

## Subcontracting

Readings: OOSCS2 Chapters 14 – 16



EECS3311 A: Software Design  
Fall 2019

CHEN-WEI WANG

## Aspects of Inheritance



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

2 of 16

## Background of Logic (1)



Given **preconditions**  $P_1$  and  $P_2$ , we say that

$P_2$  **requires less** than  $P_1$  if

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

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

More concisely:

$$P_1 \Rightarrow P_2$$

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

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

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

3 of 16

## Background of Logic (2)



Given **postconditions** or **invariants**  $Q_1$  and  $Q_2$ , we say that

$Q_2$  **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 `q1(i: INTEGER): BOOLEAN`,

$Q_2: \text{Result} = (i > 0) \wedge (i \bmod 2 = 0)$  **ensures more** than

$Q_1: \text{Result} = (i > 0) \vee (i \bmod 2 = 0)$

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

4 of 16

## 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
  htc_phone: HUAWEI_P30_PRO
do my_phone := i_phone
  my_phone := samsung_phone
  my_phone := htc_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.

5 of 16

## Inheritance and Contracts (2.2)

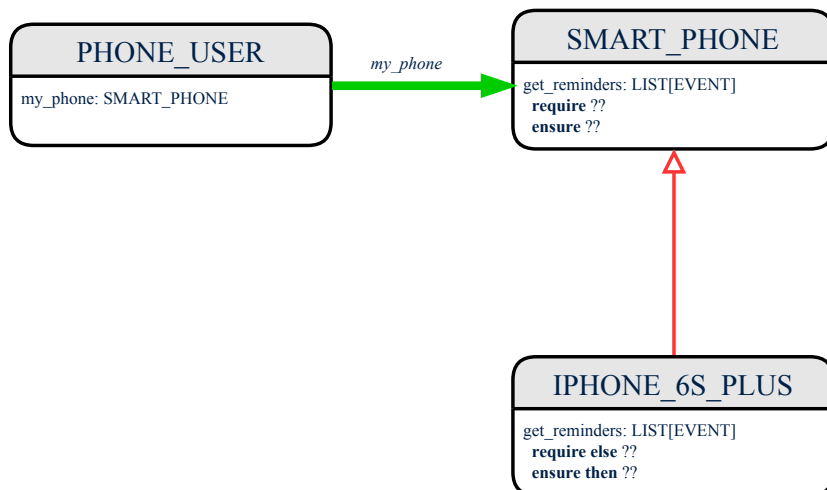
```
class SMART_PHONE
  get_reminders: LIST[EVENT]
  require
    α: battery_level ≥ 0.1 -- 10%
  ensure
    β: ∀e:Result | e happens today
end
```

```
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*.

7 of 16

## Inheritance and Contracts (2.1)



6 of 16

## 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**.  
 ( $e \text{ happens ty. or tw.} \Rightarrow e \text{ happens ty.}$ ) not tautology.  
 e.g., A client receiving today's reminders from *SMART\_PHONE* are shocked by tomorrow-only reminders from *IPHONE\_11\_PRO*.

8 of 16

## 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*.

9 of 16

## 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*.

10 of 16

## 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 precondition.
- The resulting **runtime assertions checks** are:
  - `original_pre or else new_pre`  
⇒ Clients **able to satisfy original\_pre** will not be shocked.  
∴  $true \vee new\_pre \equiv true$   
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.  
∴  $false \wedge new\_post \equiv false$   
A **postcondition violation** occurs (as expected) if clients do not receive at least those benefits promised from the ancestor classes.

11 of 16

## 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`  
∴  $true \vee new\_pre \equiv true$

```
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`  
∴  $true \wedge new\_post \equiv new\_post$

12 of 16

## 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`  
 $\therefore original\_pre \vee false \equiv original\_pre$

```
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`  
 $\therefore original\_post \wedge true \equiv original\_post$

13 of 16

## 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** :

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

(Dynamic) **Runtime** :

- $original\_pre \vee new\_pre$  is checked
- $original\_post \wedge new\_post$  is checked

15 of 16

## 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:  
 $(vertices.count \geq 3) \wedge (vertices.count = 4) \equiv (vertices.count = 4)$
- Can PENTAGON be a descendant class of RECTANGLE?  
 $(vertices.count = 5) \wedge (vertices.count = 4) \equiv false$

14 of 16

## Index (1)

- Aspects of Inheritance
- Background of Logic (1)
- Background of Logic (2)
- Inheritance and Contracts (1)
- Inheritance and Contracts (2.1)
- Inheritance and Contracts (2.2)
- Inheritance and Contracts (2.3)
- Inheritance and Contracts (2.4)
- Inheritance and Contracts (2.5)
- Contract Redeclaration Rule (1)
- Contract Redeclaration Rule (2.1)
- Contract Redeclaration Rule (2.2)
- Invariant Accumulation
- Inheritance and Contracts (3)

16 of 16