Русскоязычный перевод документации Eiffel по технологии проектирования по контракту (Design by Contract)

*Аннотация.* Широко распространенным способом тестирования программных компонент является выполнение юнит-тестов. Юнит-тесты описывают набор шагов, которые необходимо выполнить, для получения необходимого результата. Однако юнит-тесты трудно писать и поддерживать в актуальном состоянии, отсутствие декларативности и интеграции в код затрудняет понимание спецификации программного компонента, объём кода юнит-тестов, как правило, достаточно велик. Этих недостатков лишены контракты, которые накладывают ограничения и обязательства на компоненты класса. Контракты являются частью документации программной системы, позволяют легко тестировать отдельные компоненты, упрощают повторное использование и отладку. Проектирование по контракту изначально поддерживается в языке Eiffel как на уровне инструментов среды программирования EiffelStudio, так и во всех стандартных библиотеках, поставляемых с этой средой. Систематическое применение проектирования по контракту позволяет упростить проектирование программных систем, сократить время выявления ошибок, повысить качество кода и надежность разработанного ПО.

# Проектирование по контракту, утверждения и исключения

## *Design by Contract (tm), Assertions and Exceptions*

Перевод: Семченков Сергей
Редактор перевода: Когтенков Александр

# Содержание

В Eiffel непосредственно реализованы идеи проектирования по контракту, улучшающие надежность ПО и предоставляющие научную основу для спецификации ПО, документирования и тестирования, а также для обработки исключений и правильного использования наследования.

# 1. Основы проектирования по контракту

Система (в частности, программная система, однако идеи более общие) состоит из определенного количества взаимодействующих компонент. Проектирование по контракту определяет тот факт, что их взаимодействие должно быть основано на точных спецификациях – контрактах – описывающих ожидания и гарантии каждой стороны.

Контракт в Eiffel похож на контракт в реальной жизни между двумя людьми или компаниями, который удобно выразить в форме таблицы, содержащей ожидания и гарантии. Рассмотрим пример того, как можно описать в общих чертах контракт между домовладельцем и телефонной компанией.

| Предоставление телефонных услуг | **Обязательства** | **Выгоды** |
|---|---|---|
| **Клиент** | (Удовлетворение предусловия):<br>Оплата счета | (Из постусловия):<br>Получение телефонных услуг от поставщика |
| **Поставщик** | (Удовлетворение предусловия):<br>Предоставление телефонных услуг | (Из постусловия):<br>Нет необходимости в предоставлении чего-либо при неоплате счета |

Обратите внимание на то, как обязательства каждой стороны отображаются на выгоды другой. Это будет общим правилом.

Обязательство клиента, которое защищает поставщика, называется **предусловием**. Оно определяет те условия, которым должен удовлетворять клиент перед запросом определенных сервисов. Выгода клиента, описывающая, что поставщик должен выполнить (предполагается, что предусловие удовлетворено), называется **постусловием**.

В дополнение к предусловиям и постусловиям контракты класса включают **инварианты класса**, применимые к классу в целом. Более точно можно сказать, что выполнение инварианта класса должно быть проверено каждой процедурой создания (или при инициализации по умолчанию, если процедура создания отсутствует) и поддерживаться каждой экспортируемой подпрограммой класса.

# 2. Выражение утверждений

Синтаксис Eiffel позволяет выражать предусловия (**require**), постусловия (**ensure**),

инварианты класса (**invariant**), а также другие утверждения, изучаемые далее (см. раздел "Операторы"): варианты и инварианты цикла, операторы проверки.

Рассмотрим обновленный класс ACCOUNT, содержащий больше утверждений (первоначальную версию этого класса см. здесь – прим. перев.).

```
note
    description: "Простые банковские счета"

class
    ACCOUNT

create
    make

feature {NONE} -- Инициализация
    make -- Инициализировать
        do
            create all_deposits
        end

feature -- Доступ

    balance: INTEGER
            -- Текущий баланс

    deposit_count: INTEGER
            -- Количество сделанных взносов с момента открытия
        do
            Result := all_deposits.count
        end

feature -- Изменение элемента

    deposit (sum: INTEGER)
            -- Добавить `sum' на счет.
        require
            non_negative: sum >= 0
        do
            all_deposits.extend (sum)
            balance := balance + sum
        ensure
```

```
            one_more_deposit: deposit_count = old deposit_count + 1
            updated: balance = old balance + sum
        end

feature {NONE} -- Реализация

    all_deposits: DEPOSIT_LIST
            -- Список взносов с момента открытия счета

invariant
    consistent_balance: balance = all_deposits.total
    zero_if_no_deposits: all_deposits.is_empty implies (balance = 0)

end -- класс ACCOUNT
```

Любое утверждение состоит из одного или более подвыражений, каждое из которых представляет собой логическое выражение (с дополнительной конструкцией **old**). Эффект от нескольких подвыражений, как в постусловии компонента deposit и инварианте, такой же, как и от объединения их с помощью **and**. Каждому выражению может предшествовать метка, такая как consistent_balance в инварианте, и двоеточие; метка является необязательной и не влияет на семантику утверждений, за исключением сообщений об ошибках (это объяснено в следующем разделе), однако их систематическое использование является рекомендованным стилем. Логическое выражение a **implies** b принимает значение истина, за исключением случая, когда a имеет значение истина, а b – ложь.

Так как утверждения используют полноценные логические выражения, они могут включать вызовы функций. Это делает возможным выражение сложных условий согласованности, таких как “граф не содержит циклов”, которые не могут быть иначе выражены через простые выражения или исчисление предикатов первого порядка, но которые просто реализовать как функции Eiffel, возвращающие результаты логического типа.

## 2.1 Предусловия

Предусловие подпрограммы выражает условия, накладываемые подпрограммой на своих клиентов. Вызов deposit является допустимым тогда и только тогда, когда значение аргумента неотрицательно. Подпрограмма не гарантирует ничего для вызова, не удовлетворяющего предусловию. Частью методологии Eiffel является тот факт, что телу подпрограммы **никогда** не следует проверять предусловие, так как его обеспечение является обязанностью клиента. Кажущийся (т.к. не будет являться таковым к концу этого обсуждения) парадокс проектирования по контракту, отраженный в нижней правой ячейке предыдущей и следующей таблиц, заключается в том, что возможно получить

более надежное ПО с меньшим количеством явных проверок в тексте программы.

## 2.2 Постусловия

Постусловие подпрограммы выражает то, что подпрограмма гарантирует своим клиентам при вызовах, удовлетворяющих предусловию. Запись **old** expression, допустимая только в постусловиях (выражение **ensure**), обозначает значение, которое имело выражение expression при входе в подпрограмму.

Предусловие и постусловие устанавливают условия контракта между подпрограммой и ее клиентами аналогично предыдущему примеру контракта между людьми.

| deposit | **Обязательства** | **Выгоды** |
|---|---|---|
| **Клиент** | (Удовлетворение предусловия): Использовать неотрицательный аргумент | (Из постусловия): Получение обновленного списка взносов и баланса счета |
| **Поставщик** | (Удовлетворение предусловия): Обновить список взносов и баланс | (Из постусловия): Нет необходимости в контроле отрицательных аргументов |

## 2.3 Инварианты класса

Как было отмечено, инвариант класса применим ко всем компонентам. Он должен выполняться при выходе из процедуры создания, и неявно добавляется к предусловию и постусловию каждой экспортируемой подпрограммы. В этом отношении есть как хорошие, так и плохие новости для создателя подпрограммы: хорошая новость заключается в том, что первоначально объект будет находиться в стабильном состоянии, предотвращая в примере необходимость проверки того, что весь список all_deposits совместим с balance; плохие новости заключаются в том, что помимо официального контракта, выраженного в конкретном постусловии, каждая подпрограмма должна позаботиться о восстановлении инварианта при выходе из нее.

Требование осмысленных контрактов заключается в том, что они должны быть честными, то есть обеспечиваться честными партнерами. Это подразумевает правило согласованности: если подпрограмма экспортируется клиенту (всем или выборочно), каждый компонент, появившийся в ее предусловии, также должен быть доступен этому клиенту. В противном случае, например, если предусловие включает условие **require** n > 0, где n – закрытый атрибут, то поставщик мог бы создать такие требования, которые честный клиент не смог бы проверить.

Стоит отметить, что выполнение предусловия для клиента не означает его проверки. Полагая, что n экспортируется, следующий вызов проверяет предусловие с

возможной частью **else**:

```
if x.n > 0 then
    x.r
end
```

Однако если в контексте вызова в коде клиента подразумевается, что n является положительным (например, предшествующий вызов присвоил n сумму квадратных корней), то нет необходимости в **if** или подобной конструкции.

**Примечание.** В тех случаях, когда причина пропуска проверки нетривиальна, рекомендуется использовать оператор **check** (см. раздел "Операторы").

# 3. Использование контрактов для обеспечения надежности

Для чего лучше использовать контракты? Первое использование чисто методологическое. Соблюдая как можно более точно правила выражения логических предположений, стоящих за программными элементами, Вы можете изначально обеспечить надежность: ПО разрабатывается одновременно с логическими обоснованиями для обеспечения корректности.

Это простое наблюдение, неочевидное до тех пор, пока Вы не практиковались в проектировании по контракту при разработке большого проекта, влечёт столько же изменений в практике программирования и качестве, как и остальная часть объектной технологии.

# 4. Мониторинг утверждений во время выполнения

Контракты в Eiffel – это не только желаемое мышление. Они могут отслеживаться во время выполнения под управлением опций компиляции.

Из предшествующего описания должно быть ясно, что контракты не являются механизмом тестирования специальных условий, например ошибочного ввода пользователя. Для этих целей доступны обычные структуры управления (**if** deposit_sum > 0 **then** ...), дополненные в применимых случаях механизмом обработки исключений. Утверждение является **условием корректности**, контролирующим взаимосвязь между двумя программными модулями (не программным модулем и человеком и не программным модулем и внешним устройством). Если аргумент sum отрицателен на входе в подпрограмму deposit, то виноват в этом другой программный элемент, автор которого был недостаточно внимателен при соблюдении условий договора.

**Правило: Нарушение утверждения.** Нарушение утверждения во время выполнения

является проявлением программной ошибки (багом).

Более точно:

- нарушение предусловия предупреждает о баге на стороне клиента, который не соблюдает свою часть договора;
- нарушение постусловия (или инварианта) предупреждает о баге на стороне поставщика – подпрограмма не выполняет свою работу.

Эти нарушения, служащие признаком багов, объясняют, почему допустимо включение или отключение отслеживания утверждений всего лишь через опции компиляции: для корректной системы без багов утверждения всегда будут соблюдаться, опция компиляции не вносит изменений в семантику системы.

Однако для некорректной системы наилучшим способом локализации ошибки или проверки её наличия является отслеживание утверждений во время разработки и тестирования. Приведем список опций компиляции, которые EiffelStudio позволяет установить независимо для каждого класса со значениями по умолчанию для системного уровня и уровня кластеров:

- no – утверждения не отслеживаются во время выполнения;
- **require** – отслеживание только предусловий на входе в подпрограмму;
- **ensure** – отслеживание предусловий на входе, постусловий на выходе;
- **invariant** – то же, что и **ensure**, плюс инвариант класса на входе и выходе для квалифицированных вызовов;
- **all** – то же, что и **invariant**, плюс операторы **check**, варианты и инварианты циклов.

Нарушение утверждения во время выполнения при любой опции компиляции, за исключением первой, вызовет исключение (см. раздел "Обработка исключений"). Если ПО не содержит явный план "возобновления", как объяснено в обсуждении исключений, нарушение приведёт к трассировке исключения и вызовет завершение (или в EiffelStudio возврат к просмотру и отладке в точке сбоя). Для помощи в определении проблемы будет показана метка нарушенного подвыражения, если она присутствует.

Значением опции отслеживания утверждений по умолчанию является **require**. Это вызвано тем, что Eiffel настаивает на повторном использовании: для таких библиотек, как EiffelBase, содержащих значительное количество предусловий, выражающих правила использования. Ошибка в **клиентском ПО** часто приводит к нарушению одного из этих предусловий, например через некорректный аргумент. Несколько парадоксальный вывод заключается в том, что даже разработчик приложений, который сам не слишком хорошо применяет этот метод (за исключением невнимательности, спешки, безразличия или игнорирования), получит выгоду от наличия контрактов в коде чужой библиотеки.

Во время разработки и тестирования мониторинг утверждений должен быть выставлен на самом высоком уровне. Совместно со статической типизацией и технологией незамедлительной компиляции "Melting Ice" (дословно "Тающий лед" - прим. перев.) это позволяет добиться процесса разработки, описанного в разделе "Качество и функциональность", при котором ошибки искореняются как только появляются. Любой,

кто не практиковал этот метод в реальном проекте, не может представить, как много ошибок найдено этим способом; на удивление часто нарушение происходит в утверждениях, включенных на всякий случай, когда разработчик убеждён, что они никогда не дадут сбоя.

Предоставляя точное описание того, что ПО должно выполнять, в сравнении с реальным положением дел (что ПО делает в настоящий момент), проектирование по контракту сильно изменяет процессы отладки, тестирования и управления качеством.

В процессе выпуска финальной версии системы обычно выключают мониторинг или снижают его уровень до **require**. Точное правило зависит от обстоятельств, это компромисс между соображениями эффективности, потенциальной ценой ошибок, а также насколько разработчики и команда проверки качества доверяют продукту. При разработке ПО Вы должны всегда полагать (чтобы не ослабить бдительность), что в конце мониторинг будет выключен.

# 5. Контрактная форма класса

Другое приложение утверждений - документация. При щелчке по иконке Contract Form (Контракт) в EiffelStudio механизмы среды сгенерируют из текста класса абстрактную версию, которая включает только информацию необходимую авторам клиентов. Приведем контрактную форму класса ACCOUNT, рассмотренного выше.

```
note
    description: "Простые банковские счета"

class interface
    ACCOUNT

feature -- Доступ

    balance: INTEGER
            -- Текущий баланс

    deposit_count: INTEGER
            -- Количество сделанных взносов с момента открытия

feature -- Изменение элемента

    deposit (sum: INTEGER)
            -- Добавить `sum' на счет.
        require
            non_negative: sum >= 0
        ensure
```

```
        one_more_deposit: deposit_count = old deposit_count + 1
        updated: balance = old balance + sum

end -- интерфейс класса ACCOUNT
```

Слова **class** interface используются вместо обычного **class** для того, чтобы избежать путаницы с текущим текстом программы на Eiffel, так как это документация, а не исполняемое ПО. Фактически возможно сгенерировать компилируемый вариант контрактной формы в виде отложенного класса. По сравнению с полным текстом, контрактная форма класса (также называемая "краткой формой"), сохраняет все свойства интерфейса, необходимые авторам клиентов:

- имена и сигнатуры (информация об аргументах и типе результата) для экспортируемых компонентов;
- заголовочные комментарии этих компонентов, которые содержат неформальное определение их предназначения (отсюда, как отмечено в разделе "Hello World", следует важность постоянного включения этих комментариев и необходимость их тщательного написания);
- предусловия и постусловия этих компонентов (как минимум подвыражения, включающие только экспортируемые функции);
- инвариант класса (аналогично предыдущему).

Следующие элементы не входят в контрактную форму класса: любая информация о неэкспортируемых компонентах; тела всех подпрограмм (оператор **do** или варианты **external** и **once**, описанные в разделах "Внешнее ПО" и "Однократные подпрограммы и разделяемые объекты"); подвыражения утверждений, включающие неэкспортируемые функции; некоторые ключевые слова, не используемые в документации.

В соответствии с принципом унифицированного доступа (описанном в разделе "Объекты, поля, значения и ссылки") контрактная форма не проводит различие между атрибутами и запросами без аргументов. В приведенном примере balance может быть как тем, так и другим, так как для клиентов нет разницы, за исключением, возможно, производительности.

Контрактная форма является базисным инструментом для использования классов поставщика в методе Eiffel. Она позволяет авторам клиентов повторно использовать программные элементы без необходимости чтения их исходных кодов. Это ключевое требование в крупномасштабных промышленных разработках.

Контрактная форма удовлетворяет двум ключевым требованиям хорошей программной документации.

- Она является по-настоящему абстрактной, свободной от деталей реализации того, что она описывает и сконцентрированной на функциональности.
- Вместо того, чтобы разрабатываться независимо – нереалистичное требование, которое трудно навязать первоначально разработчикам и невыполнимое на практике, если мы ожидаем, что документация будет оставаться актуальной в процессе развития ПО – документация извлекается непосредственно из ПО. Она

не является отдельным продуктом, это другой вид того же самого продукта. Это продолжает принцип **единого продукта**, который лежит в основе бесшовной модели разработки Eiffel (описанной в разделе "Программный процесс в Eiffel").

Контрактная форма – это только один из способов представления. EiffelStudio, например, генерирует графическое представление системных структур для того, чтобы показать классы и их взаимосвязи – быть клиентом, наследование – в соответствии с соглашениями нотации BON (the Business Object Notation – нотация бизнес объектов). В соответствии с принципами бесшовности и обратимости, EiffelStudio позволяет одновременно работать с текстом, на лету генерируя графическое представление, или работать над графическим представлением, обновляя текст на лету, Вы можете выбрать по желанию один из двух режимов. Результирующий процесс полностью отличается от традиционных подходов, основанных на отдельных инструментах: инструментальные средства анализа и системы автоматизированной разработки программ CASE, основанные на UML, имеют дело с первоначальным описанием в виде “кружков и стрелок” ("bubble-and-arrow"), а также с отдельной средой программирования только для реализации. В Eiffel среда предоставляет целостную, бесшовную поддержку с начала до конца разработки.

Контрактная форма, или плоская контрактная форма, которая учитывает наследование (см. раздел "Плоские и плоские контрактные формы"), являются стандартной формой документации библиотек, широко применялась, например, в книге "Повторно используемое ПО" (см. список литературы). Утверждения играют главную роль в этой документации, выражая условия контракта. Как продемонстрировало падение ракеты Ariane-5 в июне 1996 года (стоимость убытков 500 млн. долларов), связанное с неправильным использованием программного модуля от ракеты Ariane-4, **повторное использование без контрактной документации** – это путь к катастрофе. Отсутствие повторного использования в этом случае было бы более предпочтительным.

# 6. Обработка исключений

Еще одно применение проектирования по контракту касается обработки непредвиденных ситуаций. Неопределенность рассуждений на эту тему следует из-за отсутствия точного определения понятия "исключение". Использование проектирования по контракту позволяет быть более точным.

- Каждая подпрограмма имеет контракт, который необходимо выполнить.
- Тело подпрограммы определяет способ достижения контракта – последовательность операций или других структур управления, включающих операции. Некоторые из этих операций вызывают подпрограммы со своими контрактами, однако даже атомарная арифметическая операция имеет неявный контракт, заключающийся в том, что результат можно будет представить.
- Каждая из этих операций может дать сбой, то есть не выполнить свой контракт, например, арифметическая операция может сгенерировать переполнение (или непредставимый результат).
- Сбой операции является **исключением** в подпрограмме, содержащей эту опера-

цию.

- Подпрограмма может также дать сбой, приводя к исключению в вызывающей подпрограмме.

Стоит отметить точное определение двух ключевых понятий: сбой и исключение. Несмотря на то, что сбой является более базовым понятием, так как он определен для атомарной операции, не вызывающей подпрограмму, определения являются взаимно рекурсивными, так как исключение может привести к сбою подпрограммы-получателя, а сбой подпрограммы приводит к исключению в вызывающей подпрограмме.

Почему исключение "может" привести к сбою? Конечно, потому, что можно "восстановить" подпрограмму от сбоя в случае исключения, включив в нее оператор с ключевым словом **rescue**, как в следующем примере:

```
read_next_character (f: FILE)
        -- Прочитать следующий символ в переменную last_character.
        -- При невозможности присвоить флагу сбоя failed значение True.
    require
        readable: file.readable
    local
        impossible: BOOLEAN
    do
        if impossible then
            failed := True
        else
            last_character := low_level_read_function (f)
        end
    rescue
        impossible := True
        retry
    end
```

Этот пример включает две конструкции, которые необходимы для обработки исключений: **rescue** и **retry**. Инструкция **retry** допустима только в выражении **rescue**; ее эффект заключается в повторном запуске подпрограммы без повторной инициализации локальных сущностей (таких, как impossible в примере, которая была инициализирована значением False при первом входе). Полагаем, что компоненты failed и last_character являются атрибутами включающего класса.

Этот пример является типичным использованием исключений: в качестве последнего средства для ситуаций, которые не должны происходить. Подпрограмма имеет предусловие, file.readable, которое проверяет существование файла и наличие доступа для чтения символов. Таким образом, клиенты должны проверить, что все в порядке перед вызовом подпрограммы. Несмотря на то, что эта проверка почти всегда гарантирует успех, редкое сочетание обстоятельств может послужить причиной изменения статуса файла (потому что пользователь или другая система манипулируют файлом) между проверкой readable и вызовом low_level_read_function. Полагая, что

последняя функция даст сбой в том случае, если файл нечитаем, мы должны обработать исключение.

Получим следующий вариант:

```
local
    attempts: INTEGER
do
    if attempts < Max_attempts then
        last_character := low_level_read_function (f)
    else
        failed := True
    end
rescue
    attempts := attempts + 1
    retry
end
```

В этом варианте будет происходить Max_attempts попыток перед отказом.

Рассмотренная подпрограмма в любом варианте никогда не даст сбой: она всегда выполняет свой контракт, в котором указано, что она должна прочесть символ или установить флаг failed в случае невозможности сделать это. Для сравнения рассмотрим новый вариант:

```
local
    attempts: INTEGER
do
    last_character := low_level_read_function (f)
rescue
    attempts := attempts + 1
    if attempts < Max_attempts then
        retry
    end
end
```

Этот вариант не содержит флаг failed. В этом случае после Max_attempts неудачных попыток подпрограмма выполнит выражение **rescue** без **retry** (так как **if** не имеет части **else**). Так происходит **сбой** подпрограммы. Как было отмечено, исключение в подпрограмме будет передано вызывающей подпрограмме.

Оператор **rescue** перед завершением должен восстановить инвариант класса, чтобы вызывающая подпрограмма и, возможно, **retry** на более высоком уровне вызовов получили объекты в согласованном состоянии. Как результат, правило для отсутствующего выражения **rescue** – этот случай для подавляющего большинства подпрограмм в большинстве систем – эквивалентен следующему:

```
rescue
    default_rescue
```

где процедура default_rescue приходит из класса ANY, в котором она описана как ничего не выполняющая; но в системах, ориентированных на надежность, классы, подверженные неявно восстанавливаемым исключениям, должны переопределять default_rescue (возможно, используя процедуру создания с теми же формальными требованиями) таким образом, чтобы всегда восстанавливать инвариант.

В основе схемы обработки исключений в Eiffel лежит следующий принцип, являющийся на первый взгляд банальным, но нарушаемый многими существующими механизмами: подпрограмма должна **успешно завершиться или дать сбой**. Это, в свою, очередь, является результатом принципов проектирования по контракту: успешность означает выполнение контракта, возможно после одного или нескольких **retry**; сбой – это другой случай, при котором должно быть вызвано исключение в вызывающей подпрограмме. В противном случае подпрограмма могла бы проигнорировать свой контракт и вернуть вызывающей подпрограмме кажущееся нормальное состояние. Это наихудший случай обработки исключений.

Исключения могут быть результатом следующих событий.

- Сбой подпрограммы (выражение **rescue** выполнилось до конца без **retry**), как только что было рассмотрено.
- Нарушение утверждения, если запуск программы происходит с включенным мониторингом.
- Исключение разработчика, см. далее.
- Сигнал ОС: арифметическое переполнение, отсутствие свободной памяти для запрашиваемого создания или клонирования, даже если сборщик мусора предпринял все возможное для поиска свободного места. В этот пункт не включаются ошибки вида "неправильный адрес указателя" (как в языках C/C++), так как эти ошибки не встречаются вследствие строгой статической типизации в Eiffel.

При обработке исключений в выражении **rescue** иногда бывает полезным выяснить точный вид исключения, которое произошло. Для этого достаточно наследовать от класса EXCEPTIONS из библиотеки Kernel, которая предоставляет такие запросы, как exception, возвращающий код последнего исключения, и символьные имена (см. раздел "Константные атрибуты") для всех кодов, таких как No_more_memory. Затем Вы можете обрабатывать различные исключения по-разному, проверяя exception на разные возможности. Метод настоятельно рекомендует, чтобы коды исключений оставались простыми; сложный алгоритм в выражении **rescue** является признаком неправильного использования механизма исключений. Класс EXCEPTIONS предоставляет различные средства для тонкой настройки возможностей исключений, таких как процедура raise, которая явным образом вызывает "исключение разработчика" с кодом, который может быть обнаружен и обработан. Обработка исключений позволяет разрабатывать ПО на Eiffel, которое не только корректно, но и надежно, планируя случаи, которые не должны возникать при нормальных условиях, однако могут произойти по закону Мерфи (закону бутерброда или, по-другому, закону подлости - прим. перев.) и убедиться в том, что они не оказывают влияние на простоту и безопасность ПО.

# 7. Иные применения проектирования по контракту

Идеи проектирования по контракту проходят через весь метод Eiffel. В дополнение к уже упомянутым приложениям они имеют два важных следствия:

- Они дают возможность использовать Eiffel для анализа и проектирования. На высоком уровне абстракции также необходима точность. За исключением BON (Business Object Notation - прим. перев.), методы объектно-ориентированного анализа и проектирования имеют тенденцию к превосходству абстракции над точностью. С помощью утверждений можно точно выразить свойства системы ("С какой скоростью должен начать звенеть будильник?") без внесения каких-либо обязательств в реализацию. В обсуждении отложенных классов (см. раздел. "Применение отложенных классов") показано, как задать чисто описательную, а не программную, модель, используя контракты для формулировки основных свойств системы без какого-либо компьютерного или программного аспекта.
- Утверждения служат также для контроля механизмов, связанных с наследованием, – повторным объявлением, полиморфизмом, динамическим связыванием, – и обеспечивают их правильное использование, задавая соответствующие семантические ограничения. Более подробно смотрите раздел "Наследование и контракты".