Two general Q/A sessions will be host

First session:

Location: DB 0006

Time: 3:00 PM – 4:30 PM Dec. 4th

Second session:

Location: TBD

Time: 3:00 PM – 4:30 PM Dec. 11th

Final Exam Review

EECS3311 2019 Fall

Song Wang

Exam Format

- **No data sheet**
  - *wp* rules will be given to you in the appendix
    - assignment, IF statements, Sequential compositions, Loop, etc.

- **Format**
  - Written questions
  - Design Architectures/BON class diagrams etc.
  - Code/contracts
  - Explanations/justifications/design decisions

No ETF questions on the exam.

Exam topics

- **Design by Contract**
- **Basic Eiffel Syntax**
- **BON diagrams**
- **Design Patterns**
- **Program Correctness**

Design by Contract

- Express Contracts/ Runtime Monitoring of Contracts

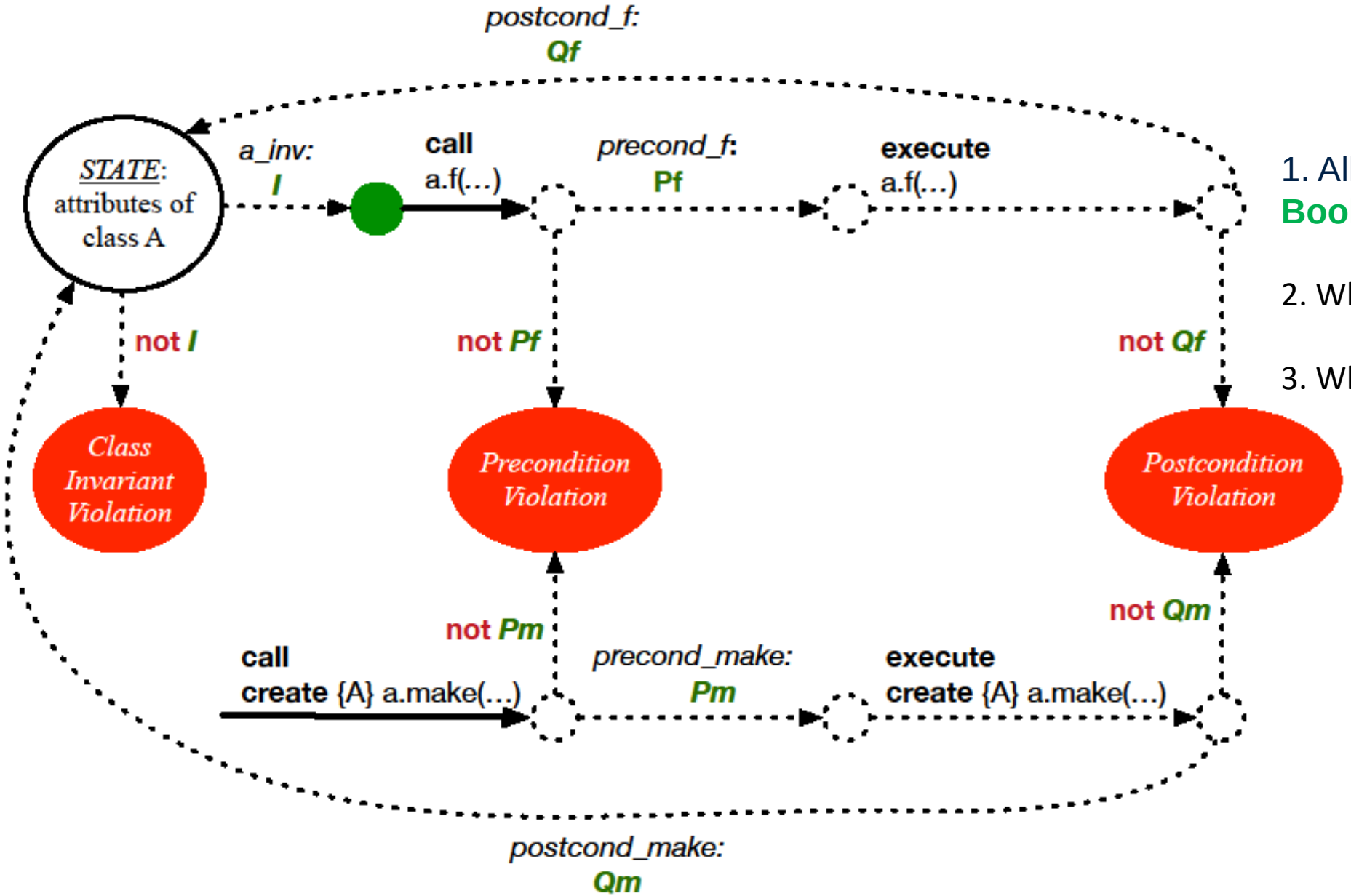
- Be able to write pre/post-conditions/invariants

- Complete post-conditions
    * reference copy, shallow copy, deep copy
    * old expressions

- Judge if contracts are appropriate
    * pre-condition and post-condition
    * invariants
    * loop invariants

- Subcontracting

- Be able to write test cases for testing expected contract violations

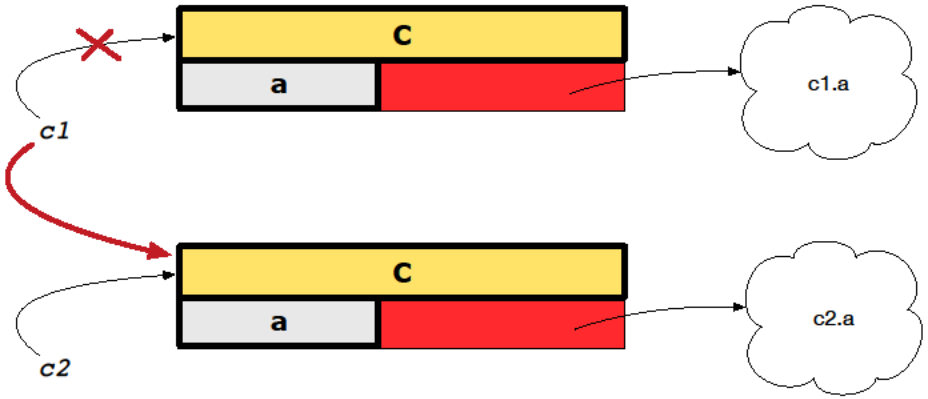


1. All contracts are specified as **Boolean expressions**

2. What happens before a feature call?

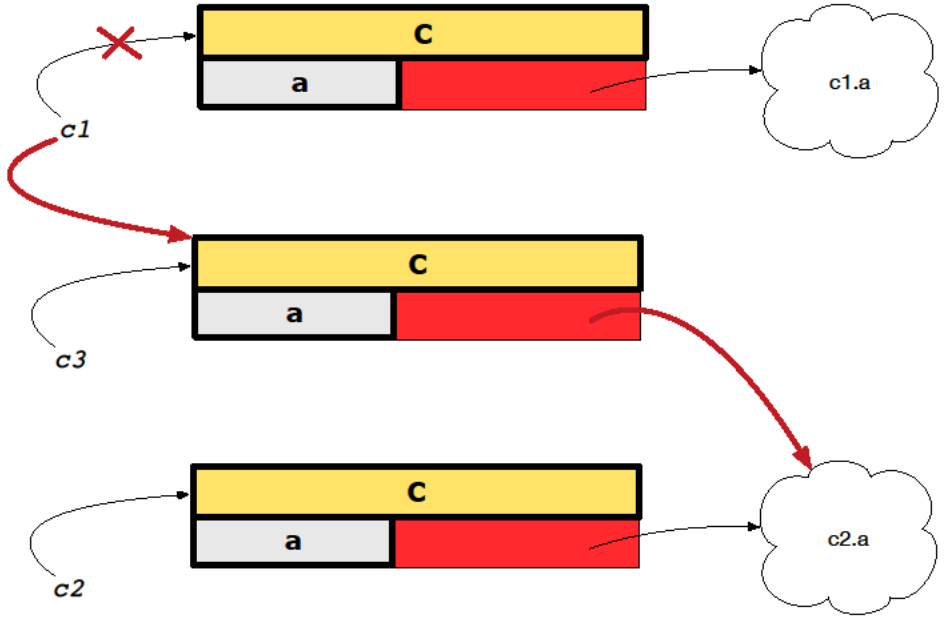
3. What happens after a feature call?

reference copy, shallow copy, and deep copy

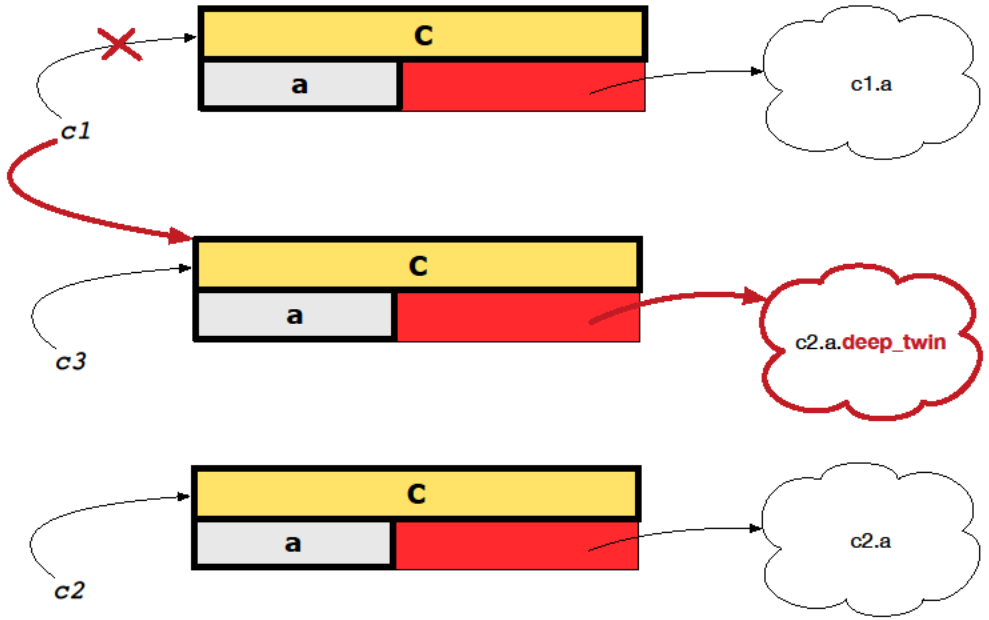


c1 := c2

c1 := c2.**twin**



c1 := c2.**deep_twin**



Your are expected to use old expression and deep copy

```
class BANK
  deposit_on_v5 (n: STRING; a: INTEGER)
    require across accounts as acc some acc.item.owner ~ n end
      local i: INTEGER
    do
      -- same loop as in version 1
      -- wrong implementation: also deposit in the first account
      accounts[accounts.lower].deposit(a)
    ensure
      num_of_accounts_unchanged: accounts.count = old accounts.count
      balance_of_n_increased:
        account_of (n).balance = old account_of (n).balance + a
      others_unchanged :
        across old accounts.deep_twin as cursor
        all cursor.item.owner /~ n implies
            cursor.item ~ account_of (cursor.item.owner)
        end
    end
end
```

```
class BANK
  deposit_on_v4 (n: STRING; a: INTEGER)
    require across accounts as acc some acc.item.owner ~ n end
    local i: INTEGER
    do
      -- same loop as in version 1
      -- wrong implementation: also deposit in the first account
      accounts[accounts.lower].deposit(a)
    ensure
      num_of_accounts_unchanged: accounts.count = old accounts.count
      balance_of_n_increased:
        account_of (n).balance = old account_of (n).balance + a
      others_unchanged :
        across old accounts.twin as cursor
        all cursor.item.owner /~ n implies
            cursor.item ~ account_of (cursor.item.owner)
      end
    end
end
```

What's the problem of the above post-condition?

```
matching_keys (d1: DATA1; d2: DATA2): ITERABLE[KEY]
        -- Keys that are associated with data items 'd1' and 'd2'.
    local
        ks: LINKED_LIST[KEY]

    do
        ...
        Result := ks
    ensure
        result_contains_correct_keys_only: -- TODO:
            across
                Result is k
            all
                data_items_1 [keys.index_of (k, 1)] ~ d1
                and
                data_items_2[k] ~ d2
            end
        correct_keys_are_in_result: -- TODO:
            across
                Current is tuple
            all
                tuple.d1 ~ d1 and tuple.d2 ~ d2
                implies
                (across
                    Result is k
                some
                    k ~ tuple.k
                end)
            end
```

Be able to use “Result” and “Current” in your post-condition

You are expected to fill post-conditions

You are expected to be able to find out the problems introduced by inappropriate Subcontracting

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

```
class IPHONE_6S_PLUS
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
end
```

Contracts in descendant class *IPHONE_6S_PLUS* are *not suitable*.

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

Basic Eiffel Syntax

- You are expected to use `across … as/is … some/all … end

Basic Eiffel Syntax

- You are expected to use `across … as/is … some/all … end

- Inheritance
  - **Polymorphism**: An object variable may have multiple possible shapes
  - **Dynamic binding**: a feature's implementation can be dynamically decided

Basic Eiffel Syntax

- You are expected to use `across … as/is … some/all … end

- Inheritance
  - **Polymorphism**: An object variable may have multiple possible shapes
  - **Dynamic binding**: a feature's implementation can be dynamically decided

You are expected to be able to explain the two concepts with real code examples

BON Diagrams

- deferred class vs. effective class

- client supplier vs. inheritance

- deferred vs. effective vs. redefined features


LIST[G]*
LINKED_LIST[G]+
ARRAYED_LIST[G]+
LIST[LIST[PERSON]]*
LINKED_LIST[INTEGER]+
ARRAYED_LIST[G]+
DATABASE[G]*
DATABASE_V1[G]+
DATABASE_V2[G]+

DATABASE[G]*

feature {**NONE**} -- Implementation
  data: **ARRAY**[G]

feature -- Commands
  add_item* (g: G)
    -- Add new item `g` into database.
  require
    non_existing_item: ¬ exists (g)
  ensure
    size_incremented: count = **old** count + 1
    item_added: exists (g)

feature -- Queries
  count+: **INTEGER**
    -- Number of items stored in database
  ensure
    correct_result: **Result** = data.count

  exists* (g: G): **BOOLEAN**
    -- Does item `g` exist in database?
  ensure
    correct_result: **Result** = (∃i : 1 ≤ i ≤ count : data[i] ~ g)

DATABASE_V1[G]+

feature {**NONE**} -- Implementation
  data: **ARRAY**[G]

feature -- Commands
  add_item+ (g: G)
    -- Append new item `g` into end of `data`.

feature -- Queries
  count+: **INTEGER**
    -- Number of items stored in database

  exists+ (g: G): **BOOLEAN**
    -- Perform a linear search on `data` array.

DATABASE_V2[G]+

feature {**NONE**} -- Implementation
  data: **ARRAY**[G]

feature -- Commands
  add_item++ (g: G)
    -- Insert new item `g` into the right slot of `data`.

feature -- Queries
  count+: **INTEGER**
    -- Number of items stored in database

  exists++ (g: G): **BOOLEAN**
    -- Perform a binary search on `data` array.

  invariant
    sorted_data: ∀i : 1 ≤ i < count : data[i] < data[i + 1]

Design Patterns

* Singleton Pattern
* Iterator Pattern
* Composite Pattern
* Visitor Pattern
* State Pattern
* Observer Pattern

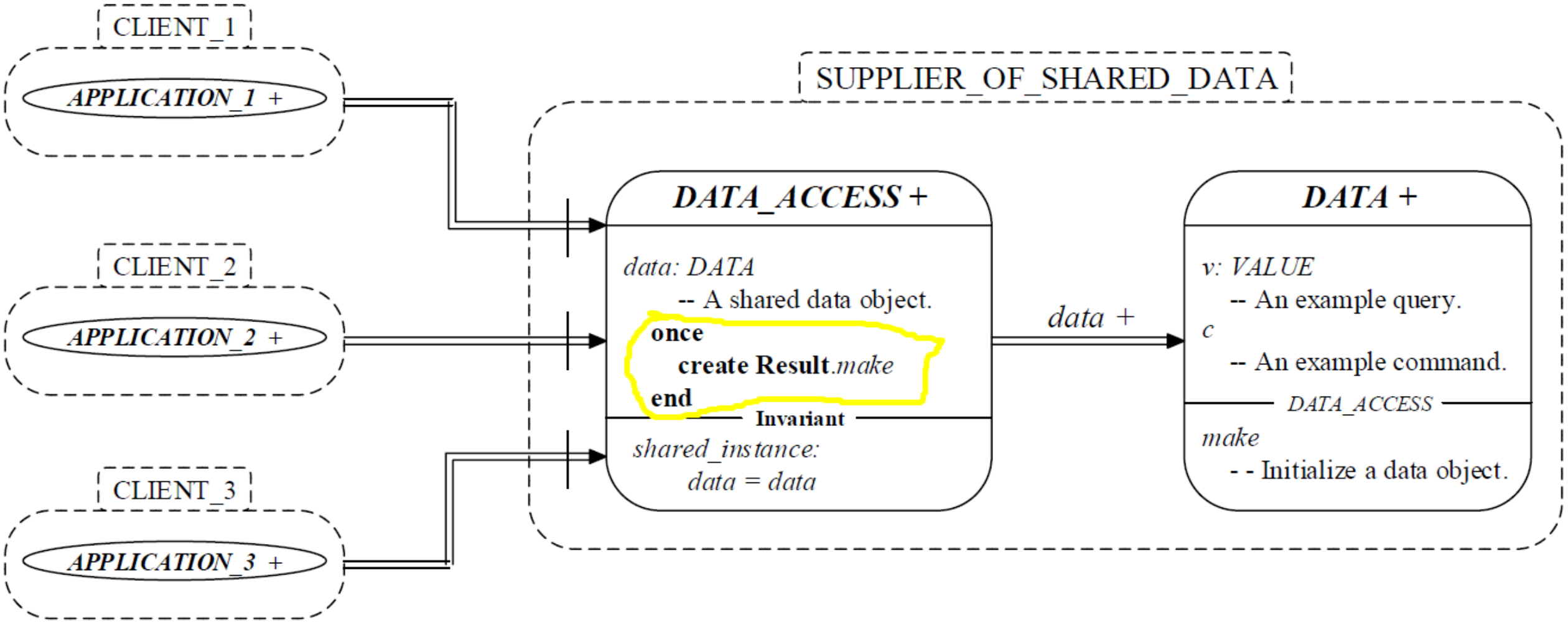
Design Patterns

* Singleton Pattern
* Iterator Pattern
* Composite Pattern
* Visitor Pattern
* State Pattern
* Observer Pattern

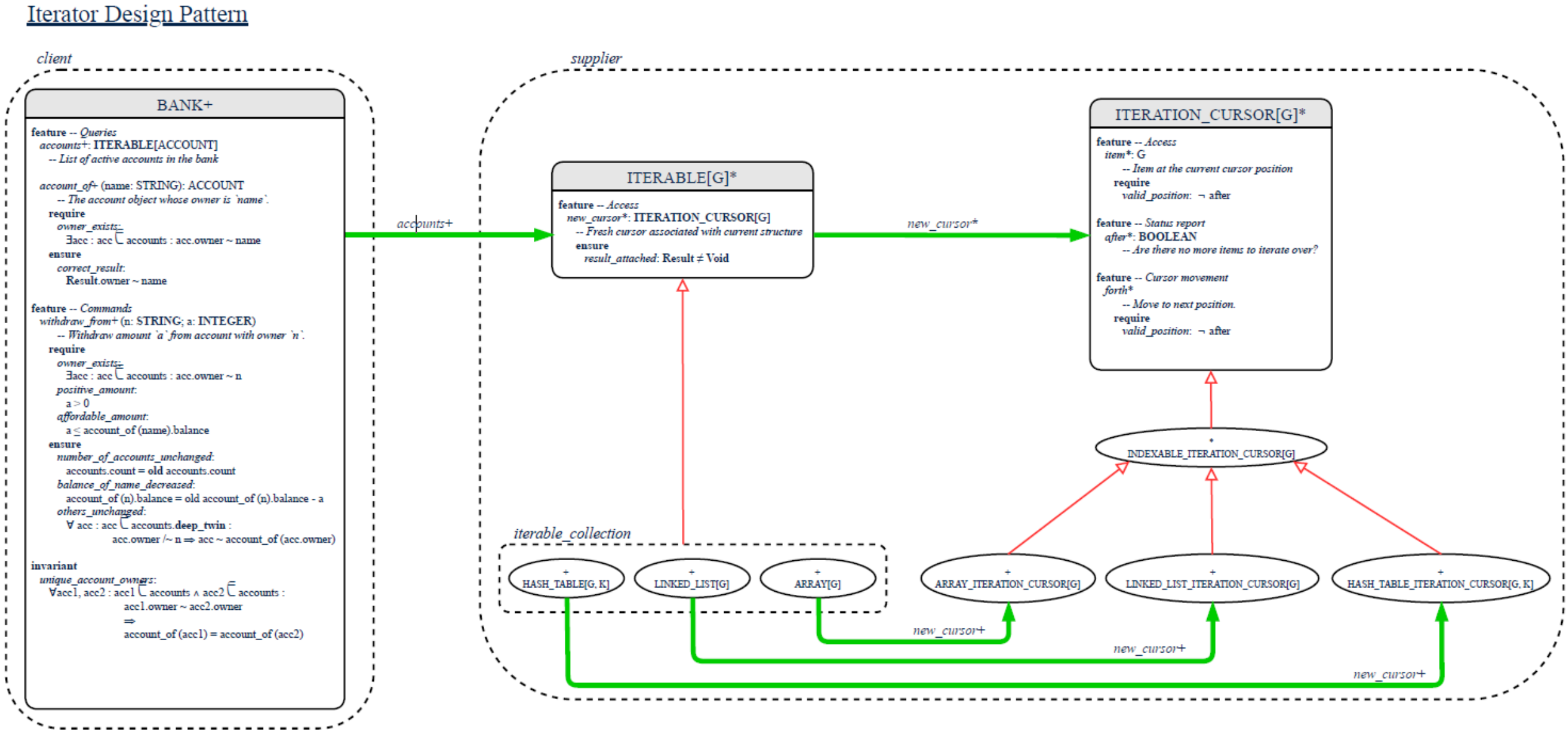
For each pattern:

- Application Context
- Architecture
- Implementation/Contracts
- Usage/Runtime/Tests
- Polymorphism and Dynamic Binding

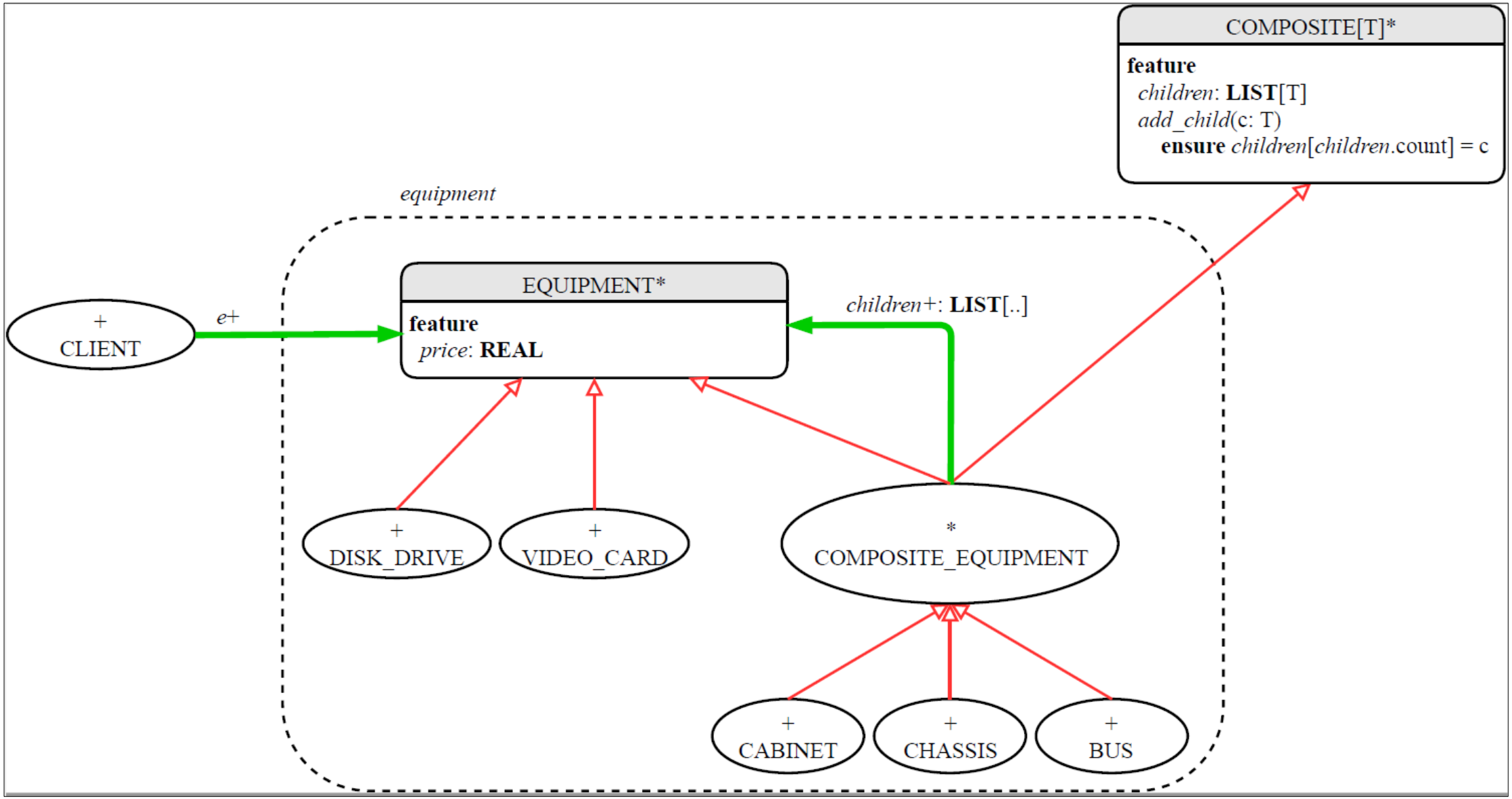
Singleton Pattern



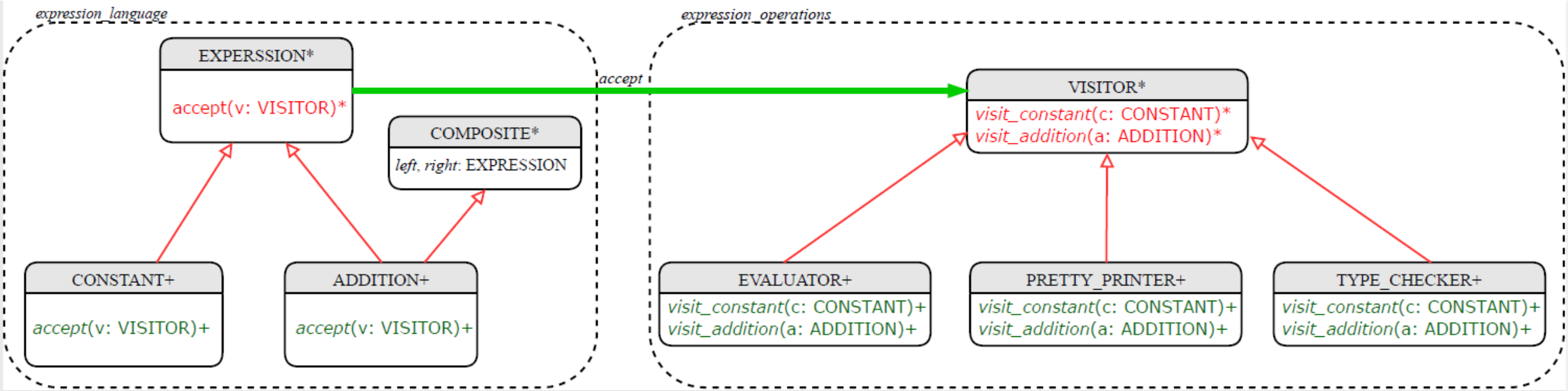
Iterator Pattern



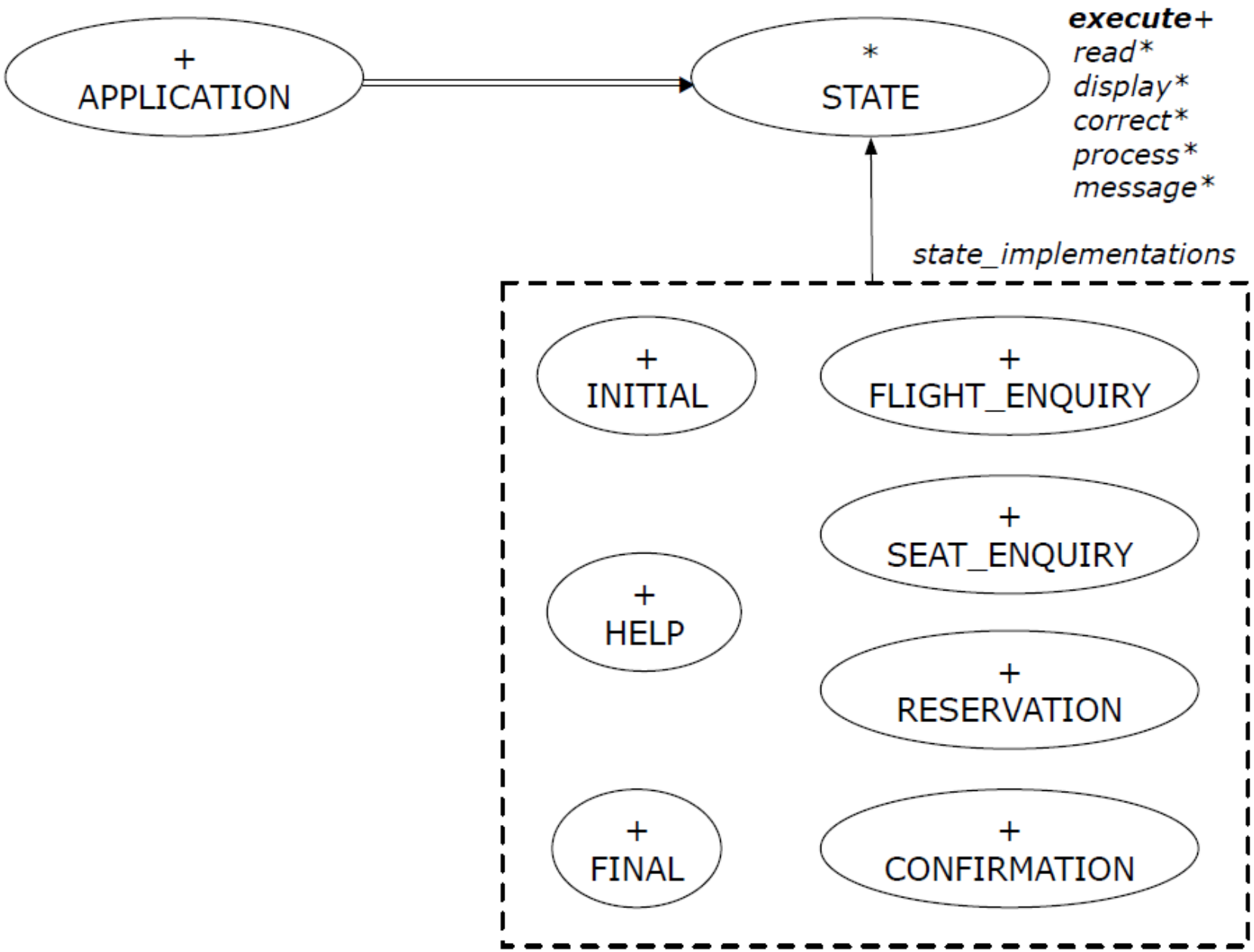
Composite Pattern



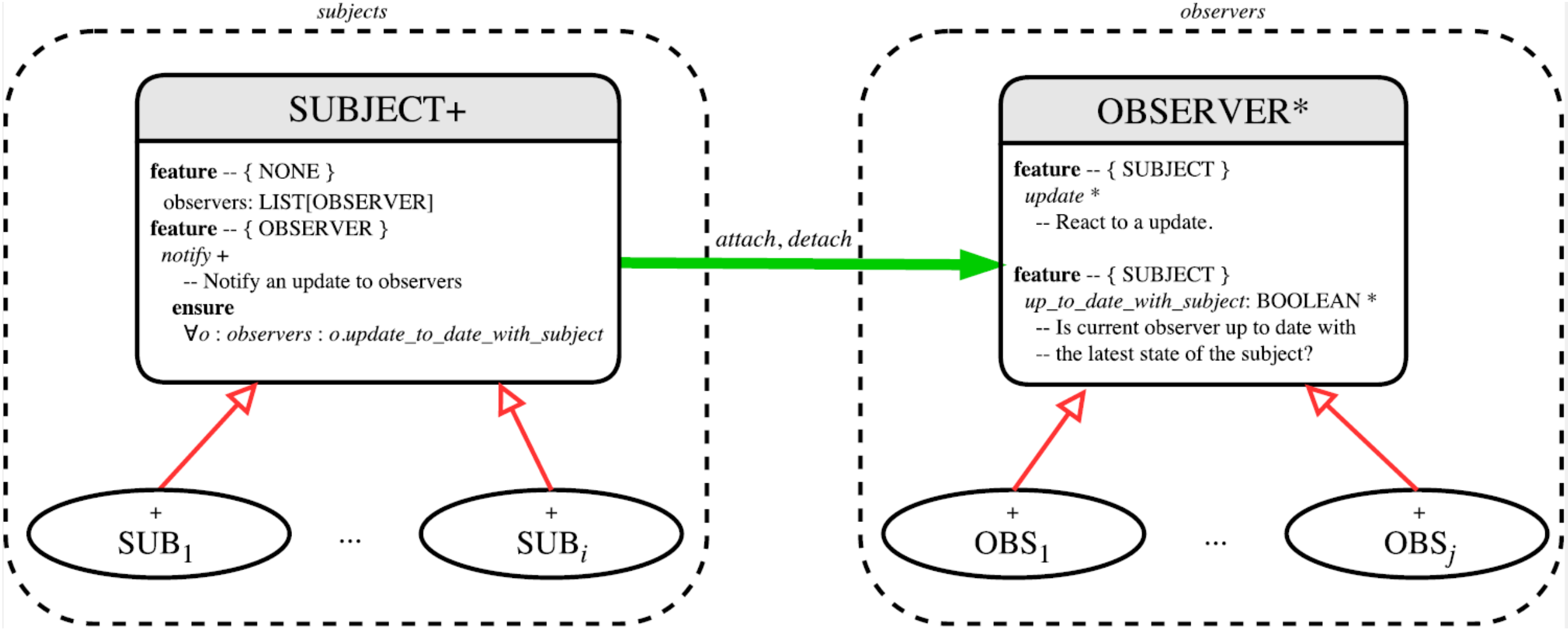
Visitor Pattern



Sate Pattern



Observer Pattern

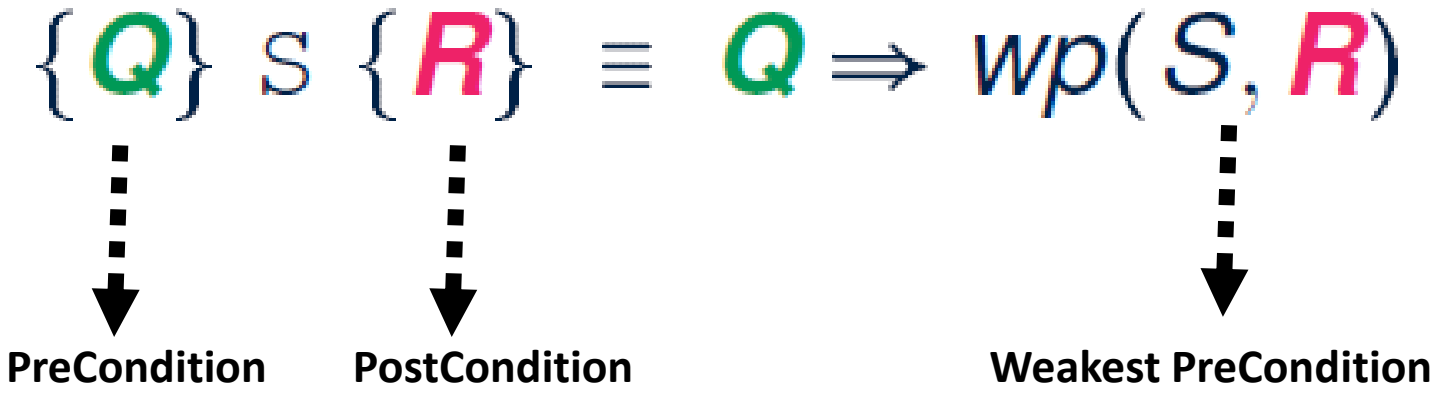


Program Correctness

Calculate the weakest precondition

- - formulate given program to {**Q**} S {**R**}
- - calculate **wp(S, R)**
- - prove/disprove **Q => wp(S, R)**
- - Proof format **MUST** conform to the equational style discussed in lecture.

Prove the program is correct



You are expected to be able to work on the following statements and their combinations

- Assignment Statements
- If Statements (***if … then … else … end***)
- Sequential compositions (*S1 ; S2*)
- Loop Statements

Assignment Statements

$$wp(\underbrace{\texttt{x := e}}_{S}, R) = R[x := e]$$

Rules:

$$
\begin{aligned}
& wp(\texttt{x := x + 1}, x > x_0) \\
= \; & \{\textit{Rule of } wp\textit{: Assignments}\} \\
& x > x_0[x := x_0 + 1] \\
= \; & \{\textit{Replacing } x \textit{ by } x_0 + 1\} \\
& x_0 + 1 > x_0 \\
= \; & \{1 > 0 \textit{ always true}\}
\end{aligned}
$$

If Statements

$$wp(\texttt{if } B \texttt{ then } S_1 \texttt{ else } S_2 \texttt{ end}, R) = \left(\begin{array}{l} B \Rightarrow wp(S_1, R) \\ \wedge \\ \neg B \Rightarrow wp(S_2, R) \end{array} \right)$$

Sequential compositions

$$wp(S_1 \; ; \; S_2, \boldsymbol{R}) = wp(S_1, wp(S_2, \boldsymbol{R}))$$

Loop Statements

total** correctness* = ***partial** correctness* + ***termination

$\{Q\}$ `from` S_{init} `invariant` I `until` B `loop` S_{body} `variant` V `end` $\{R\}$

- A loop is ***partially** correct* if:
  1. Given precondition ***Q***, the initialization step S_{init} establishes ***LI*** I.

     $$\{Q\}\ S_{init}\ \{I\}$$

  2. At the end of S_{body}, if not yet to exit, ***LI*** I is maintained.

     $$\{I \wedge \neg B\}\ S_{body}\ \{I\}$$

  3. If ready to exit and ***LI*** I maintained, postcondition ***R*** is established.

     $$I \wedge B \Rightarrow R$$

- loop *terminates* if:
  4. Given ***LI*** I, and not yet to exit, S_{body} maintains ***LV*** V as non-negative.

     $$\{I \wedge \neg B\}\ S_{body}\ \{V \geq 0\}$$

  5. Given ***LI*** I, and not yet to exit, S_{body} decrements ***LV*** V.

     $$\{I \wedge \neg B\}\ S_{body}\ \{V < V_0\}$$

Course Evaluation

https://courseevaluations.yorku.ca/