4
Подклассы

Описание подкласса

Пример подкласса

Выбор метода

Сообщения псевдопеременной self

Сообщения псевдопеременной super

Абстрактные суперклассы

Системные сообщения для работы с подклассами

Словарь терминов

Каждый объект системы Smalltalk-80 является экземпляром некоторого класса. Все экземпляры одного класса представляют один и тот же тип компонента системы. Например, каждый экземпляр класса Rectangle представляет прямоугольную область, а каждый экземпляр класса Dictionary — множество ассоциативных пар “ключ-значение”. Тот факт, что все экземпляры одного класса представляют один и тот же тип компонента системы, отражается как в способе ответа экземпляра на сообщения, так и в форме переменных экземпляра.

- Все экземпляры одного класса отвечают на один и тот же набор сообщений и используют для ответа на сообщения одни и те же методы.
- Все экземпляры класса имеют одно и то же количество именованных переменных экземпляра и для доступа к ним используют одинаковые имена.
- Объект может иметь индексированные переменные экземпляра, только если все экземпляры его класса имеют индексированные переменные.

Структура класса, как она описывалась до сих пор, ничего не говорила явно о каких-либо пересечениях между классами. Каждый объект системы — экземпляр только одного класса. Эта структура показана на рис. 4.1, где кружки представляют экземпляры класса, а рамки - сами классы. Если кружок лежит внутри рамки, то он представляет экземпляр класса, изображаемого рамкой.

Рис. 4.1.

Отсутствие пересечений между классами — существенное ограничение на разработку в объектно-ориентированной системе, так как оно не допускает какого-либо совместного описания разных классов. Мы могли бы пожелать, чтобы два объекта системы были в основном похожи, но отличались в некоторых деталях. Например, числа с плавающей точкой и целые числа одинаковы своей способностью отвечать на арифметические сообщения, но отличаются способом представления числовых значений. Упорядоченные и неупорядоченные наборы похожи в том, что они представляют накопители элементов, в которые можно добавлять и из которых можно удалять элементы, но они различны по способу доступа к своим элементам. Различие между подобными объектами может быть заметно внешне, например, по тому, что они умеют отвечать на различные сообщения. Различие может быть и чисто внутренним и проявляться в том, что в ответ на одни и те же сообщения похожие объекты выполняют различные методы. Если классам не разрешено пересекаться, то возможность такого частичного подобия не может быть предоставлена системой.

Наиболее общий способ обойти вышеуказанное ограничение — разрешить произвольные пересечения между классами (см. рис. 4.2).

Рис. 4.2.

Мы называем такой подход *множественным наследованием*. Множественное наследование допускает такую ситуацию, при которой одни объекты являются экземплярами двух классов, в то время как другие объекты — экземплярами только одного класса. Менее общий способ обойти ограничение на невозможность пересечения между классами состоит в том, что одному классу разрешается содержать в себе все экземпляры другого класса и не разрешается ничего более общего (см. рис. 4.3). В такой ситуации мы будем говорить, что один класс является *подклассом* другого класса. Этот термин взят из языка Симула, использующего подобный подход.

Рис. 4.3.

Организация подклассов строго иерархическая; если некоторые экземпляры одного класса являются экземплярами другого класса, то все экземпляры первого класса должны быть экземплярами второго класса.

Система Smalltalk-80 поддерживает иерархию классов в форме подклассов. В этой главе описывается, как подклассы определяются в своих суперклассах, как это влияет на связь

между сообщениями и методами, и как механизм подклассов обеспечивает разработку классов системы.

Описание подкласса

Подкласс определяет, что все его экземпляры будут, за исключением явно указанных отличий, такими же, как экземпляры другого класса, называемого его *суперклассом*. В Smalltalk-80 программист всегда создает новый класс как подкласс уже существующего класса. Системный класс Object (Объект) описывает общие свойства *всех* объектов системы, поэтому любой класс системы является как минимум подклассом класса Object. Описание класса (протокол или реализация) явно указывает, чем его экземпляры отличаются от экземпляров суперкласса. Существование подклассов никак не влияет на экземпляры суперкласса.

Подкласс тоже класс, следовательно, он сам может иметь подклассы. Каждый класс имеет один суперкласс, в то время как несколько классов могут быть подклассами одного и того же суперкласса; таким образом, структура классов образует дерево. Класс связан с последовательностью классов, от которых он наследует переменные и методы. Эта цепочка наследования начинается с его непосредственного суперкласса и заканчивается, пока не встретиться класс Object. Object — единственный корневой класс системы; он один не имеет суперкласса.

Напомним, что описание реализации класса состоит из трех основных частей:

1. имени класса;
2. объявления переменных;
3. набора методов.

Подкласс должен указать для себя новое имя класса, но он наследует все переменные и методы суперкласса. Подкласс может объявить для себя новые переменные и добавить новые методы. Если в подклассе объявлены имена новых переменных экземпляра, то экземпляр подкласса будет иметь больше переменных экземпляра, чем экземпляр суперкласса. Если в подклассе добавлены новые общие переменные, то они будут доступны только экземплярам подкласса, но не экземплярам суперкласса. Все добавляемые имена переменных должны отличаться от имен наследуемых переменных.

Если класс не имеет индексированных переменных экземпляра, подкласс может объявить, что его экземпляры будут иметь индексированные переменные; эти индексированные переменные добавляются к наследуемым именованным переменным. Если класс имеет индексированные переменные экземпляра, то все его подклассы тоже *должны* иметь индексированные переменные экземпляра; подклассы могут дополнительно объявить новые именованные переменные.

Если подкласс добавляет метод, чей образец сообщения содержит то же имя сообщения, что и метод суперкласса, экземпляры подкласса будут отвечать на такое сообщение выполнением нового метода. Это называется *переопределением* метода. Если в подкласс добавить метод с именем, отсутствующим в суперклассом, то экземпляры подкласса смогут отвечать на новое сообщение, которое не понимают экземпляры суперкласса.

Итак, любая часть описания реализации может быть модифицирована подклассом:

1. Имя подкласса *должно* быть переопределено на новое.
2. Переменные *могут* быть добавлены.
3. Методы *могут* быть добавлены или переопределены.

Пример подкласса

Описание реализации включает не показанную в предыдущей главе строку, в которой определяется суперкласс. В следующем примере создается подкласс введенного в главе 3 класса FinancialHistory. Экземпляры подкласса наследуют все функции класса FinancialHistory по сохранению информации о денежных доходах и расходах. Они имеют дополнительную функцию, сохраняющую информацию о расходах, облагаемых налогом. Подкласс обязательно задает свое новое имя DeductibleHistory (НалоговыйОтчет), добавляет одну переменную экземпляра и четыре метода. Один из новых методов, initialBalance:, переопределяет метод из суперкласса.

Вот описание класса DeductibleHistory.

class name	DeductibleHistory
superclass	FinancialHistory
instance variable names	deductibleExpenditures

instance methods

transaction recording

```
spendDeductible: amount for: reason
    self spend: amount for: reason.
    deductibleExpenditures ←
      deductibleExpenditures + amount

spend: amount for: reason deducting: deductibleAmount
    self spend: amount for: reason.
    deductibleExpenditures ←
      deductibleExpenditures + deductialeAmount
```

inquiries

```
totalDeductions
    ↑deductibleExpenditures
```

initialization

```
initialBalance: amount
    super initialBalance: amount.
    deductibleExpenditures ← 0
```

Для того, чтобы узнать, какие сообщения понимают экземпляры класса DeductibleHistory, необходимо исследовать протоколы классов DeductibleHistory, FinancialHistory и Object. Экземпляры класса DeductibleHistory имеют четыре переменные — три унаследованы от суперкласса FinancialHistory и одна определена в классе DeductibleHistory. Класс Object не определяет переменных экземпляра.

Рис. 4.4 показывает, что DeductibleHistory есть подкласс FinancialHistory. Каждый прямоугольник на этой диаграмме имеет метку в левом верхнем углу с именем представляемого класса.

Рис. 4.4.

Экземпляры класса DeductibleHistory могут применяться для ведения финансовых записей теми, кто платит налоги (граждане, домовладельцы, бизнесмены). Экземпляры класса FinancialHistory могут применяться для ведения финансовых записей теми организациями, которые не платят налоги (благотворительные, религиозные организации). На самом деле, экземпляр класса DeductibleHistory может быть использован и вместо экземпляра класса FinancialHistory, так как на те же самые сообщения он отвечает тем же способом. В добавление к сообщениям и методам, унаследованным от суперкласса, экземпляр класса DeductibleHistory может работать с сообщениями, указывающими, что все или часть расхо-

дов облагаются налогом. Доступны два новых сообщения: первое spendDeductible:for:, которое посылается, если вся сумма расходов облагается налогом, и второе spend:for:deducting:, которое используется, если только часть суммы облагается налогом. Полную сумму налога можно получить посылая экземпляру класса DeductibleHistory сообщение totalDeductions.

Выбор метода

Когда объекту посылается сообщение, метод с подходящим именем ищется в классе, которому принадлежит получатель сообщения. Если метод там не найден, поиск продолжается в суперклассе и так далее по цепочке суперклассов снизу вверх, пока нужный метод не будет найден. Предположим, что мы посылаем экземпляру класса DeductibleHistory сообщение с именем cashOnHand. Поиск соответствующего метода начинается в классе получателя сообщения, то есть в классе DeductibleHistory. Там нужного метода нет и поиск продолжается в суперклассе класса DeductibleHistory, то есть в классе FinancialHistory. Здесь такой метод есть, поиск прекращается и этот метод выполняется как ответ на посланное сообщение. Выполнение метода возвращает значение переменной экземпляра с именем cashOnHand, которое находится в получателе сообщения, экземпляре класса DeductibleHistory.

Поиск соответствующего метода по цепочке суперклассов продолжается до класса Object. Если нужный метод не найден ни в одном классе, получателю посылается сообщение с именем doesNotUnderstand: и аргументом, содержащим нереализованное сообщение. В классе Object есть специальный метод с таким именем, который информирует программиста о возникшей ситуации.

Допустим, мы послали экземпляру класса DeductibleHistory сообщение с именем spend:for:. Метод для него определен в классе FinancialHistory и имеет следующий вид (см. главу 3):

```
spend: amount for: reason
    expenditures  at: reason
                  put: (self totalSpentFor: reason ) + amount.
    cashOnHand ← cashOnHand – amount
```

Значения переменных экземпляра (expenditures и cashOnHand) находятся в получателе сообщения, экземпляре класса DeductibleHistory. В теле этого метода есть также ссылка на псевдопеременную self; в нем self представляет собой тот экземпляр класса DeductibleHistory, которому было послано первоначальное сообщение.

Сообщения псевдопеременной self

Когда метод содержит сообщение, адресованное self, поиск соответствующего этому сообщению метода всегда начинается с класса получателя, вне зависимости от того, в каком классе был найден метод, содержащий self. Таким образом, когда в методе spend:for:, найденном в классе FinancialHistory, выполняется выражение self totalSpentFor: reason, поиск метода, соответствующего имени сообщения totalSpentFor: начинается в классе self, то есть в классе DeductibleHistory.

Как выполняются сообщения к self, объясним на примере двух классов с именами One и Two. Пусть Two есть подкласс One, а One — подкласс Object. Оба класса имеют метод для ответа на сообщение test. Класс One еще имеет метод для ответа на сообщение result1, который возвращает результат выражения self test.

class name	One
superclass	Object
instance methods	

```
test
    ↑1

result1
    ↑self test
```

```
class name                  Two
superclass                  One

instance methods

test
    ↑2
```

Экземпляр каждого класса будет использован для демонстрации поиска метода для сообщения к self. Пусть example1 будет экземпляром класса One, а example2 — экземпляром класса Two:

```
example1 ← One new.
example2 ← Two new
```

Отношения между классами One и Two показаны на рис. 4.5. В дополнение к меткам прямоугольников, обозначающим имена классов, некоторые кружочки помечены именами соответствующих экземпляров.

Рис. 4.5.

В следующей таблице приведены результаты выполнения различных выражений:

выражение	*результат*
example1 test	1
example1 result1	1
example2 test	2
example2 result1	2

Оба сообщения result1 в двух выражениях вызывают один и тот же метод, определенный в классе One. Результат выражений различен из-за сообщения к self, содержащегося в этом методе. Когда result1 посылается объекту example2, поиск метода начинается с класса Two. Нужного метода в этом классе нет, поэтому поиск продолжается в суперклассе — классе One. Метод для result1 находится в этом классе и состоит из одного выражения ↑self test. Псевдопеременная self ссылается на получателя, example2. Поиск ответа на test начинается в классе, которому принадлежит получатель сообщения result1, то есть в классе Two. Он там обнаруживается, выполняется и возвращает 2.

Сообщения псевдопеременной super

Еще одна псевдопеременная, super (супер), может быть использована в выражениях методов. Она, как и псевдопеременная self, ссылается на получателя сообщения. Однако, когда сообщение посылается super, поиск нужного метода начинается не в классе получателя сообщения. Вместо этого поиск начинается в суперклассе класса, содержащего метод. Применение super позволяет получить доступ к методам суперкласса, переопределенным в подклассах. Действие super не в качестве получателя (например, в качестве аргумента) полностью совпадает с действием self; единственное, на что влияет super — это на класс, с которого начинается поиск метода.

Как будут выполняться сообщения к super, объясним на примере еще двух классов, с именами Three и Four. Пусть Four — подкласс Three, а Three — подкласс ранее определенного класса Two. В классе Four переопределяется метод для сообщения test, а в классе

Three определяются методы для двух новых сообщений: result2 возвращает результат выражения self result1, а result3 возвращает результат выражения super test.

```
class name                    Three
superclass                    Two

instance methods

    result2
        ↑self result1

    result3
        ↑super test


class name                    Four
superclass                    Three

instance methods

    test
        ↑4
```

Все экземпляры классов One, Two, Three и Four могут отвечать на сообщения test и result1. Ответы экземпляров классов Three и Four на посылаемые им сообщения иллюстрируют действие псевдопеременной super (см. рис. 4.6). Определим два новых объекта:

```
example3 ← Three new.
example4 ← Four new
```

Попытка послать сообщение result2 или result3 объектам example1 или example2 вызовет ошибку, так как экземпляры классов One и Two не понимают эти сообщения.

Рис. 4.6.

Следующая таблица показывает результаты посылки различных сообщений:

выражение	*результат*
example3 test	2
example4 result1	4
example3 result2	2
example4 result2	4
example3 result3	2
example4 result3	2

Когда test посылается переменной example3, используется метод из Two, так как Three не переопределяет метод с именем test. Переменная example4 отвечает на resultl четверкой по той же причине, по которой example2 возвращает 2. При посылке result2 к example3 поиск метода начинается с класса Three. Обнаруженный там метод возвращает результат выражения self resutl1. Поиск ответа на сообщение result1 также начинается в классе Three, но нужный метод не находится ни в классе Three, ни в классе Two. Он находится в классе One и возвращает результат self test. Поиск ответа на сообщение test еще раз начинается в классе Three. В этот раз нужный метод обнаруживается в Two, суперклассе класса Three.

Эффект от посылки сообщений к super иллюстрируется ответами объектов example3 и example4 на сообщение result3. Будучи послано к example3, сообщение result3 вызывает поиск метода в классе Three. Найденный там метод возвращает результат выражения super test. Так как сообщение test посылается к super, поиск начинается в не в классе Three, а в его суперклассе Two. Метод test из класса Two возвращает 2. Когда сообщение result3 посылается объекту example4, возвращается 2, несмотря на переопределение классом Four метода test.

Этот пример предостерегает от возможной ошибки: super не означает, что поиск метода начнется в суперклассе получателя сообщения, которым в последнем примере был класс Three. Использование super означает, что поиск начинается в суперклассе того класса, в котором есть метод, содержащий super. Таким классом в примере был Two. Даже если бы класс Three переопределял метод test возвратом тройки, результат example4 result3 был бы 2. Иногда, конечно, суперкласс класса, в котором находится метод, содержащий super, является тем же, что и суперкласс получателя сообщения.

Еще один пример использования super приводится в методе для initialBalance: в классе DeductibleHistory.

```
initialBalance: amount
    super initialBalance: amount.
    deductibleExpenditures ← 0
```

Этот метод переопределяет метод из класса FinancialHistory. Метод в классе DeductibleHistory состоит из двух выражений. Первое передает управление в суперкласс для инициализации баланса:

```
super initialBalance: amount
```

Псевдопеременная super ссылается на получателя сообщения, но указывает, что при поиске метода надо пропустить класс DeductibleHistory и начать с класса FinancialHistory. В этом случае выражения из FinancialHistory не нужно дублировать в DeductibleHistory. Второе выражение метода выполняет специфическую для подкласса инициализацию:

```
deductibleExpenditures ← 0
```

Если бы в методе initialBalance: из класса DeductibleHistory вместо super была бы псевдопеременная self, получилась бы бесконечная рекурсия, так как каждая посылка сообщения initialBalance: вызывала бы снова посылку этого же сообщения.

Абстрактные суперклассы

Абстрактные суперклассы создаются, когда два класса имеют нечто общее в своих описаниях и ни один из них не является подклассом другого. Тогда для этих двух классов создается общий суперкласс, содержащий все общее. Называется он *абстрактным* потому, что создается не для порождения экземпляров. Ситуацию с абстрактным суперклассом представляет рис. 4.7. Заметьте, что абстрактные классы непосредственно не содержат экземпляров.

Рис. 4.7.

В качестве примера абстрактного суперкласса рассмотрим два класса, чьи экземпляры представляют словари. Один класс, с именем SmallDictionary, минимизирует память для хранения своих компонентов; второй класс, с именем FastDictionary, требует больше памяти, но очень быстро находит необходимые данные. Оба класса используют два параллельных списка, один из которых содержит имена, второй — соответствующие им значения. SmallDictionary хранит данные последовательно и использует линейный поиск нужного имени. FastDictionary хранит имена и значения в разреженной памяти, а при поиске имени использует хеширование. За исключением этих различий, оба класса очень похожи: они имеют одинаковый протокол и используют параллельные списки для записи содержимого словаря. Это подобие представляется в абстрактном суперклассе с именем DualListDictionary. Взаимоотношения между классами показаны на рис. 4.8.

Рис. 4.8.

Описание реализации абстрактного класса DualListDictionary приведено ниже.

class name DualListDictionary
superclass Object
instance variable names names
values

instance methods

accessing

```
at: name
    | index |
    index ← self indexOf: name.
    index = 0
      ifTrue: [self error: 'Name not found']
      ifFalse: [↑values at: index]

at: name put: value
    | index |
    index ← self indexOf: name.
    index = 0
      ifTrue: [index ← self newIndexOf: name].
    ↑values at: index put: value
```

testing

```
includes: name
    ↑(self indexOf: name) ~= 0

isEmpty
    ↑self size = 0
```

initialization

```
initialize
    names ← Array new: 0.
    values ← Array new: 0
```

Это описание класса DualListDictionary использует сообщения, либо определенные в самом классе DualListDictionary, либо введенные ранее в этой главе или в предыдущих главах. Внешний протокол класса DualListDictionary состоит из сообщений at:, at:put:, isEmpty, initialize. Новый экземпляр класса DualListDictionary (на самом деле, его подкласса) создается посылкой ему сообщения new. Затем экземпляру посылается сообщение initialize для того, чтобы провести инициализацию двух переменных экземпляра; изначально это пустые массивы (Array new: 0).

Три сообщения к self, использованные в этих методах, не реализованы в классе DualListDictionary — это сообщения size, indexOf: и newIndexOf:. По этой причине DualListDictionary и называется абстрактным. Если бы был создан его экземпляр, последний не мог бы отвечать на все необходимые сообщения. Два подкласса, SmallDictionary и FastDictionary, должны реализовать методы для этих сообщений. Тот факт, что поиск всегда начинается с класса, на который указывает self, означает, что в суперклассе могут быть посылки сообщений к self, но соответствующие этим сообщениям методы будут находиться в подклассе. Таким образом, суперкласс может задавать основу метода, которая или уточняется, или фактически реализуется подклассом.

SmallDictionary — подкласс класса DualListDictionary, использующий минимальное количество памяти для представления ассоциативных пар, но тратящий много времени для их поиска. Он содержит методы для трех сообщений, не реализованных в DualListDictionary — для size, indexOf: и newIndexOf:. Новых переменных экземпляра этот класс не вводит.

class name SmallDictionary
superclass DualListDictionary

instance methods

```
accessing
    size
        ↑names size

private
    indexOf: name
        1 to: names size do:
            [:index | (names at: index) = name ifTrue: [↑index]].
        ↑0

    newIndexOf: name
        self grow.
        names at: names size put: name.
        ↑names size

    grow
        | oldNames oldValues |
        oldNames ← names.
        oldValues ← values.
        names ← Array new: names size + 1.
        values ← Array new: values size + 1.
        names replaceFrom: 1 to: oldNames size with: oldNames.
        values replaceFrom: 1 to: oldValues size with: oldValues
```

Так как имена записываются последовательно, размер экземпляра класса SmallDictionary совпадает с размером его массива имен names. Индекс выбранного имени определяется линейным поиском в массиве names. Если имя не находится, возвращается индекс 0, сообщая о неудачном завершении поиска. Когда в словарь нужно добавить новый элемент, для поиска подходящего индекса применяется метод newIndexOf:. Он предполагает, что размеры массивов names и values достаточны только для хранящихся в них данных и в них нет свободного пространства для добавления нового элемента. Сообщение grow создает два новых массива — копии предыдущих, но с одним лишним местом для элемента в конце каждого массива. В методе newIndexOf: вначале увеличиваются размеры массивов names и values, а затем в последнюю позицию массива names записывается новое имя. За сохранение значения value в словаре отвечает метод, пославший сообщение newIndexOf:.

Для примера мы можем выполнить следующие выражения:

выражение	*результат*
ages ← SmallDictionary new	новый, неинициализированный экземпляр
ages initialize	экземпляр с инициализированными переменными
ages isEmpty	true
ages at: 'Brett' put: 3	3
ages at: 'Dave' put: 30	30
ages includes: 'Sam'	false
ages includes 'Brett'	true
ages size	2
ages at: 'Dave'	30

Для каждого из приведенных выше выражений мы покажем, в каком классе каждое сообщение находит нужный ему метод и в каком классе обнаруживаются методы для сообщений, адресованных self.

имя сообщения	*сообщение к* self	*класс метода*
initialize		DualListDictionary
at:put:		DualListDictionary

	indexOf:	SmallDictionary
	newIndexOf:	SmallDictionary
includes:		DualListDictionary
	indexOf:	SmallDictionary
size		SmallDictionary
at:		DualListDictionary
	indexOf:	SmallDictionary
	error	Object

Класс FastDictionary — другой подкласс класса DualListDictionary. Он использует хеширование для поиска имен. Хеширование требует больше памяти, но действует быстрее, чем линейный поиск. Все объекты системы отвечают на сообщение hash, возвращая некоторое целое число. Все целые числа отвечают на сообщение \\, возвращая остаток от деления получателя на аргумент сообщения.

class name FastDictionary
superclass DualListDictionary

instance methods

accessing

size

```
| size |
size ← 0.
names do: [:name | name notNil ifTrue: [size ← size + 1]].
↑size
```

initialization

initialize

```
names ← Array new: 4.
values ← Array new: 4
```

private

indexOf: name

```
| index |
index ← name hash \\ names size + 1.
[(names at: index) = name]
   whileFalse: [(names at: index) isNil
                  ifTrue: [↑0]
                ifFalse: [index ← index \\ names size + 1]].
↑index
```

newIndexOf: name

```
| index |
names size – self size <= (names size / 4)
   ifTrue: [self grow].
index ← name hash \\ names size + 1.
[(names at: index) isNil]
   whileFalse: [index ← index \\ names size + 1].
names at: index put: name.
↑index
```

grow

```
| oldNames oldValues |
oldNames ← names.
oldValues ← values.
names ← Array new: names size * 2.
values ← Array new: values size * 2.
```

```
1 to: oldNames size do:
  [:index | (oldName at: index) isNil
              ifFalse: [self at: (oldNames at: index)
                          put: (oldValues at: index)]]
```

Класс FastDictionary переопределяет реализованный в классе DualListDictionary метод initialize для того, чтобы создавать массивы, которые уже содержат некоторое место для размещения элементов (Array new: 4). Размер экземпляра класса FastDictionary не равен размеру одной из своих переменных экземпляра, так как последние содержат пустые элементы. Поэтому его размер определяется подсчетом элементов, отличных от nil.

Реализация newIndexOf: в классе FastDictionary в основном похожа на реализацию этого сообщения в классе SmallDictionary, за исключением того, что когда размер массива изменяется (в данном случае удваивается в методе grow), каждый элемент из старого массива явно копируется в новый массив, так что элементы словаря хешируются заново. При добавлении элементов размер экземпляра класса FastDictionary не обязательно должен изменяться, как это было для экземпляра класса SmallDictionary. Он изменяется только тогда, когда количество пустых ячеек понизится до минимума, который составляет 25% от общего количества элементов. Эта информация содержится в выражении

```
names size – self size <= (names size / 4)
```

Системные сообщения для работы с подклассами

В соответствии с принятым стилем программирования, метод не должен содержать сообщений к self, если эти сообщения не реализуются классом или не наследуются из суперклассов. В описании класса DualListDictionary существуют три таких сообщения: size, indexOf: и newIndexOf:. Как мы увидим в следующих главах, способность отвечать на сообщение size наследуется из класса Object; ответом будет число индексированных переменных экземпляра. Предполагается, что подкласс класса DualListDictionary будет переопределять этот метод для того, чтобы возвращать число слов в словаре.

В классе Object определено специальное сообщение subclassResponsibility (реализованоВПодклассе). Его следует применять при реализации сообщений, которые не могут быть должным образом реализованы в абстрактном классе. То есть, по соглашениям языка Smalltalk-80, реализация сообщений size, indexOf: и newIndexOf: в классе DualListDictionary должна выглядеть так:

```
self subclassResponsibility
```

В ответ на это сообщение вызывается метод, определенный в классе Object:

```
subclassResponsibility
    self error: 'My subclass should have overridden one of my messages.'
```
[1]

Таким образом, если метод должен быть реализован в подклассе абстрактного класса, то такое сообщение об ошибке указывает программисту, как исправить программу. Более того, используя это сообщение, программист может создавать абстрактные классы, в которых реализуются все сообщения к псевдопеременной self, и в которых программисту напоминается, какие методы должны быть переопределены в подклассах абстрактного класса.

Если программист решит, что сообщение, наследуемое из абстрактного суперкласса, в действительности не должно быть реализовано в некотором его подклассе, то, по соглашению, чтобы переопределить этот наследуемый метод в данном подклассе используют выражение

```
self shouldNotImplement
```

[1] 'Мой подкласс должен переопределить одно из моих сообщений'

В ответ на это сообщение вызывается метод, определенный в классе Object:

```
shouldNotImplement
    self error: 'This message is not appropriate for this object.'
```
[2]

В Smalltalk-80 есть несколько основных иерархий подклассов, использующих идею создания системы сообщений, реализация которых должна быть завершена в подклассах. Например, это классы, описывающие различные виды наборов (см. главы 9, 10). Классы наборов иерархически упорядочены для того, чтобы использовать как можно больше общего между классами, описывающими различные виды наборов. Они используют сообщения shouldNotImplement и subclassResponsibility. Другие примеры использования подклассов — иерархия классов линейно упорядоченных величин и чисел (главы 7, 8).

Словарь терминов

подкласс	класс, наследующий переменные и методы от существующего класса.
суперкласс	класс, у которого наследуются переменные и методы.
Object	системный класс, корень дерева иерархии классов.
переопределение метода	определение в подклассе метода, отвечающего на то же сообщение, что и метод в суперклассе.
super	псевдопеременная, указывающая на получателя сообщения; отличается от self тем, где начинается поиск необходимого метода.
абстрактный класс	класс, определяющий протокол, но не реализующий его в полной мере; по соглашению, экземпляры такого класса создаваться не должны.
SubclassResponsibility	сообщение об ошибке, указывающее на то, что необходимый метод должен быть реализован в подклассе.
ShouldNotImplement	сообщение об ошибке, указывающее на то, что метод унаследован от суперкласса, но он явно не доступен экземплярам этого подкласса.

[2] 'Сообщение не подходит для этого объекта.'