решение о том, какая функция является самой верхней функцией системы (и имеется ли таковая), откладывается на последние этапы процесса проектирования.
- Чтобы проектируемое ПО было расширяемым и допускало повторное использование, ОО-конструирование должно выводить архитектуру из достаточно абстрактных описаний объектов.

Контрольные вопросы

1. На какой основе следует строить модуль?

объектов
процессоров
функций
спецификаций

2. Функциональная декомпозиция имеет следующие достоинства

система строится на основе хорошо понятных последовательных уточнений
уровень абстракции на каждом шаге уточнения уменьшается
выделяет главную функцию системы
позволяет справиться со сложностью исходной задачи

3. Функциональная декомпозиция имеет следующие ограничения

реальная система имеет широкий спектр сервисов, среди которых трудно выделить главную функцию

не способствует поддержке расширяемости

не поддерживает самодокументирование

плохо согласуется с предыдущими версиями

4. Порядок выполнения модулей нужно устанавливать

как можно раньше

как можно позже

с помощью логических ограничений

5. ОО-конструирование – это?

метод разработки, в котором объекты подчинены функциям

метод разработки, строящий архитектуру программной системы на функциях, которые эта система реализует

метод разработки, строящий архитектуру программной системы на модулях, выведенных из типов объектов

метод разработки, в котором функции подчинены объектам

6. Проектирование интерфейса пользователя

должно выполняться на самых ранних этапах ОО-проектирования

после завершения проектирования основных классов

должно выполняться на поздних этапах ОО-проектирования

в процессе проектирования каждого класса

7. Функциональная декомпозиция при проектировании сверху-вниз

обеспечивает хорошее соответствие проекта его начальной спецификации

облегчает понимание каждого шага декомпозиции

затрудняет связь с предыдущими версиями проекта

способствует повторному использованию

8. При проектировании системы типов можно использовать?

объекты, описанные в литературе

объекты солнечной системы

объекты физической реальности, к которой применяется ПО

повторно используемые объекты

9. Девиз ОО-разработки

не спрашивай вначале, кто в системе это делает; спроси, зачем он это делает

не спрашивай вначале, что система делает; спроси, кто в системе это делает

не спрашивай вначале, что система делает; спроси, насколько хорошо она это делает

не спрашивай вначале, кто в системе это делает; спроси, что система делает

10. Главная функция проекта

у проекта может вообще не существовать главной функции

проект может иметь несколько главных функций

наиболее стабильная и неизменная часть проекта
может меняться в процессе разработки

11. Главный вопрос, который следует задавать при проектировании системы - это?

что за функции выполняет система
кто выполняет функции системы
зачем выполнять функции системы
какова стоимость выполнения функций системы

12. При описании типов объектов следует руководствоваться требованиями

первоочередного задания структуры данных
нахождения для описания функций подходящего места в архитектуре системы
независимости описаний от представлений
независимости описаний от спецификаций

13. Отметьте истинные высказывания

архитектуру системы можно получить исходя из функций или из типов объектов
при проектировании модульной структуры объекты стабильнее функций
подход сверху-вниз хорошо подходит для описания разработки
при ОО-подходе проектируются объекты и не рассматриваются функции

14. Отметьте истинные высказывания

между типами объектов могут существовать два вида отношений: “быть клиентом” и наследование

естественно считать, что задача системы состоит в реализации только одной функции
вычисление включает три вида ингредиентов: потоки управления, функции и данные
на ранних стадиях проектирования и разработки системы нужно уделять много внимания
ограничениям на порядок действий

15. Отметьте истинные высказывания

в классической объектной технологии имеются только два отношения между типами объектов: быть клиентом и быть наследником

функциональное проектирование сверху вниз не подходит для программных систем с долгим жизненным циклом, включающим их изменения и повторное использование

отношение наследования покрывает многочисленные формы специализации

описание, основанное на анализе функций системы с течением времени обеспечивает лучшую устойчивость и лучшие возможности для повторного использования, чем описание, основанное на типах объектов

Набрано баллов

6 Абстрактные типы данных (АТД)

Чтобы объекты играли лидирующую роль в архитектуре ПО, нужно их адекватно описывать. В этой лекции показывается, как это делать.

Это открыло мне глаза, я начал понимать, что значит использовать инструмент, называемый алгеброй. Черт возьми, никто никогда не говорил мне ничего подобного раньше. Мсье Дюпюи [учитель математики] произносил напыщенные фразы об этом предмете, но ни разу не сказал этих простых слов: это разделение труда, которое, как и всякое другое разделение труда производит чудеса и позволяет уму сконцентрировать все свои силы только на одной стороне объектов, только на одном из их качеств.

Насколько другим это предстало бы перед нами, если бы мсье Дюпюи сказал нам: "Этот сыр мягкий или твердый, он белый, он синий, он старый, он молодой, он твой, он мой, он легкий или он тяжелый. Из всех его многочисленных качеств давайте рассматривать только вес. Каким ни был этот вес, давайте назовем его А. А теперь, не думая больше о весе, давайте применять к А все, что мы знаем о количестве."

Такая простая вещь, но до сих пор никто не говорил нам о ней в этой отдаленной провинции...

Стендаль, "Жизнь Анри Брюлара"

6.1 Критерии

Чтобы получить надлежащие описания объектов, наш метод должен удовлетворять трем условиям:

- Описания должны быть точными и недвусмысленными.
- Они должны быть полными - или, по крайней мере, иметь в каждом конкретном случае нужную нам полноту (некоторые детали можно намеренно опускать).
- Они не должны быть излишне специфицированы.

Последний пункт делает ответ нетривиальным. В конце концов, легко сделать описание точным, недвусмысленным и полным, если мы готовы "выдать все секреты указав все детали объектного представления. Но такое описание, как правило, будет включать чересчур много информации для авторов программ, которым требуется доступ к таким объектам.

Это замечания похожи на комментарии, которые привели к понятию скрытия информации. Там дело было в том, что, предоставляя в качестве первичного источника информации исходный код модуля (элементы, связанные с реализацией) авторам клиентских программ, зависящих от этого модуля, мы можем окунуть их в поток деталей, который помешает им сосредоточиться на своей собственной работе и затруднит перспективу развития проекта. Здесь нас ожидает та же опасность, что и в случае, когда мы позволяем модулям использовать некоторую структуру данных на основании информации, которая относится к представлению этой структуры, а не к ее существенным свойствам.

Различные реализации

Чтобы лучше понять всю важность описаний абстрактных типов данных, исследуем глубже потенциальные последствия использования физической реализации в качестве основы описания объектов.

Удобным и хорошо изученным примером является описание объектов типа стек. Объект стек служит для того, чтобы накапливать и доставать другие объекты в режиме "последним пришел - первым ушел"("LIFO"), элемент, вставленный в стек последним, будет извлечен из него первым. Стек повсеместно используется в информатике и во многих программных системах, в частности, компиляторы и интерпретаторы усыпаны разными видами стеков.

Представления стеков

Существует несколько физических представлений стеков:

Этот рисунок иллюстрирует три наиболее популярных представления стеков. Для удобства ссылок дадим каждому из них свое имя:

МАССИВ_ВВЕРХ представляет стек посредством массива **representation** и целого числа **count**, с диапазоном значений от 0 (для пустого стека) до **capacity** - размера массива **representation**, элементы стека хранятся в массиве и индексируются от 1 до **count**.

МАССИВ_ВНИЗ похож на **МАССИВ_ВВЕРХ**, но элементы помещаются в конец стека, а не в начало. Здесь число, называемое **free**, является индексом верхней свободной позиции в стеке или 0, если все позиции в массиве заняты и изменяется в диапазоне от **capacity** для пустого стека до 0 для заполненного. Элементы стека хранятся в массиве и индексируются от **capacity** до **free+1**.

СПИСОЧНОЕ при списочном представлении каждый элемент стека хранится в ячейке с двумя полями: **item**, содержащем сам элемент, и **previous**, содержащем указатель на ячейку с предыдущим элементом. Для этого представления нужен также указатель **last** на ячейку, содержащую вершину стека.

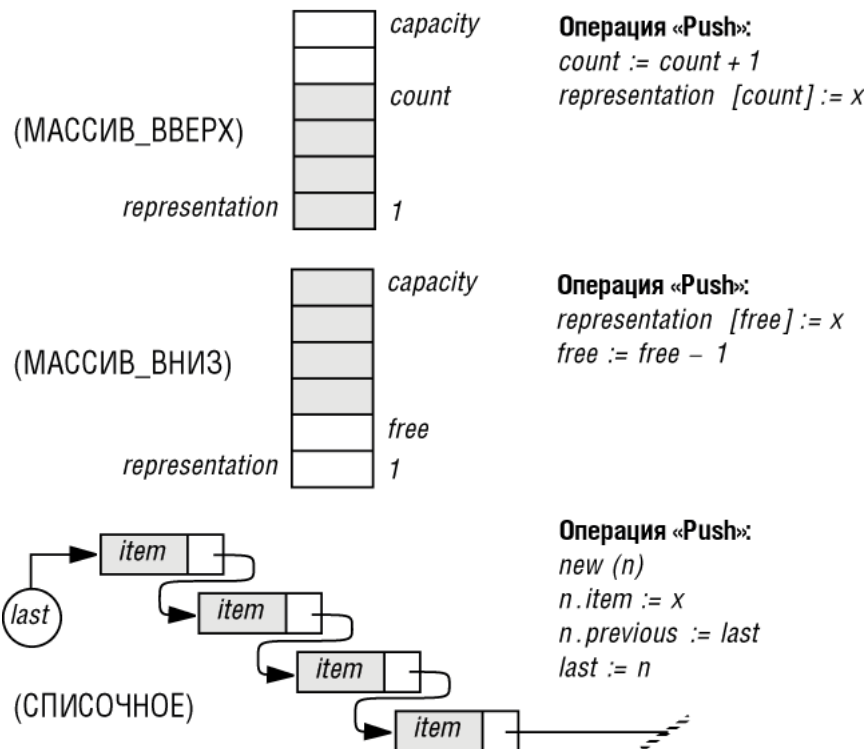


Рис. 6.1: Три возможных представления стеков

Для представлений с помощью массивов `МАССИВ_ВВЕРХ` и `МАССИВ_ВНИЗ` команды увеличивают или уменьшают указатель на вершину (`count` или `free`) и присваивают `x` соответствующему элементу массива. Так как эти представления поддерживают стеки с не более чем `capacity` элементами, то корректные реализации должны содержать защищающие от переполнения тесты соответствующего вида:

```
if count " capacity then ...  
if free " 0 then ...,
```

(на рисунке они для простоты опущены).

Для представления `СПИСОЧНОЕ` вталкивание элемента требует четырех действий:

- создания новой ячейки `p` (здесь оно выполняется с помощью процедуры Паскаля `new`, которая выделяет память для нового объекта);
- присваивания `x` полю `item` новой ячейки;
- присоединения новой ячейки к вершине стека путем присвоения ее полю `previous` текущего значения указателя `last`;
- изменения `last` так, чтобы он ссылался на только что созданную ячейку.

Хотя эти представления встречаются чаще всего, существует и много других представлений стеков. Например, если вам нужны два стека с однотипными элементами и память для их представления ограничена, то можно использовать один массив с двумя метками вершин `count` как в представлении `МАССИВ_ВВЕРХ` и `free` как в `МАССИВ_ВНИЗ`. При этом один стек будет расти вверх, а другой - вниз. Условием полного заполнения этого представления является равенство `count = free`.

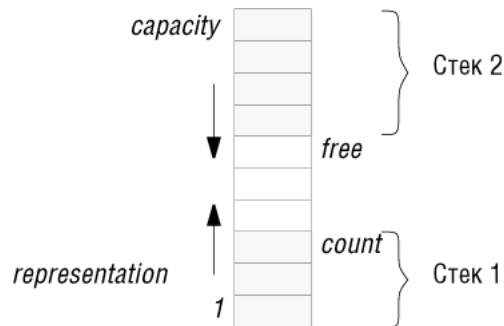


Рис. 6.2: Представление двух стеков лицом к лицу

Преимущество такого представления состоит в уменьшении риска переполнить память: при двух массивах размера n , представляющих стеки способом `МАССИВ_ВВЕРХ` или `МАССИВ_ВНИЗ`, память исчерпается, как только любой из стеков достигнет n элементов. А в случае одного массива размера $2n$, содержащего два стека лицом к лицу, работа продолжается до тех пор, пока их общая длина не превысит $2n$, что менее вероятно, если стеки растут независимо друг от друга.

Каждое из этих и другие возможные представления полезны в разных ситуациях. Выбор одного из них в качестве эталона для определения стека был бы типичным примером излишней спецификации. Почему мы должны, например, предпочесть `МАССИВ_ВВЕРХ` представлению `СПИСОЧНОЕ`? Большинство видимых свойств представления `МАССИВ_ВВЕРХ` - массив, число `count`, верхняя граница - несущественны для понимания представляемой ими структуры.

Опасность излишней спецификации

Почему так плохо использовать конкретное представление в качестве спецификации?

Можно напомнить результаты изучения Линцем (Lientz) и Свенсоном (Swanson) стоимости сопровождения. Было установлено, что более 17% стоимости ПО приходится на изменения в форматах данных. Ясно, что метод, который ставит анализ и проектирование в зависимость от физического представления структур данных, не обеспечит разработку достаточно гибкого ПО.

Поэтому при использовании объектов или типов объектов в качестве основы для архитектуры системы требуется найти лучший способ описания, чем конкретное представление.

6.2 К абстрактному взгляду на объекты

Как нам сохранить полноту, точность и однозначность, не заплатив за это излишней спецификацией?

Использование операций

Представления стека при всех их различиях объединяет то, что они описывают структуру "хранения"(т.е. структуру, используемую для хранения других объектов), к которой применяются определенные операции, обладающие определенными свойствами. Сосредоточившись не на выборе конкретного представления структуры, а на этих операциях и свойствах, можно получить достаточно абстрактное, но, тем не менее, полезное, описание понятия стек.

Обычно для стеков рассматриваются следующие операции:

- Команда вталкивания некоторого элемента на вершину стека. Назовем эту операцию `put`.
- Команда удаления верхнего элемента стека. Назовем ее `remove`.

Таблица 6.1: Имена операций над стеком

Стандартное имя операции над стеком	Имя, используемое здесь
Push (втолкнуть)	Put (поместить)
Pop (вытолкнуть)	Remove (удалить)
Top (вершина)	Item (элемент)
New (новый)	Make (создать)

- Запрос элемента, находящегося на вершине стека (если стек не пуст). Назовем его `item`.
- Запрос на проверку пустоты стека. (Он позволит клиентам заранее проверить возможность операций `remove` и `item`.)

Кроме того, нам понадобится операция-конструктор для создания пустого стека. Назовем ее `make`.

Чтобы сделать главный шаг в направлении абстракции данных, нужно стать на противоположную точку зрения: забыть на некоторое время о конкретном представлении и взять в качестве определения структуры данных операции сами по себе. Иначе говоря, стек - это любая структура, к которой клиенты могут применять перечисленные выше операции.

Согласованность имен

Зачем использовать терминологию, отличающуюся от общепринятой? Причина - в желании достичь более высокого уровня понимания структур данных - особенно "контейнеров которые используются для хранения объектов.

Стеки это просто один из видов контейнеров, точнее они относятся к категории контейнеров, которые можно назвать распределителями. Распределитель предоставляет своим клиентам механизм

для хранения (**put**), извлечения (**item**) и удаления (**remove**) объектов, но не дает им возможности управлять тем, какой объект будет извлекаться или удаляться. Например, метод доступа LIFO, используемый в стеках, позволяет извлекать или удалять только тот элемент, который был сохранен последним. Другой вид распределителей - очередь, которая использует метод доступа "первым в, первым из"(FIFO): элементы добавляются в один конец очереди, а извлекаются и удаляются - с другого конца. Пример контейнера, не являющегося распределителем, - это массив, в нем вы сами выбираете целочисленные номера позиций, в которые вставляются или из которых извлекаются объекты.

Поскольку схожесть разных видов контейнеров (распределителей, массивов и т.п.) более важна, чем различия между тем, как они хранят, извлекают или удаляют объекты, эта книга твердо придерживается стандартизированной терминологии, которая сглаживает различия между вариантами структур данных и, наоборот, подчеркивает их общность. Поэтому базисная операция извлечения элемента будет всегда называться **item**, базисная операция удаления элемента будет всегда называться **remove**, и т.д.

Формализация спецификаций

Приведенные содержательные описания явно недостаточны - **put** вталкивает элемент на "вершину"стека, **remove** выталкивает элемент, находящийся на вершине. Нам нужно точно знать, как клиенты могут использовать эти операции и что они для этого должны делать.

Спецификация АТД предоставит эту информацию. Она состоит из четырех разделов, разъясняемых в следующих разделах:

- ТИПЫ
- ФУНКЦИИ

- АКСИОМЫ
- ПРЕДУСЛОВИЯ

Для спецификации АТД в этих разделах будут использоваться простая математическая нотация.

Специфицирование типов

В разделе ТИПЫ указываются специфицируемые типы. В общем случае, может оказаться удобным определять одновременно несколько АТД, хотя в нашем примере имеется лишь один тип STACK(СТЕК). Между прочим, что такое тип? Ответ на этот вопрос объединит все положения, развиваемые далее в этой лекции: тип - это совокупность объектов, характеризуемая функциями, аксиомами и предусловиями. Не будет большой ошибкой рассматривать пока тип как множество объектов в математическом смысле слова "множество тип STACK как множество всех возможных стеков, тип INTEGER как множество всех целых чисел и т.д.

Однако при этом не должно быть никакой путаницы: АТД, такой как STACK, - это не объект (один конкретный стек), а совокупность объектов (множество всех стеков). Напомним, в чем состоит наша главная цель: найти подходящую основу для модулей наших программной систем. Очевидно, не имеет смысла делать основой для модуля один конкретный объект - один стек, один самолет, один счет в банке. ОО-проектирование даст нам возможность строить модули, отражающие свойства всех стеков, всех самолетов, всех банковских счетов, или, по крайней мере, значительной их части.

Объект, принадлежащий множеству объектов, описываемых спецификацией АТД, называется экземпляром этого АТД. Например, конкретный стек, обладающий свойствами абстрактного типа данных STACK, будет экземпляром АТД STACK. Понятие экземпляра проходит через все ОО-проектирование и программирование, и будет играть важную роль в объяснении поведения программ во время исполнения.

В разделе ТИПЫ просто перечисляются типы, вводимые в данной спецификации. Здесь:

- Типы
 - STACK[G]

Таким образом, наша спецификация относится к одному абстрактному типу данных - STACK, задающему стеки объектов произвольного типа G.

Универсализация (Genericity)

В описании STACK[G] именем G обозначен произвольный, не определяемый тип. G называется формальным родовым параметром для типов элементов АТД STACK, а сам STACK называется родовым или универсальным АТД. Механизм, допускающий такие параметризованные спецификации, известен как универсализация, мы уже сталкивались с аналогичным понятием в обзоре конструкций пакетов.

Перечисление функций

Вслед за разделом ТИПЫ идет раздел ФУНКЦИИ, в котором перечисляются операции, применяемые к экземплярам данного АТД. Как уже говорилось, эти операции будут главными компонентами определения типа, с их помощью описывается, что могут предложить его экземпляры, а не то, чем они являются.

Ниже приведен раздел ФУНКЦИИ для абстрактного типа данных STACK. Если вы разработчик ПО, то этот стиль описания вам знаком: строки этого раздела напоминают декларации типизированных языков программирования. Строка для операции new похожа на объявление переменной, остальные - на заголовки процедур.

- Функции

- `put`: $\text{STACK } [G] \times G \rightarrow \text{STACK } [G]$
- `remove`: $\text{STACK } [G] \not\rightarrow \text{STACK } [G]$
- `item`: $\text{STACK } [G] \not\rightarrow G$
- `empty`: $\text{STACK } [G] \rightarrow \text{BOOLEAN}$
- `new`: $\text{STACK } [G]$

В каждой строке вводится определенная математическая функция, моделирующая соответствующую операцию над стеком. Например, функция `put` представляет операцию, которая вталкивает элемент на вершину стека.

Почему функции? Большая часть программистов не посчитает такую операцию как `put` функцией. Когда во время работы программной системы операция `put` применяется к стеку, она, как правило, изменяет этот стек, добавляя к нему элемент. Вследствие этого в приведенной выше классификации операций `put` была "командой операцией, которая может модифицировать объекты. (Две другие категории операций - это конструкторы и запросы).

Однако спецификация АТД - это математическая модель и в ее основании должны быть корректные математические методы. В математике понятие команды или, более общо, изменение чего-либо как таковое отсутствует: вычисление квадратного корня из числа 2 не изменяет само это число. Математические выражения просто определяют одни математические объекты в терминах некоторых других математических объектов. В отличие от вычисления программы на компьютере, они никогда не изменяют никакие математические объекты. Но поскольку мы нуждаемся в некотором математическом объекте для моделирования операций компьютера, то понятие функции представляется наиболее близким приближением. Функция - это механизм для получения некоторого результата,

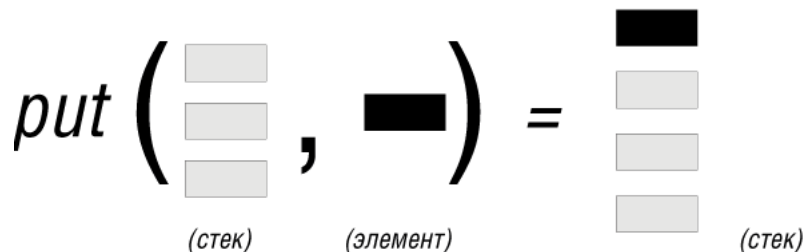


Рис. 6.3: Применение функции `put`

принадлежащего некоторому результирующему множеству по любому допустимому входу, принадлежащему некоторому исходному множеству.

Спецификации абстрактных типов данных используют именно это понятие. Например, операция `put` определяется как

$$\text{put}: \text{STACK } [G] \times G \rightarrow \text{STACK } [G]$$

и означает, что `put` будет брать два аргумента: `STACK` экземпляров типа `G` и экземпляр типа `G` и возвращать в качестве результата новый `STACK [G]`. (Более формально, множеством определения функции `put` является множество `STACK [G] × G`, являющееся декартовым произведением множеств `STACK [G]` и `G`, т.е. множеством пар $\langle s, x \rangle$, в которых первый элемент s принадлежит `STACK [G]`, а второй элемент x принадлежит `G`.) Вот рисунок, иллюстрирующий это:

АТД имеют дело только с математическими функциями, у которых нет никаких побочных эффектов и которые, на самом деле, ничего не изменяют.

Из нашего обсуждения следуют роли операций, моделируемых каждой из функций спецификации `STACK`:

- Функция `put` возвращает новое состояние стека с одним новым элементом, помещенным на его вершину. Рисунок на предыдущей странице иллюстрирует операцию `put(s, x)`, выполняемую над стеком `s` и элементом `x`.
- Функция `remove` возвращает новое состояние стека с вытолкнутым верхним элементом, если таковой был. Как и `put`, эта функция при проектировании и реализации должна превращаться в команду (операцию, изменяющую объект, обычно реализуемую как процедура). Мы увидим далее, как учесть возможность пустого стека, с вершины которого нечего удалять.
- Функция `item` возвращает верхний элемент стека, если таковой имеется.
- Функция `empty` выявляет пустоту стека, ее результатом является логическое значение (истина или ложь). Предполагается, что АТД `BOOLEAN`, задающий логические значения, определен отдельно.
- Функция `new` создает пустой стек.

В разделе ФУНКЦИИ эти функции определяются не полностью, вводятся только их сигнатуры - списки типов их аргументов и результата. Сигнатура функции `put`

$$\text{STACK } [G] \times G \rightarrow \text{STACK } [G]$$

показывает, что `put` берет в качестве аргумента пару вида $\langle s, x \rangle$, в которой `s` - экземпляр типа `STACK [G]`, а `x` - экземпляр типа `G`, и возвращает в качестве результата экземпляр типа `STACK [G]`. Вообще говоря, множество значений функции (его тип указывается в сигнатуре правее стрелки, здесь это `STACK [G]`) может само быть декартовым произведением. Это можно использовать при описании операций, возвращающих два или более результатов.

В сигнатуре функций `remove` и `item` вместо обычной стрелки используется перечеркнутая стрелка `.`. Это означает, что эти функции применимы не ко всем элементам множества входов. Описание функции `new` выглядит просто как

`new: STACK`

без всякой стрелки в сигнатуре. Здесь аргументы не нужны, поскольку `new` должна всегда возвращать один и тот же результат - пустой стек. Поэтому для простоты мы убрали здесь стрелку. Результат применения этой функции (т. е. пустой стек) будет записываться `new`.

Категории функций

В альтернативной терминологии эти три категории называются "конструктор", "аксессор" и "модификатор". Здесь мы придерживаемся терминов, более непосредственно связанных с интерпретацией функций АД как моделей операций над программными объектами.

- Функция, в сигнатуре которой `T` появляется лишь справа от стрелки, например `new`, является функцией-конструктором. Она моделирует операцию, создающую экземпляры `T` из экземпляров других типов или вообще не использующую аргументов, например как в случае константного конструктора `new`.
- Такие функции как `item` и `empty`, у которых `T` появляется только слева от стрелки, являются функциями-запросами. Они моделируют операции, которые устанавливают свойства `T`, выраженные в терминах экземпляров других типов (в наших примерах - это `BOOLEAN` и параметр типа `G`).

- Такие функции как `put` и `remove`, у которых `T` появляется с обеих сторон стрелки, являются функциями-командами. Они моделируют операции, которые по существующим экземплярам `T` и, возможно, экземплярам других типов выдают новые экземпляры типа `T`.

Раздел АКСИОМЫ

Поскольку всякое явное определение заставляет выбирать некоторое представление, обратимся к неявным определениям. При этом воздержимся от определения значений функций в спецификации АТД и вместо этого опишем свойства этих значений - все их существенные свойства, но только эти свойства.

Они формулируются в разделе АКСИОМЫ (AXIOMS). Для типа `STACK` он выглядит следующим образом.

- Аксиомы
 - Для всех `x: G`, `s: STACK [G]`,
 - A1** `item (put (s, x)) = x`
 - A2** `remove (put (s, x)) = s`
 - A3** `empty (new)`
 - A4** `not empty (put (s, x))`

Первые две аксиомы выражают основные свойства стеков (последним пришел - первым ушел) LIFO. Чтобы понять их, предположим, что у нас есть стек `s` и экземпляр `x`, и определим `s'` как результат `put(s, x)`, т. е. как результат вталкивания `x` в `s`. Приспособим один из предыдущих рисунков:

Здесь аксиома **A1**, говорит о том, что вершиной `s'` является `x` - последний элемент, который мы втолкнули, а аксиома **A2** объясняет, что при удалении верхнего элемента `s'` мы снова получаем тот

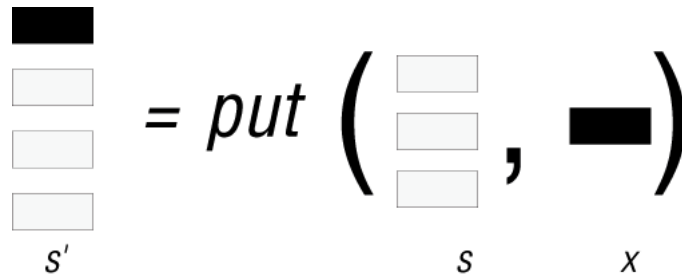


Рис. 6.4: Применение функции put

же стек s , который был до вталкивания x . Эти две аксиомы дают лаконичное описание главного свойства стеков в чисто математических терминах без всякой помощи императивных рассуждений или ссылок на свойства представлений.

Аксиомы A3 и A4 говорят о том, когда стек пуст, а когда - нет: стек, полученный в результате работы конструктора new пустой, а всякий стек, полученный после вталкивания элемента в уже существующий стек (пустой или непустой) не является пустым.

Эти аксиомы, как и остальные, являются предикатами (в смысле логики), выражающими истинность некоторых свойств для всех возможных значений s и x .

Две или три вещи, которые мы знаем о стеках

Спецификации АДТ являются неявными. Имеются два вида "неявности":

- Метод АДТ определяет неявно некоторое множество объектов, задавая применимые к ним функции. Из этого определения никогда не следует, что в нем перечислены все операции;

часто, на пути к представлению, будут добавлены и другие.

- Сами функции также определяются неявно. Вместо явных определений используются аксиомы, задающие свойства этих функций. Здесь тоже ничего не утверждается о полноте: когда вы, в конце концов, дойдете до реализации этих функций, они приобретут дополнительные свойства.

Эта неявность является ключевым аспектом абстрактных типов данных и, как следствие, - их будущих аналогов в построении ОО-ПО - классов. Когда мы определяем абстрактный тип данных или класс, мы всегда сообщаем кое-что об этом типе или классе, просто перечисляя те их свойства, которые знаем, и берем их в качестве определения. При этом никогда не предполагается, что других применимых свойств нет.

Неявность также предполагает открытость определений: всегда можно добавить новые свойства АТД или класса. Основным механизмом для выполнения таких расширений без разрушения уже существующего первоначального определения является наследование.

Частичные функции

Спецификация всякого реалистичного примера, даже такого простого как стеки, необходимо сталкивается с проблемами не всюду определенных операций: некоторые операции применимы не ко всем возможным элементам исходных множеств. Например, это имеет место для функций `remove` и `item`: нельзя удалить элемент из пустого стека, и у пустого стека нет верхнего элемента.

Решение этой проблемы, использованное в приведенной выше спецификации, состоит в том, чтобы определить эти функции как частичные. Функция из исходного множества X в результирующее множество Y является **частичной**, если она определена не для всех элементов X . Функция, не являющаяся частичной, называется **полной**.

В спецификации АД STACK эти идеи использованы для стеков при объявлении `remove` и `item` как частичных функций в разделе ФУНКЦИИ - это указано с помощью перечеркнутых стрелок в их сигнатуре. При этом возникает новая проблема, обсуждаемая в следующем пункте: как задавать области таких функций?

В некоторых случаях функцию `put` тоже желательно описывать как частичную, например, это требуется в таких реализациях как МАССИВ_ВВЕРХ и МАССИВ_ВНИЗ, которые поддерживают выполнение лишь конечного числа подряд идущих операций `put` для каждого заданного стека. Это на самом деле полезное упражнение - приспособить спецификацию STACK к тому, чтобы она описывала ограниченные стеки конечного объема, поскольку в приведенном выше виде она не содержит никаких ограничений на размеры стеков.

Это будет новым применением частичных функций, отражающим ограничения реализации. В отличие от этого, объявление функций `remove` и `item` как частичных отражает абстрактное свойство этих операций, относящееся ко всем реализациям.

Предусловия

Для этого всякая спецификация АД, содержащая частичные функции, должна задавать их области. В этом и состоит роль раздела ПРЕДУСЛОВИЯ (PRECONDITIONS). Для АД STACK этот раздел выглядит так:

- Предусловия (preconditions)
 - `remove (s: STACK [G]) require not empty (s)`
 - `item (s: STACK [G]) require not empty (s)`

В нем у каждой из функций в пункте "требуется" перечисляются условия, которым должны удовлетворять аргументы функции, чтобы входить в ее область.

Булевское выражение, которое определяет область функции, называется предусловием соответствующей частичной функции. В нашем случае предусловия обеих функций `remove` и `item` утверждают, что стек должен быть непустым.

С точки зрения математики предусловие функции f - это характеристическая функция области f . Характеристической функцией подмножества A множества X называется полная функция $ch: X \rightarrow \text{BOOLEAN}$ такая, что $ch(x)$ истинна, если x принадлежит A , и ложна в противном случае.

Полная спецификация

Раздел ПРЕДУСЛОВИЯ (PRECONDITIONS) завершает простую спецификацию абстрактного типа данных STACK. Для удобства ссылок полезно собрать вместе разные компоненты спецификации, приведенные выше. Вот полная спецификация.

Спецификация стеков как АД

- ТИПЫ (TYPES)
 - STACK [G]
- ФУНКЦИИ (FUNCTIONS)
 - put: STACK [G] \times G \rightarrow STACK [G]
 - remove: STACK [G] \nrightarrow STACK [G]
 - item: STACK [G] \nrightarrow G
 - empty: STACK [G] \rightarrow BOOLEAN

- new: STACK [G]

- АКСИОМЫ (AXIOMS)

- Для всех x : G, s : STACK [G],

- A1** item (put (s , x)) = x

- A2** remove (put (s , x)) = s

- A3** empty (new)

- A4** not empty (put (s , x))

- ПРЕДУСЛОВИЯ (PRECONDITIONS)

- remove (s : STACK [G]) require not empty (s)

- item (s : STACK [G]) require not empty (s)

Сила спецификаций АТД проистекает из их способности отражать только существенные свойства структур данных без лишних деталей. Приведенная выше спецификация стеков выражает все, что нужно по существу знать о понятии стека, и не включает ничего, что относилось бы к каким-либо конкретным реализациям стеков. Это вся правда о стеках, и ничего кроме правды.

Такие спецификации задают общую модель вычислений на соответствующих структурах данных. Определенные в спецификации абстрактного типа данных функции позволяют строить сложные выражения, а аксиомы АТД позволяют упрощать такие выражения и получать более простые результаты. Сложное стековое выражение является математическим эквивалентом программы, а процесс упрощения является математическим эквивалентом вычисления или выполнения этой программы.

Вот пример. Рассмотрим для приведенной выше спецификации АТД STACK следующее выражение `stackexp`:

```
item (remove (put (remove (put (put (
remove (put (put (put (new, x1), x2), x3)),
item (remove (put (put (new, x4), x5))))), x6)), x7)))
```

По-видимому, выражение `stackexp` будет проще понять, если мы представим его как последовательность вспомогательных выражений:

```
s1 = new
s2 = put (put (put (s1, x1), x2), x3)
s3 = remove (s2)
s4 = new
s5 = put (put (s4, x4), x5)
s6 = remove (s5)
y1 = item (s6)
s7 = put (s3, y1)
s8 = put (s7, x6)
s9 = remove (s8)
s10 = put (s9, x7)
s11 = remove (s10)
stackexp = item (s11)
```

Какой бы вариант определения вы ни выбрали, по нему несложно восстановить вычисление, математической моделью которого является `stackexp`: создать новый стек; втолкнуть в него элементы `x1`, `x2`, `x3` (в указанном порядке); удалить верхний элемент (`x3`), назвав получившийся стек `s3`; создать другой пустой стек и т. д. Этот процесс графически представлен на рис. [6.5](#).

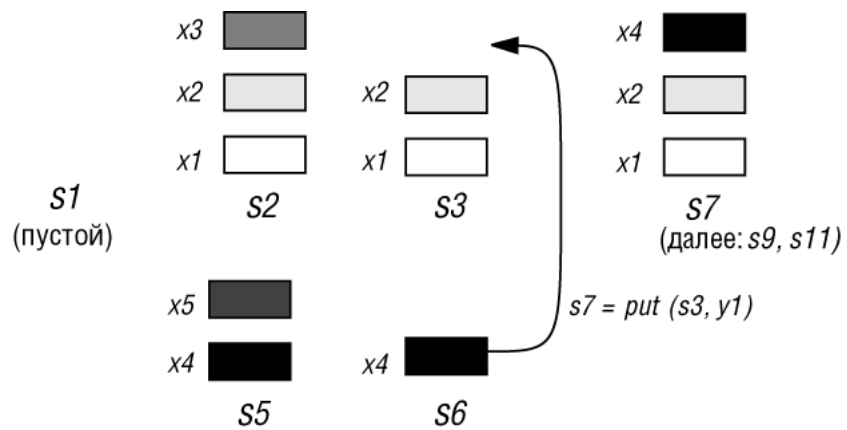


Рис. 6.5: Манипуляции со стеком

6.3 От абстрактных типов данных к классам

Классы

АТД будут служить непосредственной основой разбиения системы на модули. Точнее, ОО-система будет строиться (на уровне анализа, проектирования и реализации) как совокупность взаимодействующих, частично или полностью реализованных АТД. Основное понятие здесь - класс:

Класс - это абстрактный тип данных, снабженный некоторой (возможно частичной) реализацией

Таким образом, чтобы получить класс, мы должны построить АТД и решить, как его реализовывать. АТД - это математическое понятие, а реализация - это его версия, ориентированная на компьютер. Приведенное определение, однако, утверждает, что реализация может быть частичной. Введенные ниже термины позволяют отделить этот случай от полностью реализованного класса:

Полностью реализованный класс называется **эффективным** (effective). Класс, который реализован лишь частично или совсем не реализован, называется **отложенным** (deferred). Всякий класс является либо отложенным, либо эффективным.

Чтобы получить эффективный класс, требуется предусмотреть все детали реализации. Для отложенного класса можно выбрать определенный уровень реализации, но при этом оставить некоторые аспекты реализации незавершенными. В самом крайнем случае при частичной реализации можно вообще отказаться от принятия каких-либо решений о ее уточнении. В этом случае получившийся класс будет полностью отложенным и будет эквивалентен АТД.

Как создавать эффективный класс

Рассмотрим вначале эффективные классы. Что нужно сделать для реализации АТД? Результирующий эффективный класс будет формироваться из элементов трех видов:

E1 Спецификации АТД (множество функций с соответствующими аксиомами и предусловиями, описывающими их свойства).

E2 Выбора представления.

E3 Отображения из множества функций (E1) в представление (E2) в виде множества механизмов (или компонентов (features)), каждый из которых реализует одну из функций в терминах представления и при этом удовлетворяет аксиомам и предусловиям. Многие из этих компонентов будут методами - обычными процедурами, но некоторые могут появляться в качестве полей данных или "атрибутов"(это будет показано в следующих лекциях).

Роль отложенных классов

В определении эффективного класса должна присутствовать полная информация о реализации (пункты E2 и E3). Если она хоть в чем-то неполна, то класс является отложенным.

Чем более "отложенным" является класс, тем он ближе к АТД, одетому в некоторую синтаксическую одежду, которая скорее поможет завоевать признание разработчиков ПО, чем математиков. Отложенные классы особенно полезны при анализе и проектировании:

- При ОО-проектировании многие аспекты реализации будут опущены, проектирование должно сосредотачиваться на архитектурных свойствах высокого уровня - на том, какую функциональность обеспечивает каждый модуль системы, а не на том, как он это делает.

- При постепенном продвижении к полной реализации будут добавляться все новые и новые ее свойства до тех пор, пока не будет получен эффективный класс.

Но на этом роль отложенных классов не завершается, даже в полностью реализованной системе можно часто обнаружить много таких классов. Кое-что следует из только что перечисленных применений: когда из отложенных классов получаются эффективные, то появляется желание сохранить их в качестве предков (в смысле наследования) эффективных классов как живую память о процессе анализа и проектирования.

Очень часто при разработке ПО с помощью не ОО-подходов система в окончательном виде не содержит никаких записей о тех значительных усилиях, которые были затрачены на ее получение. Для тех, кто вынужден будет обслуживать такую систему - расширять, переносить, отлаживать - понять ее без этих записей будет так же трудно, как трудно геологу понять видимый ландшафт, не имея доступа к осадочным слоям. Один из лучших способов обеспечить необходимую для сопровождения системы информацию - это сохранить отложенные классы в ее окончательной форме.

У отложенных классов имеется также применение, полностью связанное с реализацией. Они служат для классификации групп связанных типов объектов, предоставляют некоторые наиболее важные многократно используемые модули высокого уровня, фиксируют общие свойства поведения многих вариантов и играют ключевую роль (вместе с полиморфизмом и динамическим связыванием) в обеспечении децентрализации и расширяемости программной архитектуры.

Абстрактные типы данных и скрытие информации

Особенно интересным следствием ОО-политики, в которой модули основаны на реализациях АТД (классах), является то, что она дает ясный ответ на вопрос, который остался нерешенным при обсуждении скрытия информации: как нам следует разделять общедоступные и скрытые свойства модуля - видимую и невидимую части айсберга? Если модуль является классом, полученным из АТД, то ответ

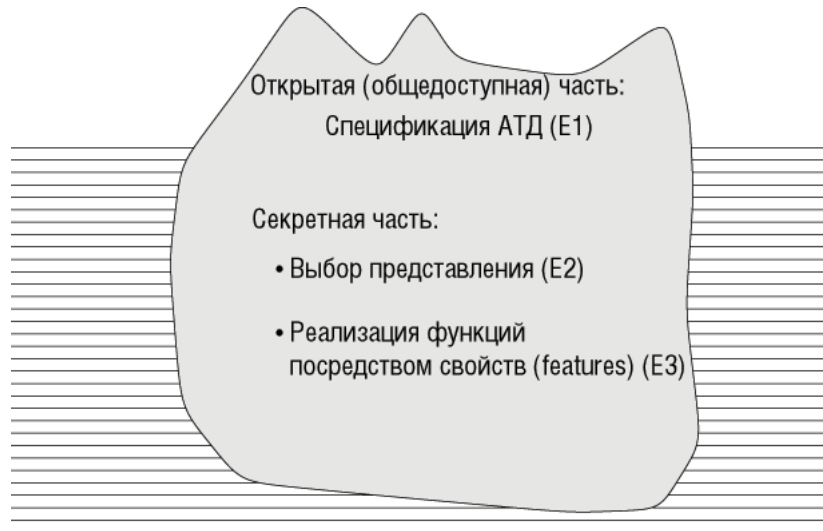


Рис. 6.6: АДТ вид модуля при скрывтии информации

ясен. Из трех частей, вовлеченных в эту эволюцию, Е1- спецификация АТД, является открытой, а Е2 и Е3 - выбор представления и реализация функций АТД в терминах этого представления - должны быть закрытыми (секретными). Когда мы начнем строить классы, то столкнемся еще с четвертой частью, также секретной, - вспомогательными свойствами, необходимыми только для внутренних нужд этих программ.

Таким образом, использование абстрактных типов данных в качестве источника модулей дает нам практичное, однозначное указание для применения скрывания информации в наших проектах.

Конструирование объектно-ориентированного ПО

Мы уже давали определение конструирования ОО-ПО: будучи весьма общим, оно представляет метод следующим образом: "основывать архитектуру всякой программной системы на модулях, полученных из типов объектов, с которыми оперирует система". Придерживаясь рамок этого определения, мы можем дополнить его теперь более техническим определением:

Конструирование ОО-ПО - это построение программной системы как структурированной совокупности реализаций (возможно частичных) абстрактных типов данных.

Это определение будет нашим рабочим определением. Все его компоненты являются важными:

- В основе лежит понятие абстрактного типа данных.
- Для конструирования программ нам нужны не сами по себе АТД (как математическое понятие), а реализации АТД - программистское понятие.
- При этом эти реализации не обязаны быть полными, оговорка "возможно частичные" позволяет использовать и отложенные классы, включая, как крайний случай, полностью отложенный класс без какой-либо реализации.

- Система представляет собой совокупность классов без выделения какого-либо главного или ответственного класса или головной программы.
- Эта совокупность является структурированной благодаря двум отношениям между классами: "быть клиентом" и наследованию.

6.4 Ключевые концепции

- Теория абстрактных типов данных (АТД) примиряет необходимость в точности и полноте спецификаций с желанием избежать лишних деталей в спецификации.
- Спецификация абстрактного типа данных является формальным математическим описанием, а не текстом программы. Она аппликативна, т.е. не включает в явном виде изменений.
- АТД может быть родовым, и он задается функциями, аксиомами и предусловиями. Аксиомы и предусловия выражают семантику данного типа и важны для полного и однозначного его описания.
- Частичные функции образуют удобную математическую модель для описания не всюду определенных операций. У каждой частичной функции имеется предусловие, задающее условие, при котором она будет выдавать результат для заданного конкретного аргумента.
- ОО-система - это совокупность классов. Каждый класс основан на некотором абстрактном типе данных и задает частичную или полную реализацию этого АТД.
- Класс является эффективным, если он полностью реализован, в противном случае он называется отложенным.

- Классы должны разрабатываться в наиболее общем виде, допускающем повторное использование; процесс их объединения в систему часто идет снизу-вверх.
- Абстрактные типы данных являются скорее неявными, чем явными описаниями. Эта неявность, которая также означает открытость, переносится на весь ОО-метод.
- Не существует формального определения интуитивно ясного понятия "полноты" спецификации абстрактного типа данных. Строго определяемое понятие достаточной полноты как правило обеспечивает удовлетворительный ответ. Хотя не существует метода, устанавливающего достаточную полноту произвольной спецификации, часто удается ее доказать для конкретных спецификаций; приведенное в этой лекции доказательство достаточной полноты для спецификации стеков может служить образцом и для других случаев.

Контрольные вопросы

1. Какими критериями следует руководствоваться при описании типов объектов?

описания должны быть точными и недвусмысленными

описания должны быть полностью специфицированы

описания могут быть неточными

описания не должны быть излишне специфицированы

2. Спецификация АТД включает разделы

предусловий
функций
типов
аксиом

3. В описании АТД функции разделяются на следующие категории

команды
ответы
запросы
конструкторы

4. При описании АТД можно?

указывать родовые параметры
задавать родительский класс
задавать реализацию функций
задавать представление данных

5. Распределитель - это?

специальный вид контейнера данных, в котором клиент управляет размещением элементов
специальный вид контейнера данных, в котором размещение элементов определяется внутренней дисциплиной
системный администратор, распределяющий задания между исполнителями

6. В основе спецификации должно лежать

набор интерфейсов для работы пользователя с объектами

конкретное представление данных

набор функций, заданных их сигнатурой и их формальными свойствами

реализация функций, выполняемых над данными

7. Какие определения, связанные с понятием класс, являются корректными?

класс называется отложенным, если его реализация задана лишь частично

класс называется абстрактным, если его реализация задана лишь частично

класс - это абстрактный тип данных, снабженный некоторой (возможно частичной) реализацией

класс называется эффективным, если при его реализации используются эффективные алгоритмы

8. Стеки могут быть представлены?

односвязным списком

набором простых переменных

массивом, растущим вверх

массивом, растущим вниз

9. Спецификация

задает множество правильно построенных выражений

может быть противоречивой

выражения, строящиеся на основе АТД, могут быть некорректными

может быть неполной

10. АД, классы и скрытие информации

все, что входит в АД, никогда не скрывается и представляет интерфейс класса

реализация данных обычно скрывается

представление данных никогда не скрывается

аксиомы АД всегда скрываются

11. Чтобы АД превратить в эффективный класс необходимо?

в соответствии с выбранным представлением задать реализацию компонентов, удовлетворяющую аксиомам

отобразить множество функций АД в компоненты класса

провести формальное доказательство соответствия реализации и аксиом АД

выбрать представление данных

12. Отметьте истинные высказывания

система представляет собой совокупность классов, без выделения какого-либо главного или ответственного класса или головной программы

для получения эффективного класса требуется предусмотреть все детали реализации

описание АД основано на описании функций, применяемых к данным

конструирование ОО ПО – это построение программной системы как структурированной совокупности реализаций (возможно частичных) абстрактных типов данных

13. Отметьте истинные высказывания

задание реализаций функций соответствует этапу реализации системы

задание представлений соответствует этапу проектирования системы

для конструирования программ нужны не сами по себе АД (как математическое понятие), а реализации АД – программистское понятие

задание спецификаций соответствует этапу анализа системы

14. Отметьте истинные высказывания

АД не может быть родовым

ОО-система – это совокупность классов. Каждый класс основан на некотором абстрактном типе данных и задает частичную или полную реализацию этого АД

класс является эффективным, если он полностью реализован, в противном случае он называется отложенным

спецификация абстрактного типа данных является текстом программы

Набрано баллов

7 Статические структуры

7.1 Классы, а не объекты - предмет обсуждения

Какова центральная концепция объектной технологии? Необходимо дважды подумать, прежде чем ответить "объект". Объекты полезны, но в них нет ничего нового.

С тех пор, как структуры используются в Cobol, с тех пор, как в Pascal существуют записи, с тех пор как программист написал на С первое определение структуры, человечество располагает объектами.

Объекты важны при описании выполнения ОО-систем. Но базовым понятием объектной технологии является класс. Обратимся вновь к его определению. (Детальное обсуждение объектов содержится в следующей лекции.)

Определение класса:

Класс - это абстрактный тип данных, поставляемый с возможно частичной реализацией.

Абстрактные типы данных (АТД) являются математическим понятием, пригодным на этапе подготовки спецификации - в процессе анализа. Понятие класса, предусматривая частичную или полную реализацию, обеспечивает необходимую связь с разработкой ПО на этапах проектирования и программирования. Напомним, класс называется эффективным, если его реализация полна, и отложенным - при частичной реализации.

Аналогично АТД, класс это тип, описывающий множество возможных структур данных, назы-

ваемых экземплярами (instances) класса. Экземпляры АТД являются абстракциями - элементами математического множества. Экземпляр класса конкретен - это структура данных, размещаемая в памяти компьютера и обрабатываемая программой.

Например, если определить класс STACK, взяв за основу спецификацию АТД из предыдущей лекции и добавив информацию, необходимую для адекватного представления, то экземплярами класса будут структуры данных - конкретные стеки. Другим примером является класс POINT, моделирующий точку на плоскости. Если для представления точки выбрана декартова система координат, то каждый экземпляр POINT представляет собой запись с полями x, y - абсциссой точки и ее ординатой.

Термин "объект" появляется как побочный продукт определения "класса". Объект это просто экземпляр некоторого класса.

Программные тексты, описывающие создаваемую систему, содержат определения классов. Объекты создаются только в процессе выполнения программ.

Настоящая лекция посвящена основным приемам создания программных элементов и объединения их в системы, именно поэтому в центре внимания - классы. В следующей лекции будут рассмотрены структуры периода выполнения, порождаемые ОО-системой, что потребует изучения некоторых особенностей реализации и более детального рассмотрения природы объектов.

7.2 Роль классов

Затратив немного времени на устранение абсурдных, но распространенных и вредных заблуждений, можно вернуться к рассмотрению центральных свойств классов и выяснить, в частности, почему они столь важны в объектной технологии.

Метакласс - это класс, экземпляры которого сами являются классами

Для понимания ОО-подхода необходимо ясно представлять, что классы выполняют две функции, которые до появления ОО-технологий всегда были разделены. Класс одновременно является модулем и типом.

Модули и типы

Средства, используемые при разработке ПО, - языки программирования, проектирования, спецификаций, графические системы обозначений для анализа, - всегда включали в себя как возможность применения модулей, так и систему типов.

Модули - это структурные единицы, из которых состоит программа. Различаются виды модулей, такие как подпрограммы и пакеты. Независимо от конкретного выбора той или иной модульной структуры, модуль всегда рассматривается как синтаксическая концепция. Отсюда следует, что разбиение на модули влияет лишь на форму записи исходных текстов программ, но не определяет их функциональность. В самом деле, принципиально можно написать программу Ada в виде единственного пакета, или программу Pascal как единую основную программу. Безусловно, такой подход не рекомендуется, и любой компетентный программист будет использовать модульные возможности языка для деления программы на обозримые и управляемые части. Но если взять существующую программу, например на Паскале, то всегда можно собрать воедино все модули и получить работоспособную программу с эквивалентной семантикой. (Присутствие рекурсивных подпрограмм делает этот процесс менее тривиальным, но не оказывает принципиального влияния на данную дискуссию.) Таким образом, деление на модули диктуется принципами управления проектами, а не внутренней необходимостью.

Концепция типов на первый взгляд совершенно иная. Тип является статическим описанием вполне определенных динамических объектов - элементов данных, которые обрабатываются во время выполнения программной системы. Набор типов обычно содержит predetermined типы, такие как

INTEGER или CHARACTER, а также пользовательские типы: записи (структуры), указатели, множества (в Pascal), массивы и другие. Понятие типа является семантической концепцией, и каждый тип непосредственно влияет на выполнение программной системы, так как описывает форму объектов, которые система создает и которыми она манипулирует.

Класс как модуль и как тип

В не ОО-подходах концепции модуля и типа существуют независимо друг от друга. Наиболее замечательным свойством класса является одновременное использование обеих концепций в рамках единой лингвистической конструкции. Класс является модулем или единицей программной декомпозиции, но одновременно класс это тип (или шаблон типа в тех случаях, когда поддерживается параметризация).

Мощь ОО-метода, во многом, следствие этого отождествления. Наследование, в частности, может быть полностью понято, только при рассмотрении его как модульного расширения, так и, одновременно, уточнения специализации типа.

Как практически соединить две столь различные на первый взгляд концепции? Последующая дискуссия и примеры позволят ответить на этот вопрос.

7.3 Унифицированная система типов

Важным аспектом ОО-подхода является простота и универсальность системы типов, которая строится на основе фундаментального принципа.

Объектный принцип:

Каждый объект является экземпляром некоторого класса

Объектный принцип будет распространяться не только на составные объекты, определяемые разработчиками (такие как структуры данных, содержащие несколько полей), но и на базовые объекты - целые и действительные числа, булевы значения и символы, которые будут рассматриваться как экземпляры предопределенных библиотечных классов (INTEGER, REAL, DOUBLE, BOOLEAN, CHARACTER).

На первый взгляд подобное стремление превратить любое сколь угодно простое значение в экземпляр некоторого класса может показаться преувеличенным и даже экстравагантным. В конце концов, математики и инженеры в течение многих лет успешно используют целые и действительные числа, не подозревая о том, что они работают с экземплярами классов. Однако настойчивое требование к унификации вполне окупается по ряду причин.

- Всегда желательно иметь простую и универсальную схему, нежели множество частных случаев. Предлагаемая система типов полностью опирается на понятие класса.
- Описание базовых типов как абстрактных структур данных и далее как классов является простым и естественным. Нетрудно представить, например, определение класса INTEGER с функциональностью включающей арифметические операции, такие как "+ операции сравнения, такие как "=" и ассоциированные свойства, следующие из соответствующих математических аксиом.
- Определение базовых типов как классов позволяет использовать все возможности ОО, главным образом наследование и родовые средства. Если базовые типы не будут классами, то придется вводить ряд ограничений и рассматривать частные случаи.

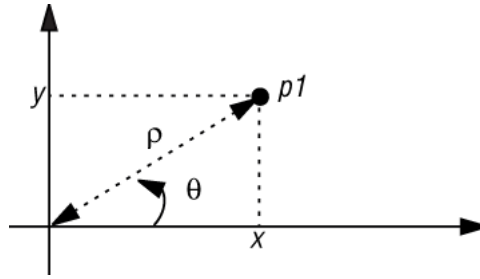


Рис. 7.1: Точка и ее координаты

7.4 Простой класс

Что представляет собой класс можно выяснить, изучая простой, но типичный пример, который демонстрирует фундаментальные свойства, применимые практически ко всем классам.

Компоненты

Пример использует представление точки в двумерной графической системе:

Для определения типа POINT как абстрактного типа данных потребуется четыре функции-запроса: x, y, ρ, θ . (В текстах подпрограмм для двух последних функций будут использоваться имена `rho` и `theta`). Функция x возвращает абсциссу точки (горизонтальную координату), y - ординату (вертикальную координату), ρ - расстояние от начала координат, θ - полярный угол, отсчитываемый от горизонтальной оси. Значения x и y являются декартовыми, а ρ и θ - полярными координатами точки. Другой полезной функцией является `distance`, возвращающая расстояние между двумя точками.

Далее спецификация АТД будет содержать такие команды, как `translate` (перемещение точки на заданное расстояние по горизонтали и вертикали), `rotate` (поворот на определенный угол вокруг

начала координат) и scale (уменьшение или увеличение расстояния до начала координат в заданное число раз).

7.5 Ключевые концепции

- Фундаментальная концепция объектной технологии основана на понятии класса. Класс это абстрактный тип данных, частично или полностью реализованный.
- Класс может иметь экземпляры, называемые объектами.
- Нельзя путать объекты (динамические элементы) с классами (статическим описанием свойств, общих для множества объектов времени выполнения).
- При последовательном подходе к объектной технологии каждый объект является экземпляром класса.
- Класс одновременно служит модулем и типом. Оригинальность и мощь ОО-модели следует частично из интеграции этих понятий.
- Класс характеризуется компонентами, включая атрибуты, представляющие поля в экземплярах класса, и подпрограммы, представляющие вычисления с участием данных экземпляров. Подпрограмма может быть функцией возвращающей результат или процедурой, если результат не возвращается.
- Базовым механизмом ОО-вычислений является вызов компонентов (обращение к компонентам) класса. Вызов компонента применяет компонент к экземпляру класса (возможно с аргументами).

- При вызове именованных компонентов используется точечная нотация, а при вызове компонент-операций - инфиксная или префиксная нотация.
- Каждая операция относительна к "текущему экземпляру" класса.
- Для клиентов класса (других классов, которые используют его компоненты) атрибут ничем не отличается от функции без аргументов, в соответствии с принципом унифицированного доступа.
- Исполняемый ансамбль классов называется системой. Система содержит корневой класс и все классы, которые необходимы корневому прямо или косвенно через клиентские отношения или наследование. Выполнение системы сводится к созданию экземпляра корневого класса и вызову процедуры создания для данного экземпляра.
- Системы имеют децентрализованную архитектуру. Порядок действий несущественен для разработки.
- Механизм скрытия информации требует гибкости. Наряду с неограниченным доступом и полным скрытием может потребоваться экспорт только для части клиентов. Атрибуты могут быть доступны только для чтения, для чтения и ограниченной модификации и в режиме полного доступа.
- Экспорт атрибута означает доступ к нему только для чтения. Модификация требует вызова соответствующей экспортированной процедуры.
- Модульный стиль ОО-разработок требует большого числа небольших подпрограмм. Потенциальная опасность снижения производительности может быть достигнута путем встраивания

этих подпрограмм оптимизирующим компилятором. Ответственность за поиск таких фрагментов следует возложить на компилятор, а не на разработчиков.

Контрольные вопросы

1. Класс - это?

программный текст
статическая структура
динамическая структура, создаваемая в момент выполнения
АТД с заданной реализацией (возможно частичной)

2. Метакласс - это?

класс, объекты которого сами являются классами
большой класс
синоним понятия абстрактный класс
класс, экземпляры которого сами являются классами

3. Объект - это?

программный текст
статическая структура
экземпляр класса

динамическая структура, создаваемая в момент выполнения

4. Метод класса - это?

компонент класса

поле класса

часть структуры данных, представляющей объект

процедура или функция

5. Модуль – это?

класс в ОО-подходе

структурный архитектурный элемент, из набора которых строятся программы

семантическое понятие

синтаксическое понятие

6. Функция всегда

имеет аргументы

возвращает значение атрибута

возвращает результат

требует вычислений

7. Текущий объект – это?

случайно выбираемый объект из множества объектов, созданных в процессе вычисления
объект, создаваемый в момент "большого взрыва"

цель неквалифицированного вызова в объектных вычислениях

цель вызова в объектных вычислениях

8. Атрибут класса - это?

метод класса

компонент класса

часть структуры данных, представляющей объект

поле класса

9. Тип - это?

синтаксическое понятие

множество объектов, над которыми определено заданное множество операций

класс в ОО-подходе

семантическое понятие

10. Создание законченной программной системы предполагает

выбор корневой процедуры создания

выбор корневого класса

замыкание системы по отношению к корневому классу

создание множества классов

11. Клиенту класса должны быть доступны

только компоненты, экспортируемые данному клиенту

все компоненты класса-поставщика

все компоненты, за исключением тех, что экспортируются наследникам класса

все компоненты, экспортируемые любому из клиентов

12. Согласно принципу унифицированного доступа клиент не может отличить

вызов атрибута от вызова метода

вызов атрибута от вызова функции

вызов функции от вызова процедуры

вызов атрибута от вызова функции без аргументов

13. Отметьте истинные высказывания

для каждого вызова должна быть явно указана его цель

атрибуты доступны только для чтения

сущность и объект – это синонимичные понятия

селективный экспорт позволяет группе концептуально связанных классов обеспечить друг другу доступ ко всем своим компонентам, скрыв их от остального мира

Набрано баллов

8 Динамические структуры: объекты

8.1 Объекты

В процессе выполнения ОО-система создает некоторое число объектов. Организация этих объектов и отношения между ними определяют конструкцию времени выполнения. Рассмотрим свойства объектов.

Что такое объект?

Определение: объект

Объект - это экземпляр некоторого класса

Во время выполнения программная система, содержащая класс *C*, может в разных точках, используя процедуры создания или клонирования, создавать экземпляры *C*, - структуры данных, соответствующие образцу, заданному классом *C*. Например, экземпляр класса POINT представляет собой структуру данных, состоящую из двух полей, соответствующих атрибутам *x* и *y* класса. Экземпляры всех возможных классов составляют множество объектов системы.

Это официальное определение в мире ОО-ПО. Но в повседневном языке термин "объект" имеет гораздо более широкий смысл. Любая программная система связана с определенной внешней системой, которая может содержать "объекты": точки, линии, поверхности и тела в графической системе;

сотрудников и их оклады в системе расчета заработной платы и т.д. В таких ситуациях, как правило, реальным объектам соответствуют программные объекты. Примером может служить класс EMPLOYEE в системе расчета зарплаты, экземпляры которого являются компьютерными моделями сотрудников.

Но не стоит переоценивать "реальность" слова "объект". В науке и технике существует большой риск в заимствовании слов естественного языка и придания им специального смысла. Термин "объект" настолько перегружен повседневным смыслом, что техническое его использование может стать источником недоразумений. В частности:

- Не все классы соответствуют типам проблемной области. Многие классы, введенные в интересах проектирования и реализации, не имеют двойников в моделируемой системе. Именно эти классы на практике могут иметь наибольшее значение и именно их труднее всего спроектировать.
- Некоторые концепции проблемной области естественно приводят к классам, хотя в проблемной области не существует реальных объектов, которые можно было бы поставить в соответствие экземплярам этих классов. Примерами могут быть класс STATE, описывающий состояние системы, или класс COMMAND.

Когда слово "объект" используется в этом пособии, то из контекста ясно, в общем или техническом смысле используется этот термин. В тех случаях, когда эту разницу необходимо подчеркнуть, используется уточнение - программный объект или внешний объект.

Базовая форма

Пусть *O* - объект. По определению он является экземпляром некоторого класса. Точнее, он является прямым экземпляром (direct instance) только одного класса, например *C*.

С учетом наследования О будет тогда косвенным экземпляром других классов, - предков С. Это тема дальнейшего обсуждения; в данной дискуссии достаточно понятия прямого экземпляра. Везде, где не может возникнуть недоразумений, слово "прямой" будет опущено.

Класс С называется порождающим классом (generating class) или просто генератором (generator) объекта О. Заметьте, С- программный текст, а О - структура данных времени выполнения, появляющаяся в результате работы рассмотренных ниже механизмов создания объектов.

Часть компонентов С является атрибутами. Эти атрибуты полностью определяют форму объекта, представляющего собой просто набор полей, по одному на каждый атрибут.

Рассмотрим класс POINT. Исходный текст имеет вид:

```
class Point:
    def __init__(self, x=0.0, y=0.0):
        self.x = x
        self.y = y
```

... Объявления подпрограмм ...

Данный класс имеет два атрибута x и y типа Float, следовательно, его экземпляр - это объект с двумя полями, содержащими значения этого типа:

Простые поля

Оба атрибута класса POINT относятся к типу Float. Следовательно, соответствующие поля прямого экземпляра POINT содержат действительные числа.

Это пример полей, соответствующих атрибутам одного из "базовых" типов. Формально эти типы определены как классы, а их экземпляры принимают значения из предопределенных диапазонов. К базовым (предопределенным, встроенным) типам конкретного языка программирования.

Ссылки

Чаще всего нам необходимы объекты с полями, представляющими другие объекты. Можно ввести понятие подобъекта. Можно в качестве решения предложить в каждом экземпляре объекта дублировать информацию в виде подобъекта. Причины неприемлемости такого решения:

- Расходуется дополнительная память. Можно привести в качестве более характерного примера совокупность объектов, представляющих людей. Каждый объект в качестве подобъекта содержит информацию о стране гражданства. Очевидно, что численность населения намного превышает число стран.
- Более важно, что такая техника не обеспечивает разделения информации. Вполне естественно желание, чтобы внесение изменений в подобъект повлекло за собой автоматическое обновление этой информации для всех объектов.

Лучшим является решение использование ссылки в качестве подобъекта.

Определение: ссылка

Ссылка это значение времени выполнения. Она может быть пустой (None) или присоединенной. Присоединенная ссылка однозначно идентифицирует объект (присоединена к конкретному объекту).

Концепция ссылки должна, безусловно, иметь аналог при реализации. Программирование на уровне машинного кода использует адресацию, многие языки программирования содержат понятие указателя. Понятие ссылки является более абстрактным. Хотя ссылка, в конечном итоге, может быть представлена адресом, не следует из этого исходить. Ссылка может содержать адрес наряду с другой информацией.

Идентичность объектов

Понятие ссылки приводит к концепции идентичности объектов. Каждый объект, созданный в процессе выполнения ОО-системы, уникален и идентифицируется независимо от значений его полей. Возможны две ситуации:

I1 Два различных объекта могут иметь абсолютно одинаковые поля

I2 Напротив, поля данного объекта могут изменяться в процессе выполнения системы, но это не влияет на идентификацию объекта.

Ссылка на себя

Ничто не препятствует объекту O1 в определенный момент выполнения системы содержать ссылку, присоединенную к самому O1. Такая ссылка на себя может быть косвенной.

Такие циклы в динамических структурах возможны, только если клиентские отношения между соответствующими классами также содержат прямые или косвенные циклы.

8.2 Объекты как средство моделирования

Рассмотренные приемы позволяют продвинуться в понимании возможностей ОО-подхода как средства моделирования. Важно, в частности, прояснить два аспекта: рассмотреть различные миры, связанные с разработкой ПО и отношения между ПО и внешней реальностью.

Четыре мира программной разработки

Из предшествующей дискуссии следует, что когда мы говорим об ОО-разработке, следует различать четыре отдельных мира:

- Моделируемую систему, - внешнюю по отношению к программной системе, описываемую типами объектов и их абстрактными отношениями.
- Частную конкретизацию внешней системы, состоящую из объектов с фиксированными отношениями.
- Программную систему, состоящую из классов, связанных ОО-отношениями ("быть клиентом" "быть наследником").
- Объектную структуру в том виде, в котором она существует в процессе выполнения программной системы, то есть множество программных объектов, связанных ссылками.

Соотношения между этими мирами представлены на рис.8.1.

И на программном и на внешнем уровне (нижняя и верхняя части рисунка) важно разграничить общие понятия и их конкретные реализации (классы и абстрактные отношения слева, объекты и отношения экземпляров справа). Данный момент уже обсуждался в дискуссии о сравнительной роли классов и объектов в предыдущей лекции. Применительно к отношениям необходимо отличать абстрактные отношения `loved_one` от множества связей `loved_one`, существующих между элементами конкретного множества объектов.

Это различие невыразимо ни в стандартных математических определениях понятия "отношение" ни в программистской терминологии, например в теории реляционных баз данных. Если ограничиться бинарными отношениями, то и в математике и в теории баз данных отношение определяется как множество пар в форме $\langle x, y \rangle$, где x и y являются элементами заданных множеств T_X и T_Y . В терминах программирования все x относятся к типу T_X , а все y - к типу T_Y . Будучи пригодными для математиков, эти определения не подходят для целей моделирования, поскольку не позволяют различать абстрактные отношения и отношения конкретных экземпляров. При моделировании си-

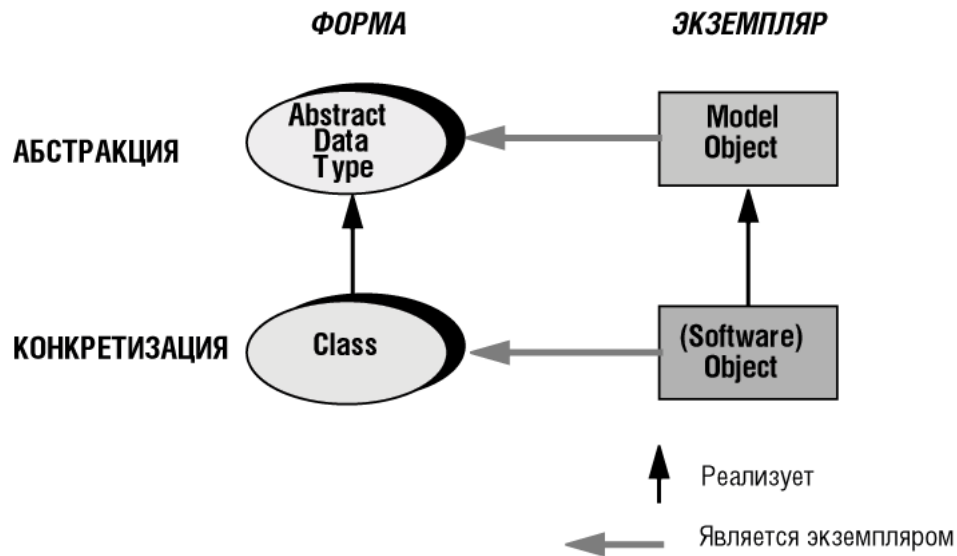


Рис. 8.1: Формы и их экземпляры

стемы отношение "любит" имеет свои общие и абстрактные свойства, совершенно не зависящие от записи того, кто кого любит в конкретной группе людей в некоторый момент времени.

8.3 Работа с объектами и ссылками

Вернемся к более приземленным проблемам и рассмотрим, как программные системы работают с объектами, как создают и используют гибкие структуры данных.

Динамическое создание и повторное связывание

Что не было показано при описании структуры объектов периода выполнения, так это в высшей степени динамичная природа настоящей ОО-модели. Статическая и ориентированная на стеки политика управления объектами характерна для языков уровня Fortran и Pascal соответственно. Противоположной является политика в настоящем ОО-окружении, позволяющая создавать объекты в период выполнения, когда в них возникает потребность. Какому образцу (типу) соответствуют создаваемые объекты, как правило, невозможно предсказать при статической проверке программного текста.

В начальном состоянии, как описано в предыдущей лекции, создается единственный корневой объект. Затем система повторно выполняет операции создания новых объектов, связывает изначально пустые ссылки с этими объектами, делает ранее присоединенные ссылки пустыми или присоединяет их к другим объектам. Динамическая и непредсказуемая природа этих операций обеспечивает гибкий подход и позволяет поддерживать динамические структуры данных, необходимые для реализации сложных алгоритмов и моделирования быстро меняющихся свойств внешних систем.

Агрегирование

В некоторых областях информатики - базах данных, моделировании, анализе требований - разработана классификация отношений, имеющих место между элементами моделируемой системы. В этих контекстах часто встречается отношение "агрегирования"(aggregation), выражающее тот факт, что каждый объект некоторого типа является агрегатом - содержит в своем составе ноль или более объектов, каждый из которых имеет свой собственный тип. Например: автомобиль является агрегатом, содержащим мотор, кузов и другие детали.

8.4 Ключевые концепции

- ОО-вычисления характеризуются высоко динамичной структурой времени выполнения, в которой объекты создаются только по запросу.
- Некоторые объекты, используемые ПО, являются моделями внешних объектов (обычно косвенными). Другие объекты служат только для целей проектирования и реализации.
- Объект состоит из ряда значений, называемых полями. Каждое поле соответствует атрибуту генератора объекта (класса, прямым экземпляром которого является объект).
- Значение, в частности поле объекта, является объектом или ссылкой.
- Ссылка может быть пустой (void) или присоединенной к объекту. Проверка условия $x = \text{Void}$ позволяет определить текущее состояние ссылки. Корректное выполнение вызова $x.f(\dots)$ возможно, если x не пустая ссылка.
- Объектные структуры могут содержать циклические цепочки ссылок.

- Для сущностей ссылочного типа присваивание ($:=$) и проверка эквивалентности ($=$) являются ссылочными операциями. Для сущностей развернутых типов используется семантика значений. Соответствующая семантика распространяется и на смешанные операнды.
- В результате ссылочных операций появляются динамические псевдонимы. Они затрудняют получение выводов о работе системы при анализе ее текста. На практике большинство нетривиальных действий со ссылками можно инкапсулировать в библиотечные классы.

Контрольные вопросы

1. К какому понятию относятся определения: "экземпляр класса, набор полей, понятие периода исполнения"?

сущности
класса
ссылки
объекта

2. К какому понятию относятся высказывания: "понятие периода исполнения, однозначно идентифицирует объект, может быть пустой"?

сущности
ссылки
объекта

класса

3. К какому понятию относятся высказывания: "статическое понятие, применимое к тексту, идентификатор, имеет значение в период выполнения"?

класса

объекта

ссылки

сущности

4. Динамическая структура объектов, создаваемая в период выполнения

имеет, как правило, сложность, значительно превосходящую сложность статической структуры классов

имеет, как правило, сложность более простую, чем сложность статической структуры классов

имеет ту же сложность, что и статическая структура классов

не имеет ничего общего со статической структурой классов

5. На основе изучения статического текста программы нельзя точно предсказать

когда будут созданы новые объекты

какие ссылки будут присоединены к объектам

какие объекты будут созданы

когда ранее присоединенные ссылки становятся пустыми

6. Прямые или косвенные ссылки объектов на самих себя в динамических структурах

возможны только прямые циклы

невозможны

возможны, только если клиентские отношения между соответствующими классами также содержат прямые или косвенные циклы

возможны, только косвенные циклы

7. Эффект инструкции создания вида *create x* состоит в?

присоединения ссылки – значения *x* к новому объекту

создании новой ссылки

инициализации полей создаваемого объекта значениями по умолчанию

создании нового объекта

8. При ссылочном присваивании (левая и правая части присваивания являются ссылками)

ссылка левой части разрывает связь с объектом, к которому она была присоединена, и присоединяется к объекту, присоединенному к ссылке правой части

создается новый объект – копия объекта, присоединенного к ссылке правой части, и ссылка левой части присоединяется к этой копии

ссылка левой части присоединяется к объекту, присоединенному к ссылке правой части, только если ссылка левой части имела значение *void*

ссылка левой части сохраняет связь с объектом, к которому она была присоединена, и дополнительно связывается с объектом, присоединенным к ссылке правой части

9. При клонировании

при поверхностном клонировании копируется только открытая часть объекта

изменяется состояние существующего объекта
при глубоком клонировании может создаваться несколько новых объектов
создается копия существующего объекта

10. Присоединение y к x

использует y как цель

использует x как цель

имеет место при присваивании $y:=x$

имеет место при вызове подпрограммы, когда формальный аргумент x заменяется фактическим y

11. Под динамическими псевдонимами понимается?

существование двух или более сущностей с одинаковыми именами

существование двух или более объектов с одинаковыми значениями полей

существование двух или более сущностей, присоединенных к одному и тому же объекту

существование двух или более объектов, связанных с одной и той же сущностью

12. Развернутый тип позволяет?

обеспечить лучшее моделирование реального мира

улучшить эффективность

экономить память

поддерживать базовые типы

13. Отметьте истинные высказывания

определение в классе процедур создания автоматически запрещает использование базовой инструкции создания

каждое поле объекта соответствует атрибуту генерирующего класса

в момент выполнения системы не могут существовать пустые ссылки (не связанные с объектами)

базовые типы (BOOLEAN, CHARACTER, INTEGER, REAL, DOUBLE) определены как развернутые классы

14. Отметьте истинные высказывания

вызов компонента для пустой ссылки приводит к ошибке в период выполнения

поле объекта является объектом или ссылкой

пустые ссылки не считаются эквивалентными

развернутый клиент не может быть своим клиентом

15. Отметьте истинные высказывания

при порождающем вызове создается новый объект с инициализацией полей, отличной от значений по умолчанию

в результате ссылочного присваивания всегда появляется висячий объект (доступный для сборщика мусора).

объектные структуры могут содержать циклические цепочки ссылок

сущность развернутого типа не может быть пустой ссылкой

Модуль 3

9 Проектирование по контракту: построение надежного ПО

9.1 Базисные механизмы надежности

Технические приемы, введенные в предыдущих лекциях, были направлены на создание надежного ПО. Дадим их краткий обзор - было бы бесполезно рассматривать более продвинутые концепции до приведения в порядок основных механизмов надежности. Первым и определяющим свойством объектной технологии является почти навязываемая структура программной системы - простая, модульная, расширяемая, - проще гарантирующая надежность, чем в случае "кривых" структур, возникающих при применении ранних методов разработки. В частности, усилия по ограничению межмодульного взаимодействия, сведения его к минимуму, были в центре дискуссии о модульности. Результатом стал запрет общих рисков, снижающих надежность, - отказ от глобальных переменных, механизм ограниченного взаимодействия модулей, отношения наследования и вложенности. Общее наблюдение: самый большой враг надежности (и качества ПО в целом) - это сложность. Создавая наши структуры настолько простыми, сколь это возможно, мы достигаем необходимого, но не достаточного условия, гарантирующего надежность. Прежнее обсуждение служит лишь верной отправной точкой в последующих систематических усилиях.

Заметьте, необходим, но также недостаточен, постоянный акцент на создание элегантного и чи-

табельного ПО. Программные тексты не только пишутся, они еще читаются и переписываются по много раз. Ясность и простота нотации языковых конструкций - основа любого изощренного подхода к надежности.

Еще одно необходимое оружие - автоматическое управление памятью, в особенности сборка мусора. В лекции, посвященной этой теме, в деталях пояснено, почему для любой системы, оперирующей динамическими структурами данных, столь опасно опираться на управление этим процессом вручную. Сборка мусора не роскошь - это ключевой компонент ОО-среды, обеспечивающий надежность.

Тоже можно сказать об еще одном, сочетающемся с параметризацией механизме, - статической типизации. Без правил строгой статической типизации пришлось бы лишь надеяться на снисхождение многочисленных ошибок, возникающих в период выполнения.

9.2 О корректности ПО

Зададимся вопросом, что означает утверждение - программный элемент корректен? Наблюдения и рассуждения, отвечающие на это вопрос, могут показаться тривиальными. Но, как заметил один известный ученый, таковы все научные результаты, - они начинаются с обычных наблюдений и продолжаются путем простых рассуждений, но все это нужно делать упорно и настойчиво.

Предположим, некто пришел к вам с программой из 300 000 строк на С и спрашивает, корректна ли она? Если вы консультант, то взыщите высокую плату и ответьте - "нет". Вы, вероятно, окажетесь правы.

Для того чтобы можно было дать разумный ответ на подобный вопрос, одной программы недостаточно, необходима еще и ее спецификация, точно описывающая, что должна делать программа. Оператор

```
x := y+1
```

сам по себе не является ни корректным, ни не корректным. Эти понятия приобретают смысл лишь по отношению к ожидаемому эффекту присваивания. Например, присваивание корректно по отношению к утверждению: "Переменные x и y имеют различные значения". Но не гарантируется его корректность по отношению к высказыванию: "переменная x отрицательна поскольку результат присваивания зависит от значения y , которое может быть положительным.

Эти примеры иллюстрируют свойство, служащее отправной точкой в обсуждении проблемы корректности: программная система или ее элемент сами по себе ни корректны, ни не корректны. Корректность подразумевается лишь по отношению к некоторой спецификации. Строго говоря, мы и не будем обсуждать проблему корректности программных элементов, а лишь их согласованность (consistent) с заданной спецификацией. В наших обсуждениях мы будем продолжать использовать хорошо понимаемый термин "корректность но всегда при этом помнить, что этот термин не применим к программному элементу, он имеет смысл лишь для пары - "программный элемент и его спецификация".

Свойство корректности ПО

Корректность - понятие относительное.

В этой лекции мы научимся выражать спецификации через утверждения (assertions), что поможет оценить корректность разработанного ПО. Но пойдем дальше и перевернем проблему, - разработка спецификации является первым, важнейшим шагом на пути, гарантирующем, что ПО действительно соответствует спецификации. Существенную выгоду можно получить, когда спецификации пишутся одновременно с написанием программы, а лучше, до ее написания. Среди следствий такого подхода можно отметить следующее.

- Разработка ПО корректного с самого начала, проектируемого так, чтобы быть корректным. Один из создателей структурного программирования Харлан Д. Миллс в семидесятые годы

написал статью со знаменательным названием "Как писать корректные программы и знать это". Слово "знать" в данном контексте означает снабжать программу в момент ее написания аргументами, характеризующими корректность.

- Значительно лучшее понимание проблемы и достижение ее решения.
- Упрощение задачи создания программной документации. Как будет позже показано, утверждения будут играть важную роль в ОО-подходе к документации.
- Обеспечение основ для систематического тестирования и отладки.

Оставшаяся часть лекции посвящена исследованию этих вопросов. Одно предупреждение: языки программирования C, C++, Python и другие имеют оператор утверждения `assert`, динамически проверяющий истинность заданного утверждения в момент выполнения программы и останавливающий вычисление, если утверждение является ложным. Эта концепция, хотя и имеет отношение к предмету обсуждения, но является лишь малой частью использования утверждений в ОО-методе. Потому, если подобно многим разработчикам вы знакомы с этим оператором, не обобщайте ваше знание на всю картину, почти все концепции этой лекции, возможно, будут новыми.

9.3 Выражение спецификаций

От неформальных высказываний перейдем к простой математической нотации, принятой в теории формальной проверки правильности программ и имеющей ценность при доказательстве корректности программных элементов.

Формула корректности

Пусть A - это некоторая операция (оператор или тело программы). Формула корректности (correctness formula) - это выражение в форме:

$$\{P\} A \{Q\}$$

Формула выражает свойство, которое может быть или не быть истинным:

Смысл формулы корректности $\{P\} A \{Q\}$ Любое выполнение A , начинающееся в состоянии, где P истинно, завершится и в заключительном состоянии будет истинно Q .

Формула корректности, называемая также триадой Хоара, - математическое понятие, а не программистская конструкция. Она не является частью языка программирования и введена для того, чтобы выражать свойства программных элементов. В этой формуле A , как было сказано, обозначает операцию, P и Q - свойства вовлекаемых в рассмотрение сущностей, называемые утверждениями (точный смысл этого термина будет определен ниже). Утверждение P называется предусловием, а Q - постусловием.

С этого момента обсуждение корректности ПО будет связываться не с программным элементом A , а с триадой, содержащей этот элемент A , предусловие P и постусловие Q . Единственной целью становится установление того, что триада Хоара $\{P\} A \{Q\}$ выполняется (истинна).

Вот пример выполняемой тривиальной формулы, в которой полагается, что x имеет тип integer:

$$\{x \geq 9\} x := x + 5 \{x \geq 13\}$$

Число 13 в постусловии не опечатка. Предполагая корректную реализацию целочисленной арифметики, данная формула действительно выполняется. Если предусловие $x \geq 9$ выполняется перед присваиванием, то $x \geq 13$ будет истинным по завершении оператора присваивания. Конечно, можно утверждать более интересную вещь: при заданном предусловии сильнейшим, насколько это возможно, будет постусловие $x \geq 14$. В свою очередь, при заданном постусловии $x \geq 13$ слабейшим

предусловием будет $x \geq 8$. Из выполняемой формулы корректности всегда можно породить новые выполняемые формулы, ослабляя постусловие или усиливая предусловие. Займемся теперь выяснением того, что означают термины "сильнее" и "слабее" в пред- и постусловиях.

Сильные и слабые условия

Понятия "сильнее" и "слабее" пришли из логики. Говорят, что $P1$ сильнее, чем $P2$, а $P2$ слабее, чем $P1$, если $P1$ влечет $P2$ и они не эквивалентны. Каждое утверждение влечет True, и из False следует все что угодно. Можно говорить, что True является слабейшим, а False сильнейшим из всех возможных утверждений.

Давайте взглянем на формулу корректности с позиций человека, собирающегося наняться на работу по выполнению операции A . Каковы с его точки зрения наилучшие предусловие P и постусловие Q , если у него есть возможность выбора? Возможность усиления предусловия означает, что можно предъявлять более жесткие требования к работодателю, что можно уменьшить число ситуаций, в которых следует приступать к выполнению работы. Так что сильное предусловие это "хорошие новости" для работника. Наилучшей для него работой - синекурой является работа, чья спецификация выражается формулой:

Синекура 1

`{False} A {...}`

Постусловие здесь не специфицировано, поскольку не имеет значения каково оно. К выполнению работы можно вообще не приступать, поскольку нет ни одного начального состояния, в котором предусловие было бы истинным. Так что если вам предложат такую синекуру, немедленно соглашайтесь, не глядя на постусловие - требования, предъявляемые к выполненной работе.

Именно такую спецификацию работ имел в виду начальник полиции одного из американских городов. Когда его спросили в интервью, почему он выбрал именно эту работу, он ответил: "Это единственная работа, где заказчик всегда неправ!"

Для постусловия ситуация меняется на противоположную. Лучшими для работника являются более слабые условия - это "хорошие новости"; в этом случае хорошо нужно уметь делать очень немного. Наилучшей работой - второй синекурой является работа, заданная спецификацией:

Синекура 2

$\{ \dots \} A \{ \text{True} \}$

Как бы не была выполнена работа, постусловие в этом случае будет истинным по определению. Кстати, почему эта работа является все-таки второй по предпочтительности? Причина, как можно видеть из определения триады Хоара, в завершаемости (terminate). Определение устанавливает, что выполнение должно завершиться в состоянии, удовлетворяющем Q, всякий раз, когда оно начинается в состоянии, удовлетворяющем P. Для синекуры 1, где нет состояний, удовлетворяющих P, не имеет значения, что делает A даже если программный текст приводит к выполнению бесконечного цикла, или ломает компьютер. Любое A будет корректным по отношению к данной спецификации. Для синекуры 2, однако, требуется завершение работы, должно существовать заключительное состояние, не важно, что делает A, но то, что делается, должно быть выполнено за конечное время.

Читатели, знакомые с теорией, могли заметить, что формула $\{P\} A \{Q\}$ определяет тотальную (total correctness) или полную корректность, включающую завершаемость наряду с соответствием спецификации. Свойство, устанавливающее, что программа удовлетворяет спецификации при условии ее завершения, известно, как частичная корректность.

Обсуждение того, будет ли усиление или ослабление утверждений "хорошей" или "плохой" новостью, шло с позиций работника, нанимающегося для выполнения работы. Обратим ситуацию, и рассмотрим ее с позиций работодателя. В этом случае слабое предусловие станет "хорошей" новостью, поскольку

означает выполнение работы для большего множества входных случаев; более предпочтительным теперь является сильное постусловие, поскольку оно расширяет получение важных результатов. Эта двойственность критериев типична в рассмотрении корректности ПО. Она вновь появится в качестве центрального понятия этой лекции при обсуждении темы: контракты между модулями - клиентами и поставщиками, в установлении которых преимущества, приобретаемые одним участником, становятся обязательствами для другого. Производство эффективного и надежного ПО проходит через составление контрактов, представляющих возможные наилучшие компромиссы во всех межмодульных коммуникациях клиентов и поставщиков.

9.4 Введение утверждений в программные тексты

Как только корректность ПО определена как согласованность реализации с ее спецификацией, следует предпринять шаги по включению спецификации в сам программный продукт. Для большинства в программистском сообществе это все еще новая идея. Привычно писать программы, устанавливая тем самым, - как делать (the how); менее привычно рассматривать описание целей - что делать (the what) - как часть программного продукта.

Спецификации будут основываться на утверждениях - выражениях, включающих сущности нашего ПО. Выражение задает свойство, которому эти сущности могут удовлетворять на некоторых этапах выполнения программы. Типичное утверждение может выражать тот факт, что определенное целое имеет положительное значение, или что некоторая ссылка не определена.

Ближайшим к утверждению математическим понятием является предикат, хотя используемый язык утверждений обладает лишь частью выразительной силы полного исчисления предикатов.

Предусловия и постусловия

Первое использование утверждений - семантическая спецификация программ. Программа - это не просто часть кода, она задает реализацию функции, входящей в спецификацию АТД. Задачу, выполняемую функцией, необходимо выразить точно, как в интересах проектирования, так и как цель последующей реализации и понимания программного текста. Два утверждения связываются с программой - предусловие и постусловие. Предусловие устанавливает свойства, которые должны выполняться всякий раз, когда программа вызывается; постусловие определяет свойства, гарантируемые программой по ее завершению.

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-
```

```
"""
```

Бинарный алгоритм возведения в степень

```
"""
```

```
def power0(a, n):
```

```
    """
```

```
    простая реализация, используется только для проверки
```

```
    """
```

```
    assert n >= 0
```

```
    r = 1
```

```
    for i in range(n):
```

```
        r *= a
```

```
    return r
```

```
def power1(a, n):
    """
    первоначальный вариант со всеми проверками
    """
    assert n >= 0
    r, a0, n0 = 1, a, n
    # инвариант цикла
    assert r*power0(a, n) == power0(a0, n0)
    while n > 0:
        if n & 1:
            r *= a
        n >>= 1
        a *= a
    # инвариант цикла
    assert r*power0(a, n) == power0(a0, n0)
    assert r == power0(a0, n0)
    return r
```

```
def power(a, n):
    """
    окончательный вариант
    """
    assert n >= 0
    r = 1
```

```
while n > 0:
    if n & 1:
        r *= a
    n >>= 1
    a *= a
return r
```

```
if __name__ == '__main__':
    for i in range(14):
        print "2**%d" % i , "=", power(2, i)
    for i in range(14):
        print "2**%d" % i , "=", power(-2, i)
```

9.5 пример Класс стек

Предусловия

Предусловия выражают ограничения, выполнение которых необходимо для корректной работы функции. Здесь:

- put не может быть вызвана, если стек заполнен;
- remove и item не могут быть применены к пустому стеку.

Предусловия применяются ко всем вызовам программы, как внутри класса, так и у клиента. Корректная система никогда не вызовет программу в состоянии, в котором не выполняется ее предусловие.

Постусловия

Постусловие выражает свойство состояния, завершающего выполнение программы. Здесь:

- После завершения `put` стек не может быть пуст; на его вершине находится только что втолкнуемый элемент, число его элементов увеличилось на единицу.
- После `remove` стек не может быть полон, число его элементов на единицу уменьшилось.

Постусловие в программе выражает гарантию, представленную создателем программы, что выполнение программы завершается и приводит к состоянию с заданными свойствами, в предположении, что программа была запущена в состоянии, удовлетворяющем предусловию.

9.6 Контракты и надежность ПО

Предусловие и постусловие программы определяют контракт со всеми ее клиентами.

Права и обязательства

Связывая с программой `г` предложения `require pre` и `ensure post`, класс говорит своим клиентам:

"Если вы обещаете вызвать `г` в состоянии, удовлетворяющем `pre`, то я обещаю в заключительном состоянии выполнить `post`".

В отношениях между людьми и компаниями контракт - это письменный документ, фиксирующий отношения. Удивительно, что в программной индустрии, где точность так важна и двусмысленность так рискованна, эта идея так долго не появлялась. Любой хороший контракт устанавливает для обоих участников как обязательства, так и приобретаемую выгоду; обычно обязательства одного оборачиваются выгодой для другого участника, и это взаимно. Все это верно и для контрактов между классами.

- Предусловие связывает клиента: определяются условия, при которых вызов программы клиентом легитимен. Обязательства клиента приносят пользу поставщику.
- Постусловие связывает класс: программа обязана обеспечить условия по ее завершению. Здесь польза клиента оборачивается обязательствами поставщика класса.

Возможно, вы не заметили, что контракт противоречит мудрости, бытующей в программной инженерии. Поначалу это шокирует, но контракт - один из главных вкладов в надежность ПО.

Правило контракта говорит, что предусловие дает преимущество поставщику, если клиентская часть контракта не выполняется, то класс перестает быть связан постусловием. В этом случае программа может делать все что угодно, например заикнуться, не нарушая при этом контракт. Это тот самый случай, когда "заказчик виноват".

Первое преимущество от такого соглашения в том, что стиль программирования существенно упрощается. Разработчик класса при написании тела программы смело может предполагать, что все ограничения, заданные предусловием, выполняются; ему нет нужды проверять их в теле программы. Так для функции, вычисляющей квадратный корень:

```
def sqrt(x):  
    assert x >= 0  
    .....
```


можно смело применять алгоритм, не учитывающий случай отрицательного x , поскольку это предусмотрено предусловием, и ответственность за его выполнение несут клиенты программы. С первого взгляда это может показаться опасным, но читайте дальше. Фактически метод Проектирования по Контракту идет дальше. Предположим, что мы написали в предложении до предыдущей программы следующий текст:

```
...
if x < 0:
    print "Обработать ошибку как-нибудь"
else:
    print "Выполнить нормальное вычисление квадратного корня"
....
```

Заметьте, в этом не только нет никакой необходимости, но это и неприемлемо! Этот факт можно отразить в следующем методологическом правиле:

Принцип Нет-Избыточности

Ни при каких обстоятельствах в теле программы не должно проверяться ее предусловие

Это правило противоречит тому, чему учат во многих учебниках по программированию, где необходимость проверок часто выступает под знаменами "защитного программирования" (defensive programming). Его идея в том, что для получения надежного ПО каждая программа должна защищать себя настолько, насколько это возможно. Лучше больше проверок, чем недостаточно; нельзя доверять незнакомцам; еще одна проверка может и не поможет, но и не навредит делу.

Проектирование по контракту утверждает противное: избыточные проверки могут нанести вред. Конечно, это кажется странным, на первый взгляд. Это естественная реакция, полагать, что дополнительная проверка в худшем случае может быть бесполезной, но не может быть причиной

неполадок. Возьмем, например, программу `sqrt`, включившую проверку $x < 0$, хотя ее клиенты были проинструктированы о необходимости обеспечения $x \geq 0$. Что в этом плохого? С микроскопической точки зрения, ограничив наше видение узким мирком `sqrt`, кажется, что включение проверки делает программу более устойчивой. Но мир системы не ограничивается одной программой - он содержит множество программ в множестве классов. Для получения надежной системы необходимо перейти к макроскопическому видению проблемы, обобщающему всю архитектуру.

С этой глобальной точки зрения простота становится критическим фактором. Сложность - главный враг качества. Когда в этот концерт привносятся излишние проверки, то это уже не покажется столь безобидным делом. Экстраполируйте на тысячи программ в системе среднего размера (или на десятки и сотни тысяч в большой системе) проверку (`if x < 0 then ...`), столь безобидную с первого взгляда, - все это начнет выглядеть подобно монстру бесполезной сложности. Добавляя избыточные проверки, добавляете больше кода. Больше кода - больше сложности, отсюда и больше источников условий, приводящих к тому, что все пойдет не так, это приведет к дальнейшему разрастанию кода и так до бесконечности. Если пойти по этой дороге, то определенно можно сказать одно - мы никогда не достигнем надежности. Чем больше пишем, тем больше придется писать.

Этот бег с препятствиями не для нас, нас ждет другая дорога. Проектирование по Контракту приглашает идентифицировать согласованные условия, необходимые для правильного функционирования каждого контракта в кооперации клиенты - поставщики. Метод вынуждает для каждого соглашения установить, кто несет ответственность - клиент или поставщик. Ответ может быть разный, частично он определяется стилем проектирования; позже будет дан ответ, как это делать лучшим образом. Но когда решение принято, нужно его придерживаться. Если требования корректности появляются в предусловии, определяя тем самым ответственность клиента, то в программе не должно быть соответствующих проверок. Требования, не указанные в предусловии, должны проверяться и выполняться в программе.

Следует отметить, что избыточные проверки широко применяются в процессе функционирования

компьютеров и другой электронной аппаратуры. Физическая система, нормально функционирующая, со временем может терять целостность; причины разные - износ, разрыв, внешние воздействия. Поэтому нормальной практикой является, когда получатель и отправитель сигнала оба проверяют его целостность. Для программных систем феномена износа не наблюдается, нет и необходимости в избыточных проверках.

Можно также заметить, что так называемая избыточная проверка аппаратуры, на самом деле таковой не является: это могут быть различные дополнительные тесты, например, проверка на четность, проверка разных устройств и т.д.

Еще одним недостатком защитного программирования является его стоимость. Потеря производительности - наказание за избыточные проверки. Иногда этого вполне достаточная причина для отказа от защитного программирования, чтобы не писалось в учебниках. Работа по удалению таких проверок может быть довольно утомительной. Приемы, рассматриваемые в этой лекции, оставляют место дополнительным проверкам, но они будут основываться на разработке такого окружения, которое возьмет на себя заботу о подобных проверках. После завершения отладки достаточно будет отключить соответствующий параметр компиляции, чтобы проверки исчезли; в самом программном продукте они не содержатся.

Не говоря уже о потере производительности, принципиальной причиной отказа от защитного программирования является наша цель - получение максимальной надежности. Для систем сколь либо существенных размеров недостаточно обеспечение качества отдельных элементов, - более важно гарантировать, что для каждого взаимодействия двух элементов задан явный список взаимных обязательств и преимуществ - контракт. В заключение сформулируем парадокс Дзен-стиля: меньше проверок - больше надежности.

Контрольные вопросы

1. Корректность программы – это понятие

неформальное

которое можно определить, используя только термины самой программы

для формализации которого необходимо задание спецификации

которое можно формализовать триадой Хоара

2. Триада Хоара

устанавливает корректность программы безотносительно каких либо условий

устанавливает корректность программы по отношению к ее постусловию

устанавливает корректность программы по отношению к ее предусловию

устанавливает корректность программы по отношению к ее предусловию и постусловию

3. Укажите истинные триады Хоара

$\{x \leq 100\} \ x = x+10 \ \{x \leq 110\}$

$\{x \leq 100\} \ x = -x \ \{abs(x) \leq 100\}$

$\{x \leq 100\} \ x = -x \ \{x \geq -100\}$

$\{x \leq 100\} \ x = x-90 \ \{abs(x) \geq 10\}$

4. Частью контракта, заключаемого между классом-поставщиком и классами, являющимися его клиентами, является?

обязательная проверка взаимных обязательств – поставщик проверяет выполнимость предусловий, клиент - постусловий

гарантия поставщиком выполнения постусловия метода класса, при условии, что клиент гарантировал при вызове метода выполнения предусловия

гарантия поставщиком выполнения всех предусловий методов класса в каждый момент выполнения клиентской программы

гарантия поставщиком выполнения постусловий всех методов класса в каждый момент выполнения клиентской программы

5. Общая техника контрактов неприменима

к библиотечным классам

к классам, осуществляющим вход исходных данных

к классам, реализующим интерактивное взаимодействие с конечным пользователем

к классам, получающим данные от внешних источников

6. Частью контракта, заключаемого между классом-поставщиком и классами, являющимися его клиентами, является ?

проверка предусловия в методе класса в тот момент, когда клиент вызывает этот метод

проверка предусловия в клиентской программе непосредственно перед вызовом метода и проверка постусловия сразу же по завершении метода

выполнение предусловия в клиентской программе непосредственно перед вызовом метода

гарантия клиентами выполнения всех предусловий методов класса в каждый момент выполнения клиентской программы

7. Инвариант класса должен выполняться

в каждый текущий момент выполнения методов класса

в клиентской программе непосредственно перед вызовом любого метода

на момент создания объекта

в программе поставщика непосредственно после завершения вызова любого метода

8. Для двух стилей разработки характерно?

профессиональная разработка предполагает толерантный стиль

профессиональная разработка предполагает требовательный стиль

для требовательного стиля – более жесткие требования к клиентам класса

для толерантного стиля – более жесткие требования к поставщику

9. Неверное решение, принятое при разработке программной системы, классифицируется как

баг

неисправность

ошибка

дефект

10. В производственной системе следует предусмотреть возможность включения мониторинга утверждений для проверки?

только предусловий

предусловий, постусловий и инвариантов цикла

вариантов и инвариантов цикла
отключения всяческих проверок

11. Каждый цикл

имеет единственный инвариант
имеет множество инвариантов
имеет инвариант, позволяющий доказать его корректность
имеет инвариант, позволяющий доказать завершаемость цикла

12. Если процедуры создания корректны (по их завершению выполняются все инварианты класса) и каждый из экспортируемых методов сохраняет инварианты, то?

гарантируется выполнение инвариантов во все стабильные моменты
эти условия еще не гарантируют выполнение инвариантов во все стабильные моменты
гарантируется выполнение инвариантов во все стабильные моменты для программ без динамических псевдонимов
гарантируется выполнение инвариантов во все стабильные моменты для программ без циклов

13. Отметьте истинные высказывания

утверждения служат четырем целям: помогают в конструировании корректных программ; помогают в создании документации, помогают в отладке, являются основой механизма исключений
включение функций в состав утверждений позволяет сохранить аппликативный характер утверждений

инвариант реализации, – часть инварианта класса – выражает корректность представления классом соответствующего АТД

инвариант цикла изменяет значение на каждом шаге выполнения цикла

14. Отметьте истинные высказывания

инварианты класса позволяют выражать только свойства и отношения между атрибутами класса

метод с предусловием False бесполезен для клиентов

класс описывает все возможные реализации АТД

компонент, появляющийся в предусловии программы, должен быть доступен каждому клиенту, которому доступна сама программа

15. Отметьте истинные высказывания

предусловия должны описываться в терминах реализации метода

инвариант класса это утверждение, выражающее общие согласованные ограничения, применимые к каждому экземпляру класса, как целому

утверждения задают семантику класса

для модулей, чьими клиентами являются другие программные модули, требовательный подход обычно является правильным выбором

Набрано баллов

10 Когда контракт нарушается: обработка исключений

10.1 Базисные концепции обработки исключений

Отказы

Неформально исключение это аномальное событие, прерывающее выполнение программы. Для получения содержательного определения полезно вначале рассмотреть понятие отказа, непосредственно следующее из идеи контракта.

Программа это не произвольная последовательность инструкций, а реализация некоторой спецификации - контракта программы. Всякий вызов программы должен завершаться в состоянии, удовлетворяющем постусловию и инварианту класса. Неявное следствие контракта - при вызове программы не должны появляться прерывания операционной системы, связанные, например, с обращением к недоступным областям памяти или переполнением при выполнении арифметических операций.

Такие ситуации будем называть **отказом (failure)**.

Определения: успех, отказ

Вызов программы успешен, если он завершается в состоянии, удовлетворяющем контракту. Вызов завершается отказом, если он не успешен.

Будем использовать термины "отказ программы" или просто "отказ как сокращения более точного термина "вызов программы, завершающийся отказом". Понятно, что сама программа не может быть ни успешной, ни давать отказ. Эти понятия применимы только по отношению к конкретному вызову.

Исключения

Программа приводит к отказу из-за возникновения некоторых специфических событий (арифметического переполнения, нарушения спецификаций), прерывающих ее выполнение. Такие события и являются исключениями.

Определение: **исключение** Исключение - событие периода выполнения, которое может стать причиной отказа программы.

Зачастую исключение будет причиной отказа. Но можно предотвратить отказ, написав программу так, что она будет захватывать возникшее исключение, пытаться восстановить состояние, допускающее нормальное продолжение вычислений. Вот почему отказ и исключение - это разные понятия: каждый отказ это следствие исключения, но не каждое исключение приводит к отказу.

Источники исключений

Исключения можно классифицировать, разделив их на категории.

Стандартные исключения могут возникать при выполнении программы на Python в результате следующих ситуаций.

- **Exception**

Базовый класс для всех встроенных классов исключений. Атрибут `exception.args` его экземпляров содержит список аргументов, которые использовались при его инициализации

- `SystemExit`

Исключения этого класса генерируются функцией `sys.exit()`

- `StopIteration`

Вызываются методом `next()` итератора для сигнала, что дальнейших значений нет.

- `StandardError`

Базовый класс исключений, предназначенных для генерации при возникновении стандартных ошибок (то есть всех ошибок, которые могут возникнуть при использовании встроенных средств языка). Производный класс от класса `Exception`

- * `KeyboardInterrupt`

По умолчанию, если интерпретатор получает сигнал `SIGINT`, например, при прерывании с клавиатуры (обычно клавишами `Ctrl-C`)

- * `ImportError`

Генерируются, если модуль с указанным в инструкции `import` именем не может быть импортирован или в модуле не найдено его имя

- * `EnvironmentError`

Базовый класс исключений, генерируемых при ошибках, возникающих за пределами интерпретатора *Python*; кортеж аргументов ошибки: `(errno, errMsg...)`

- `IOError`

Используется при ошибках ввода/вывода

- `OSError`

Исключения этого класса (доступного также как `os.error`) в основном генерируются функциями модуля `os`

- * `EOFError`

Генерируются некоторыми встроенными функциями `input()`, `raw_input()`, если

достигается конец файла и, при этом, из него не считано ни байта информации.

* **RuntimeError**

При ошибках времени выполнения, которые не попадают ни в одну из описанных выше категорий

· **NotImplementedError**

При определении абстрактных методов пользовательских классов-интерфейсов

* **NameError**

Исключения этого класса генерируются, если используемое в программе локальное или глобальное имя не найдено

· **UnboundLocalError**

Используется в случае, если вы неявно ссылаетесь на глобальную переменную (не использовав предварительно инструкцию **global**), а затем пытаетесь удалить ее или присвоить ей новое значение

* **AttributeError**

Исключения этого класса генерируются в случаях, когда невозможно получить значение атрибута с определенным именем, удалить его или присвоить ему новое значение

* **SyntaxError**

Используется синтаксическим анализатором при обнаружении синтаксических ошибок

· **IndentationError**

При синтаксических ошибках с отступами

· **TabError**

При синтаксических ошибках с отступами

* **TypeError**

Генерируются, какая-либо операция применяется к объекту несоответствующего типа

* **AssertionError**

Исключения этого класса генерируются, если не выполняется условие, указанное в инструкции `assert`

* **LookupError**

Базовый класс исключений, генерируемых, если последовательность или отображение не содержит элемента с заданным индексом или ключом

· **IndexError**

Используется в случаях, когда индекс, используемый для получения элемента последовательности, выходит за пределы диапазона

· **KeyError**

Исключения этого класса генерируются, если в отображении не найдена запись с указанным ключом

* **ArithmeticError**

Базовый класс исключений, генерируемых при возникновении арифметических ошибок

· **OverflowError** Используется, когда результат арифметической операции слишком большой, чтобы его можно было представить в рамках используемого типа

· **ZeroDivisionError** Используется, если второй операнд операторов деления и получения остатка от деления равен нулю.

· **FloatingPointError** Исключения этого класса генерируются, если возникают проблемы с выполнением операции с плавающей точкой

* **ValueError**

Генерируются, какая-либо операция применяется к объекту правильного типа, но имеющего несоответствующее значение, и ситуация не может быть описана более точно

с помощью исключения другого типа (например, `TypeError`).

- `UnicodeError`

При ошибках кодирования, связанного с `unicode`

- * `ReferenceError`

При попытке доступа к объекту “сборщику мусора” с недостаточными полномочиями для связи

- * `SystemError`

Генерируются в случае обнаружения внутренних ошибок

- * `MemoryError`

Используется, если для выполнения операции не хватает памяти

- * `Warning`

Базовый класс для предупреждений (см. модуль `warning`)

- `UserWarning`

Предупреждение, генерируемое кодом пользователя

- `PendingDeprecationWarning`

Предупреждение о коде, который в будущем будет упразднён

- `DeprecationWarning`

Предупреждение об упразднённом коде

- `SyntaxWarning`

Предупреждение о вызывающем сомнения синтаксисе

- `RuntimeWarning`

Предупреждение о странном поведении во время работы

Пример обработки исключительной ситуации при работе с базой данных

```

import psycopg2 as db

def rounding(n):
    return int(round(n+0.5))

conn = db.connect(database="CA&NT", user="postgres")
db.threadsafety=2
curs = conn.cursor()

try:
    sql = """
DELETE FROM tblЗаявкиДоп USING tblЗаявки
WHERE tblЗаявкиДоп.FK_Заявка = tblЗаявки.PK_Заявка AND
      tblЗаявки.FK_УчебныйГод = 1;
    """
    curs.execute(sql)
except db.DatabaseError, x:
    print "Ошибка: ", x
    conn.rollback()
conn.commit()
conn.close()

```

Общий вид обработки исключений на языке Python

```
try:
```

```
#здесь код, который может вызвать исключение
raise ExceptionType("message")
except (Тип исключения1, Тип исключения2, ...), Переменная:
    #Код в блоке выполняется, если тип исключения совпадает с одним из типов
    #(Тип исключения1, Тип исключения2, ...) или является наследником одного
    #из этих типов.
    #Полученное исключение доступно в необязательной Переменной.
except (Тип исключения3, Тип исключения4, ...), Переменная:
    #количество блоков except не ограничено
    raise #Сгенерировать исключение "поверх" полученного
except:
    #Будет выполнено при любом исключении, не обработанном типизированными блоками except
else:
    #Код блока выполняется, если не было поймано исключений.
finally:
    #будет исполнено в любом случае, возможно после соответствующего
    #блока except или else
```

Ключевые концепции

- Обработка исключений - это механизм, позволяющий справиться с неожиданными условиями, возникшими в период выполнения.
- Отказ - это невозможность во время выполнения программы выполнить свой контракт.
- Программа получает исключение в результате: отказа вызванной ею программы, нарушения

утверждений, сигналов аппаратуры или операционной системы об аномалиях, возникших в ходе их работы.

- Программная система может включать также исключения, спроектированные разработчиком.
- Программа имеет два способа справиться с исключениями - Повторение вычислений и Организованная Паника. При Повторении тело программы выполняется заново. Организованная Паника означает отказ и формирование исключения у вызывающей программы.
- Базисный механизм обработки исключений, включаемый в язык, должен оставаться простым, если только поощрять прямую цель обработки исключений - Организованную Панику или Повторение.

Контрольные вопросы

1. Отказ

всякое исключение приводит к отказу

понятие, противоположное понятию успех

не может возникнуть для корректных программ

понятие, связанное с вызовом программы, - вызов, заканчивающийся в состоянии, нарушающем контракт

2. Успех

понятие, связанное с вызовом программы, - вызов, заканчивающийся в состоянии, удовлетворяющему контракту

может возникнуть для некорректных программ

всякое исключение означает невозможность достижения успеха

понятие, противоположное понятию отказ

3. Исключение

возникает всегда, когда в программе есть ошибка

приводит к прерыванию нормального процесса вычислений

может служить причиной отказа

может привести в конечном счете к успеху

4. В случае, когда обработка исключения завершается отказом в обработчике исключения

можно не выполнять никаких действий

следует восстановить состояние, удовлетворяющее инварианту

следует освободить занятые ресурсы

обеспечить появление исключения у вызывающей программы

5. Дисциплинированная обработка исключения должна завершаться одной из следующих ситуаций

уведомлением о возникшем исключении и продолжением работы

повторением работы тела метода

исправлением ситуации и возвращением в точку возникновения исключения

отказом

6. Класс Exception

создавать собственные исключения

позволяет классифицировать исключения

не может иметь наследников

организовать разбор случаев при обработке исключения

7. Организованная паника

завершает работу вызванного метода и всего приложения

подразумевает отказ для вызванного метода

это один из двух нормальных способов завершения работы обработчика исключения

означает, что приложение работает некорректно

8. Отметьте истинные высказывания

программная система может включать исключения, спроектированные разработчиком

организованная паника неприемлема при обработке исключения

обработка исключений – это механизм, позволяющий справиться с неожиданными условиями, возникшими в период выполнения

программа получает исключение в результате: отказа вызванной ею программы, нарушения утверждений, сигналов аппаратуры или операционной системы об аномалиях, возникших в ходе их работы

9. Отметьте истинные высказывания?

исключения могут быть следствием аппаратных прерываний
программа во время ее выполнения имеет право нарушать инварианты
при обработке исключения возможен подъем по цепочке вызовов

10. Отметьте истинные высказывания?

отказ в работе всего приложения происходит тогда, когда при обработке возникшего исключения происходят отказы для всех программ из цепочки вызовов
отказ в работе всего приложения происходит тогда, когда при обработке возникшего исключения происходит отказ хотя бы для одной из программ цепочки вызовов
механизм исключений противоречит механизму контрактов
несмотря на возникающие исключения и отказы работа приложения может завершиться успехом

Набрано баллов

11 Введение в наследование

11.1 Основные соглашения и терминология

Кроме терминов “наследник” и “родитель” будут полезны следующие термины:

Терминология наследования

Потомок класса *C* - это любой класс, который наследует *C* явно или неявно, включая и сам класс *C*. (Формально, это либо *C*, либо, по рекурсии, потомок некоторого наследника *C*).

Собственный потомок класса *C* - это потомок, отличный от самого *C*.

Предок *C* - это такой класс *A*, для которого *C* является потомком. Собственный предок *C* - это такой класс *A*, для которого *C* является собственным потомком.

В литературе также встречаются термины “подкласс” и “суперкласс”, но мы не будем их использовать из-за неоднозначности.

Имеется также терминология для компонентов класса: компонент либо является наследуемым (перешедшим от некоторого собственного предка), либо непосредственным (введенным в данном классе).

Стрелка указывает вверх от наследника к родителю. Это соглашение легко запомнить - оно представляет отношение “наследовать от”.

Стрелка - это не просто произвольная пиктограмма, она указывает на одностороннюю связь между своими двумя концами. В данном случае:

- Всякий экземпляр наследника можно рассматривать как экземпляр родителя, а обратное неверно.
- В тексте наследника всегда упоминается его родитель, но не наоборот. Это, на самом деле, является важным свойством ОО-метода, вытекающим из принципа Открыт-Закрыт, согласно которому класс не “знает” списка своих наследников и других собственных потомков.

Наследование инварианта

Хотелось бы указать инвариант класса RECTANGLE, который говорил бы, что число сторон прямоугольника равно четырем и что длины сторон последовательно равны side1, side2, side1 и side2.

У класса POLYGON также имеется инвариант, который применим и к его наследнику:

Правило наследования инварианта

Инвариант класса является конъюнкцией утверждений из его раздела invariant и свойств инвариантов его родителей (если таковые имеются).

Поскольку у родителей класса могут быть свои родители, то это правило рекурсивно: в результате полный инвариант класса получается как конъюнкция собственного инварианта и инвариантов классов всех его предков.

Это правило отражает одну из важных характеристик наследования: сказать, что *B* наследует *A* - это утверждать, что каждый экземпляр *B* является также экземпляром *A*. Вследствие этого всякое выраженное инвариантом ограничение целостности, применимое к экземплярам *A*, будет также применимо и к экземплярам *B*.

В нашем примере второе предложение инварианта POLYGON утверждает, что число сторон должно быть не менее трех, оно является следствием предложения из инварианта класса RECTANGLE, которое требует, чтобы сторон было ровно четыре.

Наследование и конструкторы

Процедура создания (конструктор) для класса POLYGON может иметь вид

```
class Point:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    @static
    def len2(a, b):
        return (a.x - b.x)**2 + (a.y - b.y)**2

class Polygon:
    @static
    def isLine(points):
        ...

    def __init__(self, points):
        assert len(points) >= 3
        assert not isLine(points)
        self.points = points

class Convex(Polygon):
    @static
    def isConvex(points):
```

```

...

def __init__(self, points):
    Polygon.__init__(self, points)
    assert len(points) >= 3
    assert isConvex(points)

class Parallelogram(Convex):
    def __init__(self, points):
        Convex.__init__(self, points)
        assert len(points) == 4
        assert Point.len2(points[0], points[1]) == Point.len2(points[2], points[3])
        assert Point.len2(points[1], points[2]) == Point.len2(points[3], points[0])

class Rectangle(Parallelogram):
    def __init__(self, points):
        Parallelogram.__init__(self, points)
        assert (points[0].x - points[1].x)*(points[1].x - points[2].x) \
            + (points[0].y - points[1].y)*(points[1].y - points[2].y) == 0

```

Полиморфное присоединение

“Полиморфизм” означает способность обладать несколькими формами. В ОО-разработке несколькими формами обладают сущности (элементы структур данных), способные во время выполнения присоединяться к объектам разных типов.

Такие структуры данных, содержащие объекты разных типов, имеющих общего предка, называются полиморфными структурами данных. Далее будут рассмотрены многочисленные примеры таких структур. Массивы - это только одна из возможностей, полиморфными могут быть любые структуры контейнеров: списки, стеки и т.п.

Полиморфные структуры данных реализуют цель: объединение порождения и наследования для достижения максимальной гибкости и надежности.

```
a = {  
  Polygon([Point(1,2), Point(1,3), Point(3,4)],  
  Convex(...),  
  Parallelogram(...),  
  Rectangle(...),  
}
```

```
for figure in a:  
    figure.draw()
```

Правило Вызова Компонентов

Если тип сущности x основан на классе C , то в вызове компонента $x.f$ сам компонент f должен быть определен в одном из предков C .

Неограниченный полиморфизм был бы несовместим со статическим понятием типа. Допустимость полиморфных операций определяется наследственностью.

11.2 Ключевые концепции

- С помощью наследования можно определять новые классы как расширение, специализацию и комбинацию ранее определенных классов.
- Класс, наследующий другому классу, называется его наследником, а исходный класс - его родителем. Распространенные на произвольное число уровней (включая ноль) эти понятия становятся понятиями потомка и предка.
- Наследование является ключевым методом как для повторного использования, так и для расширяемости.
- Плодотворное применение наследования требует переопределения (предоставления классу возможности переписать реализацию некоторых компонентов его собственного предка), полиморфизма (возможности связывать ссылку во время выполнения с экземплярами разных классов), динамического связывания (динамического выбора подходящего варианта переопределенного компонента), совместности типов (требования, чтобы всякая сущность могла присоединяться только к экземплярам типов-наследников).
- С точки зрения модулей наследник расширяет набор служб, предоставляемых его родителями. В частности, это полезно для повторно использования.
- С точки зрения типов отношение между наследником и его родителем - это отношение “является”. Оно полезно как для повторного использования, так и для расширяемости.
- Методы наследования, в особенности, динамическое связывание, позволяют разрабатывать децентрализованную архитектуру, в которой каждый вариант операции определяется в том же модуле, где описан соответствующий вариант структуры данных.

- Для типизированных языков динамическое связывание можно реализовать с малыми накладными расходами. Связанные с ним оптимизации, в частности, применяемое компилятором статическое связывание и подстановка кода, помогают ОО-программам достичь или превзойти эффективность выполнения традиционных программ.
- Отложенные классы содержат один или более отложенный (не реализованный) компонент. Они описывают частичные реализации абстрактных типов данных.
- Способность эффективных подпрограмм вызывать отложенные позволяет примирить с помощью “классов поведения” повторное использование с расширяемостью.
- Отложенные классы являются основным средством, используемым ОО-методами на стадиях анализа и проектирования.
- Утверждения, применяемые к отложенным компонентам, позволяют точно специфицировать отложенные классы.

Контрольные вопросы

1. При наследовании

если класс A является родителем класса B , то класс B является наследником класса A
по определению класс A является потомком класса A
по определению класс A является предком класса A

если класс A является предком класса B , то класс B является потомком класса A

2. Наследник

может переопределить атрибуты родителя

наследует все компоненты родителя

может переопределить методы родителя

может определить собственные (непосредственные) компоненты

3. Инвариант класса

совпадает с собственным (непосредственным) инвариантом класса – конъюнкцией утверждений из его раздела `invariant`

является конъюнкцией собственных инвариантов всех предков класса

совпадает с инвариантом родителя

является конъюнкцией собственного инварианта класса и собственного инварианта родителя

4. Тип U согласован с типом T

только если совпадают фактические параметры при универсальном порождении типов

только если базовый класс для U является потомком базового класса для T

для универсально порожденных классов $B [Y]$ будет согласован с $A [X]$, если B является потомком A , а Y – потомком X

если типы не являются универсально порожденными

5. В присваивании $P := S$

типы P и S должны совпадать

тип S может быть потомком типа P

тип P может быть потомком типа S

присваивание возможно и в ситуации, когда типы P и S не связаны отношением наследования

6. Класс наследник

конструкторы родителя наследуются, но они не сохраняют статус конструкторов

не наследует конструкторы своего родителя

каждый класс должен определить собственный набор конструкторов

наследует все компоненты родителя

7. Динамические и статические типы

ссылка имеет динамический тип или может быть пустой

сущность имеет динамический и статический типы

объект имеет динамический и статический типы

сущность имеет только статический тип

8. Полиморфизм – это способность обладать несколькими формами. Полиморфными могут быть?

сущности

методы

структуры данных

присваивания и присоединения

9. Полиморфное присоединение (присваивание, замена формального аргумента фактическим) допускается?

для ссылок и для развернутого типа

только для ссылок

только для развернутого типа

вообще не допускается

10. Изменение объявления компонента позволяет?

переопределить атрибут в виде функции без аргументов

задать реализацию отложенного компонента

функцию без аргументов переопределить в виде атрибута

переопределить реализацию уже реализованного компонента

11. Расширение или специализация

экземпляры родительского класса являются экземплярами классов потомков

наследование классов, рассматриваемое как наследование модулей, является расширением

у экземпляров потомков свойств и методов может быть больше, чем у экземпляров родительского класса

наследование классов, рассматриваемое как наследование типов, является специализацией

12. Отложенный компонент, отложенный класс

класс является отложенным, если он имеет хотя бы один отложенный компонент

компонент является отложенным, если его реализация возлагается на потомка

компонент является отложенным, если потомок переопределил реализацию, заданную его родителем

класс является отложенным, если все его компоненты отложены

13. Отметьте истинные высказывания

всякое изменение отложенного компонента должно сделать его эффективным

повторное объявление компонента – означает определение или переопределение его реализации

тип создаваемого экземпляра в процедуре создания не может быть отложенным

у отложенного класса нет инвариантов, а для отложенного компонента не может быть задано предусловие и постусловие

14. Отметьте истинные высказывания

полностью отложенный класс соответствует *АТД*

наследование позволяет определять новые классы либо как расширение, либо как специализацию

если результат статического связывания не совпадает с результатом динамического связывания, то семантически некорректным является статическое связывание

если наследник переопределил метод родителя, то вызвать родительский метод у потомка не представляется возможным

15. Отметьте истинные высказывания

наследование является ключевым механизмом повторного использования и расширяемости
вызов динамически связываемого компонента можно реализовать за константное время

программой с дырами называется еще не завершенная программа или программа, в которой обнаружены ошибки

эффективный компонент – это компонент, снабженный реализацией

Набрано баллов

12 Множественное наследование

Полноценное применение наследования требует важного расширения этого механизма. Изучая его основы, мы столкнулись с необходимостью порождать новые классы от нескольких классов-родителей. Эта возможность, известная как множественное (multiple) наследование (именуемое так в противовес единичному (single) наследованию), действительно нужна для построения надежных ОО-решений. Множественное наследование это, по сути, прямое приложение уже рассмотренных принципов наследования, - класс вправе иметь произвольное число родителей. Однако, изучая этот вопрос более внимательно, можно обнаружить две интересные проблемы: потребность в смене имен компонентов, которая может оказаться полезной и при единичном наследовании; дублируемое (repeated) наследование, при котором два класса связаны отношением предок-потомок более чем одним способом.

12.1 Примеры множественного наследования

Выясним, прежде всего, в каких ситуациях множественное наследование и в самом деле уместно. Для этого рассмотрим ряд типичных примеров, заимствованных из разных предметных областей.

Такой краткий экскурс тем более необходим, что несмотря на элегантность, простоту множественного наследования и реальную потребность в нем, демонстрация этого механизма подчас создает впечатление чего-то сложного и таинственного. И хотя эту точку зрения не подтверждает ни

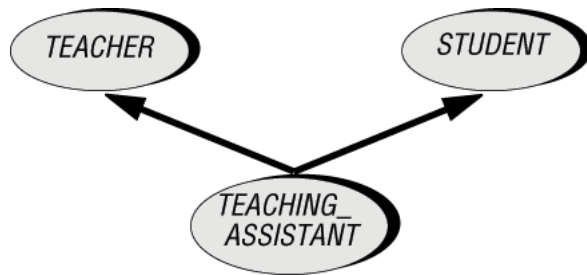


Рис. 12.1: Пример множественного наследования

практика, ни теория, она распространилась достаточно широко, и теперь мы просто обязаны потратить немного времени на изучение случаев, в которых множественное наследование действительно совершенно необходимо.

Сначала покончим с одним бытующим заблуждением. Для этого рассмотрим пример, приводимый (в том или ином виде) во многих статьях, книгах и лекциях, но зачастую порождающий недоверие к множественному наследованию. И дело не в том, что этот пример неверен; просто при первом знакомстве с проблемой он не может служить иллюстрацией, поскольку являет собой образец нетипичного применения этого механизма.

В стандартной формулировке примера речь заходит о классах *TEACHER* и *STUDENT*, и вам тут же предлагают отметить тот факт, что отдельные студенты тоже преподают, и советуют ввести класс *TEACHING_ASSISTANT*, порожденный от *TEACHER* и *STUDENT*.

Выходит, в этой схеме что-то не так? Не обязательно. Но как начальный пример он весьма неудачен. Все дело в том, что *STUDENT* и *TEACHER* - не отдельные абстрактные понятия, а вариации на одну тему *UNIVERSITY_PERSON*. Поэтому, увидев картину в целом, мы обнаружим пример

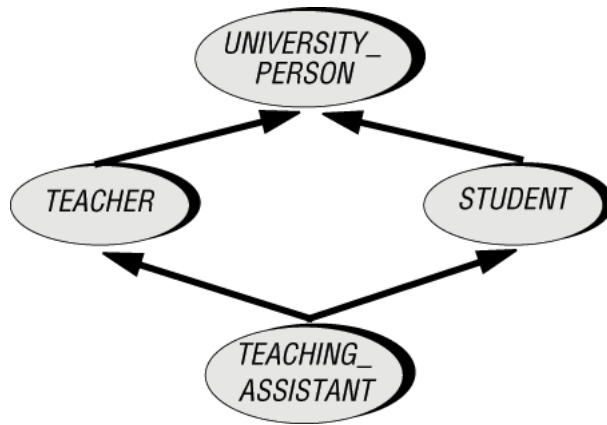


Рис. 12.2: А это пример дублируемого наследования

не просто множественного, но дублируемого (repeated) наследования - схемы, изучаемой позже в этой лекции, в которой класс является правильным наследником другого класса двумя или более различными путями:

Дублируемое наследование - это особый случай. Его применение требует большого опыта в использовании более простых форм порождения классов. Этот пример нельзя обсуждать с начинающими просто потому, что он создает впечатление конфликтов между отдельными компонентами, наследуемых от обоих родителей, в то время как речь идет о свойстве, приходящем от общего предка. При правильном подходе исправить эту проблему не составит труда. Но было бы серьезной ошибкой начинать разговор с таких исключительных и непростых случаев, делая вид, будто они характерны для всего множественного наследования.

Может ли самолет быть имуществом?

Наш первый подходящий пример относится скорее к моделированию систем, чем к проектированию программных продуктов. Однако он наглядно иллюстрирует ситуацию, в которой множественное наследование необходимо.

Пусть класс *AIRPLANE* описывает самолет. Среди запросов к нему могут быть число пассажиров (*passenger_count*), высота (*altitude*), положение (*position*), скорость (*speed*); среди команд - взлететь (*take_off*), приземлиться (*land*), набрать скорость (*set_speed*).

Независимо от него может иметься класс *ASSET*, описывающий понятие имущества. К его компонентам можно отнести такие атрибуты и методы, как цена покупки (*purchase_price*), цена продажи (*resale_value*), уменьшить в цене (*depreciate*), перепродать (*resell*), внести очередной платеж (*pay_installment*).

Наверное, вы догадались, к чему мы клоним: компания ведь может владеть самолетом! И для пилота самолет компании это просто машина, способная взлетать, садиться, набирать скорость. Для финансиста это имущество, имеющее (очень высокую) цену покупки, (слишком низкую) цену продажи, и вынуждающее компанию ежемесячно платить по кредиту.

Для моделирования понятия “самолет компании” прибегнем к множественному наследованию:

В моделировании систем найдется еще немало примеров, подобных *COMPANY_PLANE*.

- Наручные часы-калькулятор моделируются с применением множественного наследования. Один родитель позволяет устанавливать время и отвечать на такие запросы, как текущее время и текущая дата. Другой - электронный калькулятор - поддерживает арифметические операции.
- Наследником классов судно и грузовик является амфибия (*AMPHIBIOUS_VEHICLE*). Наследник классов: судно, самолет - гидросамолет (*HYDROPLANE*). (Как и с *TEACHING_ASSISTANT*, здесь также возможно дублируемое наследование, поскольку каждый из классов-родителей является потомком средства передвижения *VEHICLE*.)

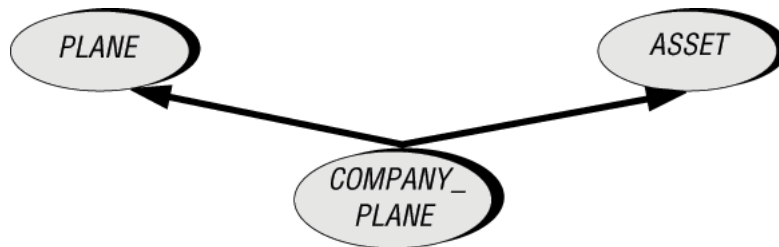


Рис. 12.3: Самолет компании

- Ужин в ресторане; поездка в вагоне поезда - вагон-ресторан (EATING_CAR). Вариант: спальный вагон (SLEEPING_CAR).
- Диван-кровать (SOFA_BED), на котором можно не только читать, но и спать.
- “Дом на колесах” (MOBILE_HOME) - вид транспорта (VEHICLE) и жилище (HOUSE) одновременно; и так далее.

Числовые и сравнимые значения

Следующий пример напрямую относится к повседневной практике ОО-разработки и неразрывно связан с построением библиотеки Kernel.

Ряд классов Kernel, потенциально необходимых всем приложениям, требуют поддержки таких операций арифметики, как $+$, $-$, $*$, $/$, а также специальных значений zero (единичный элемент группы с операцией $+$) и one (единичный элемент группы с операцией $*$). Эти компоненты используют

отдельные классы библиотеки Kernel: *INTEGER*, *REAL* и *DOUBLE*. Впрочем, они нужны и другим, заранее не определенным классам, например, классу *MATRIX*, который описывает матрицы определенного вида. Приведенные абстракции уместно объединить в отложенном классе *NUMERIC*, являющемся частью библиотеки Kernel.

NUMERIC имеет строгое математическое определение. Его экземпляры служат для представления элементов кольца (множества с двумя операциями, каждая из которых индуцирует на нем группу, причем одна из операций коммутативна, а вторая дистрибутивна относительно первой).

Многим классам необходимо отношение порядка с операциями сравнения элементов. Такая возможность полезна для классов Kernel, таких как *STRING*, и для многих других классов. Поэтому в состав библиотеки входит отложенный класс *COMPARABLE*:

Математически его экземпляры - это полностью упорядоченные множества с заданным отношением порядком.

Не все потомки *COMPARABLE* должны быть потомками *NUMERIC*. В классе *STRING* арифметика не нужна, однако нужен порядок. Обратно, не все потомки *NUMERIC* должны быть потомками *COMPARABLE*. Так, на множестве матриц с действительными коэффициентами есть сложение, умножение, единица, нуль, что придает ей свойства кольца, но нет отношения порядка. Поэтому *COMPARABLE* и *NUMERIC* должны оставаться различными классами, и ни один из них не должен быть потомком другого.

Объекты некоторых типов, однако, имеют числовую природу и одновременно допускают сравнение. (Такие классы моделируют вполне упорядоченные кольца.) Примеры таких классов - *REAL* и *INTEGER*. Целые и действительные числа сравнивают, складывают и умножают. Их описание можно построить на множественном наследовании:

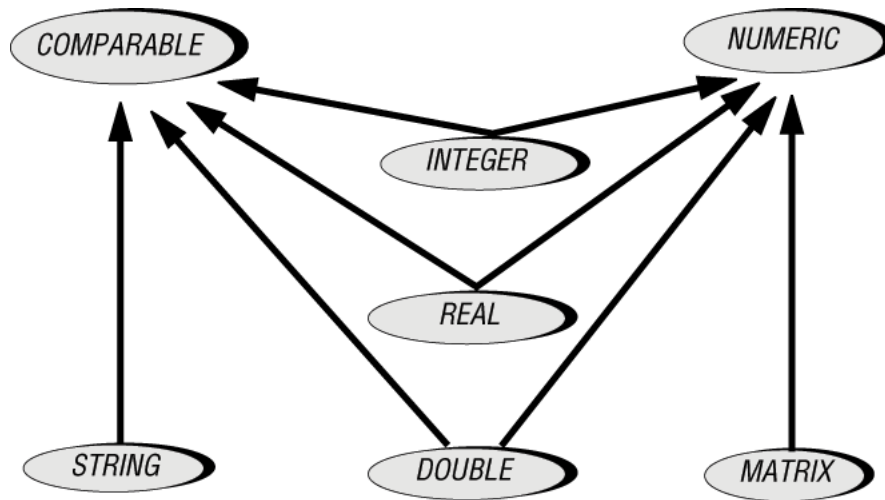


Рис. 12.4: Структура множественного и единичного наследования

Составные фигуры

Следующий пример больше чем пример, - он послужит нам образцом проектирования классов в самых различных ситуациях.

Рассмотрим структуру, введенную в предыдущей лекции для изучения наследования и содержащую классы графических фигур: *FIGURE*, *OPEN_FIGURE*, *POLYGON*, *RECTANGLE*, *ELLIPSE* и т.д. До сих пор в этой структуре использовалось лишь единичное наследование.

Пусть в этой иерархии представлены все нужные нам базовые фигуры. Однако в библиотеку классов хотелось бы включить и не базовые фигуры, имеющие широкое распространение. Конечно, любое изображение каждый раз можно строить из примитивов, но это неудобно. Поэтому мы создадим библиотеку фигур, часть которых будут базовыми, а часть - построена на их основе. Так, из экземпляров базисных классов: отрезка и окружности можно собрать колесо:

Колесо, в свою очередь, может пригодиться при рисовании велосипеда, и т. д.

Итак, нам необходим универсальный механизм создания новых фигур, построенных на основе существующих, но, будучи построенными, используемыми наравне с базовыми.

Назовем новые фигуры составными (*COMPOSITE_FIGURE*). Каждую такую фигуру, безусловно, надо порождать от *FIGURE*, что позволит ей быть “на равных” с базовыми примитивами. Составная фигура - это еще и список фигур, ее образующих, каждая из которых может быть базовой или составной. Воспользуемся множественным наследованием [12.5](#).

Для получения эффективного класса *COMPOSITE_FIGURE* выберем одну из возможных реализаций списка, например связный список - *LINKED_LIST*.

Как и в предыдущих рассмотренных случаях, мы предполагаем, что класс список предлагает механизм обхода элементов, основанный на понятии курсора. Команда *start* устанавливает курсор на первый элемент, если он есть (иначе *after* сразу же равно *True*), *after* указывает, обошел ли курсор все элементы, *item* дает значение элемента, на который указывает курсор, *forth* передвигает курсор к

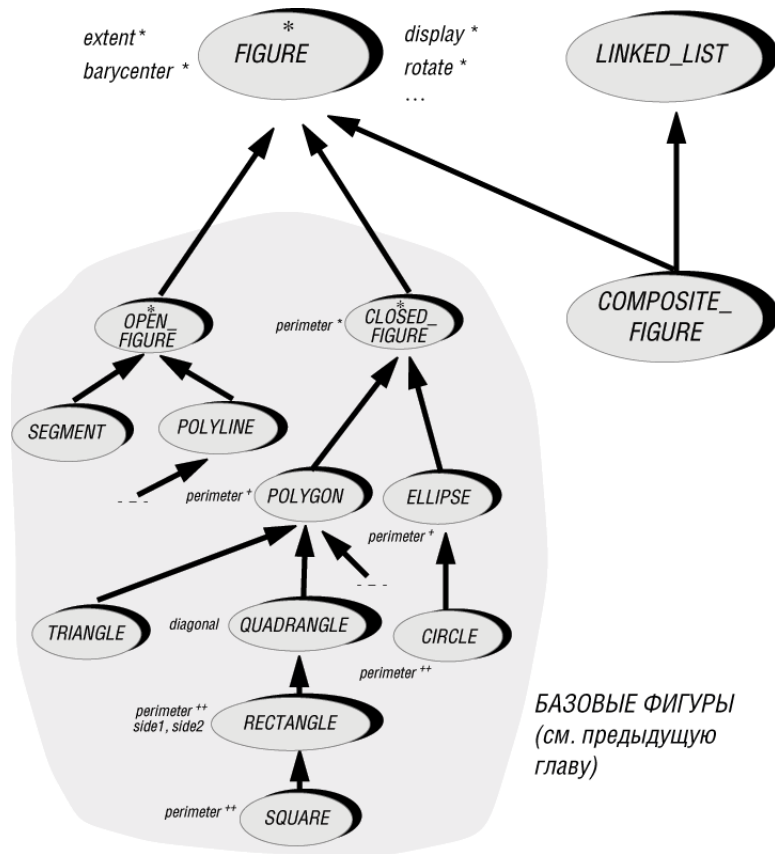


Рис. 12.5: Составная фигура - это фигура и список фигур одновременно

следующему элементу.

Я нахожу эту схему прекрасной и, надеюсь, вы тоже пленитесь ее красотой. В ней вы найдете почти весь арсенал средств: классы, множественное наследование, полиморфные структуры данных (LINKED_LIST [FIGURE]), динамическое связывание (вызов `item.display` применяет метод `display` того класса, которому принадлежит текущий элемент списка), рекурсию (каждый элемент `item` сам может быть составной фигурой без ограничения глубины вложенности).

Но можно пойти еще дальше. Обратимся к другим компонентам `COMPOSITE_FIGURE` - методам вращения (`rotate`) и переноса (`translate`). Они также должны выполнять надлежащие операции над каждым элементом фигуры, и каждый из них может во многом напоминать `display`. Для ОО-проектировщика это может стать причиной тревоги: хотелось бы избежать повторения; потому выполним преобразование - от инкапсуляции к повторному использованию. (Это могло бы стать девизом.) Техника, рассматриваемая здесь, состоит в использовании отложенного класса “итератор”, чьи экземпляры способны выполнять цикл по `COMPOSITE_FIGURE`. Его эффективным потомком может стать `DISPLAY_ITERATOR`, а также ряд других классов.

12.2 Структурное наследование

Множественное наследование просто необходимо, когда необходимо задать для класса ряд дополнительных свойств, помимо свойств, заданных базовой абстракцией.

Рассмотрим механизм создания объектов с постоянной структурой (способных сохраняться на долговременных носителях). Поскольку объект является “сохраняемым”, то у него должны быть свойства, позволяющие его чтение и запись. В библиотеке `Kernel` за эти свойства отвечает класс *STORABLE*, который может быть родителем любого класса. Очевидно, такой класс, помимо *STORABLE*, должен иметь и других родителей, а значит, схема не сможет работать, не будь множественного на-

следования. Примером может служить изученное выше наследование с родителями *COMPARABLE* и *NUMERIC*. Форма наследования при которой родитель задает общее структурное свойство, и, чаще всего, имеет имя, заканчивающееся на - *ABLE*, называется схемой наследования структурного вида.

Без множественного наследования нет способа указать, что некоторая абстракция обладает двумя структурными свойствами - числовыми и сохранения, сравнения и хеширования. Выбор только одного из родителей подобен выбору между отцом и матерью.

12.3 Наследование функциональных возможностей

Вот еще одна типичная ситуация. Многие программные инструменты должны сохранять “историю”, что позволяет пользователям:

- просмотреть список последних команд;
- вторично выполнить последнюю команду;
- выполнить новую команду, отредактировав для этого предыдущую;
- аннулировать действие последней команды, которая не сумела закончить
- свою работу.

Такой механизм привлекателен для любой интерактивной среды, однако его создание требует больших усилий. Поэтому историю поддерживают лишь немногие инструменты (к примеру, ряд “командных оболочек” Unix и Windows), да и те нередко частично. Универсальные же решения

не зависят от конкретного инструмента. Их можно инкапсулировать в класс, а от него - породить другой класс для управления рабочей сессией любого инструмента. (Решение с применением классов-клиентов допустимо, но не так привлекательно.) И снова без множественного наследования не обойтись, так как недостаточно иметь родителя, знающего только историю.

Набор полезных возможностей предоставляет класс *TEST*, инкапсулирующий ряд механизмов тестирования класса: прием и хранение данных от пользователя, вывод и хранение результата, сравнение, регрессионное тестирование и т.д. Хотя решение с использованием вложения может быть предпочтительным, неплохо иметь возможность при тестировании класса *X* определять класс *X_TEST*, порожденный от *X* и *TEST*.

Далее мы будем встречать и другие примеры наследования функциональных возможностей, при котором один класс *F* инкапсулирует набор, например констант или методов математической библиотеки, а другой, объявляя себя потомком *F*, может ими воспользоваться.

Конфликт имен

Иногда при множественном наследовании возникает проблема конфликта имен (name clash). Ее решение - обращение по полному имени – не только снимает саму проблему, но и способствует лучшему пониманию природы классов.

12.4 Ключевые концепции

- Подход к конструированию ПО, подобный конструированию из кубиков, требует возможности объединения нескольких абстракций в одну. Это достигается благодаря множественному наследованию.

- В самых простых и наиболее общих случаях множественного наследования два родителя представляют независимые абстракции.
- Множественное наследование часто необходимо как для моделирования систем, так и для повседневной разработки ПО, в частности, создания повторно используемых библиотек.
- Конфликты имен при множественном наследовании должны указываться поименно.
- Дублируемое наследование - мощная техника - возникает как результат множественного наследования, при котором один класс становится потомком другого несколькими способами.

Контрольные вопросы

1. Достоинства, недостатки и проблемы множественного наследования

проблема: компоненты родителей имеют одинаковые имена

недостаток: наследник не может отказаться от вредных или противоречащих родительских свойств

достоинство: наследование всех возможностей родителей

проблема: родители имеют общих предков

2. Дублируемое наследование

запрещено из-за возникающих конфликтов

это ситуация, при которой родительские классы имеют общих предков

это ситуация, при которой идет наследование от одного и того же класса по разным путям наследования

это ситуация, при которой родительские классы имеют компоненты с одинаковыми именами или совпадающими реализациями

3. Браком по расчету называется такое множественное наследование, когда?

один класс предоставляет атрибуты (данные), а другой – процедуры и функции (действия)

когда получено одобрение предков на множественное наследование

один класс является красивым, а другой – богатым

один класс предоставляет спецификации, а другой – возможности их реализации

4. Переименование позволяет

эффективный класс сделать отложенным

отложенный класс сделать эффективным

выбрать имена компонентов в соответствии со стилем класса наследника

разрешить конфликт имен

5. Если компоненты родителей имеют одинаковые имена, то при наследовании возникает конфликт имен. Этот конфликт разрешается за счет того, что?

наследник может выполнить склеивание компонентов – ситуация часто возникающая при дублируемом наследовании

наследник должен произвести переименование, получив два компонента с разными именами

один из родителей должен переименовать конфликтующий компонент
оба родителя должны переименовать конфликтующий компонент

6. Один и тот же класс

может быть общим предком всех классов

может быть родителем самых разных классов, предоставляя всем своим наследникам полезные функции (возможность выполнения арифметических операций и т.д.)

не может быть родителем нескольких классов

может быть родителем самых разных классов, предоставляя всем своим наследникам полезные свойства (сохраняемость, сравнимость и т.д.)

7. При дублируемом наследовании компонент многократно наследуется от общего предка. Какие ситуации приводят к конфликтам и должны быть разрешены?

компонент с одной и той же реализацией и одним и тем же именем приходит от разных родителей

родители предоставляют эффективные компоненты, имена компонентов различны

от одного родителя приходит эффективный компонент, от другого отложенный, их имена совпадают

родители предоставляют эффективные компоненты, имена компонентов совпадают, наследник переопределяет реализацию компонента

8. Наследник может

отменить реализацию, сделав компонент отложенным

запретить вызов реализации родителя

переопределить реализацию

превратить отложенный компонент в эффективный

9. Конфликт имен делает класс некорректным за исключением следующих случаев

оба компонента имеют различные сигнатуры, и оба эффективны

оба компонента имеют совместимые сигнатуры, и, по крайней мере, один из них наследуется в отложенной форме

оба компонента унаследованы от общего предка, и ни один из них не получен повторным объявлением версии предка

оба компонента имеют совместимые сигнатуры и переопределяются в новом классе

Набрано баллов

Модуль 4

13 Техника наследования

Наследование - ключевая составляющая ОО-подхода к повторному использованию и расширяемости.

13.1 Наследование и утверждения

Вкратце мы уже очертили основные правила, управляющие взаимосвязью наследования и утверждений: все утверждения (предусловие и постусловия подпрограмм, инварианты классов), заданные в классах-родителях, остаются в силе и для их потомков. В этом разделе мы уточним эти правила и используем полученные результаты, чтобы дать новый взгляд на наследование как на субподряды (subcontracts).

Инварианты

С правилом об инвариантах класса мы встречались и прежде:

Правило родительских инвариантов: Инварианты всех родителей применимы и к самому классу.

Инварианты родителей добавляются к классу. Инварианты соединяются логической операцией and then. (Если у класса нет явного инварианта, то инвариант True играет эту роль.) По индукции в классе действуют инварианты всех его предков, как прямых, так и косвенных.

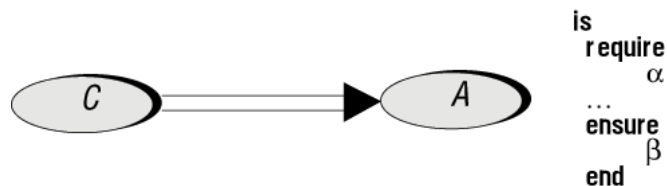


Рис. 13.1: Подпрограмма, клиент и контракт

Как следствие, выписывать инварианты родителей в инварианте потомка еще раз не нужно (хотя семантически такая избыточность не вредит: `a and then a` есть то же самое, что `a`).

Предусловия и постусловия при наличии динамического связывания

В случае с предусловиями и постусловиями ситуация чуть сложнее. Общая идея, как отмечалось, состоит в том, что любое повторное объявление должно удовлетворять утверждениям оригинальной подпрограммы. Это особенно важно, если подпрограмма отложена: без такого ограничения на будущую реализацию, задание предусловия и постусловий для отложенных подпрограмм было бы бесполезным или, хуже того, привело бы к нежелательному результату. Те же требования к предусловию и постусловию остаются и при переопределении эффективных подпрограмм.

Анализируя механизмы повторного объявления, полиморфизма и динамического связывания, можно дать точную формулировку искомого правила. Но для начала представим типичный случай.

Рассмотрим класс и его подпрограммы, имеющие как предусловие, так и постусловие:

Для простоты мы проигнорируем все аргументы, которые может требовать `g`, и положим, что `g` является процедурой, хотя наши рассуждения в равной мере применимы и к функциям.

Как обмануть клиентов

Чтобы понять, как удовлетворить клиентов, мы должны сыграть роль адвокатов дьявола и на секунду представить себе, как их обмануть. Так поступает опытный криминалист, разгадывая преступление. Как мог бы поступить поставщик, желающий ввести в заблуждение своего честного клиента С, гарантирующего при вызове и ожидающего выполнения β ? Есть два пути:

- Потребовать больше, чем предписано предусловием . Формулируя более сильное предусловие, мы позволяем себе исключить случаи, которые, согласно исходной спецификации, были совершенно приемлемы.
- Гарантировать меньше, чем это следует из начального постусловия β . Более слабое постусловие позволяет нам дать в результате меньше, чем было обещано исходной спецификацией.

Вспомните, что мы неоднократно говорили при обсуждении Проектирования по Контракту: усиление предусловия облегчает задачу поставщика (“клиент чаще не прав”), иллюстрацией чего служит крайний случай - предусловие false (когда “клиент всегда не прав”).

Как уже было сказано, утверждение А называется более сильным, чем В, если А логически влечет В, но отличается от него: например, $x \geq 5$ сильнее, чем $x \geq 0$. Если утверждение А сильнее утверждения В, говорят еще, что утверждение В слабее утверждения А.

Как быть честным

Теперь нам понятно, как обманывать. Но как же быть честным? Объявляя подпрограмму повторно, мы можем сохранить ее исходные утверждения, но также мы вправе:

- заменить предусловие более слабым;

- заменить постусловие более сильным.

Первый подход символизирует щедрость и великодушие: мы допускаем большее число случаев, чем изначально. Это не причинит вред клиенту, который на момент вызова удовлетворяет исходному предусловию. Второй подход означает, что мы выдаем больше, чем от нас требовалось. Это не причинит вред клиенту, полагающемуся на выполнение по завершении вызова исходных постусловий.

Итак, основное правило:

Правило (1) Утверждения Переобъявления (Assertion Redeclaration): При повторном объявлении подпрограммы предусловие может заменяться лишь равным ему или более слабым, постусловие - лишь равным ему или более сильным.

Это правило отражает тот факт, что новый вариант подпрограммы не должен отвергать вызовы, допустимые в оригинале, и должен, как минимум, представлять гарантии, эквивалентные гарантиям исходного варианта. Он вправе, хоть и не обязан, допускать большее число вызовов или давать более сильные гарантии.

Как явствует из названия, это правило применимо к обеим формам повторного объявления: переопределению и реализации отложенного компонента. Второй случай важен особо, - утверждения будут связаны со всеми эффективными версиями потомков.

Утверждения подпрограммы, как отложенной, так и эффективной, задают ее семантику, применимую к ней самой и ко всем повторным объявлениям ее потомков. Точнее говоря, они специфицируют область допустимого поведения подпрограммы и ее возможных версий. Любое повторное объявление может лишь сужать эту область, не нарушая ее.

Как следствие, создатель класса должен быть осторожным при написании утверждений эффективной подпрограммы, не привнося излишнюю спецификацию (overspecification). Утверждения должны описывать намерения подпрограммы, - ее абстрактную семантику, - но не свойства реализации. Иначе можно закрыть возможность создания иной реализации подпрограммы у будущих потомков.

Субподряды

Правило Утверждения Переобъявления великолепно сочетается с теорией Проектирования по Контракту.

Мы видели, что утверждения подпрограммы описывают связанный с ней контракт, в котором клиент гарантирует выполнение предусловия, получая право рассчитывать на истинность постусловия; для поставщика все наоборот.

Наследование совместно с повторным объявлением и динамическим связыванием приводит к созданию субподрядов. Приняв условия контракта, вы не обязаны выполнять его сами. Подчас вы знаете кого-то еще, способного сделать это лучше и с меньшими издержками. Так происходят, когда клиент запрашивает подпрограмму из **MATRIX**, но благодаря динамическому связыванию может на этапе выполнения фактически вызывать версию, переопределенную в потомке. “Меньшие издержки” означают здесь более эффективную реализацию, как в знакомом нам примере с периметром прямоугольника, а “лучше” - усовершенствование утверждений, в описанном здесь смысле.

Правило Утверждения Переобъявления просто устанавливает, что честный субподрядчик, приняв условия контракта, должен выполнить работу на тех же условиях, что и подрядчик или лучших, но никак не худших.

С позиции Проектирования по Контракту, инварианты классов - это ограничения общего характера, применимые и к подрядчикам, и к клиентам. Правило родительских инвариантов отражает тот факт, что все подобные ограничения передаются субподрядчикам.

Свое истинное значение для ОО-разработки наследование приобретает лишь совместно с утверждениями и двумя приведенными выше правилами. Метафора контрактов и субподрядов - прекрасная аналогия, помогающая разрабатывать корректное ОО-ПО. Несомненно, в этом - одна из центральных идей теории проектирования.

Абстрактные предусловия

Правило ослабления предусловий может оказаться чересчур жестким в случае, когда наследник понижает уровень абстракции, характерный для его предка. К счастью, есть легкий обходной путь, полностью согласующийся с теорией.

Типичным примером этого является порождение `BOUNDED_STACK` от универсального класса стека (`STACK`). Процедура занесения в стек элемента (`put`) в порожденном классе имеет предусловие `count <= capacity`, где `count` - текущее число элементов в стеке, `capacity` - физическая емкость накопителя.

В общем понятии стека нет понятия емкости. Поэтому создается впечатление, будто при переходе к `BOUNDED_STACK` предусловие приходится усилить (от бесконечной емкости перейти к конечной). Как выстроить структуру наследования, не нарушая правило Утверждения Переобъявления?

Ответ становится очевиден, если мы ближе познакомимся с требованиями к клиенту. То, что нужно сохранить или ослабить, не обязательно является конкретным предусловием, как оно видится в реализации поставщика (реализация это его забота), но касается предусловия, как оно видится клиенту.

Замечание математического характера

Неформально, правило Утверждения Переобъявления гласит: “Повторное объявление утверждений может лишь сужать область допустимого поведения, не нарушая ее”. Сейчас, завершая обсуждение этой темы, приведем строгую формулировку данного свойства.

Пусть подпрограмма реализует частичную функцию r , отображающую множество возможных входных состояний I в множество возможных выходных состояний O . Утверждения подпрограммы определяют правила действия r и ее возможных переопределений.

- Предусловие задает область определения DOM функции r (подмножество I , на котором r гарантированно вырабатывает результат).
- Постусловие задает для каждого x из DOM подмножество $RESULTS(x)$ множества O , такое, что $r(x) \in RESULTS(x)$. Так как постусловие не всегда однозначно описывает результат, это подмножество может иметь больше одного элемента.

Правило Утверждения Переобъявления означает, что повторное объявление может расширять область определения и сужать множество результатов. Пометив новые множества знаком ', запишем требования, закрепленные этим правилом:

$$DOM' \supseteq DOM$$

$$RESULTS'(x) \subseteq RESULTS(x) \text{ для всех } x \text{ из } DOM$$

Предусловие устанавливает, что подпрограмма и ее повторные объявления, как минимум, должны принимать некоторые входы (DOM), хотя повторные объявления могут это множество и расширить. Постусловие говорит, что результаты, возвращаемые подпрограммой и ее повторными объявлениями, могут, самое большее, содержать значения из $RESULTS(x)$, однако, постусловия при повторных объявлениях могут это множество сузить.

В этом описании состояние системы в период выполнения определяется состоянием (значениями) всех достижимых объектов. Кроме того, входные состояния (элементы I) также включают в себя значения аргументов. Более подробное введение в математическое описание программ и языков программирования.

13.2 Наследование и скрытие информации

Последний вопрос, оставшийся пока без ответа, как наследование взаимодействует с принципом Скрытия информации.

В отношениях между классом и его клиентами скрытие информации определяет разработчик класса. Именно он определяет политику в отношении каждого компонента класса: экспортируя его всем клиентам, разрешая выборочный экспорт, или делая компонент закрытым.

Кое-что о политике

Что происходит со статусом экспорта при передаче компонента потомку? Наследование и скрытие информации - ортогональные механизмы. Наследование определяет отношение между классом и его потомками, экспорт - между классом и его клиентами. Класс В может свободно экспортировать или скрывать любой из компонентов f , унаследованных им от класса А. При этом доступны все возможные комбинации:

- f экспортируется в классе А и в классе В (хотя и не обязательно одним и тем же клиентам);
- f скрыто в А и В;
- f скрыто в А, но полностью или частично экспортируется в В;
- f экспортируется в А, но скрыто в В.

Применение

Характерным примером является создание нескольких вариантов одной абстракции.

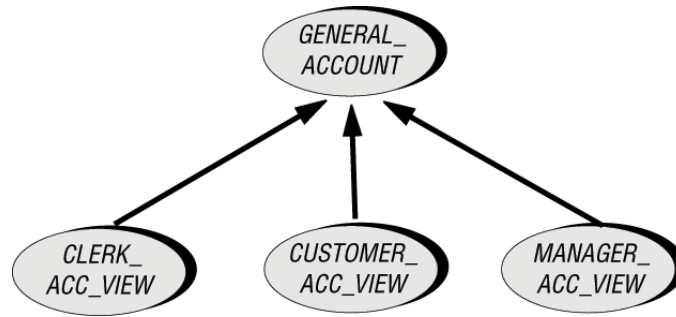


Рис. 13.2: Разные облики одной абстракции

Представим себе *GENERAL_ACCOUNT* - класс, содержащий все необходимые операции для работы с банковскими счетами: процедуры *open*, *withdraw*, *deposit*, *code* (для снятия денег через банкомат), *change_code* и т.д., - но не предназначенный для использования клиентами напрямую, а потому не экспортирующий никаких подпрограмм. Его потомки выступают как разные облики родителя: они не содержат новых компонентов и отличаются лишь предложениями экспорта. Один экспортирует *open* и *deposit*, второй, наряду с ними, - *withdraw* и *code*, и т. д.

Ключевые концепции

- К инварианту класса автоматически добавляются инварианты его родителей.
- В подходе Проектирования по Контракту наследование, переопределение и динамическое связывание приводят к идее субподрядов.

- Повторное объявление подпрограммы (переопределение или создание реализации) может сохранить или ослабить предусловие, сохранить или усилить постусловие.
- Ограниченная универсальность дает возможность использовать только родовые параметры со специфическими свойствами.
- Попытка присваивания позволяет динамически проверить, принадлежит ли объект ожидаемому типу. Эта операция не должна использоваться как замена динамического связывания.
- Потомок вправе переопределять тип любой сущности (атрибута, результата функции, формального параметра подпрограммы). Повторное определение должно быть ковариантным - заменять исходные типы соответствующими, согласуясь с требованиями потомка.
- Наследование и скрывание информации - это независимые механизмы. Потомки могут скрывать экспортированные компоненты и экспортировать скрытые компоненты.
- Компонент, доступный самому классу, доступен и его потомкам.

Контрольные вопросы

1. При повторном объявлении компонента

постусловие ослабляет требования, предъявляемые к работе компонента

предусловие сужает число ситуаций, допустимых у клиента

постусловие ужесточает требования, предъявляемые к работе компонента
предусловие расширяет число ситуаций, допустимых у клиента

2. Инвариант класса

является конъюнкцией собственного инварианта класса и собственного инварианта родителя

если у класса нет собственного инварианта, то инвариант True играет эту роль
совпадает с собственным (непосредственным) инвариантом класса – конъюнкцией утверждений из его раздела invariant

является конъюнкцией собственных инвариантов всех предков класса

3. Если наследник повторно объявляет компонент родителя, то, чтобы не обмануть клиентов, он должен

усилить или сохранить постусловие компонента

усилить или сохранить предусловие компонента

ослабить или сохранить предусловие компонента

ослабить или сохранить постусловие компонента

4. Контракты и субподряды. Под субподрядом понимается создание класса – наследника существующего класса. Субподрядчик

вправе улучшить для клиентов условия контракта

не вправе изменить существующий контракт

обязан удовлетворить всем требованиям существующего контракта

может эффективнее выполнить работу, предусмотренную контрактом

5. Попытка присваивания

предполагает динамическую проверку соответствия типов левой и правой части присваивания

это специальное обобщение операции присваивания

это присваивание цели значения `void` в случае несоответствия типов

присваивание, заканчивающееся отказом

6. Ограничение универсальности

позволяет расширить набор операций, допустимых при работе с объектами

означает, что при родовом порождении фактический параметр не может быть произвольного типа

дает возможность задавать в качестве фактического параметра тип с заданным набором операций

ограничивает возможности работы с объектами в сравнении с неограниченной универсальностью

7. При повторном объявлении

разрешается изменение типа атрибута и типа аргументов у компонентов в соответствии с правилами согласованности типов

требуется сохранение сигнатуры компонента

разрешается изменение сигнатуры компонента

разрешается произвольное изменение типа атрибута и типа аргументов у компонентов

8. Экспорт и наследование

экспорт компонента клиенту означает, что компонент экспортируется всем потомкам этого клиента

все компоненты класса должны быть доступны его потомкам

экспорт компонента клиенту еще не означает, что компонент экспортируется всем потомкам этого клиента

экспорт компонентов может быть выборочным; тот или иной компонент можно экспортировать одному или нескольким клиентам

9. Отметьте истинные высказывания

наследование и скрытие информации – это взаимозависимые механизмы

компонент, доступный классу, может быть не доступен его потомкам

утверждения дают глубокое понимание природы наследования

при повторном использовании интерфейсов скрывается информация о реализации

10. Отметьте истинные высказывания

при повторном использовании реализаций информация скрывается

попытка присваивания может заменить динамическое связывание

повторное объявление должно удовлетворять утверждениям оригинальной подпрограммы

у клиента нет возможности контроля изменения семантики операций создателями классов, пользующимися повторными объявлениями и динамическим связыванием

11. Отметьте истинные высказывания?

лишь понимание принципов Проектирования по Контракту позволяет в полной мере постичь сущность концепции наследования

при повторном использовании интерфейсов реализация не защищена

экспорт определяет отношения между классом и его потомками

заморозив компонент, можно гарантировать его семантическую уникальность

Набрано баллов

14 Типизация

Эффективное применение объектной технологии требует четкого описания в тексте системы типов всех объектов, с которыми она работает на этапе выполнения. Это правило, известное как статическая типизация (static typing), делает наше ПО: более надежным, позволяя компилятору и другим инструментальным средствам устранять несоответствия прежде, чем они смогут нанести вред; более понятным, обеспечивая точной информацией читателей: авторов клиентских систем и тех, кто будет сопровождать систему; более эффективным, поскольку информация о типах данных позволит компилятору сгенерировать оптимальный код. Хотя вопросами типизации данных активно занимались и вне объектной среды, да и сама статическая типизация применяется в языках, не поддерживающих ООП, особенно ярко эти идеи проявили себя именно при объектном подходе, во многом основанном на понятии типа, которое, сливаясь с понятием модуля, образует базовую ОО-конструкцию - класс.

14.1 Проблема типизации

О типизации при ОО-разработке можно сказать одно: эта задача проста в своей постановке, но решить ее подчас нелегко.

Базисная конструкция

Простота типизации в ОО-подходе есть следствие простоты объектной вычислительной модели. Опуская детали, можно сказать, что при выполнении ОО-системы происходят события только одного рода - вызов компонента (feature call):

`x.f (arg)`

означающий выполнение операции `f` над объектом, присоединенным к `x`, с передачей аргумента `arg` (возможно несколько аргументов или ни одного вообще). Программисты говорят в этом случае о “передаче объекту `x` сообщения `f` с аргументом `arg`”, но это - лишь отличие в терминологии, а потому оно несущественно.

То, что все основано на этой Базисной Конструкции (Basic Construct), объясняет частично ощущение красоты ОО-идей.

Из Базисной Конструкции следуют и те ненормальные ситуации, которые могут возникнуть в процессе выполнения:

Определение: **нарушение типа**

Нарушение типа в период выполнения или, для краткости, просто нарушение типа (type violation) возникает в момент вызова `x.f (arg)`, где `x` присоединен к объекту OBJ, если либо:

- не существует компонента, соответствующего `f` и применимого к OBJ,
- такой компонент имеется, однако, аргумент `arg` для него недопустим.

Ключевым является слово когда. Рано или поздно вы поймете, что имеет место нарушение типа. Например, попытка выполнить компонент “Пуск торпеды” для объекта “Служащий” не будет работать и при выполнении произойдет отказ. Однако возможно вы предпочитаете находить ошибки как можно раньше, а не позже.

Статическая и динамическая типизация

Хотя возможны и промежуточные варианты, здесь представлены два главных подхода:

- Динамическая типизация: ждать момента выполнения каждого вызова и тогда принимать решение.
- Статическая типизация: с учетом набора правил определить по исходному тексту, возможны ли нарушения типов при выполнении. Система выполняется, если правила гарантируют отсутствие ошибок.

Эти термины легко объяснимы: при динамической типизации проверка типов происходит во время работы системы (динамически), а при статической типизации проверка выполняется над текстом статически (до выполнения).

Термины типизированный и нетипизированный (typed/untyped) нередко используют вместо статически типизированный и динамически типизированный (statically/dynamically typed). Во избежание любых недоразумений мы будем придерживаться полных именований.

Статическая типизация предполагает автоматическую проверку, возлагаемую, как правило, на компилятор. В итоге имеем простое определение:

Определение: **статически типизированный язык**

ОО-язык статически типизирован, если он поставляется с набором согласованных правил, проверяемых компилятором, соблюдение которых гарантирует, что выполнение системы не приведет к нарушению типов.

В литературе встречается термин “сильная типизация” (strong). Он соответствует ультимативной природе определения, требующей полного отсутствия нарушения типов. Возможны и слабые (weak) формы статической типизации, при которых правила устраняют определенные нарушения, не

ликвидируя их целиком. В этом смысле некоторые ОО-языки являются статически слабо типизированными. Мы будем бороться за наиболее сильную типизацию.

В динамически типизированных языках, например Python, известных как нетипизированные, отсутствуют объявления типов, а к сущностям в период выполнения могут присоединяться любые значения. Статическая проверка типов в них невозможна.

Преимущества

Причины применения статической типизации в объектной технологии мы перечислили в начале лекции. Это надежность, простота понимания и эффективность.

Надежность обусловлена обнаружением ошибок, которые иначе могли проявить себя лишь во время работы, и только в некоторых случаях. Первое из правил, заставляющее объявлять сущности, как, впрочем, и функции, вносит в программный текст избыточность, что позволяет компилятору, используя два других правила, обнаруживать несоответствия между задуманным и реальным применением сущностей, компонентов и выражений.

Раннее выявление ошибок важно еще и потому, что чем дольше мы будем откладывать их поиск, тем сильнее вырастут издержки на исправление. Это свойство, интуитивно понятное всем программистам-профессионалам, количественно подтверждают широко известные работы Бема (Boehm). Зависимость издержек на исправление от времени отыскания ошибок приведена на графике, построенном по данным ряда больших промышленных проектов и проведенных экспериментов с небольшим управляемым проектом:

Читабельность или Простота понимания (readability) имеет свои преимущества. Во всех примерах этой книги появление типа у сущности дает читателю информацию о ее назначении. Читабельность крайне важна на этапе сопровождения.

Исключив читабельность из круга приоритетов, можно было бы получить другие преимущества,

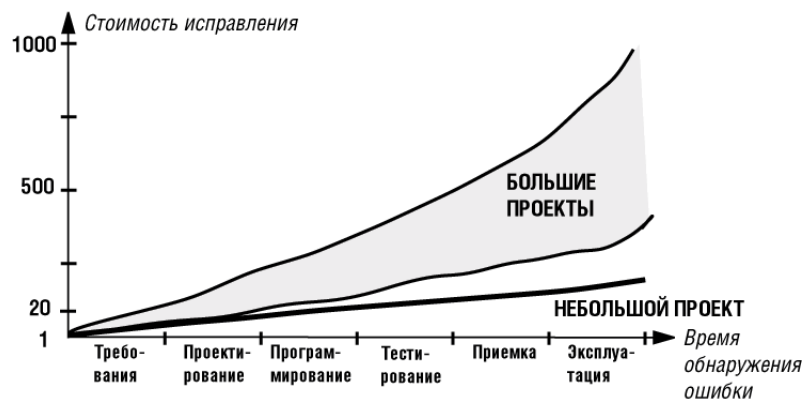


Рис. 14.1: Сравнительные издержки на исправление ошибок

не вводя явных объявлений. В самом деле, возможна неявная форма типизации, когда компилятор, не требуя явного указания типа, пытается автоматически определить его из контекста применения сущности. Эта стратегия известна как выведение типов (type inference). Но в программной инженерии явные объявления типов это помощь, а не наказание, - тип должен быть ясен не только машине, но и читающему текст человеку.

Наконец, эффективность может определять успех или отказ от объектной технологии на практике. В отсутствие статической типизации на выполнение $x.f$ (arg) может уйти сколько угодно времени. Причина этого в том, что на этапе выполнения, не найдя f в базовом классе цели x , поиск будет продолжен у ее потомков, а это верная дорога к неэффективности. Снять остроту проблемы можно, улучшив поиск компонента по иерархии. Авторы языка Self провели большую работу, стремясь генерировать лучший код для языка с динамической типизацией. Но именно статическая типизация позволила такому ОО-продукту приблизиться или сравняться по эффективности с традиционным ПО.

Ключом к статической типизации является уже высказанная идея о том, что компилятор, генерирующий код для конструкции $x.f$ (arg), знает тип x . Из-за полиморфизма нет возможности однозначно определить подходящую версию компонента f . Но объявление сужает множество возможных типов, позволяя компилятору построить таблицу, обеспечивающую доступ к правильному f с минимальными издержками, - с ограниченной константой сложностью доступа. Дополнительно выполняемые оптимизации статического связывания (static binding) и подстановки (inlining) - также облегчаются благодаря статической типизации, полностью устраняя издержки в тех случаях, когда они применимы.

Аргументы в пользу динамической типизации

Несмотря на все это, динамическая типизация не теряет своих приверженцев. Их аргументы основаны прежде всего на реализме, речь о котором шла выше. Они уверены, что статическая типизация чересчур ограничивает их, не давая им свободно выражать свои творческие идеи, называя иногда ее “поясом целомудрия”.

С такой аргументацией можно согласиться, но лишь для статически типизированных языков, не поддерживающих ряд возможностей. Стоит отметить, что все концепции, связанные с понятием типа и введенные в предыдущих лекциях, необходимы - отказ от любой из них чреват серьезными ограничениями, а их введение, напротив, придает нашим действиям гибкость, а нам самим дает возможность в полной мере насладиться практичностью статической типизации.

Возможно, вы заметили, что попытка присваивания - неотъемлемый компонент реалистичной системы типов - напоминает приведение. Однако есть существенное отличие: попытка присваивания выполняет проверку, действительно ли текущий тип соответствует заданному типу, - это безопасно, а иногда и необходимо.

Типизация и связывание

Хотя как читатель этого пособия вы наверняка отличите статическую типизацию от статического связывания, есть люди, которым подобное не под силу. Отчасти это может быть связано с влиянием языка Smalltalk, отстаивающего динамический подход к обоим задачам и способного сформировать неверное представление, будто они имеют одинаковое решение. (Для создания надежных и гибких программ желательно объединить статическую типизацию и динамическое связывание.)

Как типизация, так и связывание имеют дело с семантикой Базисной Конструкции $x.f$ (arg), но отвечают на два разных вопроса:

Типизация и связывание

- Вопрос о типизации: когда мы должны точно знать, что во время выполнения появится операция, соответствующая f , применимая к объекту, присоединенному к сущности x (с параметром arg)?
- Вопрос о связывании: когда мы должны знать, какую операцию инициирует данный вызов?

Типизация отвечает на вопрос о наличии как минимум одной операции, связывание отвечает за выбор нужной.

В рамках объектного подхода:

- проблема, возникающая при типизации, связана с полиморфизмом: поскольку x во время выполнения может обозначать объекты нескольких различных типов, мы должны быть уверены, что операция, представляющая f , доступна в каждом из этих случаев;
- проблема связывания вызвана повторными объявлениями: так как класс может менять наследуемые компоненты, то могут найтись две или более операции, претендующие на то, чтобы представлять f в данном вызове.

Обе задачи могут быть решены как динамически, так и статически. В существующих языках представлены все четыре варианта решения.

- Ряд необъектных языков, скажем, Pascal и Ada, реализуют как статическую типизацию, так и статическое связывание. Каждая сущность представляет объекты только одного типа, заданного статически. Тем самым обеспечивается надежность решения, платой за которую является его гибкость.

- Smalltalk и другие ОО-языки содержат средства динамического связывания и динамической типизации. При этом предпочтение отдается гибкости в ущерб надежности языка.
- Отдельные необъектные языки поддерживают динамическую типизацию и статическое связывание. Среди них - языки ассемблера и ряд языков сценариев (scripting languages).

Отметим своеобразие языка C++, поддерживающего статическую типизацию, хотя и не строгую ввиду наличия приведения типов, статическое связывание (по умолчанию), динамическое связывание при явном указании виртуальных (virtual) объявлений.

Причина выбора статической типизации и динамического связывания очевидна. Первый вопрос: “Когда мы будем знать о существовании компонентов?” - предполагает статический ответ: “Чем раньше, тем лучше”, что означает: во время компиляции. Второй вопрос: “Какой из компонентов использовать?” предполагает динамический ответ: “тот, который нужен”, - соответствующий динамическому типу объекта, определяемому во время выполнения. Это единственно приемлемое решение, если статическое и динамическое связывание дает различные результаты.

Контрольные вопросы

1. Для успешного применения статической типизации требуется совместное применение механизмов

ограниченной и неограниченной универсальности
попытки присваивания

утверждений
множественного наследования

2. Статическая типизация

анализирует состояние объектов в период выполнения
позволяет установить динамический тип сущности
возможна только для ОО-языков
позволяет обнаруживать многие ошибки еще на этапе компиляции

3. Динамическая типизация

позволяет обнаруживать многие ошибки еще на этапе компиляции
позволяет установить динамический тип сущности
возможна только для ОО-языков
анализирует состояние объектов в период выполнения

4. Динамическое связывание

выбирает связываемый компонент из класса, соответствующего статическому типу цели
позволяет связать цель вызова с вызываемым компонентом еще на этапе компиляции
имеет тот же эффект, что и статическое связывание
выбирает связываемый компонент из класса, соответствующего динамическому типу цели

5. Статическое связывание

выбирает связываемый компонент из класса, соответствующего статическому типу цели

позволяет связать цель вызова с вызываемым компонентом еще на этапе компиляции имеет тот же эффект, что и динамическое связывание
выбирает связываемый компонент из класса, соответствующего динамическому типу цели

6. Отметьте истинные высказывания

совместное действие ряда полезных механизмов наследования, статической типизации, скрытия потомком может приводить к некорректной работе системы; удачное решение этой проблемы еще не найдено

язык статически типизирован, если он поставляется с набором согласованных правил, проверяемых компилятором, соблюдение которых гарантирует, что выполнение системы не приведет к нарушению типов

статическое связывание и статическая типизация – это синонимичные понятия

для классово-корректной системы не могут возникать нарушения типа в период выполнения системы

7. Отметьте истинные высказывания

нарушение типа в момент вызова $x.f(arg)$, где x присоединен к объекту OBJ, не возникает, если существует компонент, соответствующий f и применимый к OBJ

полиморфизм не дает возможности осуществить статическое связывание

статическая типизация позволяет обнаружить нарушения типа в момент выполнения системы

наследование позволяет перейти от жесткого требования совпадения типов источника и цели к мягкому требованию соответствия типов

8. Отметьте истинные высказывания

задачи типизации и связывания могут быть решены только динамически
приведение типа (кастинг) препятствует строгой статической типизации
задачи типизации и связывания могут быть решены только статически
в ряде случаев универсальность требуется ограничить

Набрано баллов

15 Глобальные объекты и константы

Локальных знаний не достаточно - компонентам ПО необходима глобальная информация: разделяемые данные, общее окно для вывода ошибок, шлюз для подключения к базе данных или сети. В классическом подходе достаточно объявить такой объект глобальной переменной главной программы. В ОО-системах нет ни главной программы, ни глобальных переменных. Но разделяемые (shared) объекты по-прежнему нужны. Глобальные объекты - некий вызов ОО-методу, провозглашающему идеи децентрализации, модульности и автономности. Борьба шла за независимость модулей, за избавление от произвола центральной власти. Теперь этой власти нет. Как же построить систему, в которой компоненты совместно используют данные, не теряя своей автономности, гибкости, допускают повторное использование? Передавать модулю разделяемые объекты как параметры не разумно, поскольку число их может быть достаточно велико. Да и сама передача параметров предполагает существование владельца, хотя при подлинном разделении владеть значениями не может ни один модуль. Поиск более удачного решения мы начнем с хорошо известного понятия, необходимого как в объектной, так и в традиционной методологии проектирования. Речь пойдет о константах. Что такое константа P_i , как не простой, совместно используемый объект? Обобщив это понятие на более сложные объекты, мы сделаем первый шаг на пути к разделению объектов.

15.1 Константы базовых типов

Начнем с формы записи констант.

Правило стиля - принцип символических констант - гласит, что обращение к конкретному значению (числу, символу или строке) почти всегда должно быть косвенным. Должно существовать определение константы, задающее имя, играющее роль символической константы (symbolic constant), и связанное с ним значение - константа, называемая манифестной (manifest constant). Далее в алгоритме следует использовать символическую константу. Тому есть два объяснения.

- Читабельность: читающему текст легче понять смысл `US_states_count`, чем числа 50;
- Расширяемость: символическую константу легко обновить, исправив лишь ее определение.

Атрибуты-константы

Как и все сущности, символические константы должны быть определены внутри класса. Будем рассматривать константы как атрибуты с фиксированным значением, одинаковым для всех экземпляров класса.

Потомки не могут переопределять значения атрибутов-констант.

Как и другие атрибуты, класс может экспортировать константы или скрывать. Так, если `C` - класс, экспортирующий выше объявленные константы, а у клиента класса к сущности `x` присоединен объект типа `C`, то выражение `x.Backslash` обозначает символ `'\'`.

В отличие от атрибутов-переменных, константы не занимают в памяти места. Их введение не связано с издержками в период выполнения, а потому не страшно, если их в классе достаточно много.

Константы пользовательских классов

Символические константы полезны не только при работе с предопределенными типами, такими как `INTEGER`. Они нужны и тогда, когда их значениями являются объекты классов, созданных разработчиком. В этом случае решение не столь очевидно.

Ключевые концепции

- При любом подходе к конструированию ПО возникает проблема работы с глобальными объектами, совместно используемыми компонентами разных модулей, и инициализируемыми в период выполнения, когда какой-либо из компонентов первым к ним обратился.
- Константы могут быть манифестными и символическими. Первые задаются значениями, синтаксис которых определен так, что значение одновременно описывает и тип константы, а потому является манифестом. Символические константы представлены именами, а их значение указывается в определении константы.
- Манифестные константы базовых типов можно объявлять как константные атрибуты, не требующие памяти в объектах.
- За исключением строк, типы, определенные пользователем, не имеют манифестных констант, нарушающих принципы Скрытия информации и расширяемости.

Контрольные вопросы

1. Для нормального функционирования системы глобальная информация необходима, как?

единые точки подключения к внешним мирам: базам данных, сети, окнам ввода и вывода
не нужна в ОО-системах
источник разделяемых данных
единственный способ обмена данными между модулями

2. В ОО-системах глобальные объекты

не нужны
существуют
могут быть смоделированы
не существуют

3. Константы встроенных типов

могут быть манифестными (заданными значением)
могут быть символическими (именованными)

4. Константы специальных классов

представляют собой обычную функцию, вызывающую процедуру создания
создаются, используя специальные правила, заданные для каждого класса
представляют собой однократную функцию, вызывающую процедуру создания
создаются обычной процедурой создания класса

5. Константы

классы, которым требуются константы, как правило являются клиентами специальных классов с набором констант

собираются вместе в специальных классах, предоставляющих этот набор другим классам
разных классов объявляются в самих классах

классы, которым требуются константы, как правило являются наследниками специальных классов с набором констант

6. Разделяемые объекты

представляют собой однократную функцию, вызывающую процедуру создания

создаются, используя специальные правила, заданные для каждого класса

создаются обычной процедурой создания класса

представляют собой обычную функцию, вызывающую процедуру создания

7. Создание разделяемых объектов и разделяемых констант отличается тем, что?

как константы, так и разделяемые объекты являются константными ссылками

для констант задаются инварианты, запрещающие изменять их значения

ничем не отличается

в отличие от констант значения полей ссылки для разделяемых объектов изменяются в процессе работы

8. Строковые константы

могут использоваться как выражения при передаче аргументов или присваивании

являются такими же константами, как и константы других базовых типов
допускают изменение символов строки
являются разделяемыми объектами

9. Отметьте истинные высказывания

однократные функции могут применяться для моделирования глобальных значений – "системных параметров"
потомки могут переопределять значения атрибутов-констант
константы занимают память, также как и обычные атрибуты
класс не может экспортировать или скрывать константы

10. Отметьте истинные высказывания

манифестные константы базовых типов можно объявлять как константные атрибуты, не требующие памяти в объектах
в ОО-системах не используется глобальный способ передачи информации
строковые константы нарушают принципы скрытия информации
однократные процедуры используются для инициализации свойств

Набрано баллов

Литература

- [1] *Анисимов, А. Е.* Сборник заданий по основам программирования / А. Е. Анисимов, В. В. Пупышев. — Интернет-университет информационных технологий - ИНТУИТ.ру, БИНОМ. Лаборатория знаний, 2006. — С. 352.

Сборник заданий содержит ряд задач, вопросов и проблем по основам программирования. Целью заданий является освоение и закрепление основных идей теоретического и прикладного программирования. Рекомендовано для студентов высших учебных заведений, обучающихся по специальностям в области информационных технологий. Сборник состоит из двух частей - раздела заданий и раздела ответов и решений; задания разделены тематически на 16 глав. В каждой главе авторы ставят перед читателями проблемы определенной области, направления или технологии программирования разного уровня сложности.

- [2] *Мейер, Б.* Объектно-ориентированное конструирование программных систем + CD / Б. Мейер. — Русская Редакция, Интернет-университет информационных технологий - ИНТУИТ.ру, 2005. — С. 1232.

Книга посвящена обоснованию и технологии применения объектного подхода при разработке программных систем. Основное внимание уделяется вопросам качества, повторного использования и расширяемости проектируемых систем. Рассматриваемый объектный подход охватывает весь жизненный цикл разработки - анализ, проектирование, программирование и сопровождение. Книга в первую очередь ориентирова-

на на профессиональных разработчиков программных продуктов, но представляет несомненный интерес для всех, кто изучает и использует объектный подход в программировании. Отдельные разделы могут использоваться в качестве учебника при изучении таких дисциплин, как "Объектно-ориентированное программирование" "Объектно-ориентированный анализ и проектирование".

- [3] *Сузи, Р. А. Язык программирования Python / Р. А. Сузи. — Интернет-университет информационных технологий - ИНТУИТ.ру, БИНОМ. Лаборатория знаний, 2006. — С. 328.*

Изучается язык программирования Python, его основные библиотеки и некоторые приложения. Рекомендовано для студентов высших учебных заведений, обучающихся по специальностям в области информационных технологий. Курс посвящен одному из бурно развивающихся и популярных в настоящее время сценарных языков программирования - Python. Язык Python позволяет быстро создавать как прототипы программных систем, так и сами программные системы, помогает в интеграции программного обеспечения для решения производственных задач. Python имеет богатую стандартную библиотеку и большое количество модулей расширения практически для всех нужд отрасли информационных технологий. Благодаря ясному синтаксису изучение языка не составляет большой проблемы. Написанные на нем программы получаются структурированными по форме, и в них легко проследить логику работы. На примере языка Python рассматриваются такие важные понятия как: объектно-ориентированное программирование, функциональное программирование, событийно-управляемые программы (GUI-приложения), форматы представления данных (Unicode, XML и т.п.). Возможность диалогового режима работы интерпретатора Python позволяет существенно сократить время изучения самого языка и перейти к решению задач в соответствующих предметных областях. Python свободно доступен для многих платформ, а написанные на нем программы обычно переносимы между платформами без изменений. Это обстоятельство позволяет применять для изучения языка любую имеющуюся аппаратную платформу.

Список иллюстраций

- 1.1 Слои в разработке ПО
- 1.2 Уровни в процессе разработки, включающем повторное использование
- 1.3 Устойчивость против корректности
- 1.4 Кривые Осмонда
- 1.5 Распределение расходов на сопровождение
- 2.1 Дублируемое наследование
- 3.1 Декомпозиция
- 3.2 Иерархия нисходящего проектирования
- 3.3 Композиция
- 3.4 Понятность
- 3.5 Непрерывность
- 3.6 Нарушение защищенности
- 3.7 Виды структур межмодульных связей
- 3.8 Канал связи между модулями
- 3.9 Совместное использование данных
- 3.10 Модуль в условиях скрытия информации

4.1 Некоторые возможные реализации таблицы

5.1 Разработка сверху вниз: структура дерева

5.2 Структура простой системы расчета зарплаты

6.1 Три возможных представления стеков

6.2 Представление двух стеков лицом к лицу

6.3 Применение функции put

6.4 Применение функции put

6.5 Манипуляции со стеком

6.6 АТД вид модуля при скрытии информации

7.1 Точка и ее координаты

8.1 Формы и их экземпляры

12.1 Пример множественного наследования

12.2 А это пример дублируемого наследования

12.3 Самолет компании

12.4 Структура множественного и единичного наследования

12.5 Составная фигура - это фигура и список фигур одновременно

13.1 Подпрограмма, клиент и контракт

13.2 Разные облики одной абстракции

14.1 Сравнительные издержки на исправление ошибок

Список таблиц

6.1 Имена операций над стеком

Предметный указатель

АТД, 3, 11

ПО, 77

абстрактный класс, 11

агрегирование, 11

целостность, 91

декомпозиция, 4

динамическое связывание, 11

дублируемое наследование, 8

эффективность, 85

экономичность, 91

функциональность, 88

интерфейс, 4

исключение, 4

класс, 3, 4

коллизия, 18

корректность, 78

лингвистические модульные единицы, 19

метакласс, 4

минимум интерфейсов, 13

множественное наследование, 8

модульная композиция, 6

модульная непрерывность, 9

модульная понятность, 8

модульная защищенность, 11

нарушение типа, 4

наследование, 7

объект, 6

образец, 7

переносимость, 86

переопределение, 10

полиморфизм, 10

пользователь, 77

повторное использование, 83

простое поле, 5

простота использования, 87
прямое отображение, 12
расширяемость, 82
самодокументирование, 20
сериализация, 14
скрытие информации, 17
слабая связность интерфейсов, 14
сопровождение, 95
совместимость, 84
ссылка, 6
статически типизированный язык, 5
своевременность, 90
унифицированный доступ, 20
универсализация, 13
устойчивость, 81
утверждение, 5
верифицируемость, 90
внешний факторами качества, 77
внутренний факторами качества, 78
восстанавливаемость, 91
явные интерфейсы, 15

RAD, 92