契约式设计

Design by Contract

2022/3/3

摘要

- 引言
- Eiffel 的 DbC 机制
- DbC与继承
- 如何应用DbC

引言

- Design by Contract (DbC) 契约式设计
  - 与面向对象技术中的其它技术同等重要
    - 类
    - 对象
    - 继承
    - 多态
    - 动态绑定
    - 其它

引言

- Design by Contract (DbC) 契约式设计
  - Bertrand Meyer：DbC是构建面向对象软件系统方法的核心！
  - James McKim：“只要你会写程序，你就会写契约”

引言

- 存在的问题
  - 通过软件开发技术，获得高生产率
  - 高生产率不仅取决于软件开发技术（如复用技术），也取决于**软件质量**
  - **契约式设计**是一种保证软件质量（可靠性）的手段
  - Eiffel语言直接支持

引言

- A discipline of analysis, design, implementation, management

  作用：（可以贯穿于软件创建的全过程，从分析到设计，从文档到调试，甚至可以渗透到项目管理中）

- Viewing the relationship between a class and its clients as a formal agreement, expressing each party's rights and obligations.

  做法：（把类和它的客户程序之间的关系看做正式的协议，描述双方的权利和义务）

引言

- Every software element is intended to satisfy a certain goal, for the benefit of other software elements (and ultimately of human users). 目标
- This goal is the element's **contract**. 契约
- The contract of any software element should be
  - Explicit. **显式**
  - Part of the software element itself.

A human contract

deliver	OBLIGATIONS(义务)	BENEFITS(权益/权利)
Client	(Satisfy precondition:) Bring package before 4 p.m.; pay fee.	(From postcondition:) Get package delivered by 10 a.m. next day.
Supplier	(Satisfy postcondition:) Deliver package by 10 a.m. next day.	(From precondition:) Not required to do anything if package delivered after 4 p.m., or fee not paid.

A view of software construction

- Constructing systems as structured collections of cooperating software elements — suppliers and clients — cooperating on the basis of clear **definitions** of obligations and benefits. 软件系统（软件中的元素以客户以及服务提供者的角色，根据权利义务相互协作）

- These definitions are the contracts.

Properties of contracts

- A contract:
  - Binds two parties (or more): supplier, client. 绑定双方或多方
  - Is explicit (written). 显式的
  - Specifies mutual obligations and benefits. 规定相互的义务和权益
  - Usually maps obligation for one of the parties into benefit for the other, and conversely. 一方的义务对应另一方的权益，反之亦然

Properties of contracts

- A contract:
  - Has no hidden clauses: obligations are those specified. 没有隐式条约
  - Often relies, implicitly or explicitly, on general rules applicable to all contracts (laws, regulations, standard practices). 通常，依赖适用所有契约的一般规则

Contracts for analysis

```
deferred class PLANE inherit
    AIRCRAFT
feature
    start_take_off is
                    -- Initiate take-off procedures.
            require
                    controls.passed
                    assigned_runway.clear
            deferred
            ensure
                    assigned_runway.owner = Current
                    moving
            end
    start_landing, increase_altitude, decrease_altitude, moving,
    altitude, speed, time_since_take_off

    ... [Other features] ...

invariant
    (time_since_take_off <= 20) implies (assigned_runway.owner = Current)
    moving = (speed > 10)
end
```

Precondition

-- i.e. specified only.

-- not implemented.

Postcondition

Class invariant

Contracts for analysis (cont'd)

```
deferred class VAT inherit
    TANK
feature
    in_valve, out_valve: VALVE
    fill is
                    -- Fill the vat.
            require
                    in_valve.open
                    out_valve.closed
            deferred
            ensure
                    in_valve.closed
                    out_valve.closed
                    is_full
            end
    empty, is_full, is_empty, gauge, maximum, ... [Other features] ...
invariant

    is_full = (gauge >= 0.97 * maximum)  and  (gauge <= 1.03 * maximum)

end
```

Precondition

-- i.e. specified only.

-- not implemented.

Postcondition

Class invariant

Contracts for analysis (cont'd)

fill	OBLIGATIONS	BENEFITS
Client	(Satisfy precondition:) Make sure input valve is open, output valve is closed.	(From postcondition:) Get filled-up vat, with both valves closed.
Supplier	(Satisfy postcondition:) Fill the vat and close both valves.	(From precondition:) Simpler processing thanks to assumption that valves are in the proper initial position.

So, is it like “assert.h”?

(Source: Reto Kramer)

- Design by Contract goes further:
  - “Assert” does not provide a contract.
  - Clients cannot see asserts as part of the interface.
  - Asserts do not have associated semantic specifications.
  - Not explicit whether an assert represents a precondition, post-conditions or invariant.
  - Asserts do not support inheritance.
  - Asserts do not yield automatic documentation.

Contracts

- 契约就是“规范和检查”！
  - Precondition：针对method，它规定了在调用该方法之前**必须为真的条件**
  - Postcondition：针对method，它规定了方法顺利执行完毕之后**必须为真的条件**
  - Invariant：针对整个类，它规定了该类任何实例调用任何方法都**必须为真的条件**

Correctness in software

- Correctness is a relative notion: consistency of implementation vis-a-vis specification. (This assumes there is a specification!)

- Basic notation: (*P*, *Q*: assertions, i.e. properties of the state of the computation. *A*: instructions).

$$\{P\}\ A\ \{Q\}$$

- "Hoare triple"

- What this means (**total correctness**):
  - Any execution of *A* started in a state satisfying *P* will terminate in a state satisfying *Q*.

Hoare triples: a simple example

{n > 5} *n* := *n* + 9 {n > 13}

- Most interesting properties:
  - *Strongest* postcondition (from given precondition). → n>14
  - *Weakest* precondition (from given postcondition). → n>4
- "*P* is stronger than or equal to *Q*" means:

  P implies *Q*

- QUIZ: What is the strongest possible assertion ? The weakest ?

Software correctness

- Consider

$$\{P\}\ A\ \{Q\}$$

- Take this as a job ad in the classifieds.

"We are looking for someone whose work will be to start from initial situations as characterized by *P*, and deliver results as defined by *Q*

- Should a lazy employment candidate hope for a weak or strong *P*? What about *Q*?

- *Two special offers:*
  - 1. {*False*} *A* {...}
  - 2. {...} *A* {*True*}

Strongest precond.

Weakest postcond.

Contracts for analysis (cont'd)

```
deferred class VAT inherit
    TANK
feature
    in_valve, out_valve: VALVE
    fill is
                    -- Fill the vat.
            require
                    in_valve.open
                    out_valve.closed
            deferred
            ensure
                    in_valve.closed
                    out_valve.closed
                    is_full
            end
    empty, is_full, is_empty, gauge, maximum, ... [Other features] ...
invariant

    is_full = (gauge >= 0.97 * maximum)  and  (gauge <= 1.03 * maximum)

end
```

Precondition

-- i.e. specified only.

-- not implemented.

Postcondition

Class invariant

Contracts for analysis (cont'd)

fill	OBLIGATIONS	BENEFITS
Client	(Satisfy precondition:) Make sure input valve is open, output valve is closed.	(From postcondition:) Get filled-up vat, with both valves closed.
Supplier	(Satisfy postcondition:) Fill the vat and close both valves.	(From precondition:) Simpler processing thanks to assumption that valves are in the proper initial position.

So, is it like "assert.h"?

(Source: Reto Kramer)

- Design by Contract goes further:
  - "Assert" does not provide a contract.
  - Clients cannot see asserts as part of the interface.
  - Asserts do not have associated semantic specifications.
  - Not explicit whether an assert represents a precondition, post-conditions or invariant.
  - Asserts do not support inheritance.
  - Asserts do not yield automatic documentation.

Contracts

- 契约就是“规范和检查”！
  - Precondition：针对method，它规定了在调用该方法之前**<u>必须为真的条件</u>**
  - Postcondition：针对method，它规定了方法顺利执行完毕之后**<u>必须为真的条件</u>**
  - Invariant：针对整个类，它规定了该类任何实例调用任何方法都**<u>必须为真的条件</u>**

摘要

- 引言
- **Eiffel 的 DbC 机制**
- DbC与继承
- 如何应用DbC

Design by Contract: The Mechanism

- Preconditions and Postconditions
- Class Invariant
- Run-time effect

The contract

Routine	OBLIGATIONS	BENEFITS
Client	PRECONDITION	POSTCONDITION
Supplier	POSTCONDITION	PRECONDITION

A class without contracts

```
class ACCOUNT feature -- Access

    balance: INTEGER
                        -- Balance

    Minimum_balance: INTEGER is 1000
                        -- Minimum balance

feature {NONE} -- Implementation of deposit and withdrawal

    add (sum: INTEGER) is
                        -- Add sum to the balance (secret procedure).
            do
                        balance := balance + sum
            end
```

Without contracts (cont'd)

```
feature -- Deposit and withdrawal operations

   deposit (sum: INTEGER) is
                     -- Deposit sum into the account.
          do
                     add (sum)
          end

   withdraw (sum: INTEGER) is
                     -- Withdraw sum from the account.
          do
                     add (–sum)
          end

   may_withdraw (sum: INTEGER): BOOLEAN is
                     -- Is it permitted to withdraw sum from the account?
          do
                     Result := (balance - sum >= Minimum_balance)
          end

end
```

Introducing contracts

```
class ACCOUNT create

    make

feature {NONE} -- Initialization

    make (initial_amount: INTEGER) is
                    -- Set up account with initial_amount.
            require
                    large_enough: initial_amount >= Minimum_balance

            do
                    balance := initial_amount

            ensure
                    balance_set: balance = initial_amount

end
```

Introducing contracts (cont'd)

```
feature -- Access

   balance: INTEGER
            -- Balance

   Minimum_balance: INTEGER is 1000
            -- Minimum balance

feature {NONE} -- Implementation of deposit and withdrawal

   add (sum: INTEGER) is
                  -- Add sum to the balance (secret procedure).
            do
                  balance := balance + sum

            ensure

                  increased: balance = old balance + sum

            end
```

With contracts (cont’d)

```
feature -- Deposit and withdrawal operations

    deposit (sum: INTEGER) is
                -- Deposit sum into the account.
        require
                not_too_small: sum >= 0
        do
                add (sum)
        ensure
                increased: balance = old balance + sum
        end
```

With contracts (cont'd)

```
withdraw (sum: INTEGER) is
            -- Withdraw sum from the account.
    require
            not_too_small: sum >= 0
            not_too_big:
                sum <= balance – Minimum_balance
    do
            add (– sum)
                    -- i.e. balance := balance – sum
    ensure
            decreased: balance = old balance - sum
    end
```

The contract

withdraw	OBLIGATIONS	BENEFITS
Client	(Satisfy precondition:) Make sure *sum* is neither too small nor too big.	(From postcondition:) Get account updated with *sum* withdrawn.
Supplier	(Satisfy postcondition:) Update account for withdrawal of *sum*.	(From precondition:) Simpler processing: may assume *sum* is within allowable bounds.

With contracts (end)

```
may_withdraw (sum: INTEGER): BOOLEAN is
                -- Is it permitted to withdraw sum from the
                -- account?
        do
                Result := (balance - sum >= Minimum_balance)
        end

invariant

    not_under_minimum: balance >= Minimum_balance

end
```

The class invariant

- Consistency constraint applicable to all instances of a class.

- Must be satisfied:
  - After creation.
  - After execution of any feature by any client.
    (Qualified calls only: *a.f* (...))

The correctness of a class

- For every creation procedure *cp*:

  $\{pre_{cp}\}\ do_{cp}\ \{post_{cp}\ \text{and}\ INV\}$

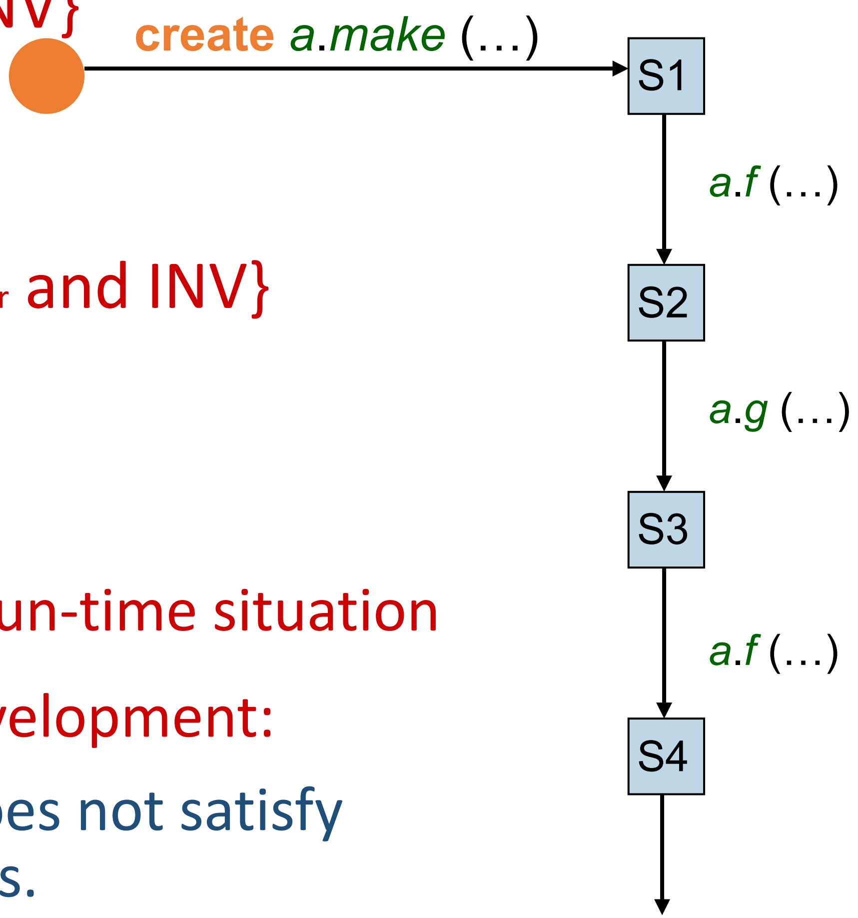
- For every exported routine *r*:

  $\{INV\ \text{and}\ pre_r\}\ do_r\ \{post_r\ \text{and}\ INV\}$

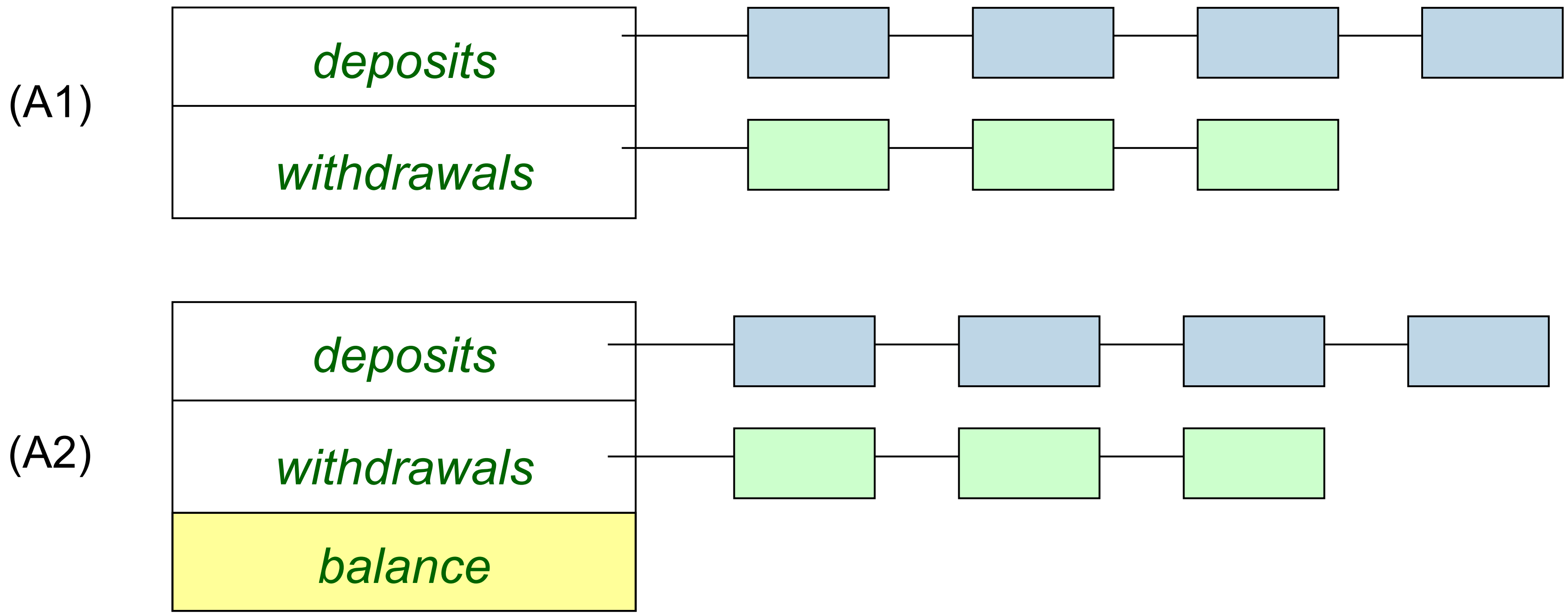
- The worst possible erroneous run-time situation

  in object-oriented software development:

  - Producing an object that does not satisfy the invariant of its own class.



Example



balance = *deposits*.*total* − *withdrawals*.*total*

A more sophisticated version

```
class ACCOUNT create
    make
feature {NONE} -- Implementation
    add (sum: INTEGER) is
                        -- Add sum to the balance (secret procedure).
            do
                        balance := balance + sum
            ensure
                        balance_increased: balance = old balance + sum
          end
    deposits: DEPOSIT_LIST
    withdrawals: WITHDRAWAL_LIST
```

New version (cont'd)

```
feature {NONE} -- Initialization
    make (initial_amount: INTEGER) is
                -- Set up account with initial_amount.
require
        large_enough: initial_amount >= Minimum_balance
do
        balance := initial_amount
        create deposits.make
        create withdrawals.make
ensure
        balance_set: balance = initial_amount
        end
feature -- Access
        balance: INTEGER
                -- Balance
        Minimum_balance: INTEGER is 1000
                -- Minimum balance
```

New version (cont'd)

```
feature -- Deposit and withdrawal operations

   deposit (sum: INTEGER) is
                 -- Deposit sum into the account.
         require
                 not_too_small: sum >= 0
         do
                 add (sum)
                 deposits.extend (create {DEPOSIT}.make (sum))
         ensure
                 increased: balance = old balance + sum
         end
```

New version (cont'd)

```
withdraw (sum: INTEGER) is
            -- Withdraw sum from the account.
    require
            not_too_small: sum >= 0
            not_too_big: sum <= balance – Minimum_balance
    do
        add (–sum)
        withdrawals.extend (create {WITHDRAWAL}.make (sum))
    ensure
        decreased: balance = old balance – sum
        one_more: withdrawals.count = old withdrawals.count + 1
    end
```

New version (end)

```
may_withdraw (sum: INTEGER): BOOLEAN is
            -- Is it permitted to withdraw sum from the
            -- account?
      do
            Result := (balance - sum >= Minimum_balance)
      end

invariant

   not_under_minimum: balance >= Minimum_balance

   consistent: balance = deposits.total – withdrawals.total
end
```

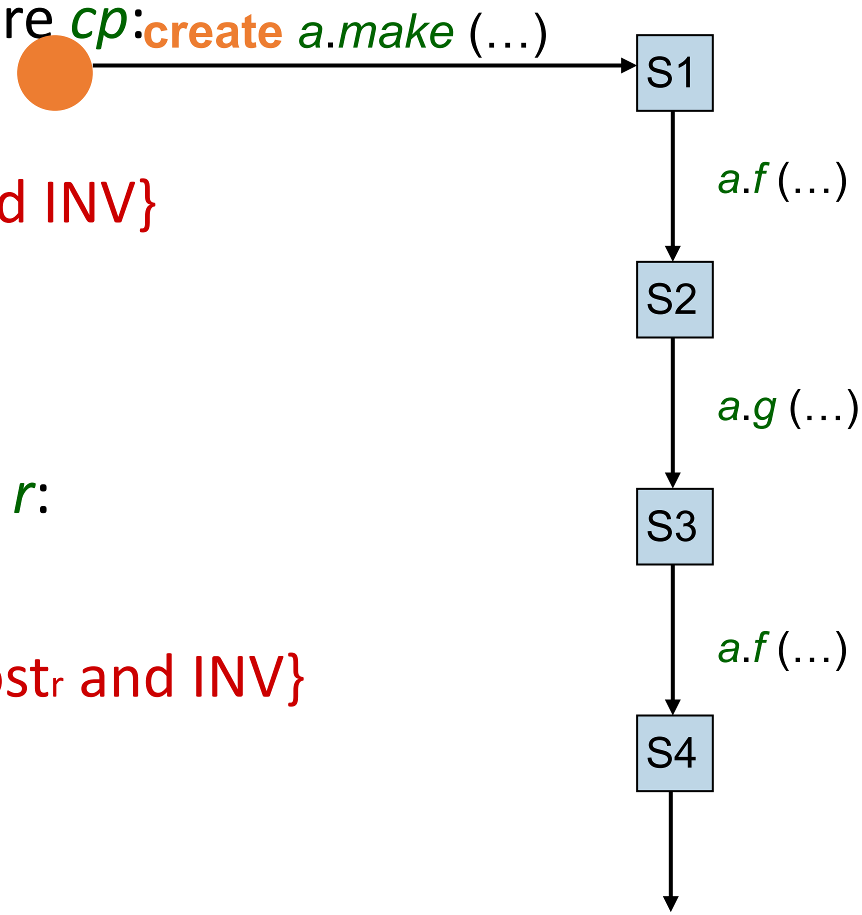
The correctness of a class

- For every creation procedure *cp*:

$$\{pre_{cp}\}\ do_{cp}\ \{post_{cp} \text{ and } INV\}$$

- For every exported routine *r*:

$$\{INV \text{ and } pre_r\}\ do_r\ \{post_r \text{ and } INV\}$$



Initial version

```
feature {NONE} -- Initialization
   make (initial_amount: INTEGER) is
                  -- Set up account with initial_amount.
         require
                  large_enough: initial_amount >= Minimum_balance
         do
                  balance := initial_amount
                  create deposits.make
                  create withdrawals.make
         ensure
                  balance_set: balance = initial_amount
   end
```

Correct version

```
feature {NONE} -- Initialization
   make (initial_amount: INTEGER) is
                  -- Set up account with initial_amount.
         require
                  large_enough: initial_amount >= Minimum_balance
         do
                  create deposits.make
                  create withdrawals.make
                  deposit (initial_amount)
         ensure
                  balance_set: balance = initial_amount
end
```

Contracts: run-time effect

- Compilation options (per class, in Eiffel):
  - No assertion checking
  - Preconditions only
  - Preconditions and postconditions
  - Preconditions, postconditions, class invariants
  - All assertions

摘要

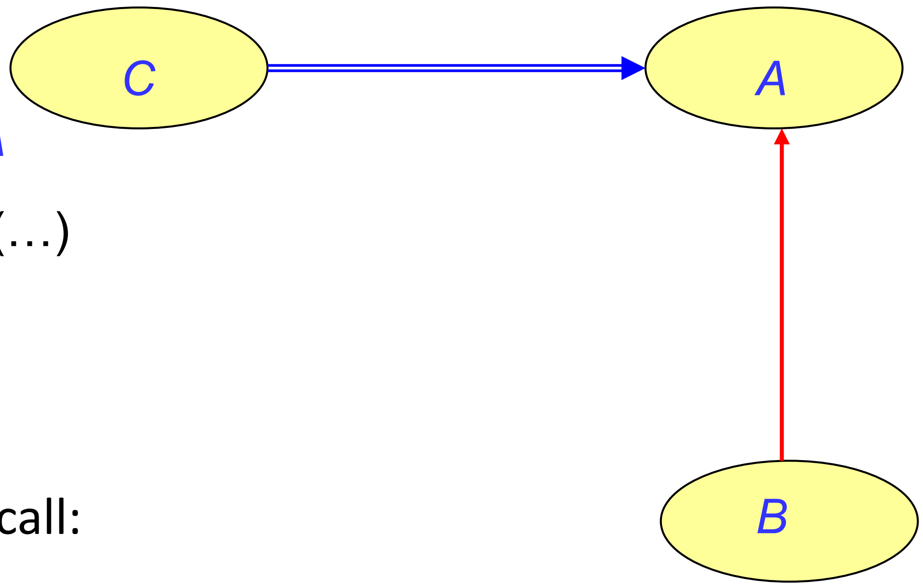
- 引言
- Eiffel 的 DbC 机制
- **DbC与继承**
- 如何应用DbC

继承与 Design by Contract

- 问题：
  - 子类中的断言与父类中的断言是什么关系？
- 依据
  - 子类乃父类的特化，子类的实例也是父类的合法实例。
  - 申明为父类的引用运行时可能指向子类实例
- 因而
  - ？

Inheritance and assertions

a1: *A*
…
a1.r (…)



r **is**
α
ensure
β

r **is**
γ
ensure
δ

Correct call:

if *a1*.α **then**
a1.r (...)
else
...
end

Contract

delivery	OBLIGATIONS	BENEFITS
Client	(Satisfy precondition:) 不得要求投递超过5kg的包裹	(From postcondition:) 3个工作日内包裹到位
Supplier	(Satisfy postcondition:) 在3个工作日内投送到位	(From precondition:) 不受理超过5kg的包裹

Contract

```
class COURIER
  feature
    deliver(p:Package, d:Destination)
      require
        --包裹重量不超过5kg
      ensure
        --3个工作日内投送到指定地点
    …
end
```

More desirable contract

delivery	OBLIGATIONS	BENEFITS
Client	(Satisfy precondition:) 不得要求投递超过8kg的包裹	(From postcondition:) 2个工作日内包裹到位
Supplier	(Satisfy postcondition:) 在2个工作日内投送到位	(From precondition:) 不受理超过8kg的包裹

More desirable contract

```
class DIFFERENT_COURIER
Inherit COURIER
redefine  deliver
   feature
      deliver(p:Package, d:Destination)
         require
            --包裹重量不超过5kg
         require else
            --包裹重量不超过8kg
         ensure
            --3天内投送到指定地点
         ensure then
            --2天内投送到指定地点
     …
end
```

```
require
      --包裹重量不超过8kg
```

```
ensure
      -- 2天内投送到指定地点
```

Assertion redeclaration rule

- Redefined version may not have **require** or **ensure**.
- May have nothing (assertions kept by default), or

require else *new_pre*

ensure then *new_post*

- Resulting assertions are:
  - *original_precondition* **or** *new_pre*
  - *original_postcondition* **and** *new_post*

Invariant accumulation

- Every class inherits all the invariant clauses of its parents.
- These clauses are conceptually “and”-ed.

简言之...

- 可以使用*require else*削弱先验条件
- 可以使用*ensure then*加强后验条件
- 用*and*把不变式子句和你所继承的不变式子句结合起来，就可以加强不变式

摘要

- 引言
- Eiffel 的 DbC 机制
- DbC与继承
- **如何应用DbC**
- 其它

Design by Contract: How to apply

- 目的：构造高质量的程序
- DbC与Quality Assurance（QA）
- 理解Contract violation
- Precondition Design
  - Not defensive programming
- Class Invariants and business logic

Design by Contract: How to apply

- 目的：构造高质量的程序
- DbC与Quality Assurance（QA）
- 理解Contract violation
- Precondition Design
    - Not defensive programming
- Class Invariants and business logic

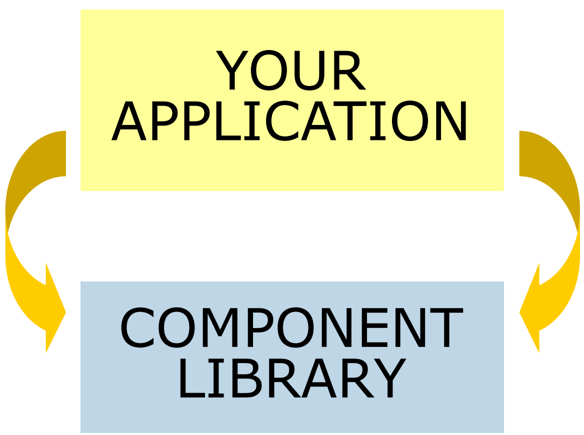
Contracts and quality assurance

- Precondition violation: Bug in the client.
- Postcondition violation: Bug in the supplier.
- Invariant violation: Bug in the supplier.

$\{P\}\ A\ \{Q\}$

Contracts and bug types

- Preconditions are particularly useful to find bugs in client code:




```
your_list.insert (y, a + b + 1)
```

```
class LIST [G]
…
      insert (x: G; i: INTEGER) is
            require
                  i >= 0
                  i <= count + 1
```

Contracts and quality assurance

- Use run-time assertion monitoring for quality assurance, testing, debugging.

- Compilation options (reminder):
  - No assertion checking
  - Preconditions only
  - Preconditions and postconditions
  - Preconditions, postconditions, class invariants
  - All assertions

Contracts and quality assurance

- Contracts enable QA activities to be based on a precise description of what they expect.

契约使得质量保证可以依赖于更精确的描述

- Profoundly transform the activities of testing, debugging and maintenance.

深切的改变了测试、调试以及维护等一系列的活动

Contract monitoring

- Enabled or disabled by compile-time options.
- Default: preconditions only.
- In development: use “all assertions” whenever possible.
- During operation: normally, should disable monitoring. But have an assertion-monitoring version ready for shipping.
- Result of an assertion violation: exception.
- Ideally: static checking (proofs) rather than dynamic monitoring.

Contract form of ACCOUNT class

```
class interface ACCOUNT create

    make

feature

    balance: INTEGER
            -- Balance

    Minimum_balance: INTEGER is 1000                                    --
    Minimum balance

    deposit (sum: INTEGER)
                -- Deposit sum into the account.
            require
                not_too_small: sum >= 0
            ensure
                increased: balance = old balance + sum
```

Contract form of ACCOUNT class (cont'd)

```
withdraw (sum: INTEGER)
        -- Withdraw sum from the account.
    require
        not_too_small: sum >= 0
        not_too_big: sum <= balance – Minimum_balance
    ensure
        decreased: balance = old balance – sum
        one_more: withdrawals.count = old withdrawals.count + 1
        may_withdraw (sum: INTEGER): BOOLEAN
                -- Is it permitted to withdraw sum from the
                -- account?
    invariant
        not_under_minimum: balance >= Minimum_balance
        consistent: balance = deposits.total – withdrawals.total
end
```

Contracts and documentation

- 契约能使文档更出色
  - 更清晰的文档
    - 契约乃是类特性的公开视图中的固有成分
  - 更可靠的文档
    - 运行时要检查断言，以便保证制定的契约与程序的实际运行情况一致
  - 明确的测试指导
    - 断言定义了测试的预期结果，并且由代码进行维护
  - 更精确的规范
    - 既能够获得精确规范得到的益处，同时还使得程序员继续以他们所熟悉的方式工作

Uses of the contract and interface forms

- 文档，用户手册
- 设计
- 开发者之间交流
- 开发者和管理者之间交流

Contracts and reuse

- 库使用者手中的优秀文档
  - 契约清楚地解释了程序库中各个类、各个例程的任务，以及使用中的限制条件
- 对库使用者的帮助
  - 运行时的契约检查为那些学习使用别人的类的人们提供了反馈

Reuse without a contract is sheer folly.

DbC vs. Defensive programming

- 什么是防御性编程？
  - 防止程序接受错误的输入？
  - 防止用错误参数或者在不适当的情况下调用程序？

“防御性编程是一种细致、谨慎的编程方法。为了开发可靠的软件，我们要设计系统中的每个组件，以使其尽可能地“保护”自己。我们通过明确地在代码中对设想进行检查，击碎了未记录下来的设想。这是一种努力，防止（或至少是观察）我们的代码以将会展现错误行为的方式被调用。” (Goodliffe, P: 《编程匠艺：编写卓越的代码》)

DbC vs. Defensive programming

- 防止程序接受错误的输入
  - “一个关键的防御性策略就是检查所有的程序输入”
- 给程序穿上“防弹衣”

```
placeCard(c:INTEGER,x:INTEGER,y:INTEGER) is
  do
    if (c<1) or (c>MAXCARDS) then return
    …
  end
```

bulletproofing

not a good style

DbC vs. Defensive programming

- 防御性编程

```
placeCard(c:INTEGER,x:INTEGER,y:INTEGER) is
        --网格(x,y)点放一张c牌
    do
      if (c<1) or (c>MAXCARDS)
      then
         raise PRECONDITION_EXCEPTION(
                 "Grid: placeCard: bad card number")
      else
      ...
    end
```

异常指明发生问题的类和程序以及问题本质

DbC vs. Defensive programming

- DbC

placeCard(c:INTEGER,x:INTEGER,y:INTEGER) is

require

valid_card_number: (c>=1) and (c<=MAXCARDS)

do

映射：从契约的设计到产生异常的实现

...

end

DbC vs. Defensive programming

- 差异
  - DbC中先验条件是程序文档的组成部分，而产生异常的语句是程序体本身的组成部分。
  - 采用注释来描述例程对参数的限制时，很难保证这个注释正确地描述了该限制。但可以相信具有显式先验条件检查的文档，因为断言在测试时经受了考验。

How strong should a precondition be?

- Two opposite styles:
  - Tolerant: weak preconditions (including the weakest, *True*: no precondition). 弱的前置条件
  - Demanding: strong preconditions, requiring the client to make sure all logically necessary conditions are satisfied before each call. 强的前置条件
- Partly a matter of taste.
- But: demanding style leads to a better distribution of roles, provided the precondition is:
  - Justifiable in terms of the specification only.
  - Documented (through the short form).
  - Reasonable!

A demanding style

```
sqrt (x, epsilon: REAL): REAL is
        -- Square root of x, precision epsilon
        -- Same version as before

  require

        x >= 0
        epsilon >= 0

  do
        ...
  ensure

        abs (Result ^ 2 – x) <= 2 * epsilon * Result

  end
```

A tolerant style

```
sqrt (x, epsilon: REAL): REAL is
            -- Square root of x, precision epsilon
      require
            True

            if x < 0 then
                  … Do something about it (?) …
            else
                  … normal square root computation …

                  computed := True
            end

      ensure
            computed implies

            abs (Result ^ 2 – x) <= 2 * epsilon * Result

   end
```

Contrasting styles

```
put (x: G) is
        -- Push x on top of stack.
    require
        not is_full
    do
        ....
    end

tolerant_put (x: G) is
        -- Push x if possible, otherwise set impossible to True.
    do
        if not is_full then
            put (x)
        else
            impossible := True
        end
    end
```

Invariants and business rules

- Invariants are absolute consistency conditions.
- They can serve to represent business rules if knowledge is to be built into the software.
- Form 1

```
invariant
        not_under_minimum: balance >= Minimum_balance
```

- Form 2

```
invariant
        not_under_minimum_if_normal:
                normal_state implies
                        (balance >= Minimum_balance)
```

小结

- Design by Contract
  - 原理
    - 借鉴“契约”原理，界定模块之间的权利义务，规范软件的开发，提高软件质量。
  - 应用
    - 可以贯穿于软件创建的全过程，从分析到设计，从文档到调试，甚至可以渗透到项目管理中
  - 优势

参考书籍

- Bertrand Meyer, ***Object-Oriented Software Construction***, Second Edition, Prentice Hall, 1997.（ Chapter 11 ）

作业

- 解释应用DbC时子类断言与父类断言的关系
- 解释DbC和防御性编程的异同
- 了解C++或者Java的断言机制，解释DbC和断言的区别

提交作业到教学立方（3月17号24点截止）

作业

- **自学Contract4J**
  - Contract4J 是一个开源的开发人员工具，它用 Java 5 标注实现契约式设计。
  - 在幕后，它用方面在应当执行测试的程序连接点处（例如，对方法的调用）插入 “建议” ，它还对这些测试的失败进行处理，即终止程序执行。
- **自学在C++中使用断言**