Subcontracting

Readings: OOSCS2 Chapters 14 – 16

EECS3311 A & E: Software Design
Fall 2020

CHEN-WEI WANG

Aspects of Inheritance

- ***Code Reuse***
- Substitutability
  - ***Polymorphism*** and ***Dynamic Binding*** [compile-time type checks]
  - *Sub-contracting* [runtime behaviour checks]

Learning Objectives

1. ***Preconditions***: require less vs. require more
2. ***Postconditions***: ensure less vs. ensure more
3. Inheritance and Contracts: ***Static Analysis***
4. Inheritance and Contracts: ***Runtime Checks***

Background of Logic (1)

Given *preconditions* P_1 and P_2, we say that

$\boxed{P_2 \textbf{ requires less} \text{ than } P_1}$ if

P_2 is ***less strict*** on (thus ***allowing more***) inputs than P_1 does.

$$\{ x \mid P_1(x) \} \subseteq \{ x \mid P_2(x) \}$$

More concisely:

$$P_1 \Rightarrow P_2$$

e.g., For command `withdraw(amount: amount)`,

$\boxed{P_2 : amount \geq 0}$ ***requires less*** than $\boxed{P_1 : amount > 0}$

What is the *precondition* that ***requires the least***? [***true***]

Background of Logic (2)

Given *postconditions* or *invariants* Q_1 and Q_2, we say that $\boxed{Q_2 \text{ ensures more than } Q_1}$ if
Q_2 is ***stricter*** on (thus ***allowing less***) outputs than Q_1 does.

$$\{ x \mid Q_2(x) \} \subseteq \{ x \mid Q_1(x) \}$$

More concisely:

$$Q_2 \Rightarrow Q_1$$

e.g., For query `q(i: INTEGER): BOOLEAN`,

$\boxed{Q_2 : \textbf{Result} = (i > 0) \land (i \textbf{ mod } 2 = 0)}$ ***ensures more*** than

$\boxed{Q_1 : \textbf{Result} = (i > 0) \lor (i \textbf{ mod } 2 = 0)}$

What is the *postcondition* that ***ensures the most***? [***false***]

Inheritance and Contracts (1)

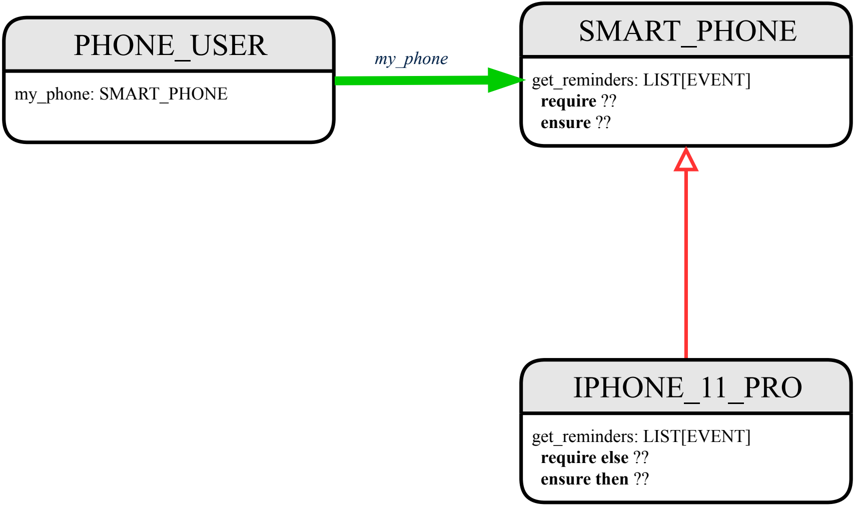
- The fact that we allow *polymorphism* :

```
local my_phone: SMART_PHONE
      i_phone: IPHONE_11_PRO
      samsung_phone: GALAXY_S10_PLUS
      huawei_phone: HUAWEI_P30_PRO
do my_phone := i_phone
   my_phone := samsung_phone
   my_phone := huawei_phone
```

  suggests that these instances may *substitute* for each other.
- Intuitively, when expecting `SMART_PHONE`, we can substitute it by instances of any of its **descendant** classes.

  ∵ Descendants *accumulate code* from its ancestors and can thus *meet expectations* on their ancestors.
- Such *substitutability* can be reflected on contracts, where a *substitutable instance* will:
  - ***Not*** require more from clients for using the services.
  - ***Not*** ensure less to clients for using the services.

Inheritance and Contracts (2.1)



Inheritance and Contracts (2.2)

```
class SMART_PHONE
  get_reminders: LIST[EVENT]
    require
      α: battery_level ≥ 0.1 -- 10%
    ensure
      β: ∀e : Result | e happens today
end
```

```
class IPHONE_11_PRO
inherit SMART_PHONE redefine get_reminders end
  get_reminders: LIST[EVENT]
    require else
      γ: battery_level ≥ 0.15 -- 15%
    ensure then
      δ: ∀e : Result | e happens today or tomorrow
end
```

Contracts in descendant class IPHONE_11_PRO are *not suitable*.

($battery\_level \geq 0.1 \Rightarrow battery\_level \geq 0.15$) is not a tautology.

e.g., A client able to get reminders on a SMART_PHONE, when battery level is 12%, will fail to do so on an IPHONE_11_PRO.

Inheritance and Contracts (2.3)

```
class SMART_PHONE
 get_reminders: LIST[EVENT]
   require
     α: battery_level ≥ 0.1 -- 10%
   ensure
     β: ∀e:Result | e happens today
end
```

```
class IPHONE_11_PRO
inherit SMART_PHONE redefine get_reminders end
 get_reminders: LIST[EVENT]
   require else
     γ: battery_level ≥ 0.15 -- 15%
   ensure then
     δ: ∀e:Result | e happens today or tomorrow
end
```

Contracts in descendant class `IPHONE_11_PRO` are *not suitable*.

(e happens ty. or tw.) ⇒ (e happens ty.) not tautology.

e.g., A client receiving today's reminders from `SMART_PHONE` are shocked by tomorrow-only reminders from `IPHONE_11_PRO`.

Inheritance and Contracts (2.4)

```
class SMART_PHONE
  get_reminders: LIST[EVENT]
    require
      α: battery_level ≥ 0.1 -- 10%
    ensure
      β: ∀e : Result | e happens today
end
```

```
class IPHONE_11_PRO
inherit SMART_PHONE redefine get_reminders end
  get_reminders: LIST[EVENT]
    require else
      γ: battery_level ≥ 0.05 -- 5%
    ensure then
      δ: ∀e : Result | e happens today between 9am and 5pm
end
```

Contracts in descendant class *IPHONE_11_PRO* are *suitable*.

- ***Require the same or less*** $\alpha \Rightarrow \gamma$

  Clients satisfying the precondition for *SMART_PHONE* are ***not*** shocked by not being to use the same feature for *IPHONE_11_PRO*.

Inheritance and Contracts (2.5)

```
class SMART_PHONE
 get_reminders: LIST[EVENT]
  require
   α: battery_level ≥ 0.1 -- 10%
  ensure
   β: ∀e:Result | e happens today
end
```

```
class IPHONE_11_PRO
inherit SMART_PHONE redefine get_reminders end
 get_reminders: LIST[EVENT]
  require else
   γ: battery_level ≥ 0.05 -- 5%
  ensure then
   δ: ∀e:Result | e happens today between 9am and 5pm
end
```

Contracts in descendant class *IPHONE_11_PRO* are *suitable*.

- ***Ensure the same or more*** $\delta \Rightarrow \beta$

  Clients benefiting from *SMART_PHONE* are ***not*** shocked by failing to gain at least those benefits from same feature in *IPHONE_11_PRO*.

Contract Redeclaration Rule (1)

- In the context of some feature in a descendant class:
  - Use `require else` to redeclare its precondition.
  - Use `ensure then` to redeclare its postcondition.
- The resulting *runtime assertions checks* are:
  - `original_pre or else new_pre`

    ⇒ Clients ***able to satisfy*** *original_pre* will not be shocked.

    $\because$ ***true*** $\lor$ *new_pre* $\equiv$ ***true***

    A *precondition violation* will ***not*** occur as long as clients are able to satisfy what is required from the ancestor classes.
  - `original_post and then new_post`

    ⇒ ***Failing to gain*** *original_post* will be reported as an issue.

    $\because$ ***false*** $\land$ *new_post* $\equiv$ ***false***

    A *postcondition violation* occurs (as expected) if clients do not receive at least those benefits promised from the ancestor classes.

Contract Redeclaration Rule (2.1)

```
class FOO
  f
     do ...
     end
end
```

```
class BAR
inherit FOO redefine f end
  f require else new_pre
     do ...
     end
end
```

- Unspecified *original_pre* is as if declaring `require true`

$\because \textbf{\textit{true}} \lor \textit{new\_pre} \equiv \textbf{\textit{true}}$

```
class FOO
  f
     do ...
     end
end
```

```
class BAR
inherit FOO redefine f end
  f
    do ...
    ensure then new_post
    end
end
```

- Unspecified *original_post* is as if declaring `ensure true`

$\because \textbf{\textit{true}} \land \textit{new\_post} \equiv \textit{new\_post}$

Contract Redeclaration Rule (2.2)

```
class FOO
  f require
       original_pre
     do ...
     end
end
```

```
class BAR
inherit FOO redefine f end
  f
     do ...
     end
end
```

- Unspecified *new_pre* is as if declaring `require else false`

$\because \mathit{original\_pre} \lor \textbf{\textit{false}} \equiv \mathit{original\_pre}$

```
class FOO
  f
     do ...
     ensure
       original_post
     end
end
```

```
class BAR
inherit FOO redefine f end
  f
    do ...
    end
end
```

- Unspecified *new_post* is as if declaring `ensure then true`

$\because \mathit{original\_post} \land \textbf{\textit{true}} \equiv \mathit{original\_post}$

Invariant Accumulation

- Every class inherits *invariants* from all its ancestor classes.
- Since invariants are like postconditions of all features, they are "***conjoined***" to be checked at runtime.

```
class POLYGON
  vertices: ARRAY[POINT]
invariant
  vertices.count ≥ 3
end
```

```
class RECTANGLE
inherit POLYGON
invariant
  vertices.count = 4
end
```

- What is checked on a `RECTANGLE` instance at runtime:

$$(vertices.count \geq 3) \land (vertices.count = 4) \equiv (vertices.count = 4)$$

- Can `PENTAGON` be a descendant class of `RECTANGLE`?

$$(vertices.count = 5) \land (vertices.count = 4) \equiv \textit{\textbf{false}}$$

Inheritance and Contracts (3)

```
class FOO
  f
     require
       original_pre
     ensure
       original_post
     end
end
```

```
class BAR
inherit FOO redefine f end
  f
    require else
       new_pre
     ensure then
       new_post
     end
end
```

(Static) ***Design Time*** :

- $\mathit{original\_pre} \Rightarrow \mathit{new\_pre}$ should be proved as a tautology
- $\mathit{new\_post} \Rightarrow \mathit{original\_post}$ should be proved as a tautology

(Dynamic) ***Runtime*** :

- $\mathit{original\_pre} \vee \mathit{new\_pre}$ is checked
- $\mathit{original\_post} \wedge \mathit{new\_post}$ is checked

Index (1)

Index (2)