Fondamenti della Programmazione: Metodi Evoluti

Prof. Enrico Nardelli

Esercitazione 4

Class invariants explained in 60 seconds

- Consistency requirements for a class
- Established after object creation
- Hold
  - Before any feature execution (like pre-conditions)
  - After any feature execution (like post-conditions)

```
class
    ACCOUNT
feature
    balance: INTEGER
invariant
    balance >= 0
end
```

Pay attention to class invariants!

- Class invariants
    - Marriage problems
    - Violating the invariant

Modeling people and marriage

Hands-On

Allow creation of a *PERSON* with a **Void** reference to *spouse*

```
class
    PERSON
feature
    name: STRING
        -- name of Current.

    spouse: detachable PERSON
        -- Spouse of Current.

    marry (a_other: PERSON)
        -- Marry `a_other'.
        do
        end
end
```

```
class
    MARRIAGE
create
    make
feature
    make
        local
            alice: PERSON
            bob: PERSON
        do
            create alice
            create bob
            bob.marry (alice)
        end
end
```

Do they compile correctly in Void Safe mode?

Let's remember to use creation procedures

```
class
    MARRIAGE
feature
    make
        local
            alice: PERSON
            bob: PERSON
        do
            create alice.set_name("Alice")
            create bob.set_name("Bob")
            bob.marry (alice)
        end
end
```

Let's remember to use creation procedures

```
class
    PERSON
create
    set_name
feature
    name: STRING
        -- name of Current.
    set_name (a_name: STRING)
        -- assign name
          do  name := a_name
          ensure name = a_name
          end
    spouse: detachable PERSON
        -- Spouse of Current.
    marry (a_other: PERSON)
        -- Marry `a_other’.
          do
          end
end
```

Write the contracts

Hands-On

```
class PERSON
feature
    name: STRING
    spouse: detachable PERSON
    marry (a_other: PERSON)
        require
            ??
        ensure
            ??

invariant
    ??
end
```

Here *a_other* must be attached to an instance of *PERSON*

A possible solution

```
class PERSON
feature
    name: TEXT
    spouse: detachable PERSON
    marry (a_other: PERSON)
        require
            -- NB a_other is attached hence cannot be Void
            spouse = Void
            a_other.spouse = Void
            a_other /= Current
        ensure
            spouse = a_other
            a_other.spouse = Current
        end

invariant
    attached spouse as s implies s.spouse = Current
end
```

Implementing *marry* (1)

Hands-On

```
class PERSON
feature
    name: STRING
    spouse: detachable PERSON
    marry (a_other: PERSON)
        require
            -- NB a_other is attached hence cannot be Void
            a_other.spouse = Void
            spouse = Void
            a_other /= Current
        do
            ??
            ??
        ensure
            spouse = a_other
            a_other.spouse = Current
        end

invariant
    attached spouse as s implies s.spouse = Current
end
```

Implementing *marry* (2)

```
class PERSON
feature
    name: STRING
    spouse: detachable PERSON
    marry (a_other: PERSON)
        require
            -- NB a_other is attached hence cannot be Void
            a_other.spouse = Void
            spouse = Void
            a_other /= Current
        do
            a_other.spouse := Current
            spouse := a_other
        ensure
            spouse = a_other
            a_other.spouse = Current
        end

invariant
    attached spouse as s implies s.spouse = Current
end
```

Implementing *marry* (3)

Hands-On

```
class PERSON
feature
    name: STRING
    spouse: detachable PERSON
    marry (a_other: PERSON)
        require
            -- NB a_other is attached hence cannot be Void
            a_other.spouse = Void
            spouse = Void
            a_other /= Current
        do
            a_other.spouse := Current
            spouse := a_other
        ensure
            spouse = a_other
            a_other.spouse = Current
        end

invariant
    attached spouse as s implies s.spouse = Current
end
```

Compiler Error:

No assigner command

Implementing *marry* (4)

Hands-On

```
class PERSON
feature
    name: STRING
    spouse: detachable PERSON
    marry (a_other: PERSON)
        require
            -- NB a_other is attached hence cannot be Void
            a_other.spouse = Void
            spouse = Void
            a_other /= Current
        do
            a_other.set_spouse (Current)
            spouse := a_other
        ensure
            spouse = a_other
            a_other.spouse = Current
        end
    set_spouse (a_other: PERSON)
            do spouse := a_other
            ensure spouse = a_other
            end
invariant
    attached spouse as s implies s.spouse = Current
end
```

Implementing *marry* (5)

Hands-On

```
class PERSON
feature
    name: STRING
    spouse: detachable PERSON
    marry (a_other: PERSON)
        require
            -- NB a_other is attached hence cannot be Void
            a_other.spouse = Void
            spouse = Void
            a_other /= Current
        do
            a_other.set_spouse (Current)
            spouse := a_other
        ensure
            spouse = a_other
            a_other.spouse = Current
        end
    set_spouse (a_other: PERSON)
            do  spouse := a_other
            ensure spouse = a_other
            end
invariant
    attached spouse as s implies s.spouse = Current
end
```

Invariant of a_other?

Violated after call to set_spouse

What happened?

In *MARRIAGE* class: *bob.marry (alice)*

In *marry* feature: *a_other.set_spouse* (**Current**)

In *PERSON* class:

attached *spouse* **as** *s* **implies** *s.spouse* = **Current**

- During execution of *marry* for **Bob**, *set_spouse* is executed for **Alice** and set **Alice**.*spouse* to **Bob** value. When *set_spouse* ends the class invariant is checked for **Alice**. **Alice**.*spouse* is attached to **Bob** but **Bob**.*spouse* value is not **Alice** and the invariant is violated

Violating the class invariant

- When one first changes *spouse* of *a_other*, then - after the execution of *a_other.set_spouse* terminates – the class invariant is checked for *a_other* and found violated

- Instead, if one first changes *spouse* of **Current**, then right after execution of *spouse := a_other* no invariant is checked (since only a **Current**'s attribute is modified) hence it's possible to update *a_other* status so as to keep class invariants true for both objects

Implementing *marry* (6)

```
class PERSON
feature
    name: STRING
    spouse: detachable PERSON
    marry (a_other: PERSON)
        require
            -- NB a_other by definition cannot be
            a_other.spouse = Void
            spouse = Void
            a_other /= Current
        do
            spouse := a_other
            a_other.set_spouse (Current)
        ensure
            spouse = a_other
            a_other.spouse = Current
        end

feature {PERSON}
    set_spouse (a_other: PERSON)
        do  spouse := a_other
        ensure spouse = a_other
        end

invariant
    attached spouse as s implies s.spouse = Current
end
```

Hands-On

Divorcing?

```
local
  bob, alice: PERSON
do
  create bob; create alice
  bob.marry (alice)
  -- let's implement divorce as
  bob.set_spouse (Void)
  alice.set_spouse (Void)
  -- the argument has to be detachable...
  -- does it make sense?!?
  -- let's try and see what happens...
end
```

Class invariant violation during divorce

- Executing **Bob**.*set_spouse*(**Void**) keeps class invariant true for **Current**, that is **Bob**, since antecedent is false. Makes the invariant false for **Alice**, but system does not become aware of it
  - class invariants are checked for an object only before and after the qualified call of a feature on the object itself
  - class invariants are **NOT** checked for a given object
    - inside the execution of any of its features
    - if other features on the same object are called in an unqualified way
    - if features of other objects of the same class are called (but invariants are checked on called objects!)
- When starting **Alice**.*set_spouse*(**Void**), the class invariant is checked for **Alice** and found violated
- Changing the order of execution does not solve the problem

Ending the marriage

Hands-On

```
class PERSON
feature
    name: STRING
    spouse: detachable PERSON
    divorce
        require
            spouse /= Void
        do
            spouse := Void
            if attached spouse as s then s.set_spouse (Void) end
        ensure
            spouse = Void
        end

invariant
    attached spouse as s implies s.spouse = Current
end
```

Is the order of instructions correct?

Let's see... N.B.: just one invocation of *divorce* is needed

There is a problem…

- Setting first the value of **Current**.*spouse* to **Void** makes the call *spouse.set_spouse* useless: it is not executed since the **if attached** test fails and the **Void** call is not issued
- Class invariant is checked after **Bob**.*divorce* and is found satisfied since its antecedent is false
- But if **Alice** is accessed then its class invariant is found violated

How to solve it

- For divorcing one has to first to set the value of *spouse.spouse* to **Void** and then to set the value of **Current**.*spouse* to **Void**
- Class invariant for *spouse* object
  - is checked after *spouse.set_spouse(***Void***)* ends
  - is satisfied since the antecedent is false
- Class invariant for **Current** object
  - is NOT checked after *spouse.set_spouse(***Void***)* ends
  - is checked at the end of *divorce*
  - is satisfied since the antecedent is false

Ending the marriage

Hands-On

```
class PERSON
feature
    name: STRING
    spouse: detachable PERSON
    divorce
        require
            spouse /= Void
        do
            if attached spouse as s then s.set_spouse (Void) end
            spouse := Void
        ensure
            spouse = Void
            attached (old spouse) as os implies os.spouse = Void
        end

invariant
    attached spouse as s implies s.spouse = Current
end
```

There is a little bit still missing...

What we have seen

- Class invariant should only depend on **Current** object

- If class invariant depends on other objects
  - Take care who can change state
  - Take care in which order you change state

- Class invariant can be temporarily violated
  - You can still call features on **Current** object
  - Violation detected when object is accessed
  - Take care calling other objects, they might call back

Although writing invariants is not that easy, they are necessary to do formal proofs. This is also the case for loop invariants (which will come later).