Design-by-Contract (Dbc)
Test-Driven Development (TDD)

Readings: OOSC2 Chapter 11

EECS3311: Software Design
Fall 2017

CHEN-WEI WANG

Terminology: Contract, Client, Supplier

- A *supplier* implements/provides a service (e.g., microwave).
- A *client* uses a service provided by some supplier.
  - The client must follow certain instructions to obtain the service (e.g., supplier **assumes** that client powers on, closes door, and heats something that is not explosive).
  - If instructions are followed, the client would **expect** that the service does what is required (e.g., a lunch box is heated).
  - The client does not care how the supplier implements it.
- What then are the *benefits* and *obligations* os the two parties?

	benefits	*obligations*
CLIENT	obtain a service	follow instructions
SUPPLIER	give instructions	provide a service

- There is a *contract* between two parties, violated if:
  - The instructions are not followed. [Client's fault]
  - Instructions followed, but service not satisfactory. [Supplier's fault]

Client, Supplier, Contract in OOP (1)

```
class Microwave {
  private boolean on;
  private boolean locked;
  void power() {on = true;}
  void lock() {locked = true;}
  void heat(Object stuff) {
    /* Assume: on && locked */
    /* stuff not explosive. */
  } }
```

```
class MicrowaveUser {
  public static void main(...) {
    Microwave m = new Microwave();
    Object obj = ???;
    m.power(); m.lock();]
    m.heat(obj);
  } }
```

Method call ***m.heat(obj)*** indicates a client-supplier relation.

- **Client**: resident class of the method call [`MicrowaveUser`]
- **Supplier**: type of context object (or call target) `m` [`Microwave`]

Client, Supplier, Contract in OOP (2)

```
class Microwave {
  private boolean on;
  private boolean locked;
  void power() {on = true;}
  void lock() {locked = true;}
  void heat(Object stuff) {
    /* Assume: on && locked */
    /* stuff not explosive. */ }
```

```
class MicrowaveUser {
  public static void main(...) {
    Microwave m = new Microwave();
    Object obj = ???;
    m.power(); m.lock();
    m.heat(obj);
}}
```

- The *contract* is *honoured* if:

  Right **before** the method call:
  - State of `m` is as assumed: `m.on==true` and `m.locked==ture`
  - The input argument `obj` is valid (i.e., not explosive).

  Right **after** the method call: `obj` is properly heated.

- If any of these fails, there is a *contract violation*.
  - `m.on` or `m.locked` is `false` ⇒ `MicrowaveUser`'s fault.
  - `obj` is an explosive ⇒ `MicrowaveUser`'s fault.
  - A fault from the client is identified ⇒ Method call will not start.
  - Method executed but `obj` not properly heated ⇒ `Microwave`'s fault

What is a Good Design?

- A “good” design should *explicitly* and *unambiguously* describe the *contract* between **clients** (e.g., users of Java classes) and **suppliers** (e.g., developers of Java classes).
  We such a contractual relation a *specification*.
- When you conduct *software design*, you should be guided by the “appropriate” contracts between users and developers.
  - Instructions to **clients** should *not be unreasonable*.
    e.g., asking them to assemble internal parts of a microwave
  - Working conditions for **suppliers** should *not be unconditional*.
    e.g., expecting them to produce a microwave which can safely heat an explosive with its door open!
  - You as a designer should strike proper balance between **obligations** and **benefits** of clients and suppliers.
    e.g., What is the obligation of a binary-search user (also benefit of a binary-search implementer)? [The input array is sorted.]
  - Upon contract violation, there should be the fault of **only one side**.
  - This design process is called *Design by Contract (DbC)*.

A Simple Problem: Bank Accounts

Provide an object-oriented solution to the following problem:

REQ1: Each account is associated with the *name* of its owner (e.g., `"Jim"`) and an integer *balance* that is always positive.

REQ2: We may *withdraw* an integer amount from an account.

REQ3: Each bank stores a list of *accounts*.

REQ4: Given a bank, we may *add* a new account in it.

REQ5: Given a bank, we may *query* about the associated account of a owner (e.g., the account of `"Jim"`).

REQ6: Given a bank, we may *withdraw* from a specific account, identified by its name, for an integer amount.

Let's first try to work on **REQ1** and **REQ2** in Java.
This may not be as easy as you might think!

Playing the Various Versions in Java

- **Download** the project archive (a zip file) here:
  `http://www.eecs.yorku.ca/~jackie/teaching/lectures/src/2017/F/EECS3311/DbCIntro.zip`
- Follow this tutorial to learn how to **import** an project archive into your workspace in Eclipse:
  `https://youtu.be/h-rgdQZg2qY`
- Follow this tutorial to learn how to **enable** assertions in Eclipse:
  `https://youtu.be/OEgRV4a5Dzg`

Version 1: An Account Class

```
1  public class AccountV1 {
2        private String owner;
3        private int balance;
4        public String getOwner() { return owner; }
5        public int getBalance() { return balance; }
6        public AccountV1(String owner, int balance) {
7              this.owner = owner; this.balance = balance;
8        }
9        public void withdraw(int amount) {
10             this.balance = this.balance - amount;
11       }
12       public String toString() {
13             return owner + "'s current balance is: " + balance;
14       }
15 }
```

- Is this a good design? Recall **REQ1**: Each account is associated with ... an integer balance that is *always positive*.
- This requirement is *not* reflected in the above Java code.

Version 1: Why Not a Good Design? (1)

```
public class BankAppV1 {
  public static void main(String[] args) {
    System.out.println("Create an account for Alan with balance -10:");
    AccountV1 alan = new AccountV1("Alan", -10);
    System.out.println(alan);
```

Console Output:

```
Create an account for Alan with balance -10:
Alan's current balance is: -10
```

- Executing `AccountV1`'s constructor results in an account object whose *state* (i.e., values of attributes) is *invalid* (i.e., Alan's balance is negative). ⇒ Violation of **REQ1**
- Unfortunately, both client and supplier are to be blamed: `BankAppV1` passed an invalid balance, but the API of `AccountV1` does not require that! ⇒ A lack of defined contract

Version 1: Why Not a Good Design? (2)

```
public class BankAppV1 {
  public static void main(String[] args) {
    System.out.println("Create an account for Mark with balance 100:");
    AccountV1 mark = new AccountV1("Mark", 100);
    System.out.println(mark);
    System.out.println("Withdraw -1000000 from Mark's account:");
    mark.withdraw(-1000000);
    System.out.println(mark);
```

```
Create an account for Mark with balance 100:
Mark's current balance is: 100
Withdraw -1000000 from Mark's account:
Mark's current balance is: 1000100
```

- Mark's account state is always valid (i.e., 100 and 1000100).
- Withdraw amount is never negative! ⇒ Violation of **REQ2**
- Again a lack of contract between `BankAppV1` and `AccountV1`.

Version 1: Why Not a Good Design? (3)

```
public class BankAppV1 {
  public static void main(String[] args) {
    System.out.println("Create an account for Tom with balance 100:");
    AccountV1 tom = new AccountV1("Tom", 100);
    System.out.println(tom);
    System.out.println("Withdraw 150 from Tom's account:");
    tom.withdraw(150);
    System.out.println(tom);
```

```
Create an account for Tom with balance 100:
Tom's current balance is: 100
Withdraw 150 from Tom's account:
Tom's current balance is: -50
```

- Withdrawal was done via an "appropriate" reduction, but the resulting balance of Tom is *invalid*. ⇒ Violation of **REQ1**
- Again a lack of contract between `BankAppV1` and `AccountV1`.

Version 1: How Should We Improve it?

- *Preconditions* of a method specify the precise circumstances under which that method can be executed.
  - Precond. of `divide(int x, int y)`? [`y != 0`]
  - Precond. of `binSearch(int x, int[] xs)`? [`xs` is sorted]
- The best we can do in Java is to encode the *logical negations* of preconditions as *exceptions*:
  - `divide(int x, int y)`
    throws `DivisionByZeroException` when `y == 0`.
  - `binSearch(int x, int[] xs)`
    throws `ArrayNotSortedException` when `xs` is *not* sorted.
  - It should be preferred to design your method by specifying the *preconditions* (i.e., *valid* inputs) it requires, rather than the *exceptions* (i.e., *erroneous* inputs) that it might trigger.
- Create Version 2 by adding *exceptional conditions* (an *approximation* of *preconditions*) to the constructor and `withdraw` method of the `Account` class.

Version 2: Added Exceptions to Approximate Method Preconditions

```
1  public class AccountV2 {
2    public AccountV2(String owner, int balance) throws
3       BalanceNegativeException
4    {
5      if( balance < 0 ) { /* negated precondition */
6        throw new BalanceNegativeException(); }
7      else { this.owner = owner; this.balance = balance; }
8    }
9    public void withdraw(int amount) throws
10      WithdrawAmountNegativeException, WithdrawAmountTooLargeException {
11     if( amount < 0 ) { /* negated precondition */
12       throw new WithdrawAmountNegativeException(); }
13     else if ( balance < amount ) { /* negated precondition */
14       throw new WithdrawAmountTooLargeException(); }
15     else { this.balance = this.balance - amount; }
16   }
```

Version 2: Why Better than Version 1? (1)

```
1  public class BankAppV2 {
2    public static void main(String[] args) {
3      System.out.println("Create an account for Alan with balance -10:");
4      try {
5        AccountV2 alan = new AccountV2("Alan", -10);
6        System.out.println(alan);
7      }
8      catch (BalanceNegativeException bne) {
9        System.out.println("Illegal negative account balance.");
10     }
```

```
Create an account for Alan with balance -10:
Illegal negative account balance.
```

L6: When attempting to call the constructor `AccountV2` with a negative balance `-10`, a `BalanceNegativeException` (i.e., *precondition* violation) occurs, *preventing further operations upon this invalid object*.

Version 2: Why Better than Version 1? (2.1)

```
1  public class BankAppV2 {
2    public static void main(String[] args) {
3      System.out.println("Create an account for Mark with balance 100:");
4      try {
5        AccountV2 mark = new AccountV2("Mark", 100);
6        System.out.println(mark);
7        System.out.println("Withdraw -1000000 from Mark's account:");
8        mark.withdraw(-1000000);
9        System.out.println(mark);
10     }
11     catch (BalanceNegativeException bne) {
12       System.out.println("Illegal negative account balance.");
13     }
14     catch (WithdrawAmountNegativeException wane) {
15       System.out.println("Illegal negative withdraw amount.");
16     }
17     catch (WithdrawAmountTooLargeException wane) {
18       System.out.println("Illegal too large withdraw amount.");
19     }
```

Version 2: Why Better than Version 1? (2.2)

Console Output:

```
Create an account for Mark with balance 100:
Mark's current balance is: 100
Withdraw -1000000 from Mark's account:
Illegal negative withdraw amount.
```

- **L9**: When attempting to call method `withdraw` with a positive but too large amount `150`, a `WithdrawAmountTooLargeException` (i.e., *precondition* violation) occurs, *preventing the withdrawal from proceeding*.
- We should observe that *adding preconditions* to the supplier `BankV2`'s code forces the client `BankAppV2`'s code to *get complicated by the* `try-catch` *statements*.
- Adding clear contract (*preconditions* in this case) to the design ***should not*** be at the cost of complicating the client's code!!

Version 2: Why Better than Version 1? (3.1)

```
1  public class BankAppV2 {
2    public static void main(String[] args) {
3      System.out.println("Create an account for Tom with balance 100:");
4      try {
5        AccountV2 tom = new AccountV2("Tom", 100);
6        System.out.println(tom);
7        System.out.println("Withdraw 150 from Tom's account:");
8        tom.withdraw(150);
9        System.out.println(tom);
10     }
11     catch (BalanceNegativeException bne) {
12       System.out.println("Illegal negative account balance.");
13     }
14     catch (WithdrawAmountNegativeException wane) {
15       System.out.println("Illegal negative withdraw amount.");
16     }
17     catch (WithdrawAmountTooLargeException wane) {
18       System.out.println("Illegal too large withdraw amount.");
19     }
```

Version 2: Why Better than Version 1? (3.2)

Console Output:

```
Create an account for Tom with balance 100:
Tom's current balance is: 100
Withdraw 150 from Tom's account:
Illegal too large withdraw amount.
```

- **L9**: When attempting to call method `withdraw` with a negative amount `-1000000`, a `WithdrawAmountNegativeException` (i.e., *precondition* violation) occurs, *preventing the withdrawal from proceeding*.
- We should observe that due to the *added preconditions* to the supplier `BankV2`'s code, the client `BankAppV2`'s code is forced to *repeat the long list of the `try-catch` statements*.
- Indeed, adding clear contract (*preconditions* in this case) ***should not*** be at the cost of complicating the client's code!!

Version 2: Why Still Not a Good Design? (1)

```
1  public class AccountV2 {
2    public AccountV2(String owner, int balance) throws
3       BalanceNegativeException
4    {
5      if( balance < 0 ) { /* negated precondition */
6        throw new BalanceNegativeException(); }
7      else { this.owner = owner; this.balance = balance; }
8    }
9    public void withdraw(int amount) throws
10      WithdrawAmountNegativeException, WithdrawAmountTooLargeException {
11     if( amount < 0 ) { /* negated precondition */
12       throw new WithdrawAmountNegativeException(); }
13     else if ( balance < amount ) { /* negated precondition */
14       throw new WithdrawAmountTooLargeException(); }
15     else { this.balance = this.balance - amount; }
16   }
```

- Are all the *exception* conditions (¬ *preconditions*) appropriate?
- What if `amount == balance` when calling `withdraw`?

Version 2: Why Still Not a Good Design? (2.1)

```
1  public class BankAppV2 {
2    public static void main(String[] args) {
3      System.out.println("Create an account for Jim with balance 100:");
4      try {
5        AccountV2 jim = new AccountV2("Jim", 100);
6        System.out.println(jim);
7        System.out.println("Withdraw 100 from Jim's account:");
8        jim.withdraw(100);
9        System.out.println(jim);
10     }
11     catch (BalanceNegativeException bne) {
12       System.out.println("Illegal negative account balance.");
13     }
14     catch (WithdrawAmountNegativeException wane) {
15       System.out.println("Illegal negative withdraw amount.");
16     }
17     catch (WithdrawAmountTooLargeException wane) {
18       System.out.println("Illegal too large withdraw amount.");
19     }
```

Version 2: Why Still Not a Good Design? (2.2)

```
Create an account for Jim with balance 100:
Jim's current balance is: 100
Withdraw 100 from Jim's account:
Jim's current balance is: 0
```

L9: When attempting to call method `withdraw` with an amount `100` (i.e., equal to Jim's current balance) that would result in a **zero** balance (clearly a violation of **REQ1**), there should have been a *precondition* violation.

Supplier `AccountV2`'s *exception* condition `balance < amount` has a *missing case*:

- Calling `withdraw` with `amount == balance` will also result in an invalid account state (i.e., the resulting account balance is **zero**).
- ∴ **L13** of `AccountV2` should be `balance <= amount`.

Version 2: How Should We Improve it?

- **Even without** fixing this insufficient *precondition*, we could have avoided the above scenario by *checking at the end of each method that the resulting account is valid*.

  ⇒ We consider the condition `this.balance > 0` as *invariant* throughout the lifetime of all instances of `Account`.

- *Invariants* of a class specify the precise conditions which all instances/objects of that class must satisfy.
  - Inv. of `CSMajoarStudent`? [`gpa >= 4.5`]
  - Inv. of `BinarySearchTree`? [in-order trav. → sorted key seq.]
- The best we can do in Java is encode invariants as *assertions*:
  - `CSMajorStudent`: **`assert`** `this.gpa >= 4.5`
  - `BinarySearchTree`: **`assert`** `this.inOrder() is sorted`
  - Unlike exceptions, assertions are not in the class/method API.
- Create Version 3 by adding *assertions* to the end of constructor and `withdraw` method of the `Account` class.

Version 3: Added Assertions to Approximate Class Invariants

```
1  public class AccountV3 {
2    public AccountV3(String owner, int balance) throws
3       BalanceNegativeException
4    {
5      if(balance < 0) { /* negated precondition */
6        throw new BalanceNegativeException(); }
7      else { this.owner = owner; this.balance = balance; }
8      assert this.getBalance() > 0 : "Invariant:  positive balance";
9    }
10   public void withdraw(int amount) throws
11      WithdrawAmountNegativeException, WithdrawAmountTooLargeException {
12     if(amount < 0) { /* negated precondition */
13       throw new WithdrawAmountNegativeException(); }
14     else if (balance < amount) { /* negated precondition */
15       throw new WithdrawAmountTooLargeException(); }
16     else { this.balance = this.balance - amount; }
17     assert this.getBalance() > 0 : "Invariant:  positive balance";
18   }
```

Version 3: Why Better than Version 2?

```
1  public class BankAppV3 {
2    public static void main(String[] args) {
3      System.out.println("Create an account for Jim with balance 100:");
4      try { AccountV3 jim = new AccountV3("Jim", 100);
5            System.out.println(jim);
6            System.out.println("Withdraw 100 from Jim's account:");
7            jim.withdraw(100);
8            System.out.println(jim); }
9            /* catch statements same as this previous slide:
10            * Version 2: Why Still Not a Good Design? (2.1) */
```

```
Create an account for Jim with balance 100:
Jim's current balance is: 100
Withdraw 100 from Jim's account:
Exception in thread "main"
    java.lang.AssertionError: Invariant: positive balance
```

L8: Upon completion of `jim.withdraw(100)`, Jim has a **zero** balance, an assertion failure (i.e., *invariant* violation) occurs, *preventing further operations on this invalid account object*.

Version 3: Why Still Not a Good Design? (1)

Let's review what we have added to the method `withdraw`:

- From Version 2: *exceptions* encoding **negated** *preconditions*
- From Version 3: *assertions* encoding the *class invariants*

```
1  public class AccountV3 {
2    public void withdraw(int amount) throws
3        WithdrawAmountNegativeException, WithdrawAmountTooLargeException {
4      if( amount < 0 ) { /* negated precondition */
5        throw new WithdrawAmountNegativeException(); }
6      else if ( balance < amount ) { /* negated precondition */
7        throw new WithdrawAmountTooLargeException(); }
8      else { this.balance = this.balance - amount; }
9      assert this.getBalance() > 0 : "Invariant:  positive balance"; }
```

However, there is ***no*** *contract* in `withdraw` which specifies:

- Obligations of supplier (`AccountV3`) if preconditions are met.
- Benefits of client (`BankAppV3`) after meeting preconditions.

  ⇒ We illustrate how problematic this can be by creating Version 4, where deliberately mistakenly implement `withdraw`.

Version 4: What If the Implementation of `withdraw` is Wrong? (1)

```
1  public class AccountV4 {
2    public void withdraw(int amount) throws
3      WithdrawAmountNegativeException, WithdrawAmountTooLargeException
4    { if(amount < 0) { /* negated precondition */
5        throw new WithdrawAmountNegativeException(); }
6      else if (balance < amount) { /* negated precondition */
7        throw new WithdrawAmountTooLargeException(); }
8      else { /* WRONT IMPLEMENTATION */
9        this.balance = this.balance + amount; }
10     assert this.getBalance() > 0 :
11       owner + "Invariant: positive balance"; }
```

- Apparently the implementation at **L11** is ***wrong***.
- Adding a positive amount to a valid (positive) account balance would not result in an invalid (negative) one.
  ⇒ The **class invariant** will ***not*** catch this flaw.
- When something goes wrong, a good *design* (with an appropriate *contract*) should report it via a *contract violation*.

Version 4: What If the Implementation of `withdraw` is Wrong? (2)

```
1  public class BankAppV4 {
2    public static void main(String[] args) {
3      System.out.println("Create an account for Jeremy with balance 100:");
4      try { AccountV4 jeremy = new AccountV4("Jeremy", 100);
5            System.out.println(jeremy);
6            System.out.println("Withdraw 50 from Jeremy's account:");
7            jeremy.withdraw(50);
8            System.out.println(jeremy); }
9      /* catch statements same as this previous slide:
10      * Version 2: Why Still Not a Good Design? (2.1) */
```

```
Create an account for Jeremy with balance 100:
Jeremy's current balance is: 100
Withdraw 50 from Jeremy's account:
Jeremy's current balance is: 150
```

L7: The resulting balance of Jeremy is valid (150), but withdrawal was done via an *mistaken* increase. ⇒ Violation of **REQ2**

Version 4: How Should We Improve it?

- *Postconditions* of a method specify the precise conditions which it will satisfy upon its completion.

  This relies on the assumption that right before the method starts, its preconditions are satisfied (i.e., inputs valid) and invariants are satisfied (i.e,. object state valid).

  - Postcondition of `divide(int x, int y)`?

    [**Result** $\times$ y == x]
  - Postcondition of `binarySearch(int x, int[] xs)`?

    [$x \in xs \Rightarrow$ **Result** == x]

- The best we can do in Java is, similar to the case of invariants, encode postconditions as *assertions*.

  But again, unlike exceptions, these assertions will not be part of the class/method API.

- Create Version 5 by adding *assertions* to the end of textttwithdraw method of the `Account` class.

Version 5: Added Assertions to Approximate Method Postconditions

```
1  public class AccountV5 {
2    public void withdraw(int amount) throws
3       WithdrawAmountNegativeException, WithdrawAmountTooLargeException {
4      int oldBalance = this.balance;
5      if(amount < 0) { /* negated precondition */
6        throw new WithdrawAmountNegativeException(); }
7      else if (balance < amount) { /* negated precondition */
8        throw new WithdrawAmountTooLargeException(); }
9      else { this.balance = this.balance - amount; }
10     assert this.getBalance() > 0 :"Invariant: positive balance";
11     assert this.getBalance() == oldBalance - amount :
12       "Postcondition:  balance deducted"; }
```

A postcondition typically relates the pre-execution value and the post-execution value of each relevant attribute (e.g.,`balance` in the case of `withdraw`).

⇒ Extra code (**L4**) to capture the pre-execution value of `balance` for the comparison at **L11**.

Version 5: Why Better than Version 4?

```
1  public class BankAppV5 {
2    public static void main(String[] args) {
3      System.out.println("Create an account for Jeremy with balance 100:");
4      try { AccountV5 jeremy = new AccountV5("Jeremy", 100);
5            System.out.println(jeremy);
6            System.out.println("Withdraw 50 from Jeremy's account:");
7            jeremy.withdraw(50);
8            System.out.println(jeremy); }
9            /* catch statements same as this previous slide:
10            * Version 2: Why Still Not a Good Design? (2.1) */
```

```
Create an account for Jeremy with balance 100:
Jeremy's current balance is: 100
Withdraw 50 from Jeremy's account:
Exception in thread "main"
  java.lang.AssertionError: Postcondition: balance deducted
```

L8: Upon completion of `jeremy.withdraw(50)`, Jeremy has a wrong balance 150, an assertion failure (i.e., *postcondition* violation) occurs, *preventing further operations on this invalid account object*.

Evolving from Version 1 to Version 5

	Improvements Made	Design *Flaws*
V1	–	Complete lack of Contract
V2	Added exceptions as *method preconditions*	Preconditions not strong enough (i.e., with missing cases) may result in an invalid account state.
V3	Added assertions as *class invariants*	Incorrect implementations do not necessarily result in a state that violates the class invariants.
V4	Deliberately changed `withdraw`'s implementation to be **incorrect**.	The incorrect implementation does not result in a state that violates the class invariants.
V5	Added assertions as *method postconditions*	–

- In Versions 2, 3, 4, 5, **preconditions** approximated as *exceptions*.
  ☹ These are ***not preconditions***, but their *logical negation*.
  ☹ Client `BankApp`'s code ***complicated*** by repeating the list of `try-catch` statements.
- In Versions 3, 4, 5, **class invariants** and **postconditions** approximated as *assertions*.
  ☹ Unlike exceptions, these assertions will ***not appear in the API*** of `withdraw`.
  Potential clients of this method ***cannot know***: **1)** what their benefits are; and **2)** what their suppliers' obligations are.
  ☹ For postconditions, ***extra code*** needed to capture pre-execution values of attributes.

Version 5: Contract between Client and Supplier

	benefits	*obligations*
`BankAppV5.main` (CLIENT)	balance deduction positive balance	amount non-negative amount not too large
`BankV5.withdraw` (SUPPLIER)	amount non-negative amount not too large	balance deduction positive balance

	benefits	*obligations*
CLIENT	postcondition & invariant	precondition
SUPPLIER	precondition	postcondition & invariant

DbC in Java

DbC is possible in Java, but not appropriate for your learning:

- *Preconditions* of a method:
  Supplier
  - Encode their logical negations as exceptions.
  - In the **beginning** of that method, a list of `if`-statements for throwing the appropriate exceptions.

  Client
  - A list of `try-catch`-statements for handling exceptions.
- *Postconditions* of a method:
  Supplier
  - Encoded as a list of assertions, placed at the **end** of that method.

  Client
  - All such assertions do not appear in the API of that method.
- *Invariants* of a class:
  Supplier
  - Encoded as a list of assertions, placed at the **end** of **every** method.

  Client
  - All such assertions do not appear in the API of that class.

DbC in Eiffel: Supplier

DbC is supported natively in Eiffel for **supplier**:

```
class ACCOUNT
create
      make
feature -- Attributes
      owner : STRING
      balance : INTEGER
feature -- Constructors
      make(nn: STRING; nb: INTEGER)
            require -- precondition
                  positive_balance: nb >= 0
            do
                  owner := nn
                  balance := nb
            end
feature -- Commands
      withdraw(amount: INTEGER)
            require -- precondition
                  non_negative_amount: amount >= 0
                  affordable_amount: amount <= balance
            do
                  balance := balance - amount
            ensure -- postcondition
                  balance_deducted: balance = old balance - amount
            end
invariant -- class invariant
      positive_balance: balance > 0
end
```

DbC in Eiffel: Contract View of Supplier

Any potential **client** who is interested in learning about the kind of services provided by a **supplier** can look through the *contract view* (without showing any implementation details):

```
class ACCOUNT
create
      make
feature -- Attributes
      owner : STRING
      balance : INTEGER
feature -- Constructors
      make(nn: STRING; nb: INTEGER)
            require -- precondition
                  positive_balance: nb >= 0
            end
feature -- Commands
      withdraw(amount: INTEGER)
            require -- precondition
                  non_negative_amount: amount >= 0
                  affordable_amount: amount <= balance
            ensure -- postcondition
                  balance_deducted: balance = old balance - amount
            end
invariant -- class invariant
      positive_balance: balance > 0
end
```

DbC in Eiffel: Anatomy of a Class

```
class SOME_CLASS
create
  -- Explicitly list here commands used as constructors
feature -- Attributes
  -- Declare attribute here
feature -- Commands
  -- Declare commands (mutators) here
feature -- Queries
  -- Declare queries (accessors) here
invariant
  -- List of tagged boolean expressions for class invariants
end
```

- Use `feature` clauses to group attributes, commands, queries.
- Explicitly declare list of commands under `create` clause, so that they can be used as class constructors.

  [See the `groups` panel in Eiffel Studio.]
- The *class invariant* `invariant` clause may be omitted:
  - There's no class invariant: any resulting object state is acceptable.
  - The class invariant is equivalent to writing **`invariant`** ***true***

DbC in Eiffel: Anatomy of a Feature

```
some_command
    -- Description of the command.
  require
    -- List of tagged boolean expressions for preconditions
  local
    -- List of local variable declarations
  do
    -- List of instructions as implementation
  ensure
    -- List of tagged boolean expressions for postconditions
  end
```

- The *precondition* `require` clause may be omitted:
  - There's no precondition: any starting state is acceptable.
  - The precondition is equivalent to writing **`require`** ***true***
- The *postcondition* `ensure` clause may be omitted:
  - There's no postcondition: any resulting state is acceptable.
  - The postcondition is equivalent to writing **`ensure`** ***true***

Runtime Monitoring of Contracts

- All *contracts* are specified as ***Boolean expressions***.
- Right before a feature call (e.g., *acc.withdraw(10)*):
  - The current state of *acc* is called the *pre-state*.
  - Evaluate feature `withdraw`'s *pre-condition* using current values of attributes and queries.
  - ***Cache*** values (**implicitly**) of all expressions involving the ***old*** keyword in the *post-condition*.

    e.g., cache the value of *old balance* via *old_balance := balance*
- Right after the feature call:
  - The current state of *acc* is called the *post-state*.
  - Evaluate class `ACCOUNT`'s *invariant* using current values of attributes and queries.
  - Evaluate feature `withdraw`'s *post-condition* using both current and ***"cached"*** values of attributes and queries.

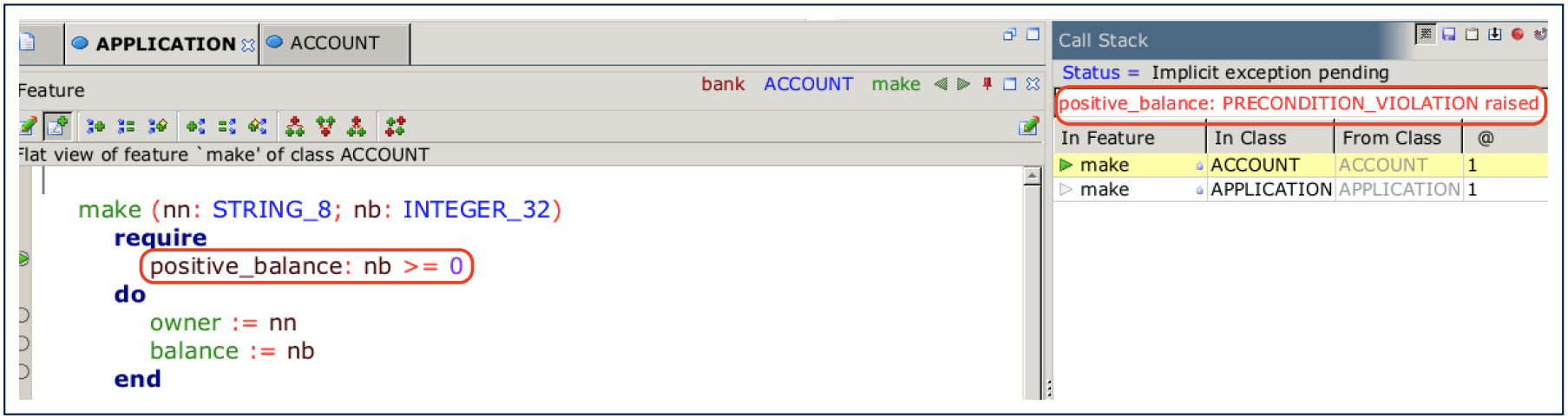
DbC in Eiffel: Precondition Violation (1.1)

The **client** need not handle all possible contract violations:

```
class BANK_APP
inherit
  ARGUMENTS
create
  make
feature -- Initialization
  make
    -- Run application.
  local
    alan: ACCOUNT
  do
    -- A precondition violation with tag "positive_balance"
    create {ACCOUNT} alan.make ("Alan", -10)
  end
end
```

By executing the above code, the runtime monitor of Eiffel Studio will report a *contract violation* (precondition violation with tag `"positive_balance"`).

DbC in Eiffel: Precondition Violation (1.2)



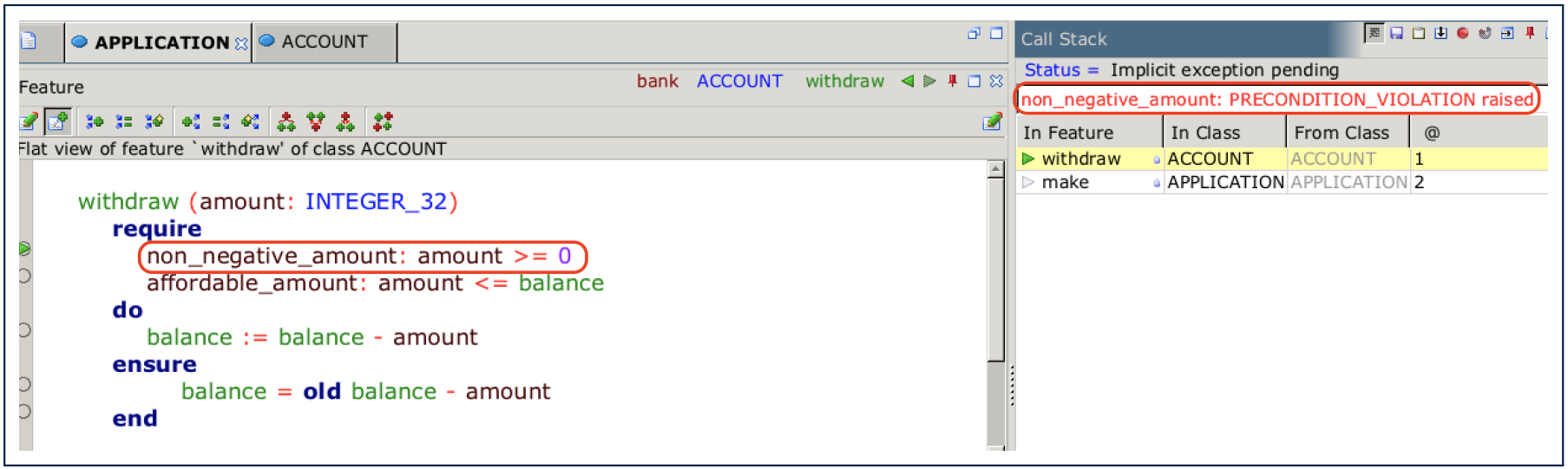
DbC in Eiffel: Precondition Violation (2.1)

The **client** need not handle all possible contract violations:

```
class BANK_APP
inherit
  ARGUMENTS
create
  make
feature -- Initialization
  make
    -- Run application.
  local
    mark: ACCOUNT
  do
    -- A precondition violation with tag "non_negative_amount"
    create {ACCOUNT} mark.make ("Mark", 100)
    mark.withdraw(-1000000)
  end
end
```

By executing the above code, the runtime monitor of Eiffel Studio will report a *contract violation* (precondition violation with tag "non_negative_amount").

DbC in Eiffel: Precondition Violation (2.2)



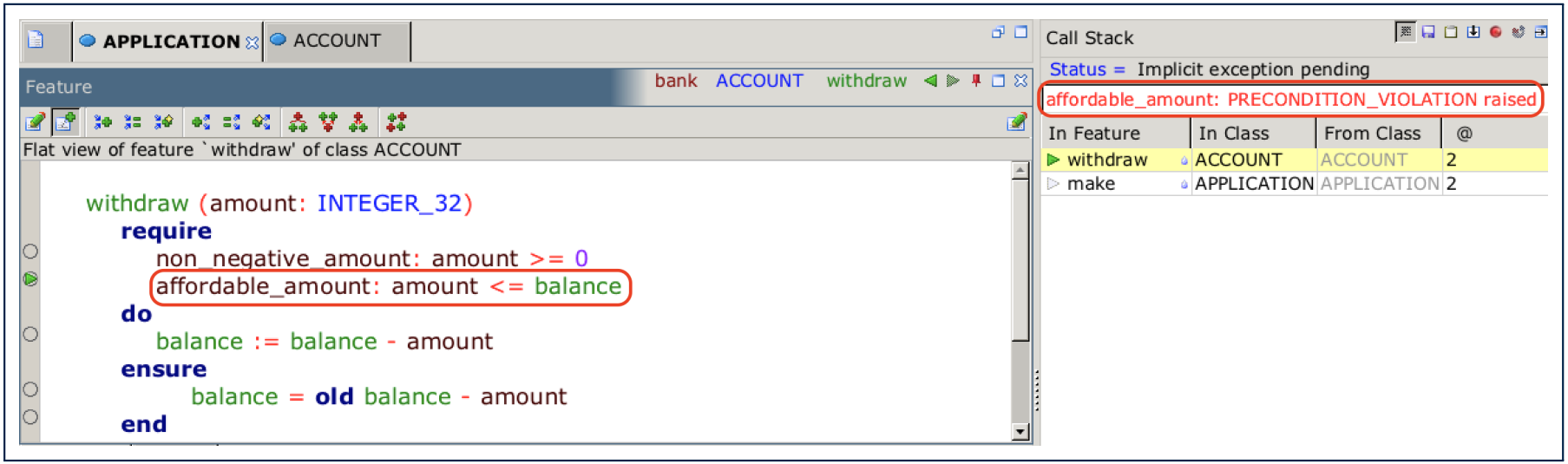
DbC in Eiffel: Precondition Violation (3.1)

The **client** need not handle all possible contract violations:

```
class BANK_APP
inherit
  ARGUMENTS
create
  make
feature -- Initialization
  make
    -- Run application.
  local
    tom: ACCOUNT
  do
    -- A precondition violation with tag "affordable_amount"
    create {ACCOUNT} tom.make ("Tom", 100)
    tom.withdraw(150)
  end
end
```

By executing the above code, the runtime monitor of Eiffel Studio will report a *contract violation* (precondition violation with tag `"affordable_amount"`).

DbC in Eiffel: Precondition Violation (3.2)



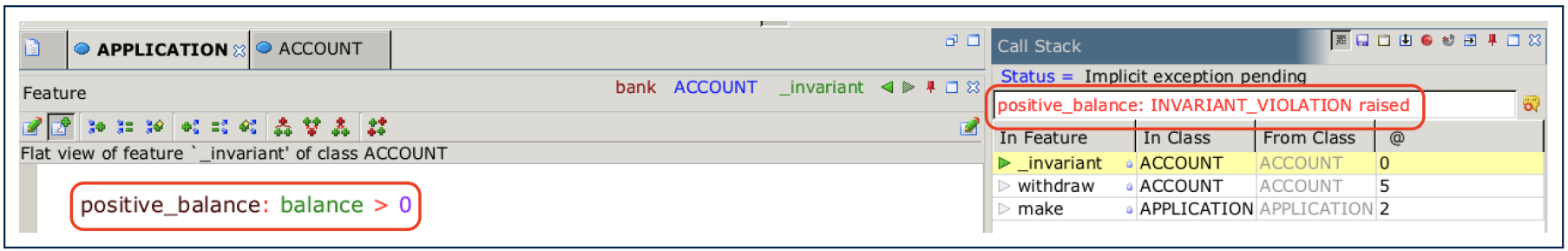
DbC in Eiffel: Class Invariant Violation (4.1)

The **client** need not handle all possible contract violations:

```
class BANK_APP
inherit
  ARGUMENTS
create
  make
feature -- Initialization
  make
    -- Run application.
  local
    jim: ACCOUNT
  do
    -- A class invariant violation with tag "positive_balance"
    create {ACCOUNT} tom.make ("Jim", 100)
    jim.withdraw(100)
  end
end
```

By executing the above code, the runtime monitor of Eiffel Studio will report a *contract violation* (class invariant violation with tag "positive_balance").

DbC in Eiffel: Class Invariant Violation (4.2)



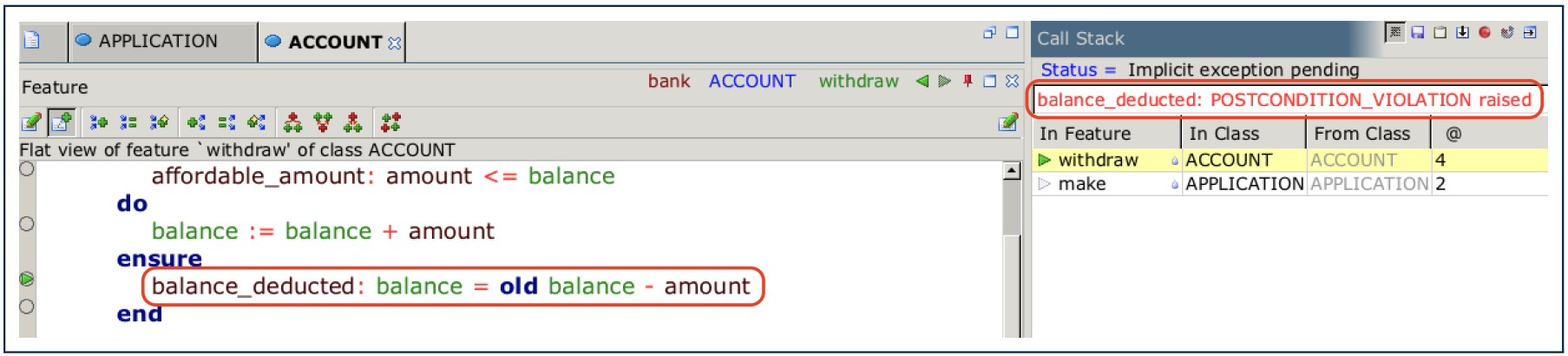
DbC in Eiffel: Class Invariant Violation (5.1)

The **client** need not handle all possible contract violations:

```
class BANK_APP
inherit ARGUMENTS
create make
feature -- Initialization
  make
    -- Run application.
  local
    jeremy: ACCOUNT
  do
    -- Change withdraw in ACCOUNT to: balance := balance + amount
    -- A postcondition violation with tag "balance_deducted"
    create {ACCOUNT} jeremy.make ("Jeremy", 100)
    jeremy.withdraw(150)
    -- Change withdraw in ACCOUNT back to: balance := balance - amount
  end
end
```

By executing the above code, the runtime monitor of Eiffel Studio will report a *contract violation* (postcondition violation with tag `"balance_deducted"`).

DbC in Eiffel: Class Invariant Violation (5.2)



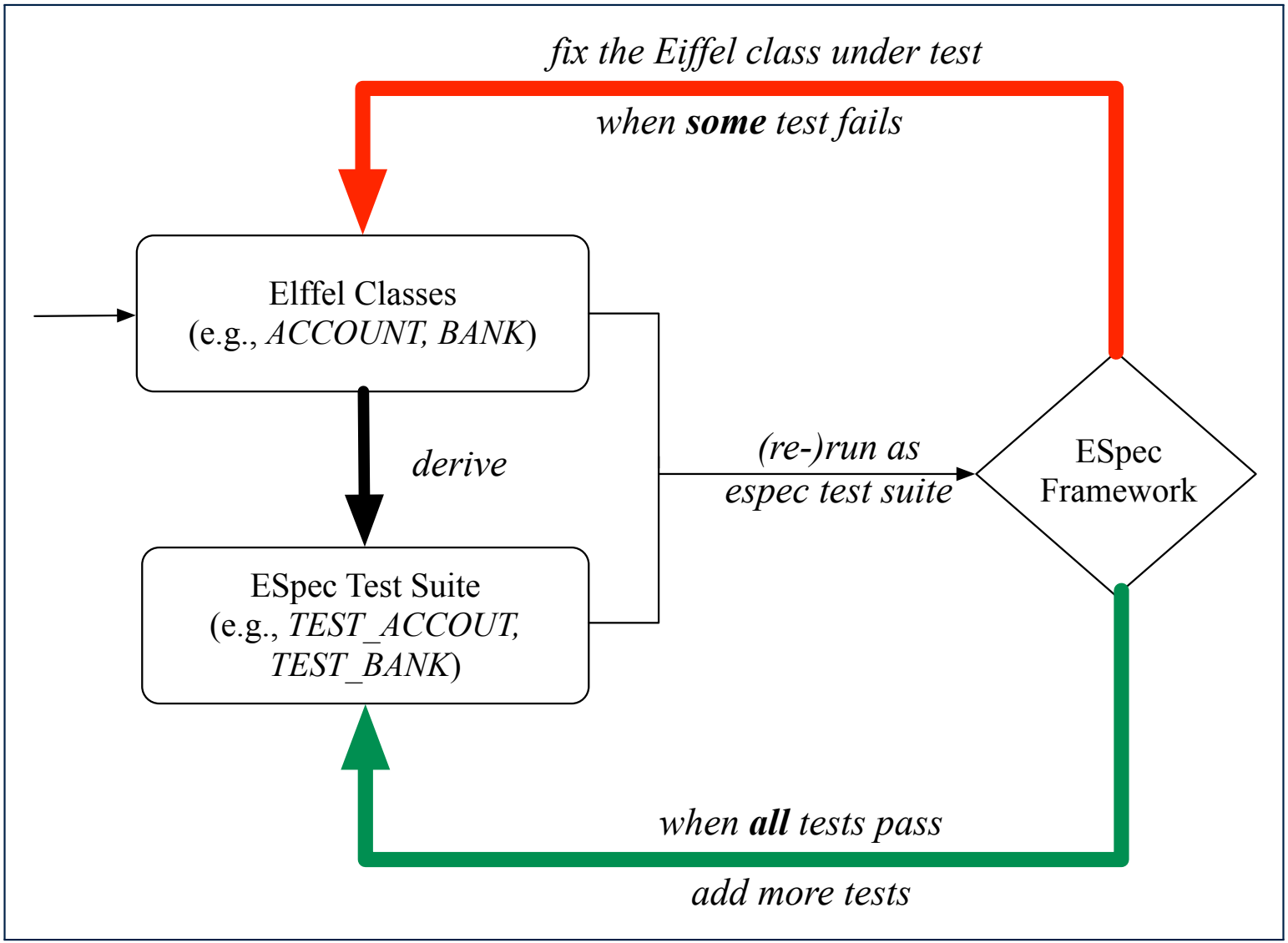
TDD: Test-Driven Development (1)

- How we have tested the software so far:
  - Executed each test case **manually** (by clicking `Run` in EStudio).
  - Compared **with our eyes** if ***actual results*** (produced by program) match ***expected results*** (according to requirements).
- Software is subject to numerous revisions before delivery.
  ⇒ Testing manually, repetitively, is tedious and error-prone.
  ⇒ We need *automation* in order to be cost-effective.
- *Test-Driven Development*
  - **Test Case**: Expected ***working*** scenario (**expected** outcome) or ***problematic*** scenario (**expected** contract violation).
  - As soon as your code becomes *executable* (with **a unit of functionality** completed), start translating relevant test cases into an executable form and execute them.
  - **Test Suite**: Collection of test cases.
    ⇒ A test suite is supposed to measure "correctness" of software.
    ⇒ The larger the suite, the more confident you are.

TDD: Test-Driven Development (2)

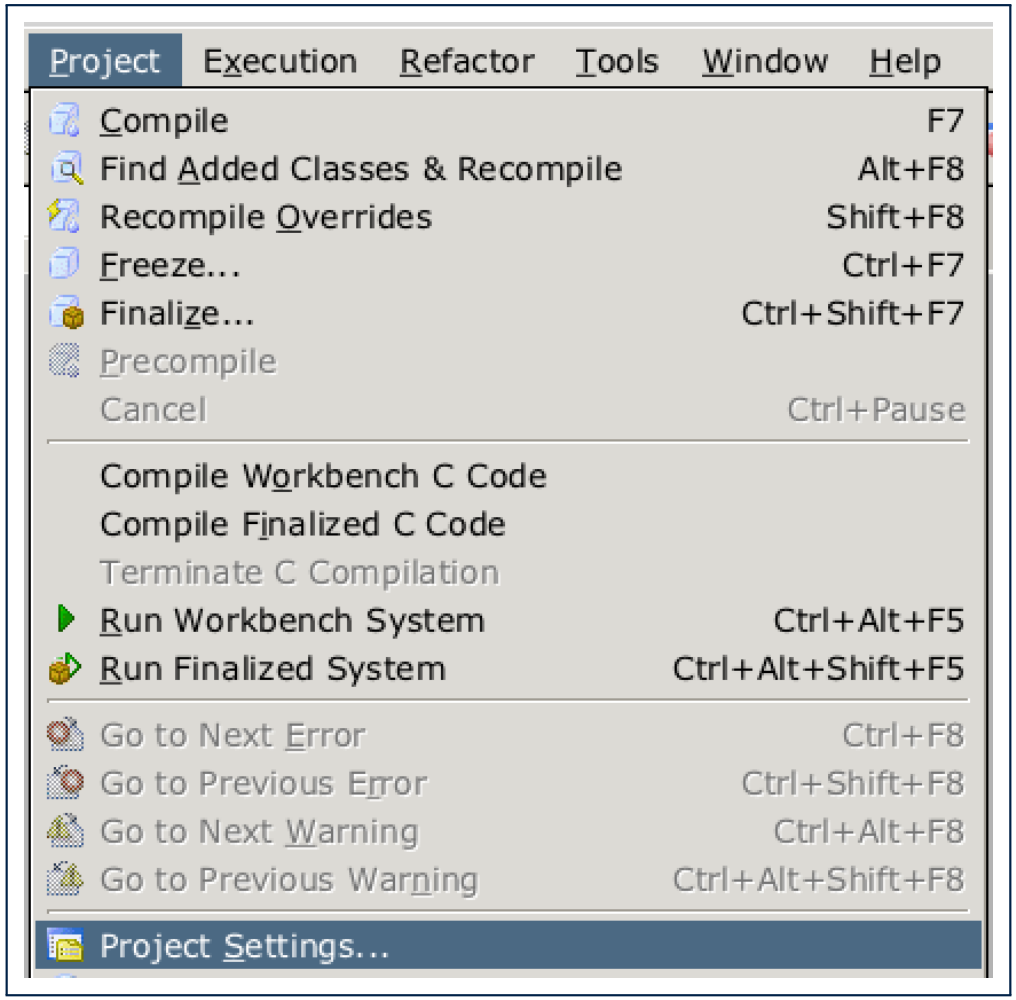
- The *ESpec* (Eiffel Specification) library is a framework for:
  - Writing and accumulating ***test cases***
    Each list of *relevant test cases* is grouped into an `ES_TEST` class, which is just an Eiffel class that you can execute upon.
  - Executing the ***test suite*** whenever software undergoes a change
    e.g., a bug fix
    e.g., extension of a new functionality
- ESpec tests are *helpful client* of your classes, which may:
  - Either attempt to use a feature in a ***legal*** way (i.e., ***satisfying*** its precondition), and report:
    - ***Success*** if the result is as expected
    - ***Failure*** if the result is ***not*** as expected:
      e.g., state of object has not been updated properly
      e.g., a ***postcondition violation*** or ***class invariant violation*** occurs
  - Or attempt to use a feature in an ***illegal*** way (e.g., ***not satisfying*** its precondition), and report:
    - ***Success*** if precondition violation occurs.
    - ***Failure*** if precondition violation does ***not*** occur.

TDD: Test-Driven Development (3)



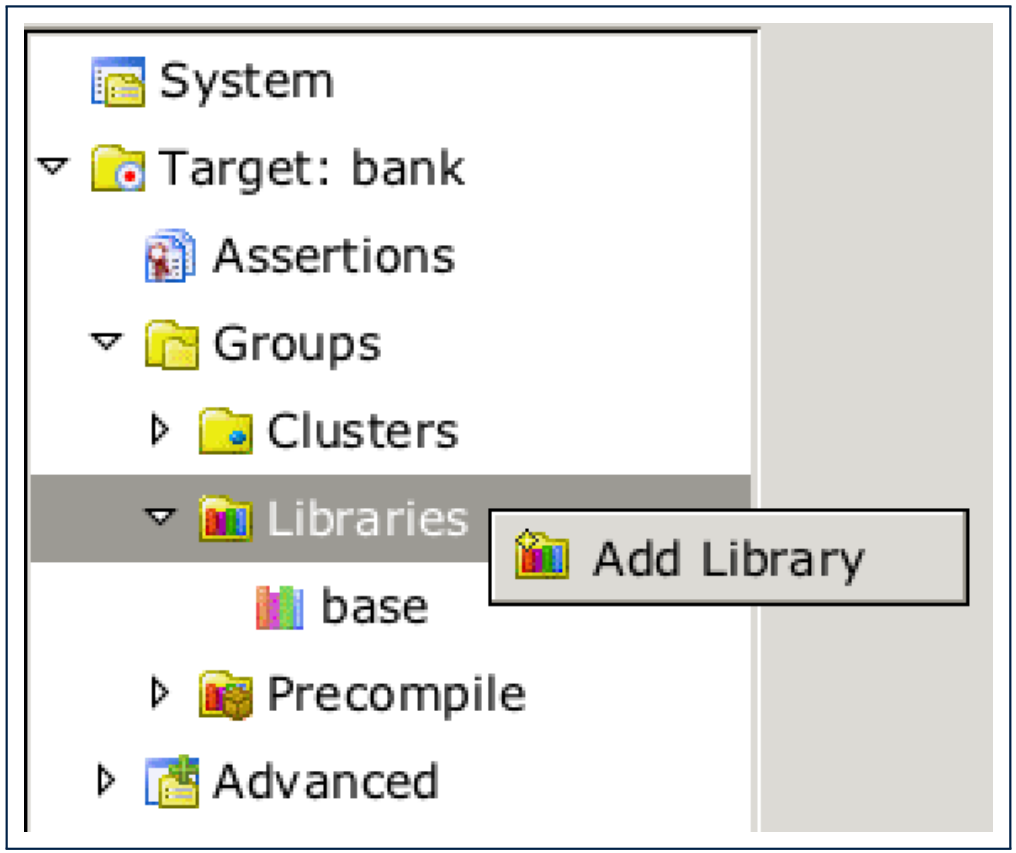
Adding the ESpec Library (1)

Step 1: Go to Project Settings.



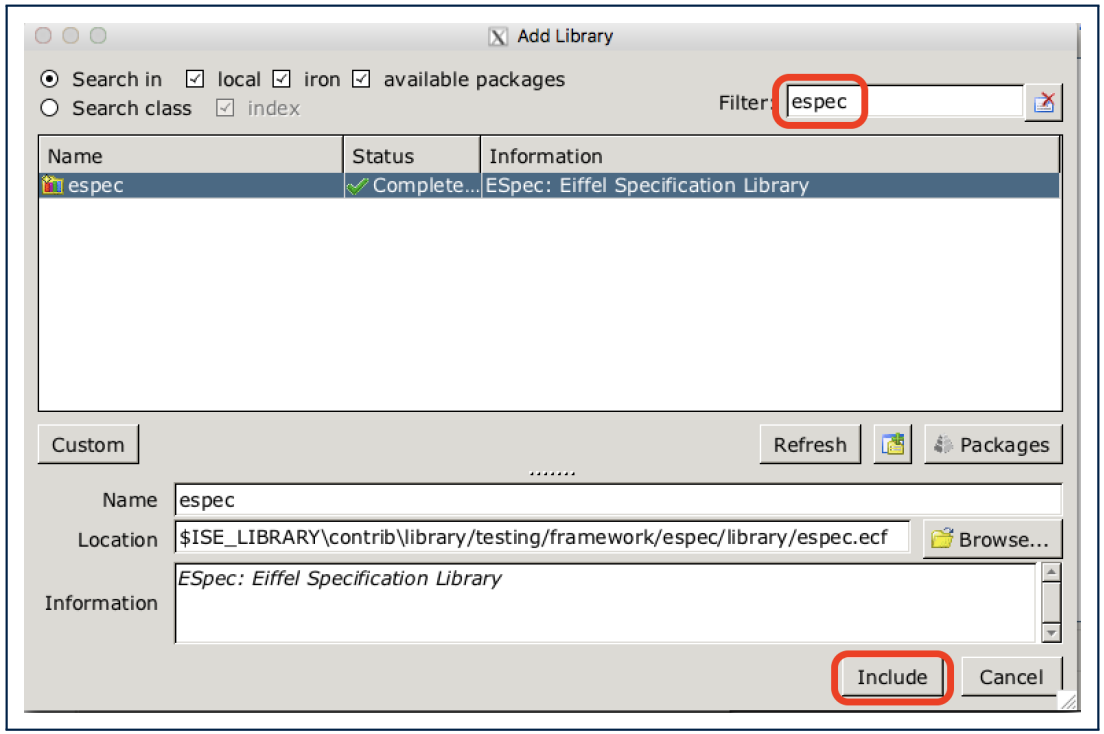
Adding the ESpec Library (2)

Step 2: Right click on `Libraries` to add a library.



Adding the ESpec Library (3)

Step 3: Search for `espec` and then include it.



This will make two classes available to you:

- `ES_TEST` for adding test cases
- `ES_SUITE` for adding instances of `ES_TEST`.
  - To run, an instance of this class must be set as the `root`.

ES_TEST: Expecting to Succeed (1)

```
1  class TEST_ACCOUNT
2  inherit ES_TEST
3  create make
4  feature -- Add tests in constructor
5    make
6      do
7        add_boolean_case (agent test_valid_withdraw)
8      end
9  feature -- Tests
10   test_valid_withdraw: BOOLEAN
11     local
12       acc: ACCOUNT
13     do
14       comment("Test a valid withdrawal.")
15       create {ACCOUNT} acc.make ("Alan", 100)
16       Result := acc.balance = 100
17       check Result end
18       acc.withdraw (20)
19       Result := acc.balance = 80
20     end
21 end
```

`ES_TEST`: Expecting to Succeed (2)

- **L2**: A test class is a subclass of `ES_TEST`.
- **L10 – 20** define a `BOOLEAN` test *query*. At runtime:
  - ***Success***: Return value of `test_valid_withdraw` (final value of variable **`Result`**) evaluates to *true* upon its termination.
  - ***Failure***:
    - The return value evaluates to *false* upon termination; or
    - Some contract violation (which is *unexpected*) occurs.
- **L7** calls feature `add_boolean_case` from `ES_TEST`, which expects to take as input a *query* that returns a Boolean value.
  - We pass *query* `test_valid_withdraw` as an input.
  - Think of the keyword `agent` acts like a function pointer.
    - `test_invalid_withdraw` alone denotes its return value
    - **`agent`** `test_invalid_withdraw` denotes address of *query*
- **L14**: Each test feature ***must*** call **`comment(...)`** (inherited from `ES_TEST`) to include the description in test report.
- **L17**: Check that *each* intermediate value of `Result` is ***true***.

ES_TEST: Expecting to Succeed (3)

- Why is the `check Result end` statement at **L7** necessary?
  - When there are two or more *assertions* to make, some of which (except the last one) may ***temporarily falsify*** return value **Result**.
  - As long as the last *assertion* assigns ***true*** to **Result**, then the entire *test query* is considered as a ***success***.
    ⇒ A ***false positive*** is possible!
- For the sake of demonstrating a false positive, imagine:
  - Constructor `make` ***mistakenly*** deducts 20 from input amount.
  - Command `withdraw` ***mistakenly*** deducts nothing.

```
1  test_query_giving_false_positive: BOOLEAN
2   local acc: ACCOUNT
3   do comment("Result temporarily false, but finally true.")
4      create {ACCOUNT} acc.make ("Jim", 100) -- balance set as 80
5      Result := acc.balance = 100 -- Result assigned to false
6      acc.withdraw (20) -- balance not deducted
7      Result := acc.balance = 80 -- Result re-assigned to true
8      -- Upon termination, Result being true makes the test query
9      -- considered as a success ==> false positive!
10  end
```

Fix? [insert ***check Result end***] between **L6** and **L7**.

ES_TEST: Expecting to Fail (1)

```
1  class TEST_ACCOUNT
2  inherit ES_TEST
3  create make
4  feature -- Add tests in constructor
5    make
6      do
7        add_violation_case_with_tag (
8          "non_negative_amount", agent test_invalid_withdraw)
9      end
10 feature -- Tests
11   test_invalid_withdraw
12     local
13       acc: ACCOUNT
14     do
15       comment("Test an invalid withdrawal.")
16       create {ACCOUNT} acc.make ("Mark", 100)
17       -- Precondition Violation
18       -- with tag "non_negative_amount" is expected.
19       Result := acc.withdraw (-1000000)
20     end
21 end
```

ES_TEST: Expecting to Fail (2)

- **L2**: A test class is a subclass of `ES_TEST`.
- **L11 – 20** define a test *command*. At runtime:
  - ***Success***: A precondition violation (with tag `"non_negative_amount"`) occurs at **L19** before its termination.
  - ***Failure***:
    - No contract violation with the expected tag occurs before its termination; or
    - Some other contract violation (with a different tag) occurs.
- **L7** calls feature `add_violation_case_with_tag` from `ES_TEST`, which expects to take as input a *command*.
  - We pass *command* `test_invalid_withdraw` as an input.
  - Think of the keyword `agent` acts like a function pointer.
    - `test_invalid_withdraw` alone denotes a call to it
    - **`agent`** `test_invalid_withdraw` denotes address of *command*
- **L15**: Each test feature ***must*** call **`comment(...)`** (inherited from `ES_TEST`) to include the description in test report.

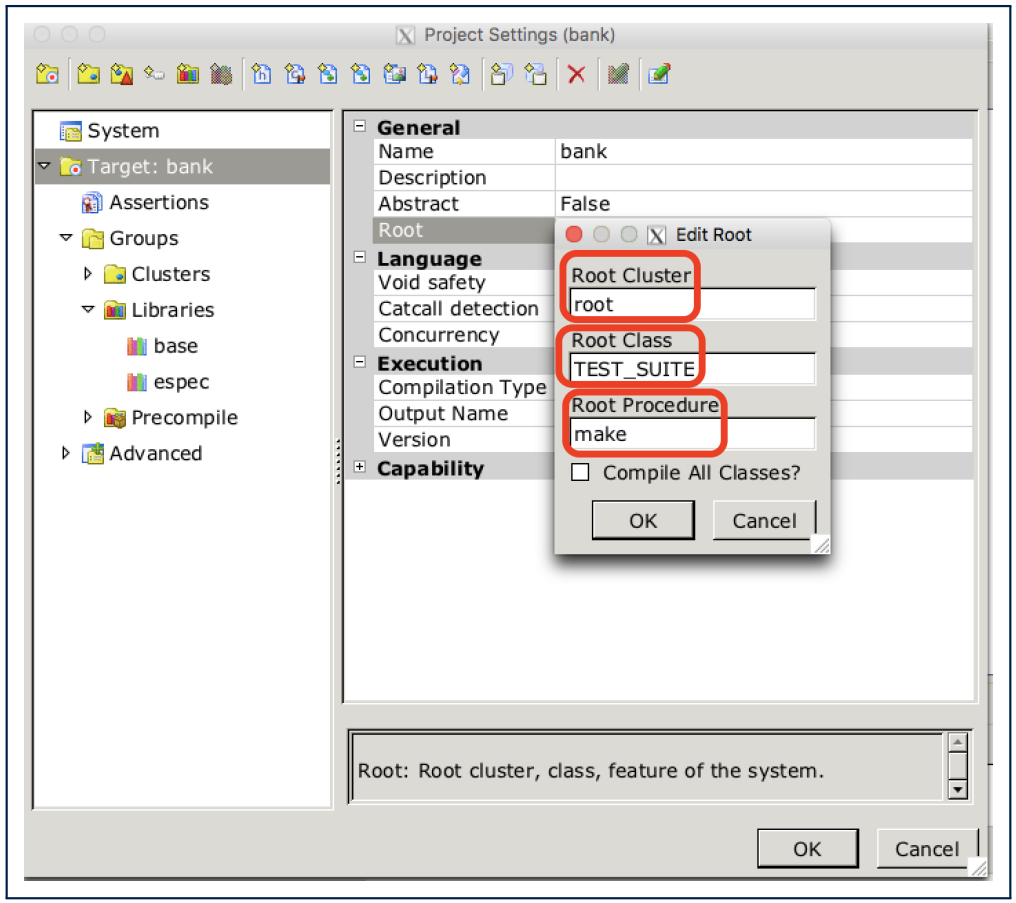
ES_SUITE: Collecting Test Classes

```
1  class TEST_SUITE
2  inherit ES_SUITE
3  create make
4  feature -- Constructor for adding test classes
5    make
6      do
7        add_test (create {TEST_ACCOUNT}.make)
8        show_browser
9        run_espec
10     end
11 end
```

- **L2**: A test suite is a subclass of `ES_SUITE`.
- **L7** passes an **anonymous** object of type `TEST_ACCOUNT` to `add_test` inherited from `ES_SUITE`).
- **L8 & L9** have to be entered in this order!

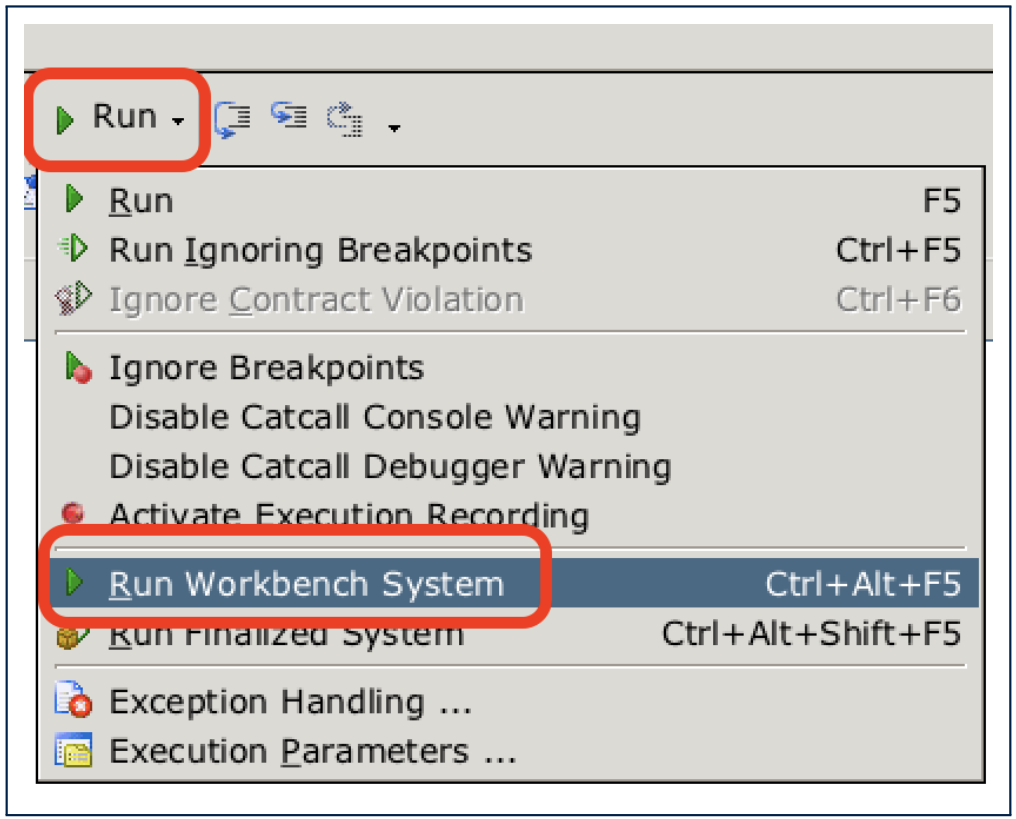
Running ES_SUITE (1)

Step 1: Change the *root class* (i.e., entry point of execution) to be `TEST_SUITE`.



Running `ES_SUITE` (2)

Step 2: Run the **Workbench System**.



Running ES_SUITE (3)

Step 3: See the generated test report.

TEST_SUITE

Note: * indicates a violation test case

PASSED (2 out of 2)

Case Type	Passed	Total
Violation	1	1
Boolean	1	1
All Cases	2	2

State	Contract Violation	Test Name
Test1	TEST_ACCOUNT	
PASSED	NONE	Test an ivalid withdrawl.
PASSED	NONE	*Test a valid withdrawl.

Beyond this lecture...

- Study this tutorial series on DbC and TDD:

  `https://www.youtube.com/playlist?list=PL5dxAmCmjv_6r5VfzCQ5bTznoDDgh__KS`

Index (1)

Index (2)

Index (3)

Index (4)

Index (5)

Syntax of Eiffel: a Brief Overview

EECS3311: Software Design
Fall 2017

CHEN-WEI WANG

Escape Sequences

Escape sequences are special characters to be placed in your program text.

- In Java, an escape sequence starts with a backward slash `\` e.g., `\n` for a new line character.
- In Eiffel, an escape sequence starts with a percentage sign `%` e.g., `%N` for a new line characgter.

See here for more escape sequences in Eiffel: `https://www.eiffel.org/doc/eiffel/Eiffel%20programming%20language%20syntax#Special_characters`

Commands, and Queries, and Features

- In a Java class:
  - **Attributes**: Data
  - **Mutators**: Methods that change attributes without returning
  - **Accessors**: Methods that access attribute values and returning
- In an Eiffel class:
  - Everything can be called a *feature*.
  - But if you want to be specific:
    - Use *attributes* for data
    - Use *commands* for mutators
    - Use *queries* for accessors

Naming Conventions

- Cluster names: all lower-cases separated by underscores

  e.g., `root`, `model`, `tests`, `cluster_number_one`
- Classes/Type names: all upper-cases separated by underscores

  e.g., `ACCOUNT`, `BANK_ACCOUNT_APPLICATION`
- Feature names (attributes, commands, and queries): all lower-cases separated by underscores

  e.g., `account_balance`, `deposit_into`, `withdraw_from`

Operators: Assignment vs. Equality

- In Java:
  - Equal sign `=` is for assigning a value expression to some variable.
    e.g., `x = 5 * y` changes `x`'s value to `5 * y`
    This is actually controversial, since when we first learned about `=`, it means the mathematical equality between numbers.
  - Equal-equal `==` and bang-equal `!=` are used to denote the equality and inequality.
    e.g., `x == 5 * y` evaluates to *true* if `x`'s value is equal to the value of `5 * y`, or otherwise it evaluates to *false*.
- In Eiffel:
  - Equal `=` and slash equal `/=` denote equality and inequality.
    e.g., `x = 5 * y` evaluates to *true* if `x`'s value is equal to the value of `5 * y`, or otherwise it evaluates to *false*.
  - We use `:=` to denote variable assignment.
    e.g., `x := 5 * y` changes `x`'s value to `5 * y`
  - Also, you are not allowed to write shorthands like `x++`, just write `x := x + 1`.

Attribute Declarations

- In Java, you write: `int i, Account acc`
- In Eiffel, you write: `i: INTEGER, acc: ACCOUNT`

  Think of `:` as the set membership operator $\in$:

  e.g., The declaration `acc: ACCOUNT` means object `acc` is a member of all possible instances of `ACCOUNT`.

Method Declaration

- **Command**

```
deposit (amount: INTEGER)
  do
    balance := balance + amount
  end
```

Notice that you don't use the return type `void`

- **Query**

```
sum_of (x: INTEGER; y: INTEGER): INTEGER
  do
    Result := x + y
  end
```

  - Input parameters are separated by semicolons `;`
  - Notice that you don't use `return`; instead assign the return value to the **pre-defined variable Result**

Operators: Logical Operators (1)

- Logical operators (what you learned from EECS1090) are for combining Boolean expressions.
- In Eiffel, we have operators that *EXACTLY* correspond to these logical operators:

	LOGIC	EIFFEL
Conjunction	$\wedge$	**and**
Disjunction	$\vee$	**or**
Implication	$\Rightarrow$	**implies**
Equivalence	$\equiv$	**=**

Review of Propositional Logic (1)

- A *proposition* is a statement of claim that must be of either *true* or *false*, but not both.
- Basic logical operands are of type Boolean: *true* and *false*.
- We use logical operators to construct compound statements.
  - Binary logical operators: conjunction ($\wedge$), disjunction ($\vee$), implication ($\Rightarrow$), and equivalence (a.k.a if-and-only-if $\Longleftrightarrow$)

p	q	$p \wedge q$	$p \vee q$	$p \Rightarrow q$	$p \Longleftrightarrow q$
true	*true*	*true*	*true*	*true*	*true*
true	*false*	*false*	*true*	*false*	*false*
false	*true*	*false*	*true*	*true*	*false*
false	*false*	*false*	*false*	*true*	*true*

  - Unary logical operator: negation ($\neg$)

p	$\neg p$
true	*false*
false	*true*

Review of Propositional Logic: Implication

- Written as $p \Rightarrow q$
- Pronounced as “p implies q”
- We call p the antecedent, assumption, or premise.
- We call q the consequence or conclusion.
- Compare the *truth* of $p \Rightarrow q$ to whether a contract is *honoured*: $p \approx$ promised terms; and $q \approx$ obligations.
- When the promised terms are met, then:
  - The contract is *honoured* if the obligations are fulfilled.
  - The contract is *breached* if the obligations are not fulfilled.
- When the promised terms are not met, then:
  - Fulfilling the obligation (q) or not ($\neg q$) does *not breach* the contract.

p	q	$p \Rightarrow q$
true	*true*	*true*
true	*false*	*false*
false	*true*	*true*
false	*false*	*true*

Review of Propositional Logic (2)

- **Axiom**: Definition of $\Rightarrow$

$$p \Rightarrow q \equiv \neg p \vee q$$

- **Theorem**: Identity of $\Rightarrow$

$$true \Rightarrow p \equiv p$$

- **Theorem**: Zero of $\Rightarrow$

$$false \Rightarrow p \equiv true$$

- **Axiom**: De Morgan

$$\begin{aligned} \neg(p \wedge q) &\equiv \neg p \vee \neg q \\ \neg(p \vee q) &\equiv \neg p \wedge \neg q \end{aligned}$$

- **Axiom**: Double Negation

$$p \equiv \neg\ (\neg\ p)$$

- **Theorem**: Contrapositive

$$p \Rightarrow q \equiv \neg q \Rightarrow \neg p$$

Review of Predicate Logic (1)

- A *predicate* is a *universal* or *existential* statement about objects in some universe of disclosure.
- Unlike propositions, predicates are typically specified using *variables*, each of which declared with some *range* of values.
- We use the following symbols for common numerical ranges:
  - $\mathbb{Z}$: the set of integers
  - $\mathbb{N}$: the set of natural numbers
- Variable(s) in a predicate may be *quantified*:
  - *Universal quantification*:
    All values that a variable may take satisfy certain property.
    e.g., Given that *i* is a natural number, *i* is *always* non-negative.
  - *Existential quantification*:
    Some value that a variable may take satisfies certain property.
    e.g., Given that *i* is an integer, *i* *can be* negative.

Review of Predicate Logic (2.1)

- A *universal quantification* has the form $(\forall X \mid R \bullet P)$
  - X is a list of variable *declarations*
  - R is a *constraint on ranges* of declared variables
  - P is a *property*
  - $(\forall X \mid R \bullet P) \equiv (\forall X \bullet R \Rightarrow P)$
    e.g., $(\forall X \mid True \bullet P) \equiv (\forall X \bullet True \Rightarrow P) \equiv (\forall X \bullet P)$
    e.g., $(\forall X \mid False \bullet P) \equiv (\forall X \bullet False \Rightarrow P) \equiv (\forall X \bullet True) \equiv True$
- *For all* (combinations of) values of variables declared in X that satisfies R, it is the case that P is satisfied.
  - $\forall i \mid i \in \mathbb{N} \bullet i \geq 0$ [*true*]
  - $\forall i \mid i \in \mathbb{Z} \bullet i \geq 0$ [*false*]
  - $\forall i, j \mid i \in \mathbb{Z} \wedge j \in \mathbb{Z} \bullet i < j \vee i > j$ [*false*]
- The range constraint of a variable may be moved to where the variable is declared.
  - $\forall i : \mathbb{N} \bullet i \geq 0$
  - $\forall i : \mathbb{Z} \bullet i \geq 0$
  - $\forall i, j : \mathbb{Z} \bullet i < j \vee i > j$

Review of Predicate Logic (2.2)

- An *existential quantification* has the form $(\exists X \mid R \bullet P)$
  - X is a list of variable *declarations*
  - R is a *constraint on ranges* of declared variables
  - P is a *property*
  - $(\exists X \mid R \bullet P) \equiv (\exists X \bullet R \wedge P)$
    e.g., $(\exists X \mid True \bullet P) \equiv (\exists X \bullet True \wedge P) \equiv (\forall X \bullet P)$
    e.g., $(\exists X \mid False \bullet P) \equiv (\exists X \bullet False \wedge P) \equiv (\exists X \bullet False) \equiv False$
- *There exists* a combination of values of variables declared in X that satisfies R and P.
  - $\exists i \mid i \in \mathbb{N} \bullet i \geq 0$ [*true*]
  - $\exists i \mid i \in \mathbb{Z} \bullet i \geq 0$ [*true*]
  - $\exists i, j \mid i \in \mathbb{Z} \wedge j \in \mathbb{Z} \bullet i < j \vee i > j$ [*true*]
- The range constraint of a variable may be moved to where the variable is declared.
  - $\exists i : \mathbb{N} \bullet i \geq 0$
  - $\exists i : \mathbb{Z} \bullet i \geq 0$
  - $\exists i, j : \mathbb{Z} \bullet i < j \vee i > j$

Predicate Logic (3)

- Conversion between $\forall$ and $\exists$

$$(\forall X \mid R \bullet P) \iff \neg(\exists X \bullet R \Rightarrow \neg P)$$
$$(\exists X \mid R \bullet P) \iff \neg(\forall X \bullet R \Rightarrow \neg P)$$

- Range Elimination

$$(\forall X \mid R \bullet P) \iff (\forall X \bullet R \Rightarrow P)$$
$$(\exists X \mid R \bullet P) \iff (\exists X \bullet R \wedge P)$$

Operators: Logical Operators (2)

- How about Java?
  - Java does not have an operator for logical implication.
  - The `==` operator can be used for logical equivalence.
  - The `&&` and `||` operators only **approximate** conjunction and disjunction, due to the ***short-circuit effect*** *(SCE)*:
    - When evaluating `e1 && e2`, if `e1` already evaluates to *false*, then `e1` will **not** be evaluated.
      e.g., In `(y != 0) && (x / y > 10)`, the SCE guards the division against division-by-zero error.
    - When evaluating `e1 || e2`, if `e1` already evaluates to *true*, then `e1` will **not** be evaluated.
      e.g., In `(y == 0) || (x / y > 10)`, the SCE guards the division against division-by-zero error.
  - However, in math, we always evaluate both sides.
- In Eiffel, we also have the version of operators with SCE:

	short-circuit conjunction	short-circuit disjunction
Java	`&&`	`||`
Eiffel	**and then**	**or else**

Operators: Division and Modulo

	Division	Modulo (Remainder)
Java	`20 / 3` is 6	`20 % 3` is 2
Eiffel	`20 // 3` is 6	`20 \\ 3` is 2

Class Declarations

- In Java:

```
class BankAccount {
  /* attributes and methods */
}
```

- In Eiffel:

```
class BANK_ACCOUNT
  /* attributes, commands, and queries */
end
```

Class Constructor Declarations (1)

- In Eiffel, constructors are just commands that have been *explicitly* declared as **creation features**:

```
class BANK_ACCOUNT
-- List names commands that can be used as constru
create
  make
feature -- Commands
  make (b: INTEGER)
    do balance := b end
  make2
    do balance := 10 end
end
```

- Only the command `make` can be used as a constructor.
- Command `make2` is not declared explicitly, so it cannot be used as a constructor.

Creations of Objects (1)

- In Java, we use a constructor `Accont(int b)` by:
  - Writing `Account acc = new Account(10)` to create a named object `acc`
  - Writing `new Account(10)` to create an anonymous object
- In Eiffel, we use a creation feature (i.e., a command explicitly declared under `create`) `make (int b)` in class `ACCOUNT` by:
  - Writing `create {ACCOUNT} acc.make (10)` to create a named object `acc`
  - Writing `create {ACCOUNT}.make (10)` to create an anonymous object
- Writing `create {ACCOUNT} acc.make (10)`

  is really equivalent to writing

```
acc := create {ACCOUNT}.make (10)
```

Selections

```
if B1 then
  -- B1
  -- do something
elseif B2 then
  -- B2 ∧ (¬B1)
  -- do something else
else
  -- (¬B1) ∧ (¬B2)
  -- default action
end
```

Loops (1)

- In Java, the Boolean conditions in `for` and `while` loops are **stay** conditions.

```
void printStuffs() {
  int i = 0;
  while( i < 10 ) {
    System.out.println(i);
    i = i + 1;
  }
}
```

- In the above Java loop, we *stay* in the loop as long as `i < 10` is true.
- In Eiffel, we think the opposite: we *exit* the loop as soon as `i >= 10` is true.

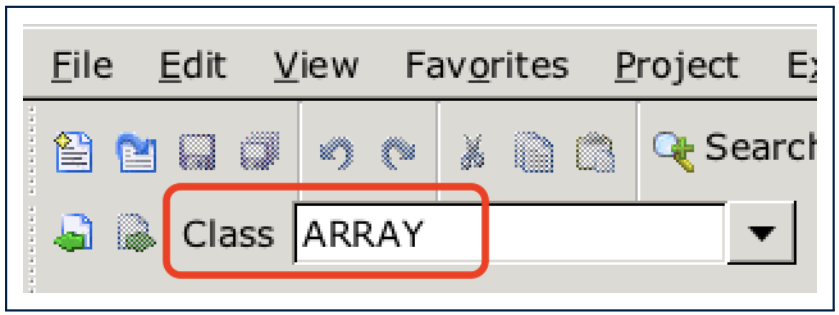
Loops (2)

In Eiffel, the Boolean conditions you need to specify for loops are **exit** conditions (logical negations of the stay conditions).

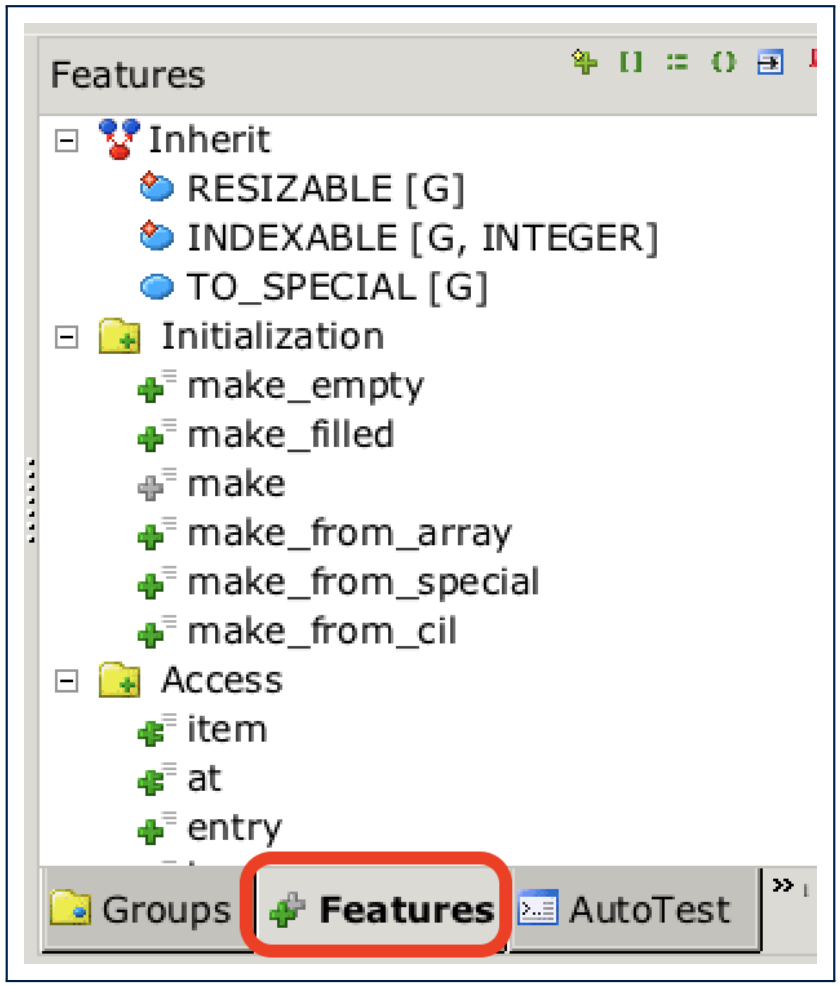
```
print_stuffs
  local
    i: INTEGER
  do
    from
      i := 0
    until
      i >= 10
    loop
      print (i)
      i := i + 1
    end -- end loop
  end -- end command
```

Library Data Structures

Enter a DS name.



Explore supported features.



Data Structures: Arrays

- Creating an empty array:

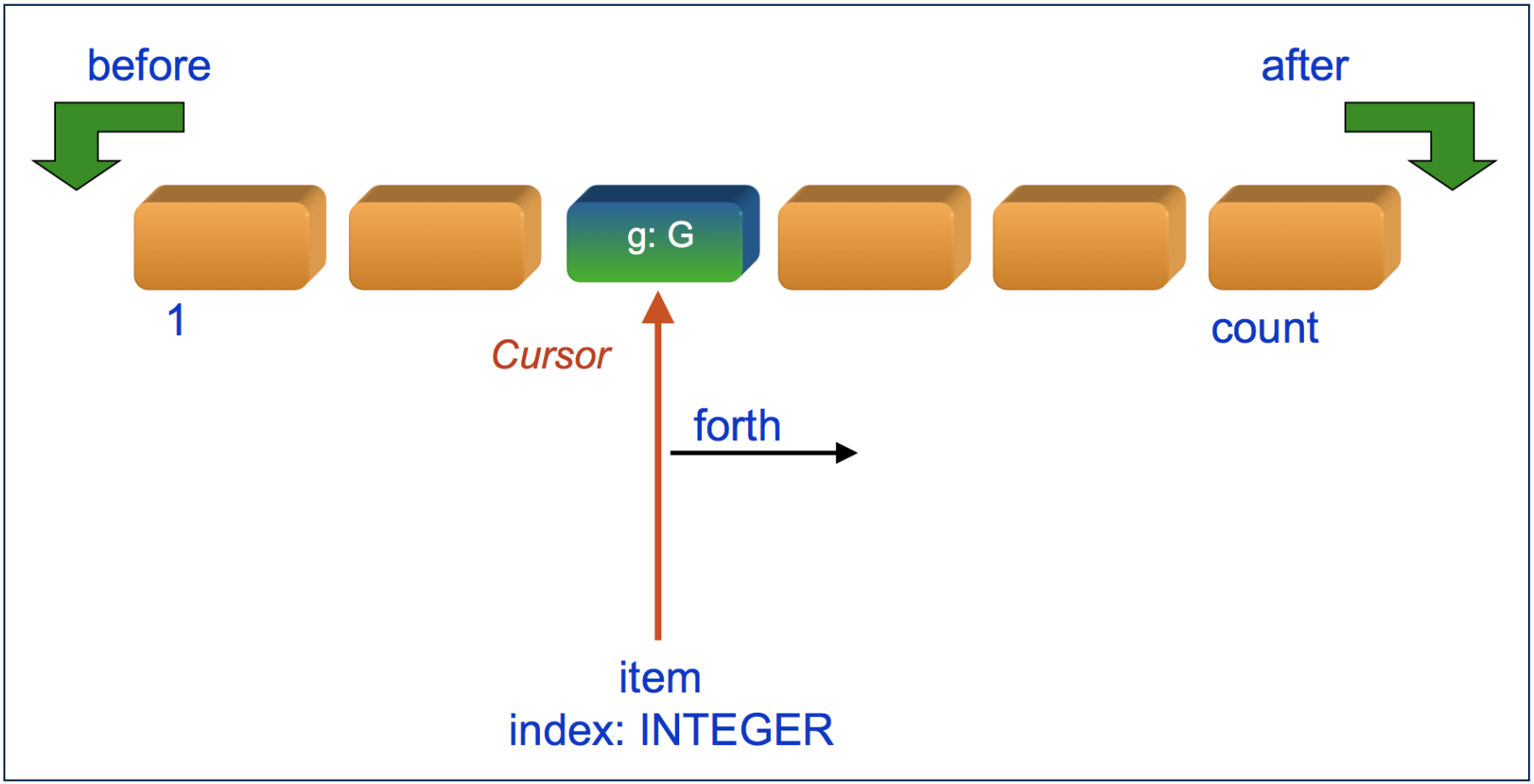
```
local a: ARRAY[INTEGER]
do create {ARRAY[INTEGER]} a.make_empty
```

  - This creates an array of `lower` and `upper` indices 1 and 0.
  - Size of array `a`: `a.upper - a.lower + 1`.

- Typical loop structure to iterate through an array:

```
local
  a: ARRAY[INTEGER]
  i, j: INTEGER
do
  ...
from
  j := a.lower
until
  j > a.upper
do
  i := a [j]
  j := j + 1
end
```

Data Structures: Linked Lists (1)



Data Structures: Linked Lists (2)

- Creating an empty linked list:

```
local
  list: LINKED_LIST[INTEGER]
do
  create {LINKED_LIST[INTEGER]} list.make
```

- Typical loop structure to iterate through a linked list:

```
local
  list: LINKED_LIST[INTEGER]
  i: INTEGER
do
  ...
from
  list.start
until
  list.after
do
  i := list.item
  list.forth
end
```

Using `across` for Quantifications

- ***across*** `...` ***as*** `...` ***all*** `...` ***end***
  A Boolean expression acting as a universal quantification (∀)

```
1  local
2    allPositive: BOOLEAN
3    a: ARRAY[INTEGER]
4  do
5    ...
6    Result :=
7      across
8        a.lower |..| a.upper as i
9      all
10       a [i.item] > 0
11     end
```

  - **L8**: `a.lower |..| a.upper` denotes a list of integers.
  - **L8**: `as i` declares a list cursor for this list.
  - **L10**: `i.item` denotes the value pointed to by cursor `i`.
- **L9**: Changing the keyword **all** to ***some*** makes it act like an existential quantification ∃.

Index (1)

Index (2)

Abstract Data Types (ADTs), Classes, and Objects

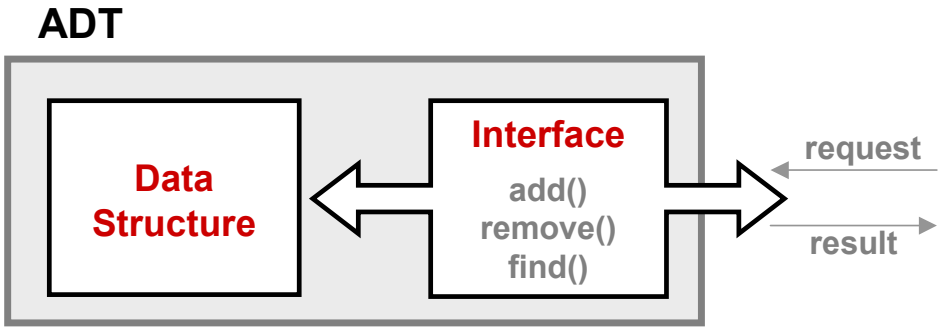
Readings: OOSC2 Chapters 6, 7, 8

EECS3311: Software Design
Fall 2017

CHEN-WEI WANG

Abstract Data Types (ADTs)

- Given a problem, you are required to filter out *irrelevant* details.
- The result is an *abstract data type (ADT)*, whose *interface* consists of a list of (unimplemented) operations.



- *Supplier*'s *Obligations*:
  - Implement all operations
  - Choose the "right" data structure (DS)
- *Client*'s *Benefits*:
  - *Correct* output
  - Efficient performance
- The internal details of an *implemented ADT* should be **hidden**.

Building ADTs for Reusability

- ADTs are *reusable software components*
  e.g., Stacks, Queues, Lists, Dictionaries, Trees, Graphs
- An ADT, once thoroughly tested, can be reused by:
  - Suppliers of other ADTs
  - Clients of Applications
- As a supplier, you are obliged to:
  - *Implement* given ADTs using other ADTs (e.g., arrays, linked lists, hash tables, etc.)
  - *Design* algorithms that make use of standard ADTs
- For each ADT that you build, you ought to be clear about:
  - The list of supported operations (i.e., *interface*)
    - The interface of an ADT should be *more than* method signatures and natural language descriptions:
      - How are clients supposed to use these methods? [*preconditions*]
      - What are the services provided by suppliers? [*postconditions*]
  - Time (and sometimes space) *complexity* of each operation

Why Java Interfaces Unacceptable ADTs (1)

Interface List<E>

Type Parameters:

`E - the type of elements in this list`

All Superinterfaces:

`Collection<E>, Iterable<E>`

All Known Implementing Classes:

`AbstractList, AbstractSequentialList, ArrayList, AttributeList, CopyOnWriteArrayList, LinkedList, RoleList, RoleUnresolvedList, Stack, Vector`

```
public interface List<E>
extends Collection<E>
```

An ordered collection (also known as a *sequence*). The user of this interface has precise control over where in the list each element is inserted. The user can access elements by their integer index (position in the list), and search for elements in the list.

It is useful to have:

- A *generic collection class* where the ***homogeneous type*** of elements are parameterized as `E`.
- A reasonably ***intuitive overview*** of the ADT.

Why Java Interfaces Unacceptable ADTs (2)

Methods described in a ***natural language*** can be ***ambiguous***:

`E`	**`set`**`(int index, E element)` Replaces the element at the specified position in this list with the specified element (optional operation).

set

```
E set(int index,
      E element)
```

Replaces the element at the specified position in this list with the specified element (optional operation).

Parameters:

`index` - index of the element to replace

`element` - element to be stored at the specified position

Returns:

the element previously at the specified position

Throws:

`UnsupportedOperationException` - if the set operation is not supported by this list

`ClassCastException` - if the class of the specified element prevents it from being added to this list

`NullPointerException` - if the specified element is null and this list does not permit null elements

`IllegalArgumentException` - if some property of the specified element prevents it from being added to this list

`IndexOutOfBoundsException` - if the index is out of range (index < 0 || index >= size())

Why Eiffel Contract Views are ADTs (1)

```
class interface ARRAYED_CONTAINER
feature -- Commands
  assign_at (i: INTEGER; s: STRING)
      -- Change the value at position 'i' to 's'.
    require
      valid_index: 1 <= i and i <= count
    ensure
      size_unchanged:
        imp.count = (old imp.twin).count
      item_assigned:
        imp [i] ~ s
      others_unchanged:
        across
          1 |..| imp.count as j
        all
          j.item /= i implies imp [j.item] ~ (old imp.twin) [j.item]
        end
  count: INTEGER
invariant
  consistency: imp.count = count
end -- class ARRAYED_CONTAINER
```

Why Eiffel Contract Views are ADTs (2)

Even better, the direct correspondence from Eiffel operators to logic allow us to present a *precise behavioural* view.

ARRAYED_CONTAINER

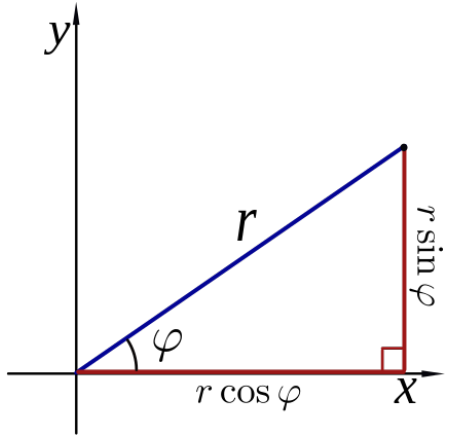
feature -- Commands
  assign_at (i: **INTEGER**; s: **STRING**)
    -- Change the value at position 'i' to 's'.
   require
    valid_index: 1 ≤ i ≤ count
   ensure
    size_unchanged: imp.count = (**old** imp.twin).count
    item_assigned: imp[i] ~ s
    others_unchanged: ∀j : 1 ≤ j ≤ imp.count : j ≠ i ⇒imp[j] ~ (**old** imp.twin) [j]

feature -- { **NONE** }
  -- Implementation of an arrayed-container
  imp: ARRAY[STRING]

invariant
consistency: imp.count = count

Uniform Access Principle (1)

- We may implement `Point` using two representation systems:



  - The *Cartesian system* stores the *absolute* positions of `x` and `y`.
  - The *Polar system* stores the *relative* position: the angle (in radian) `phi` and distance `r` from the origin (0.0).
- How the `Point` is implemented is irrelevant to users:
  - **Imp. 1**: Store `x` and `y`. [Compute `r` and `phi` on demand]
  - **Imp. 2**: Store `r` and `phi`. [Compute `x` and `y` on demand]
- As far as users of a `Point` object `p` is concerned, having a *uniform access* by always being able to call **`p.x`** and **`p.y`** is what matters, despite **Imp. 1** or **Imp. 2** being current strategy.

Uniform Access Principle (2)

```
class
  POINT
create
  make_cartisian, make_polar
feature -- Public, Uniform Access to x- and y-coordinates
  x : REAL
  y : REAL
end
```

- A class `Point` declares how users may access a point: either get its *x* coordinate or its *y* coordinate.
- We offer two possible ways to instantiating a 2-D point:
  - `make_cartisian (nx: REAL; ny: REAL)`
  - `make_polar (nr: REAL; np: REAL)`
- Features `x` and `y`, from the client's point of view, cannot tell whether it is implemented via:
  - ***Storage*** [`x` and `y` stored as real-valued ***attributes***]
  - ***Computation*** [`x` and `y` defined as ***queries*** returning real values]

Uniform Access Principle (3)

Let's say the supplier decides to adopt strategy **Imp. 1**.

```
class POINT -- Version 1
feature -- Attributes
  x : REAL
  y : REAL
feature -- Constructors
  make_cartisian(nx: REAL; nx: REAL)
    do
      x := nx
      y := ny
    end
end
```

- Attributes `x` and `y` represent the *Cartesian system*
- A client accesses a point `p` via **`p.x`** and **`p.y`**.
  - ***No Extra Computations***: just returning current values of `x` and `y`.
- However, it's harder to implement the other constructor: the body of `make_polar (nr: REAL; np: REAL)` has to compute and store `x` and `y` according to the inputs `nr` and `np`.

Uniform Access Principle (4)

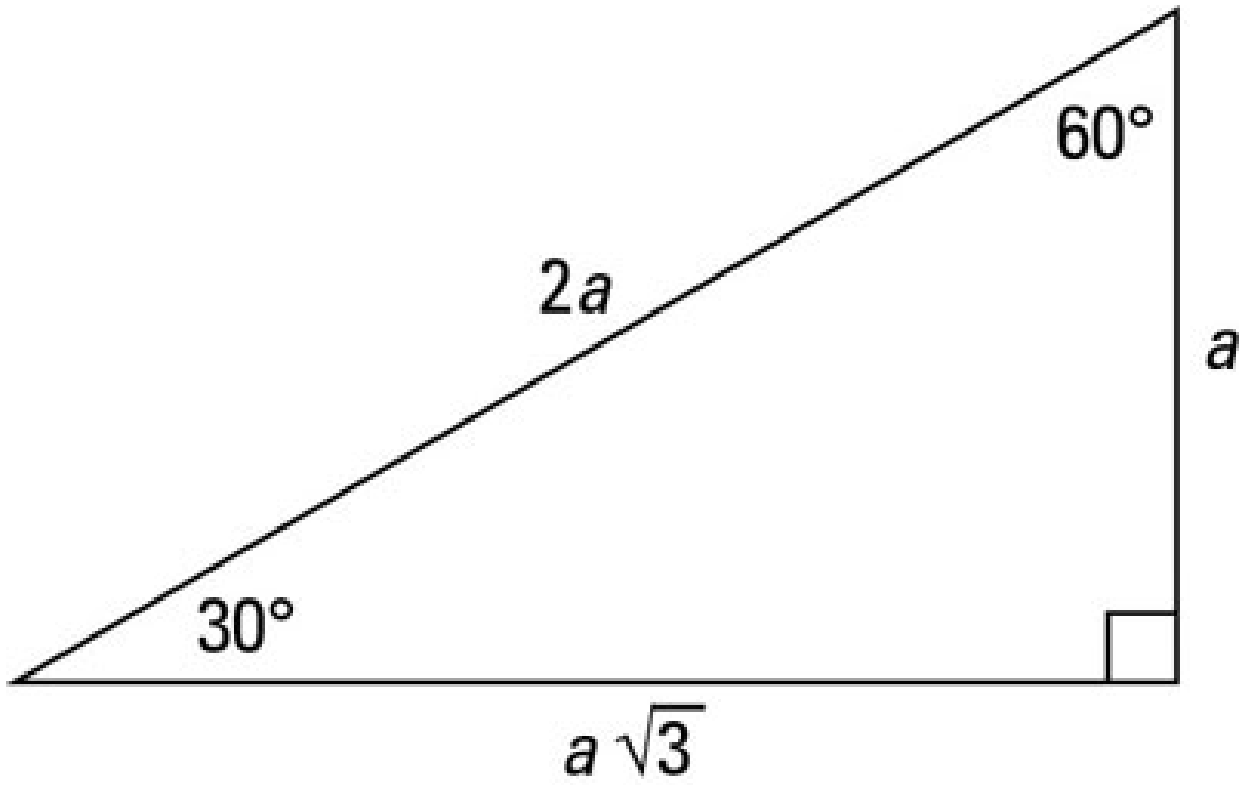
Let's say the supplier decides (*secretly*) to adopt strategy **Imp. 2**.

```
class POINT -- Version 2
feature -- Attributes
  r : REAL
  p : REAL
feature -- Constructors
  make_polar(nr: REAL; np: REAL)
    do
      r := nr
      p := np
    end
feature -- Queries
  x : REAL do Result := r × cos(p) end
  x : REAL do Result := r × sin(p) end
end
```

- Attributes `r` and `p` represent the *Polar system*
- A client **still** accesses a point `p` via **`p.x`** and **`p.y`**.
  - ***Extra Computations***: computing `x` and `y` according to the current values of `r` and `p`.

Uniform Access Principle (5.1)

Let's consider the following scenario as an example:



Note: 360° = 2π

Uniform Access Principle (5.2)

```
1  test_points: BOOLEAN
2   local
3     A, X, Y: REAL
4     p1, p2: POINT
5   do
6     comment("test: two systems of points")
7     A := 5; X := A × √3; Y := A
8     create {POINT} p1.make_cartisian (X, Y)
9     create {POINT} p2.make_polar (2 × A, 1/6 π)
10    Result := p1.x = p2.x and p1.y = p2.y
11  end
```

- If strategy **Imp. 1** is adopted:
  - **L8** is computationally cheaper than **L9**. [`x` and `y` attributes]
  - **L10** requires no computations to access `x` and `y`.

  If strategy **Imp. 2** is adopted:
  - **L9** is computationally cheaper than **L8**. [`r` and `p` attributes]
  - **L10** requires computations to access `x` and `y`.

Uniform Access Principle (6)

The *Uniform Access Principle*:

- Allows clients to use services (e.g., `p.x` and `p.y`) regardless of how they are implemented.
- Gives suppliers complete freedom as to how to implement the services (e.g., Cartesian vs. Polar).
  - No right or wrong implementation; it depends!
  - Choose for ***storage*** if the services are frequently accessed and their computations are expensive.
    e.g. `balance` of a bank involves a large number of accounts
    ⇒ Implement `balance` as an attribute
  - Choose for ***computation*** if the services are **not** keeping their values in sync is complicated.
    e.g., update `balance` upon a local deposit or withdrawal
    ⇒ Implement `balance` as a query
- Whether it's storage or computation, you can always change *secretly*, since the clients' access to the services is *uniform*.

Generic Collection Class: Motivation (1)

```
class STRING_STACK
feature {NONE} -- Implementation
  imp: ARRAY[ STRING ] ; i: INTEGER
feature -- Queries
  count: INTEGER do Result := i end
      -- Number of items on stack.
  top: STRING do Result := imp [i] end
      -- Return top of stack.
feature -- Commands
  push (v: STRING ) do imp[i] := v; i := i + 1 end
      -- Add 'v' to top of stack.
  pop do i := i - 1 end
      -- Remove top of stack.
end
```

- Does how we implement integer stack operations (e.g., `top`, `push`, `pop`) depends on features specific to element type `STRING` (e.g., `at`, `append`)? [***NO!***]
- How would you implement another class `ACCOUNT_STACK`?

Generic Collection Class: Motivation (2)

```
class ACCOUNT_STACK
feature {NONE} -- Implementation
  imp: ARRAY[ ACCOUNT ] ; i: INTEGER
feature -- Queries
  count: INTEGER do Result := i end
     -- Number of items on stack.
  top: ACCOUNT do Result := imp [i] end
     -- Return top of stack.
feature -- Commands
  push (v: ACCOUNT ) do imp[i] := v; i := i + 1 end
     -- Add 'v' to top of stack.
  pop do i := i - 1 end
     -- Remove top of stack.
end
```

- Does how we implement integer stack operations (e.g., `top`, `push`, `pop`) depends on features specific to element type `ACCOUNT` (e.g., `deposit`, `withdraw`)? [***NO!***]

Generic Collection Class: Supplier

- Your design *"smells"* if you have to create an ***almost identical*** new class (hence *code duplicates*) for every stack element type you need (e.g., `INTEGER`, `CHARACTER`, `PERSON`, etc.).
- Instead, as **supplier**, use `G` to *parameterize* element type:

```
class STACK [G]
feature {NONE} -- Implementation
  imp: ARRAY[ G ] ; i: INTEGER
feature -- Queries
  count: INTEGER do Result := i end
      -- Number of items on stack.
  top: G do Result := imp [i] end
      -- Return top of stack.
feature -- Commands
  push (v: G ) do imp[i] := v; i := i + 1 end
      -- Add 'v' to top of stack.
  pop do i := i - 1 end
      -- Remove top of stack.
end
```

Generic Collection Class: Client (1.1)

As **client**, declaring `ss: STACK[ STRING ]` instantiates every occurrence of `G` as `STRING`.

```
class STACK [G ≠ STRING]
feature {NONE} -- Implementation
  imp: ARRAY[ G ≠ STRING ] ; i: INTEGER
feature -- Queries
  count: INTEGER do Result := i end
     -- Number of items on stack.
  top: G ≠ STRING do Result := imp [i] end
     -- Return top of stack.
feature -- Commands
  push (v: G ≠ STRING) do imp[i] := v; i := i + 1 end
     -- Add 'v' to top of stack.
  pop do i := i - 1 end
     -- Remove top of stack.
end
```

Generic Collection Class: Client (1.2)

As **client**, declaring `ss: STACK[ ACCOUNT ]` instantiates every occurrence of `G` as `ACCOUNT`.

```
class STACK [G ACCOUNT]
feature {NONE} -- Implementation
  imp: ARRAY[ G ACCOUNT ] ; i: INTEGER
feature -- Queries
  count: INTEGER do Result := i end
     -- Number of items on stack.
  top: G ACCOUNT do Result := imp [i] end
     -- Return top of stack.
feature -- Commands
  push (v: G ACCOUNT ) do imp[i] := v; i := i + 1 end
     -- Add 'v' to top of stack.
  pop do i := i - 1 end
     -- Remove top of stack.
end
```

Generic Collection Class: Client (2)

As **client**, instantiate the type of `G` to be the one needed.

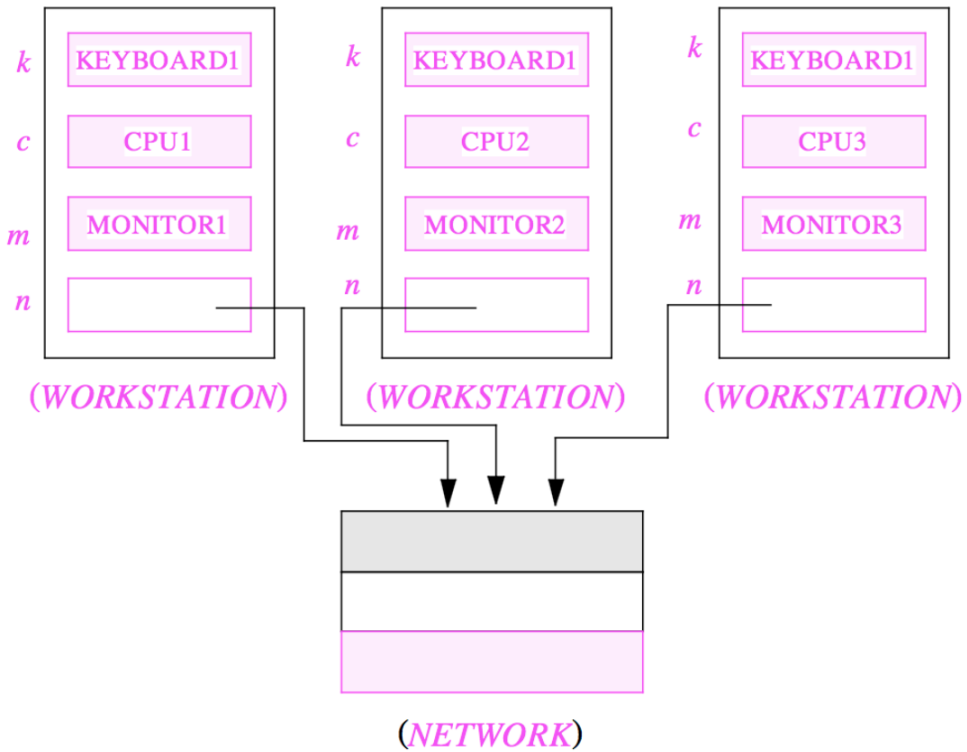
```
1   test_stacks: BOOLEAN
2    local
3      ss: STACK[STRING] ; sa: STACK[ACCOUNT]
4      s: STRING ; a: ACCOUNT
5    do
6      ss.push("A")
7      ss.push(create {ACCOUNT}.make ("Mark", 200))
8      s := ss.top
9      a := ss.top
10     sa.push(create {ACCOUNT}.make ("Alan", 100))
11     sa.push("B")
12     a := sa.top
13     s := sa.top
14   end
```

- **L3** commits that `ss` stores `STRING` objects only.
  - **L8** and **L10** ***valid***; **L9** and **L11** ***invalid***.
- **L4** commits that `sa` stores `ACCOUNT` objects only.
  - **L12** and **L14** ***valid***; **L13** and **L15** ***invalid***.

Expanded Class: Modelling

- We may want to have objects which are:
  - Integral parts of some other objects
  - ***Not*** shared among objects

  e.g., Each workstation has its own CPU, monitor, and keyword. All workstations share the same network.



Expanded Class: Programming (2)

```
class KEYBOARD ... end class CPU ... end
class MONITOR ... end class NETWORK ... end
class WORKSTATION
  k: expanded KEYBOARD
  c: expanded CPU
  m: expanded MONITOR
  n: NETWORK
end
```

Alternatively:

```
expanded class KEYBOARD ... end
expanded class CPU ... end
expanded class MONITOR ... end
class NETWORK ... end
class WORKSTATION
  k: KEYBOARD
  c: CPU
  m: MONITOR
  n: NETWORK
end
```

Expanded Class: Programming (3)

```
expanded class
  B
feature
  change_i (ni: INTEGER)
    do
      i := ni
    end
feature
  i: INTEGER
end
```

```
1  test_expanded: BOOLEAN
2   local
3     eb1, eb2: B
4   do
5     Result := eb1.i = 0 and eb2.i = 0
6     check Result end
7     Result := eb1 = eb2
8     check Result end
9     eb2.change_i (15)
10    Result := eb1.i = 0 and eb2.i = 15
11    check Result end
12    Result := eb1 /= eb2
13    check Result end
14  end
```

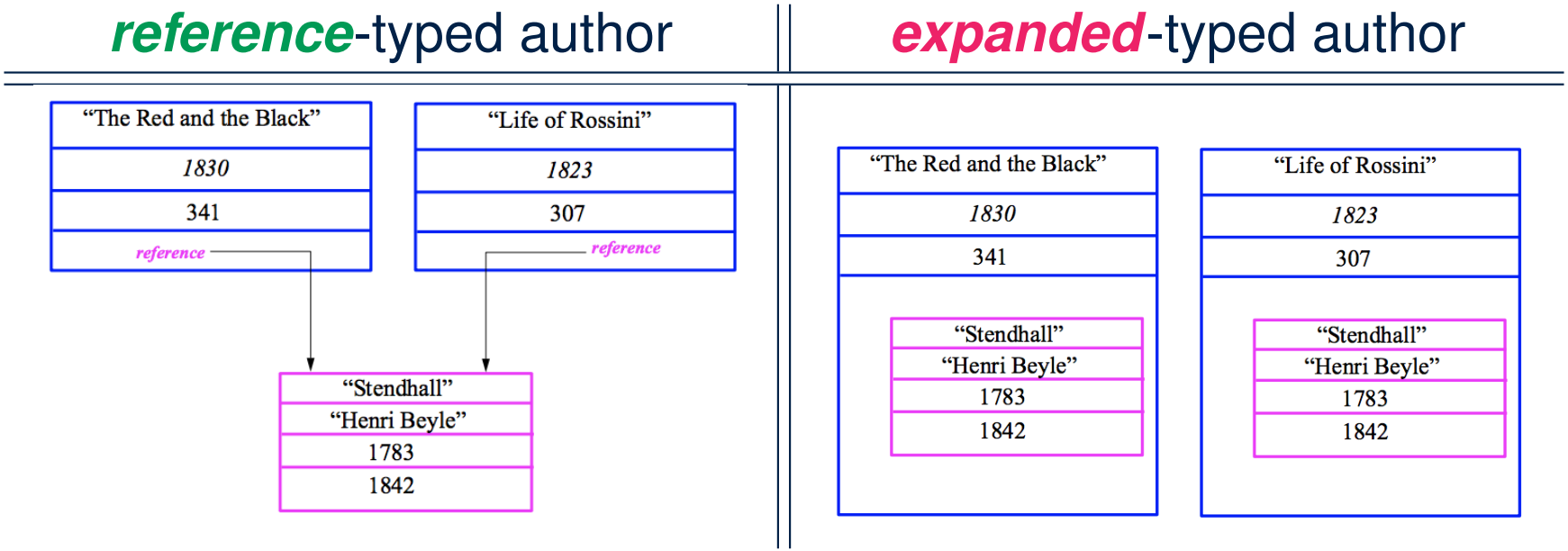
- **L5**: object of expanded type is automatically initialized.
- **L9 & L10**: no sharing among objects of expanded type.
- **L7 & L12**: = between expanded objects compare their contents.

Reference vs. Expanded (1)

- Every entity must be declared to be of a certain type (based on a class).
- Every type is either ***referenced*** or ***expanded***.
- In ***reference*** types:
  - `y` denotes ***a reference*** to some object
  - `x := y` attaches `x` to same object as does `y`
  - `x = y` compares references
- In ***expanded*** types:
  - `y` denotes ***some object*** (of expanded type)
  - `x := y` copies contents of `y` into `x`
  - `x = y` compares contents `[x ~ y]`

Reference vs. Expanded (2)

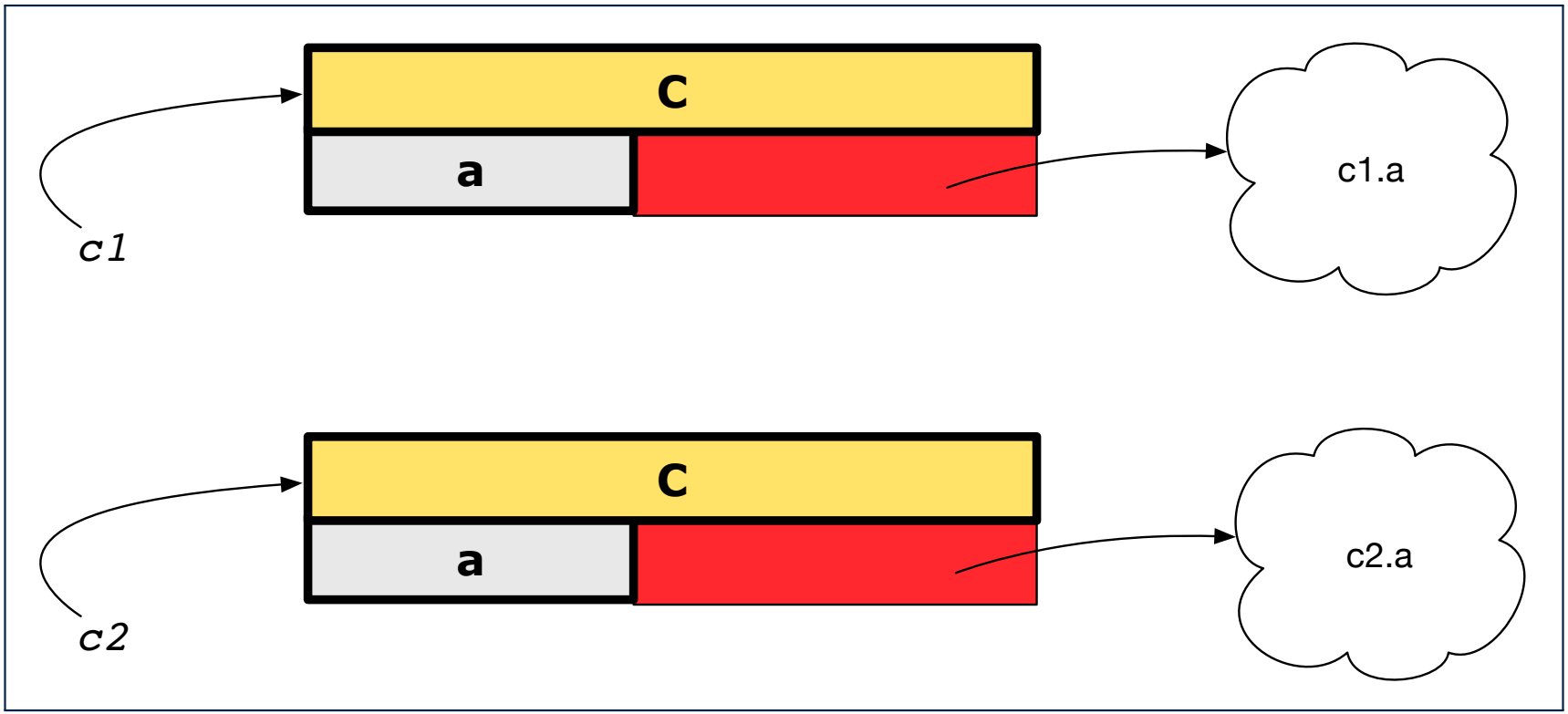
Problem: Every published book has an author. Every author may publish more than one books. Should the author field of a book ***reference***-typed or ***expanded***-typed?



Copying Objects

Say variables `c1` and `c2` are both declared of type `C`. [*c1, c2: C*]

- There is only one attribute `a` declared in class `C`.
- `c1.a` and `c2.a` may be of either:
  - ***expanded*** type or
  - ***reference*** type

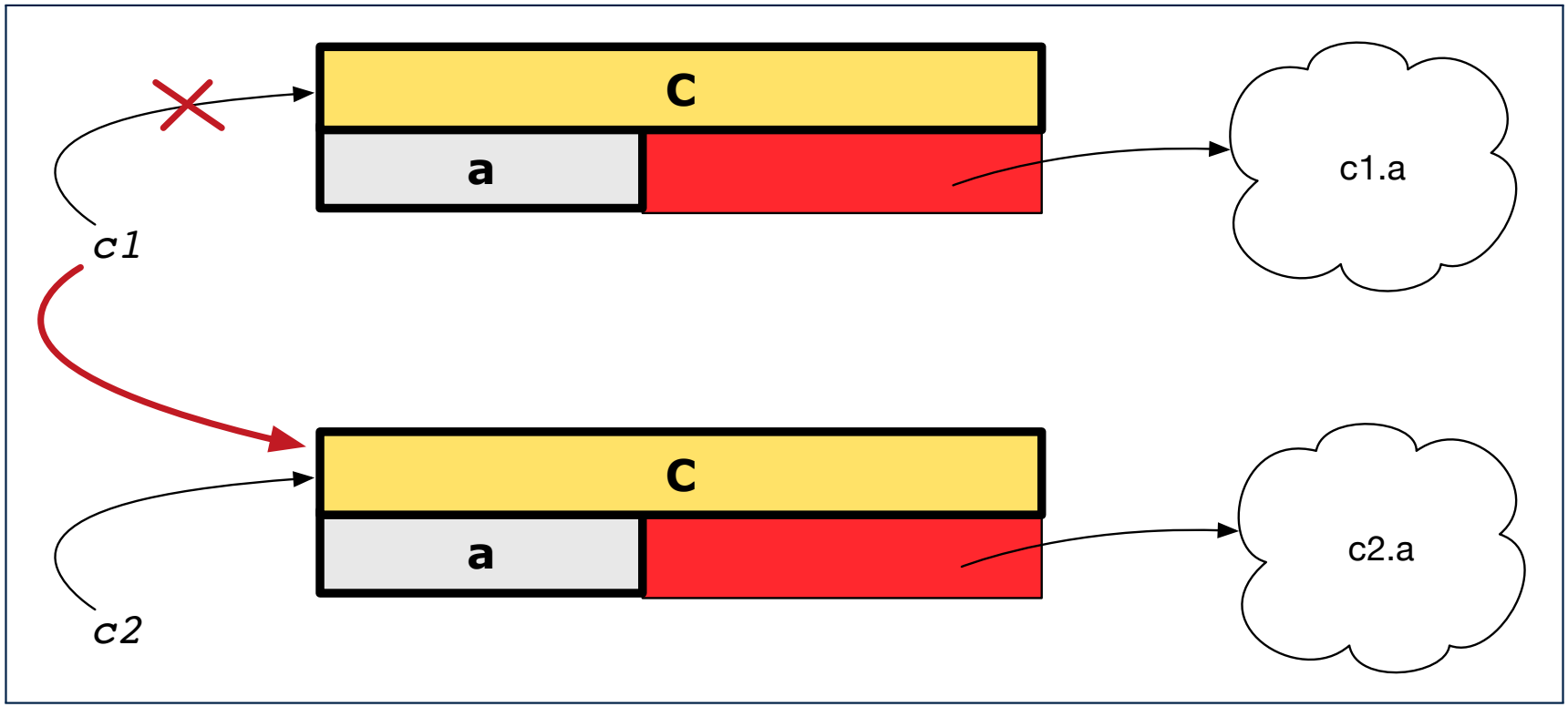


Copying Objects: Reference Copy

Reference Copy

`c1 := c2`

- Copy the address stored in variable `c2` and store it in `c1`.
  ⇒ Both `c1` and `c2` point to the same object.
  ⇒ Updates performed via `c1` also visible to `c2`. [***aliasing***]



Copying Objects: Shallow Copy

Shallow Copy

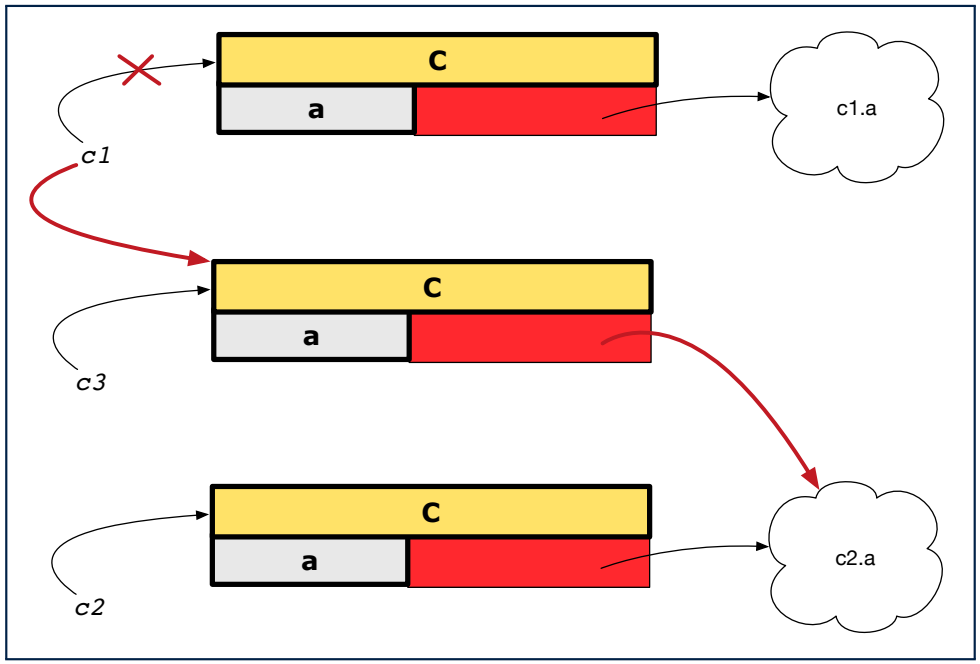
`c1 := c2.twin`

- Create a temporary, behind-the-scene object `c3` of type `C`.
- Initialize each attribute `a` of `c3` via ***reference copy***: `c3.a := c2.a`
- Make a ***reference copy*** of `c3`: `c1 := c3`

  ⇒ `c1` and `c2` ***are not*** pointing to the same object. [`c1 /= c2`]

  ⇒ `c1.a` and `c2.a` ***are*** pointing to the same object.

  ⇒ ***Aliasing*** still occurs: at 1st level (i.e., attributes of `c1` and `c2`)



Copying Objects: Deep Copy

Deep Copy

`c1 := c2.deep_twin`

- Create a temporary, behind-the-scene object `c3` of type `C`.
- *Recursively* initialize each attribute `a` of `c3` as follows:

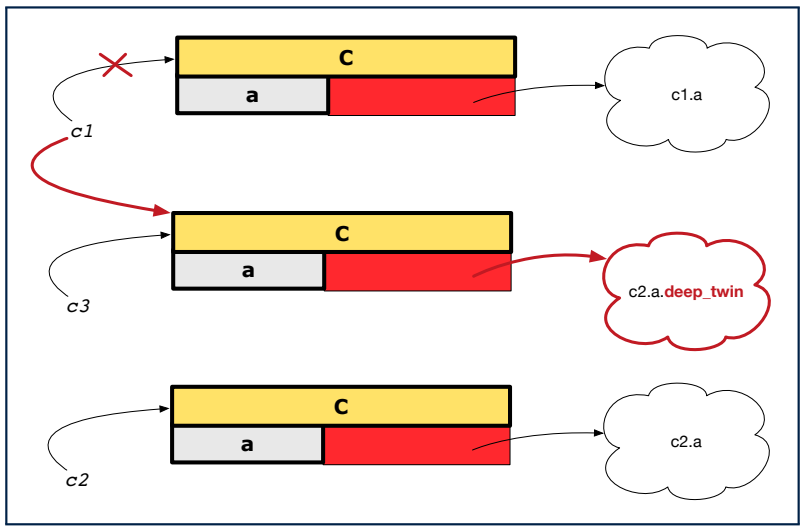
  Base Case: `a` is expanded (e.g., `INTEGER`). ⇒ `c3.a := c2.a`.

  Recursive Case: `a` is referenced. ⇒ `c3.a := c2.a.deep_twin`
- Make a ***reference copy*** of `c3`: `c1 := c3`

  ⇒ `c1` and `c2` ***are not*** pointing to the same object.

  ⇒ `c1.a` and `c2.a` ***are not*** pointing to the same object.

  ⇒ ***No aliasing*** occurs at any levels.



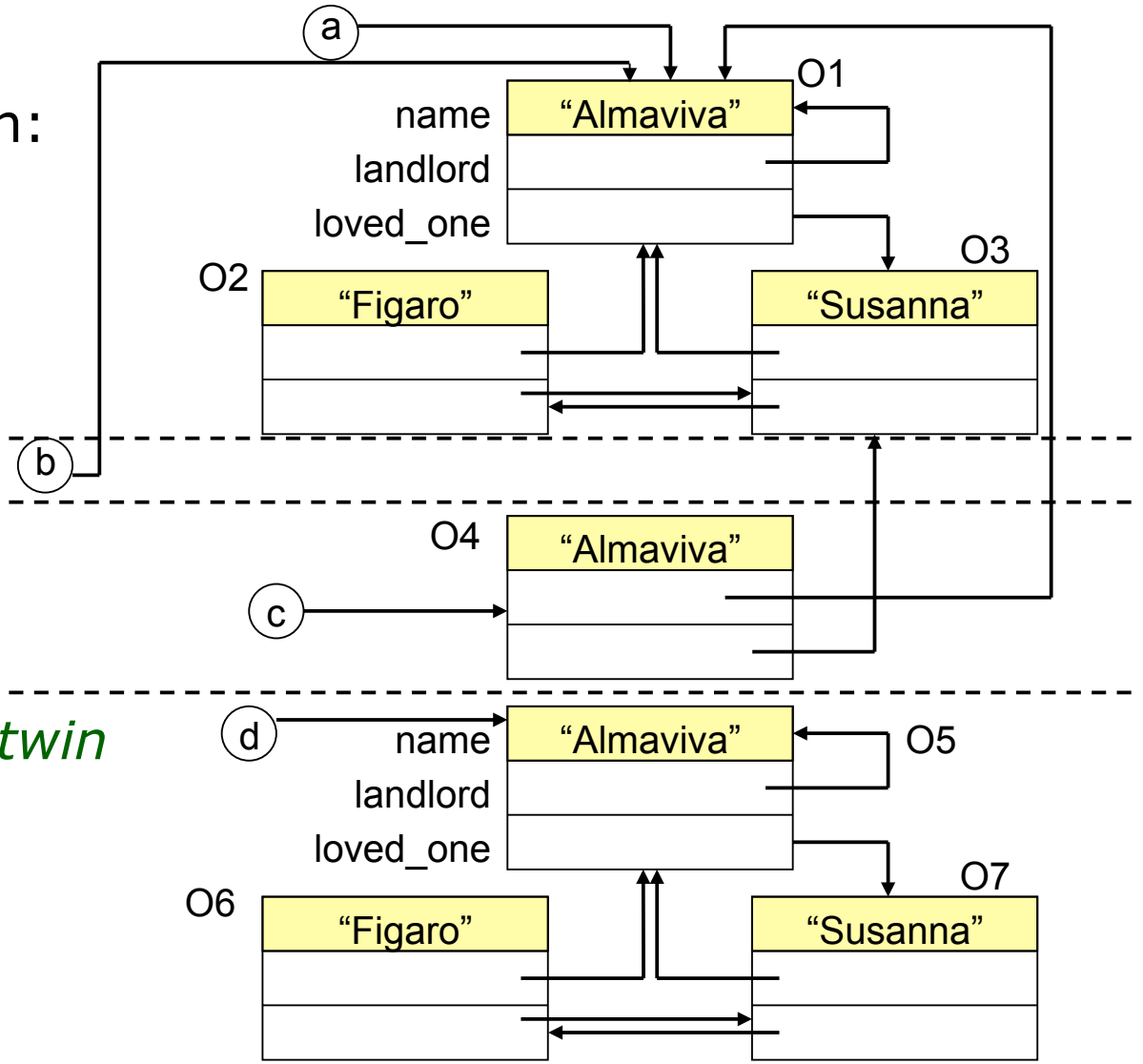
Copying Objects: Example

- Initial situation:
- Result of:

b := *a*

c := *a.twin*

d := *a.deep_twin*



Index (1)

Index (2)

Index (3)

Copying Objects: Example

Design Patterns: Singleton and Iterator

EECS3311: Software Design
Fall 2017

CHEN-WEI WANG

What are design patterns?

- Solutions to problems that arise when software is being developed within a particular context.
  - Heuristics for structuring your code so that it can be systematically maintained and extended.
  - *Caveat* : A pattern is only suitable for a particular problem.
  - Therefore, always understand *problems* before *solutions*!

Singleton Pattern: Motivation

Consider two problems:

1. Bank accounts share a set of data.

   e.g., interest and exchange rates, minimum and maximum balance, *etc*.

2. Processes are regulated to access some shared, limited resources.

Shared Data through Inheritance

Client:

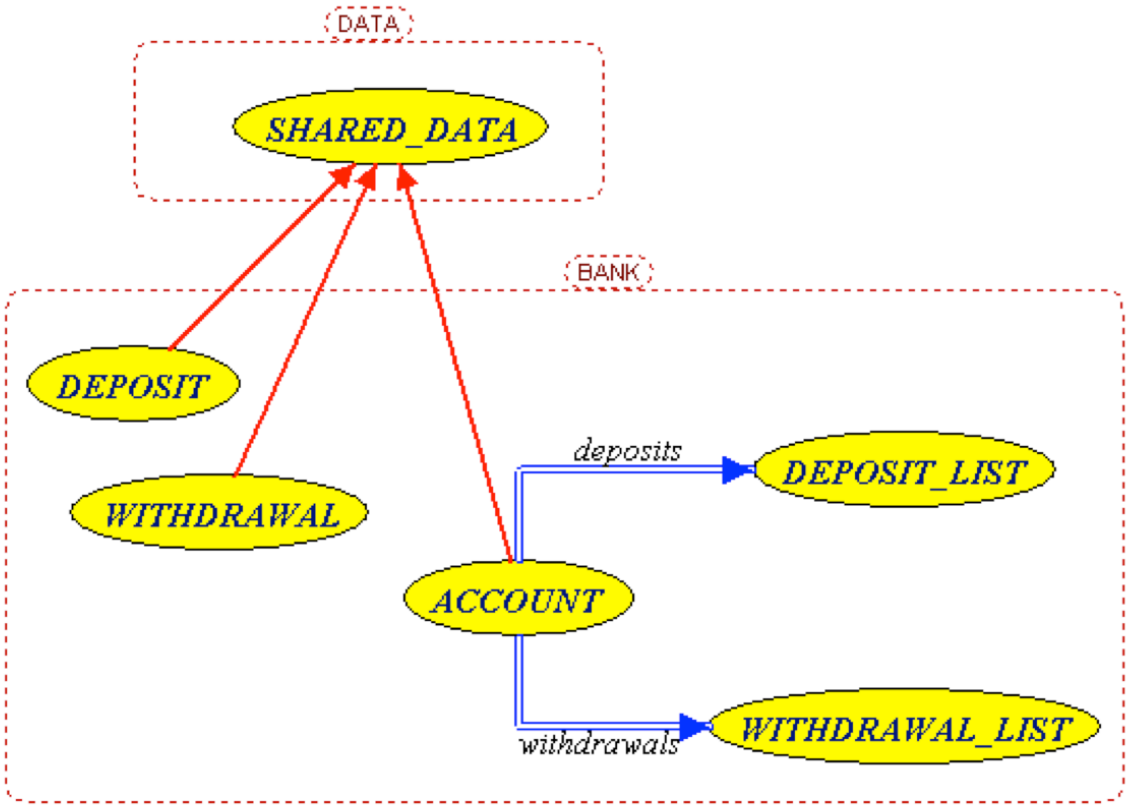
```
class DEPOSIT inherit SHARED_DATA
      ...
end

class WITHDRAW inherit SHARED_DATA
      ...
end

class ACCOUNT inherit SHARED_DATA
feature
      deposits: DEPOSIT_LIST
      withdraws: WITHDRAW_LIST
      ...
end
```

Supplier:

```
class
  SHARED_DATA
feature
  interest_rate: REAL
  exchange_rate: REAL
  minimum_balance: INTEGER
  maximum_balance: INTEGER
  ...
end
```

Problems?

Sharing Data through Inheritance: Architecture



- Irreverent features are inherited, breaking descendants' **cohesion**.
- Same set of data is duplicated as instances are created.

Sharing Data through Inheritance: Limitation

- Each instance at runtime owns a separate copy of the shared data.
- This makes inheritance *not* an appropriate solution for both problems:
  - What if the interest rate changes? Apply the change to all instantiated account objects?
  - An update to the global lock must be observable by all regulated processes.

  Solution:
  - Separate notions of *data* and its *shared access* in two separate classes.
  - Encapsulate the shared access itself in a separate class.

Introducing the Once Routine in Eiffel (1.1)

```
1  class A
2  create make
3  feature -- Constructor
4    make do end
5  feature -- Query
6    new_once_array (s: STRING): ARRAY[STRING]
7        -- A once query that returns an array.
8      once
9        create {ARRAY[STRING]} Result.make_empty
10       Result.force (s, Result.count + 1)
11     end
12   new_array (s: STRING): ARRAY[STRING]
13       -- An ordinary query that returns an array.
14     do
15       create {ARRAY[STRING]} Result.make_empty
16       Result.force (s, Result.count + 1)
17     end
18 end
```

L9 & L10 executed **only once** for initialization.

L15 & L16 executed **whenever** the feature is called.

Introducing the Once Routine in Eiffel (1.2)

```
1  test_query: BOOLEAN
2   local
3    a: A
4    arr1, arr2: ARRAY[STRING]
5   do
6    create a.make
7
8    arr1 := a.new_array ("Alan")
9    Result := arr1.count = 1 and arr1[1] ~ "Alan"
10   check Result end
11
12   arr2 := a.new_array ("Mark")
13   Result := arr2.count = 1 and arr2[1] ~ "Mark"
14   check Result end
15
16   Result := not (arr1 = arr2)
17   check Result end
18  end
```

Introducing the Once Routine in Eiffel (1.3)

```
1  test_once_query: BOOLEAN
2    local
3      a: A
4      arr1, arr2: ARRAY[STRING]
5    do
6      create a.make
7
8      arr1 := a.new_once_array ("Alan")
9      Result := arr1.count = 1 and arr1[1] ~ "Alan"
10     check Result end
11
12     arr2 := a.new_once_array ("Mark")
13     Result := arr2.count = 1 and arr2[1] ~ "Alan"
14     check Result end
15
16     Result := arr1 = arr2
17     check Result end
18 end
```

Introducing the Once Routine in Eiffel (2)

```
r (...): T
        once
                -- Some computations on Result
                ...
        end
```

- The ordinary **do** ... **end** is replaced by **once** ... **end**.
- The first time the **once** routine *r* is called by some client, it executes the body of computations and returns the computed result.
- From then on, the computed result is "cached".
- In every subsequent call to *r*, possibly by different clients, the body of *r* is not executed at all; instead, it just returns the "cached" result, which was computed in the very first call.
- **How does this help us?**
  Cache the reference to the same shared object!

Introducing the Once Routine in Eiffel (3)

- In Eiffel, the `once` routine:
  - Initializes its return value `Result` by some computation.
  - The initial computation is invoked only once.
  - Resulting value from the initial computation is cached and returned for all later calls to the once routine.
- Eiffel `once` routines are *different* from Java `static` accessors

  In Java, a `static` accessor
  - Does not have its computed return value "cached"
  - Has its computation performed ***freshly*** on every invocation
- Eiffel `once` routines are *different* from Java `static` attributes

  In Java, a `static` attribute
  - Is a value on storage
  - May be initialized via some simple expression
    e.g., `static int counter = 20;`
    but cannot be initialized via some sophisticated computation.
  - **Note**. By putting such initialization computation in a constructor, there would be a ***fresh*** computation whenever a new object is created.

Singleton Pattern in Eiffel

Supplier:

```
class BANK_DATA
create {BANK_DATA_ACCESS} make
feature {BANK_DATA_ACCESS}
  make do ... end
feature -- Data Attributes
  interest_rate: REAL
  set_interest_rate (r: REAL)
end
```

```
expanded class
  BANK_DATA_ACCESS
feature
  data: BANK_DATA
      -- The one and only access
    once create Result.make end
invariant data = data
```

Client:

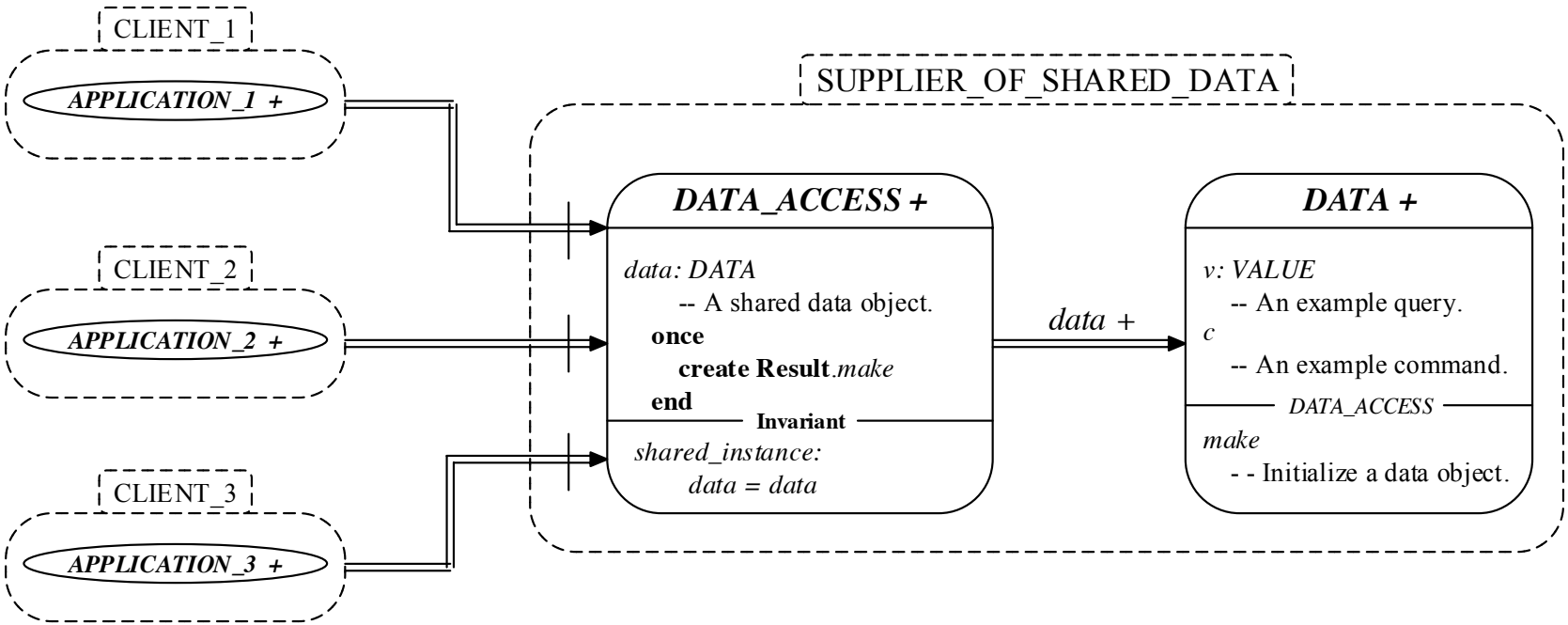
```
class
  ACCOUNT
feature
  data: BANK_DATA
  make (...)
      -- Init. access to bank data.
    local
      data_access: BANK_DATA_ACCESS
    do
      data := data_access.data
      ...
    end
end
```

Writing **create** *data.make* in client's *make* feature does not compile. Why?

Testing Singleton Pattern in Eiffel

```
test_bank_shared_data: BOOLEAN
    -- Test that a single data object is manipulated
  local
    acc1, acc2: ACCOUNT
  do
    comment("t1: test that a single data object is shared")
    create acc1.make ("Bill")
    create acc2.make ("Steve")

    Result := acc1.data ~ acc2.data
    check Result end

    Result := acc1.data = acc2.data
    check Result end

    acc1.data.set_interest_rate (3.11)
    Result := acc1.data.interest_rate = acc2.data.interest_rate
  end
```

Singleton Pattern: Architecture



Important Exercises: Instantiate this architecture to both problems of shared bank data and shared lock. Draw them in `draw.io`.

Iterator Pattern: Motivation

Supplier:

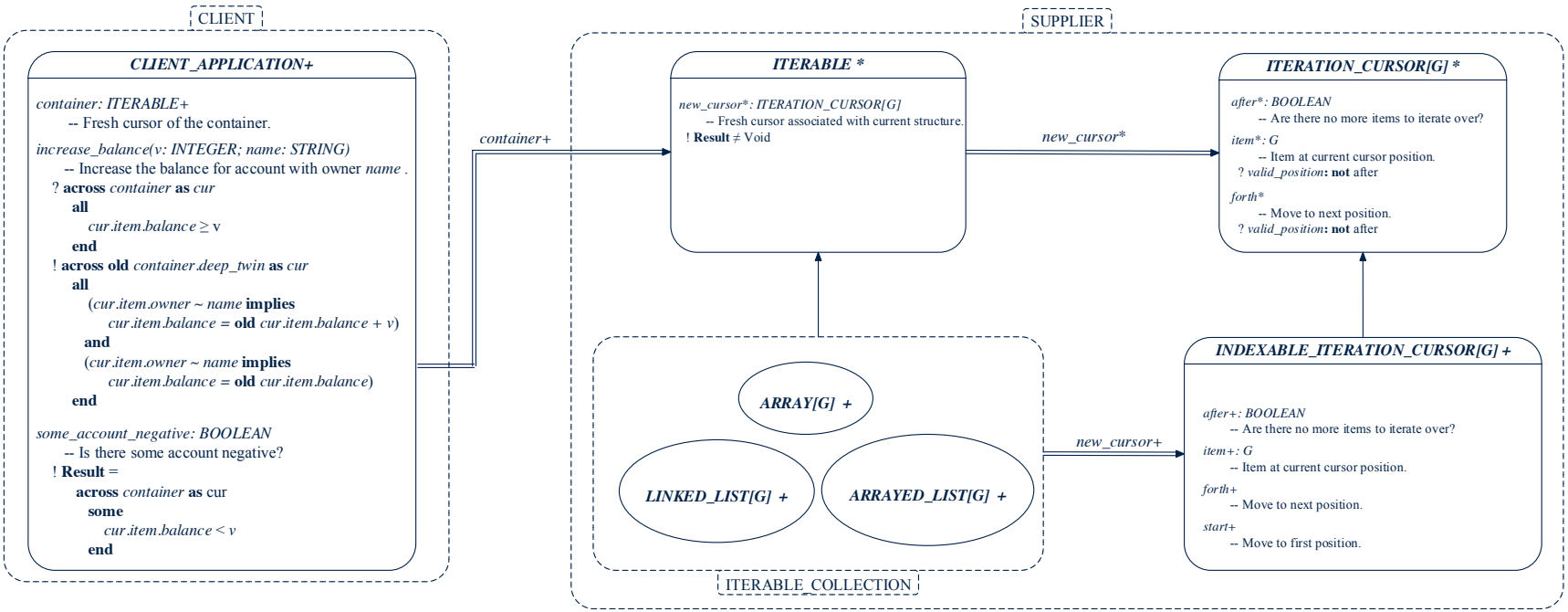
```
class
  CART
feature
  orders: ARRAY [ORDER]
end

class
  ORDER
feature
  price: INTEGER
  quantity: INTEGER
end
```

Problems?

Client:

```
class
  SHOP
feature
  cart: CART
  checkout: INTEGER
    do
      from
        i := cart.orders.lower
      until
        i > cart.orders.upper
      do
        Result := Result +
          cart.orders[i].price
          *
          cart.orders[i].quantity
        i := i + 1
      end
    end
end
```

Iterator Pattern: Architecture



Iterator Pattern: Supplier's Side

- **Information hiding**: changing the secret, internal workings of data structures should not affect any existing clients.

  e.g., changing from *ARRAY* to *LINKED_LIST* in the *CART* class

- Steps:

  1. Let the supplier class inherit from the deferred class *ITERABLE[G]*.

  2. This forces the supplier class to implement the inherited feature: *new_cursor: ITERATION_CURSOR [G]*, where the type parameter *G* may be instantiated (e.g., *ITERATION_CURSOR[ORDER]*).

  2.1 If the internal, library data structure is already *iterable* e.g., *imp: ARRAY[ORDER]*, then simply return *imp.new_cursor*.

  2.2 Otherwise, say *imp: MY_TREE[ORDER]*, then create a new class *MY_TREE_ITERATION_CURSOR* that inherits from *ITERATION_CURSOR[ORDER]*, then implement the 3 inherited features *after*, *item*, and *forth* accordingly.

Iterator Pattern: Supplier's Implementation (1)

```
class
  CART
inherit
  ITERABLE[ORDER]

...

feature {NONE} -- Information Hiding
  orders: ARRAY[ORDER]

feature -- Iteration
  new_cursor: ITERATION_CURSOR[ORDER]
    do
      Result := orders.new_cursor
    end
```

When the secrete implementation is already *iterable*, reuse it!

Iterator Pattern: Supplier's Imp. (2.1)

```
class
  GENERIC_BOOK[G]
inherit
  ITERABLE[ TUPLE[STRING, G] ]
...
feature {NONE} -- Information Hiding
  names: ARRAY[STRING]
  records: ARRAY[G]
feature -- Iteration
  new_cursor: ITERATION_CURSOR[ TUPLE[STRING, G] ]
    local
      cursor: MY_ITERATION_CURSOR[G]
    do
      create cursor.make (names, records)
      Result := cursor
    end
```

No Eiffel library support for iterable arrays ⇒ Implement it yourself!

Iterator Pattern: Supplier's Imp. (2.2)

```
class
  MY_ITERATION_CURSOR[G]
inherit
  ITERATION_CURSOR[ TUPLE[STRING, G] ]
feature -- Constructor
  make (ns: ARRAY[STRING]; rs: ARRAY[G])
    do ... end
feature {NONE} -- Information Hiding
  i: cursor_position
  names: ARRAY[STRING]
  records: ARRAY[G]
feature -- Cursor Operations
  item: TUPLE[STRING, G]
    do ... end
  after: Boolean
    do ... end
  forth
    do ... end
```

You need to implement the three inherited features: *item*, *after*, and *forth*.

Exercises

1. Draw the BON diagram showing how the iterator pattern is applied to the *CART* (supplier) and *SHOP* (client) classes.
2. Draw the BON diagram showing how the iterator pattern is applied to the supplier classes:
   - *GENERIC_BOOK* (a descendant of *ITERABLE*) and
   - *MY_ITERATION_CURSOR* (a descendant of *ITERATION_CURSOR*).

Iterator Pattern: Client's Side

Information hiding: the clients do not at all depend on *how* the supplier implements the collection of data; they are only interested in iterating through the collection in a linear manner.
Steps:

1. Obey the *code to interface, not to implementation* principle.
2. Let the client declare an attribute of type *ITERABLE[G]* (rather than *ARRAY*, *LINKED_LIST*, or *MY_TREE*).

   e.g., `cart: CART`, where *CART* inherits `ITERATBLE[ORDER]`
3. Eiffel supports, in both implementation and *contracts*, the **across** syntax for iterating through anything that's *iterable*.

Iterator Pattern: Clients using `across` for Contracts (1)

```
class
  CHECKER
feature -- Attributes
  collection: ITERABLE [INTEGER]
feature -- Queries
  is_all_positive: BOOLEAN
      -- Are all items in collection positive?
    do
      ...
    ensure
      across
        collection as cursor
      all
        cursor.item > 0
      end
  end
```

- Using **all** corresponds to a universal quantification (i.e., ∀).
- Using **some** corresponds to an existential quantification (i.e., ∃).

Iterator Pattern: Clients using `across` for Contracts (2)

```
class BANK
...
  accounts: LIST [ACCOUNT]
  binary_search (acc_id: INTEGER): ACCOUNT
      -- Search on accounts sorted in non-descending order.
    require
      across
        1 |..| (accounts.count - 1) as cursor
      all
        accounts [cursor.item].id <= accounts [cursor.item + 1].id
      end
    do
      ...
    ensure
      Result.id = acc_id
    end
```

This precondition corresponds to:

$$\forall i : INTEGER \mid 1 \leq i < accounts.count \bullet accounts[i].id \leq accounts[i+1].id$$

Iterator Pattern: Clients using `across` for Contracts (3)

```
class BANK
...
  accounts: LIST [ACCOUNT]
  contains_duplicate: BOOLEAN
     -- Does the account list contain duplicate?
    do
      ...
    ensure
       ∀ i, j : INTEGER  |
          1 ≤ i ≤ accounts.count  ∧  1 ≤ j ≤ accounts.count  •
             accounts[i] ~ accounts[j] ⇒ i = j
    end
```

- **Exercise:** Convert this mathematical predicate for postcondition into Eiffel.
- **Hint:** Each **across** construct can only introduce one dummy variable, but you may nest as many **across** constructs as necessary.

Iterator Pattern: Clients using Iterable in Imp. (1)

```
class BANK
  accounts: ITERABLE [ACCOUNT]
  max_balance: ACCOUNT
      -- Account with the maximum balance value.
    require ??
    local
      cursor: ITERATION_CURSOR[ACCOUNT]; max: ACCOUNT
    do
      from max := accounts [1]; cursor := accounts. new_cursor
      until cursor. after
      do
        if cursor. item .balance > max.balance then
          max := cursor. item
        end
        cursor. forth
      end
    ensure ??
    end
```

Iterator Pattern: Clients using Iterable in Imp. (2)

```
1  class SHOP
2    cart: CART
3    checkout: INTEGER
4       -- Total price calculated based on orders in the cart.
5      require ??
6      local
7        order: ORDER
8      do
9        across
10         cart as cursor
11       loop
12         order := cursor. item
13         Result := Result + order.price * order.quantity
14       end
15     ensure ??
16   end
```

- Class *CART* should inherit from *ITERABLE[ORDER]*.
- **L10** implicitly declares: `cursor: ITERATION_CURSOR[ORDER]`

Iterator Pattern: Clients using Iterable in Imp. (3)

```
class BANK
  accounts: ITERABLE [ACCOUNT]
  max_balance: ACCOUNT
      -- Account with the maximum balance value.
    require ??
    local
      max: ACCOUNT
    do
      max := accounts [1]
      across
        accounts as cursor
      loop
        if cursor.item.balance > max.balance then
          max := cursor. item
        end
      end
    ensure ??
    end
```

Index (1)

Index (2)

Index (3)

Writing Complete Contracts

EECS3311: Software Design
Fall 2017

CHEN-WEI WANG

How are contracts checked at runtime?

- All contracts are specified as Boolean expressions.
- Right **before** a feature call (e.g., *acc.withdraw(10)*):
  - The current state of *acc* is called its ***pre-state***.
  - Evaluate *pre-condition* using ***current values*** of attributes/queries.
  - Cache values of *all expressions involving the **old** keyword* in the *post-condition*.

    e.g., cache the value of **old** *balance* via *old_balance* := *balance*
- Right **after** the feature call:
  - The current state of *acc* is called its ***post-state***.
  - Evaluate *invariant* using ***current values*** of attributes and queries.
  - Evaluate *post-condition* using both ***current values*** and ***"cached" values*** of attributes and queries.

When are contracts complete?

LASSONDE SCHOOL OF ENGINEERING

- In *post-condition*, for *each attribute*, specify the relationship between its ***pre-state*** value and its ***post-state*** value.
  - Eiffel supports this purpose using the **old** keyword.
- This is tricky for attributes whose structures are **composite** rather than **simple**:

  e.g., *ARRAY*, *LINKED_LIST* are composite-structured.
  e.g., *INTEGER*, *BOOLEAN* are simple-structured.
- **Rule of thumb:** For an attribute whose structure is composite, we should specify that after the update:
  1. The intended change is present; **and**
  2. *The rest of the structure is unchanged*.
- The second contract is much harder to specify:
  - Reference aliasing [ref copy vs. shallow copy vs. deep copy]
  - Iterable structure [use `across`]

Account

```
class
  ACCOUNT
inherit
  ANY
    redefine is_equal end
create
  make

feature
  owner: STRING
  balance: INTEGER

  make (n: STRING)
    do
      owner := n
      balance := 0
    end
```

```
  deposit(a: INTEGER)
    do
      balance := balance + a
    ensure
      balance = old balance + a
    end

  is_equal(other: ACCOUNT): BOOLEAN
    do
      Result :=
            owner ~ other.owner
        and balance = other.balance
    end
end
```

Bank

```
class BANK
create make
feature
  accounts: ARRAY[ACCOUNT]
  make do create accounts.make_empty end
  account_of (n: STRING): ACCOUNT
    require
      existing: across accounts as acc some acc.item.owner ~ n end
    do ...
    ensure Result.owner ~ n
    end
  add (n: STRING)
    require
      non_existing:
        across accounts as acc all acc.item.owner /~ n end
    local new_account: ACCOUNT
    do
      create new_account.make (n)
      accounts.force (new_account, accounts.upper + 1)
    end
end
```

Roadmap of Illustrations

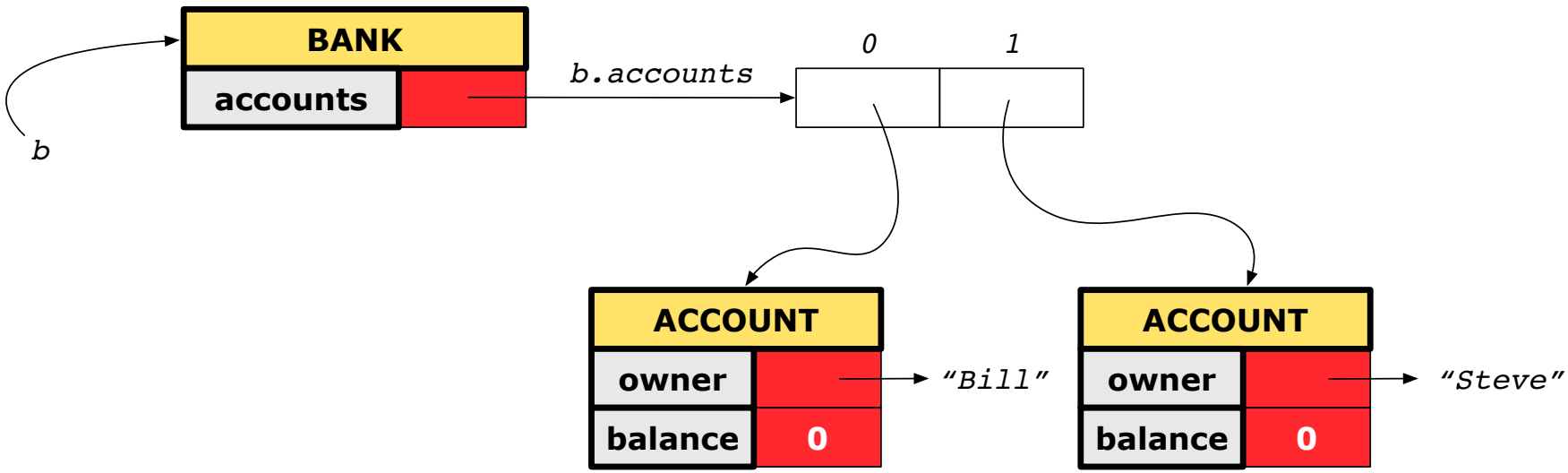
We examine 5 different versions of a command

$$deposit\_on\ (n : STRING;\ a : INTEGER)$$

VERSION	IMPLEMENTATION	CONTRACTS	SATISFACTORY?
1	*Correct*	*Incomplete*	*No*
2	*Wrong*	*Incomplete*	*No*
3	*Wrong*	*Complete* (reference copy)	*No*
4	*Wrong*	*Complete* (shallow copy)	*No*
5	*Wrong*	*Complete* (deep copy)	*Yes*

Object Structure for Illustration

We will test each version by starting with the same runtime object structure:



Version 1: Incomplete Contracts, Correct Implementation

```
class BANK
  deposit_on_v1 (n: STRING; a: INTEGER)
    require across accounts as acc some acc.item.owner ~ n end
    local i: INTEGER
    do
      from i := accounts.lower
      until i > accounts.upper
      loop
        if accounts[i].owner ~ n then accounts[i].deposit(a) end
        i := i + 1
      end
    ensure
      num_of_accounts_unchanged:
        accounts.count = old accounts.count
      balance_of_n_increased:
        account_of (n).balance = old account_of (n).balance + a
    end
end
```

Test of Version 1

```
class TEST_BANK
  test_bank_deposit_correct_imp_incomplete_contract: BOOLEAN
    local
      b: BANK
    do
      comment("t1: correct imp and incomplete contract")
      create b.make
      b.add ("Bill")
      b.add ("Steve")

      -- deposit 100 dollars to Steve's account
      b.deposit_on_v1 ("Steve", 100)
      Result :=
           b.account_of ("Bill").balance = 0
        and b.account_of ("Steve").balance = 100
      check Result end
  end
end
```

Test of Version 1: Result

APPLICATION

Note: * indicates a violation test case

PASSED (1 out of 1)		
Case Type	Passed	Total
Violation	0	0
Boolean	1	1
All Cases	1	1
State	Contract Violation	Test Name
Test1	TEST_BANK	
PASSED	NONE	t1: test deposit_on with correct imp and incomplete contract

Version 2: Incomplete Contracts, Wrong Implementation

```
class BANK
  deposit_on_v2 (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
    end
end
```

Current postconditions lack a check that accounts other than `n` are unchanged.

Test of Version 2

```
class TEST_BANK
test_bank_deposit_wrong_imp_incomplete_contract: BOOLEAN
  local
    b: BANK
  do
    comment("t2: wrong imp and incomplete contract")
    create b.make
    b.add ("Bill")
    b.add ("Steve")

    -- deposit 100 dollars to Steve's account
    b.deposit_on_v2 ("Steve", 100)
    Result :=
         b.account_of ("Bill").balance = 0
      and b.account_of ("Steve").balance = 100
    check Result end
  end
end
```

Test of Version 2: Result

APPLICATION

Note: * indicates a violation test case

FAILED (1 failed & 1 passed out of 2)		
Case Type	**Passed**	**Total**
Violation	0	0
Boolean	1	2
All Cases	1	2
State	**Contract Violation**	**Test Name**
Test1	**TEST_BANK**	
PASSED	NONE	t1: test deposit_on with correct imp and incomplete contract
FAILED	Check assertion violated.	t2: test deposit_on with wrong imp but incomplete contract

Version 3: Complete Contracts with Reference Copy

```
class BANK
  deposit_on_v3 (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 as cursor
        all cursor.item.owner /~ n implies
             cursor.item ~ account_of (cursor.item.owner)
        end
    end
end
```

Test of Version 3

```
class TEST_BANK
  test_bank_deposit_wrong_imp_complete_contract_ref_copy: BOOLEAN
    local
      b: BANK
    do
      comment("t3: wrong imp and complete contract with ref copy")
      create b.make
      b.add ("Bill")
      b.add ("Steve")

      -- deposit 100 dollars to Steve's account
      b.deposit_on_v3 ("Steve", 100)
      Result :=
           b.account_of ("Bill").balance = 0
        and b.account_of ("Steve").balance = 100
      check Result end
    end
end
```

Test of Version 3: Result

APPLICATION

Note: * indicates a violation test case

FAILED (2 failed & 1 passed out of 3)		
Case Type	Passed	Total
Violation	0	0
Boolean	1	3
All Cases	1	3
State	Contract Violation	Test Name
Test1	TEST_BANK	
PASSED	NONE	t1: test deposit_on with correct imp and incomplete contract
FAILED	Check assertion violated.	t2: test deposit_on with wrong imp but incomplete contract
FAILED	Check assertion violated.	t3: test deposit_on with wrong imp, complete contract with reference copy

Version 4: Complete Contracts with Shallow Object Copy

```
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
```

Test of Version 4

```
class TEST_BANK
  test_bank_deposit_wrong_imp_complete_contract_shallow_copy: BOOLEAN
    local
      b: BANK
    do
      comment("t4: wrong imp and complete contract with shallow copy")
      create b.make
      b.add ("Bill")
      b.add ("Steve")

      -- deposit 100 dollars to Steve's account
      b.deposit_on_v4 ("Steve", 100)
      Result :=
           b.account_of ("Bill").balance = 0
        and b.account_of ("Steve").balance = 100
      check Result end
    end
end
```

Test of Version 4: Result

APPLICATION

Note: * indicates a violation test case

FAILED (3 failed & 1 passed out of 4)

Case Type	Passed	Total
Violation	0	0
Boolean	1	4
All Cases	1	4

State	Contract Violation	Test Name
Test1	TEST_BANK	
PASSED	NONE	t1: test deposit_on with correct imp and incomplete contract
FAILED	Check assertion violated.	t2: test deposit_on with wrong imp but incomplete contract
FAILED	Check assertion violated.	t3: test deposit_on with wrong imp, complete contract with reference copy
FAILED	Check assertion violated.	t4: test deposit_on with wrong imp, complete contract with shallow object copy

Version 5: Complete Contracts with Deep Object 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
```

Test of Version 5

```
class TEST_BANK
  test_bank_deposit_wrong_imp_complete_contract_deep_copy: BOOLEAN
    local
      b: BANK
    do
      comment("t5: wrong imp and complete contract with deep copy")
      create b.make
      b.add ("Bill")
      b.add ("Steve")

      -- deposit 100 dollars to Steve's account
      b.deposit_on_v5 ("Steve", 100)
      Result :=
           b.account_of ("Bill").balance = 0
        and b.account_of ("Steve").balance = 100
      check Result end
    end
end
```

Test of Version 5: Result

APPLICATION

Note: * indicates a violation test case

FAILED (4 failed & 1 passed out of 5)		
Case Type	**Passed**	**Total**
Violation	0	0
Boolean	1	5
All Cases	1	5
State	**Contract Violation**	**Test Name**
Test1	TEST_BANK	
PASSED	NONE	t1: test deposit_on with correct imp and incomplete contract
FAILED	Check assertion violated.	t2: test deposit_on with wrong imp but incomplete contract
FAILED	Check assertion violated.	t3: test deposit_on with wrong imp, complete contract with reference copy
FAILED	Check assertion violated.	t4: test deposit_on with wrong imp, complete contract with shallow object copy
FAILED	Postcondition violated.	t5: test deposit_on with wrong imp, complete contract with deep object copy

Exercise

- Consider the query *account_of (n: STRING)* of *BANK*.
- How do we specify (part of) its postcondition to assert that the state of the bank remains unchanged:
  - `accounts = old accounts` [×]
  - `accounts = old accounts.twin` [×]
  - `accounts = old accounts.deep_twin` [×]
  - `accounts ~ old accounts` [×]
  - `accounts ~ old accounts.twin` [×]
  - `accounts ~ old accounts.deep_twin` [✓]
- Which equality of the above is appropriate for the postcondition?
- Why is each one of the other equalities not appropriate?

Index (1)

Index (2)

Inheritance

Readings: OOSCS2 Chapters 14 – 16

EECS3311: Software Design
Fall 2017

CHEN-WEI WANG

Why Inheritance: A Motivating Example

Problem: A *student management system* stores data about students. There are two kinds of university students: *resident* students and *non-resident* students. Both kinds of students have a *name* and a list of *registered courses*. Both kinds of students are restricted to *register* for no more than 30 courses. When *calculating the tuition* for a student, a base amount is first determined from the list of courses they are currently registered (each course has an associated fee). For a non-resident student, there is a *discount rate* applied to the base amount to waive the fee for on-campus accommodation. For a resident student, there is a *premium rate* applied to the base amount to account for the fee for on-campus accommodation and meals.

Tasks: Design classes that satisfy the above problem statement. At runtime, each type of student must be able to register a course and calculate their tuition fee.

The COURSE Class

```
class
  COURSE

create -- Declare commands that can be used as constructors
  make

feature -- Attributes
  title: STRING
  fee: REAL

feature -- Commands
  make (t: STRING; f: REAL)
      -- Initialize a course with title 't' and fee 'f'.
    do
      title := t
      fee := f
    end
end
```

No Inheritance: RESIDENT_STUDENT Class

```
class RESIDENT_STUDENT
create make
feature -- Attributes
  name: STRING
  courses: LINKED_LIST[COURSE]
  premium_rate:  REAL
feature -- Constructor
  make (n: STRING)
    do name := n ; create courses.make end
feature -- Commands
  set_pr (r:  REAL) do premium_rate := r end
  register (c: COURSE) do courses.extend (c) end
feature -- Queries
  tuition: REAL
    local base: REAL
    do base := 0.0
       across courses as c loop base := base + c.item.fee end
       Result := base  * premium_rate
    end
end
```

No Inheritance: RESIDENT_STUDENT Class

```
class NON_RESIDENT_STUDENT
create make
feature -- Attributes
  name: STRING
  courses: LINKED_LIST[COURSE]
  discount_rate:  REAL
feature -- Constructor
  make (n: STRING)
    do name := n ; create courses.make end
feature -- Commands
  set_dr (r:  REAL) do discount_rate := r end
  register (c: COURSE) do courses.extend (c) end
feature -- Queries
  tuition: REAL
    local base: REAL
    do base := 0.0
       across courses as c loop base := base + c.item.fee end
       Result := base  * discount_rate
    end
end
```

No Inheritance: Testing Student Classes

```
test_students: BOOLEAN
  local
    c1, c2: COURSE
    jim: RESIDENT_STUDENT
    jeremy: NON_RESIDENT_STUDENT
  do
    create c1.make ("EECS2030", 500.0)
    create c2.make ("EECS3311", 500.0)
    create jim.make ("J. Davis")
    jim.set_pr (1.25)
    jim.register (c1)
    jim.register (c2)
    Result := jim.tuition = 1250
    check Result end
    create jeremy.make ("J. Gibbons")
    jeremy.set_dr (0.75)
    jeremy.register (c1)
    jeremy.register (c2)
    Result := jeremy.tuition = 750
  end
```

No Inheritance: Issues with the Student Classes

- Implementations for the two student classes seem to work. But can you see any potential problems with it?
- The code of the two student classes share a lot in common.
- *Duplicates of code make it hard to maintain your software!*
- This means that when there is a change of policy on the common part, we need modify *more than one places*.

  $\Rightarrow$ This violates the *Single Choice Principle*

No Inheritance: Maintainability of Code (1)

What if a ***new*** way for course registration is to be implemented?

e.g.,

```
register(Course c)
  do
    if courses.count >= MAX_CAPACITY then
      -- Error: maximum capacity reached.
    else
      courses.extend (c)
    end
  end
```

We need to change the `register` commands in ***both*** student classes!

⇒ ***Violation*** of the *Single Choice Principle*

No Inheritance: Maintainability of Code (2)

What if a ***new*** way for base tuition calculation is to be implemented?

e.g.,

```
tuition: REAL
   local base: REAL
 do base := 0.0
    across courses as c loop base := base + c.item.fee end
    Result := base * inflation_rate * ...
 end
```

We need to change the `tuition` query in ***both*** student classes.

⇒ ***Violation*** of the *Single Choice Principle*

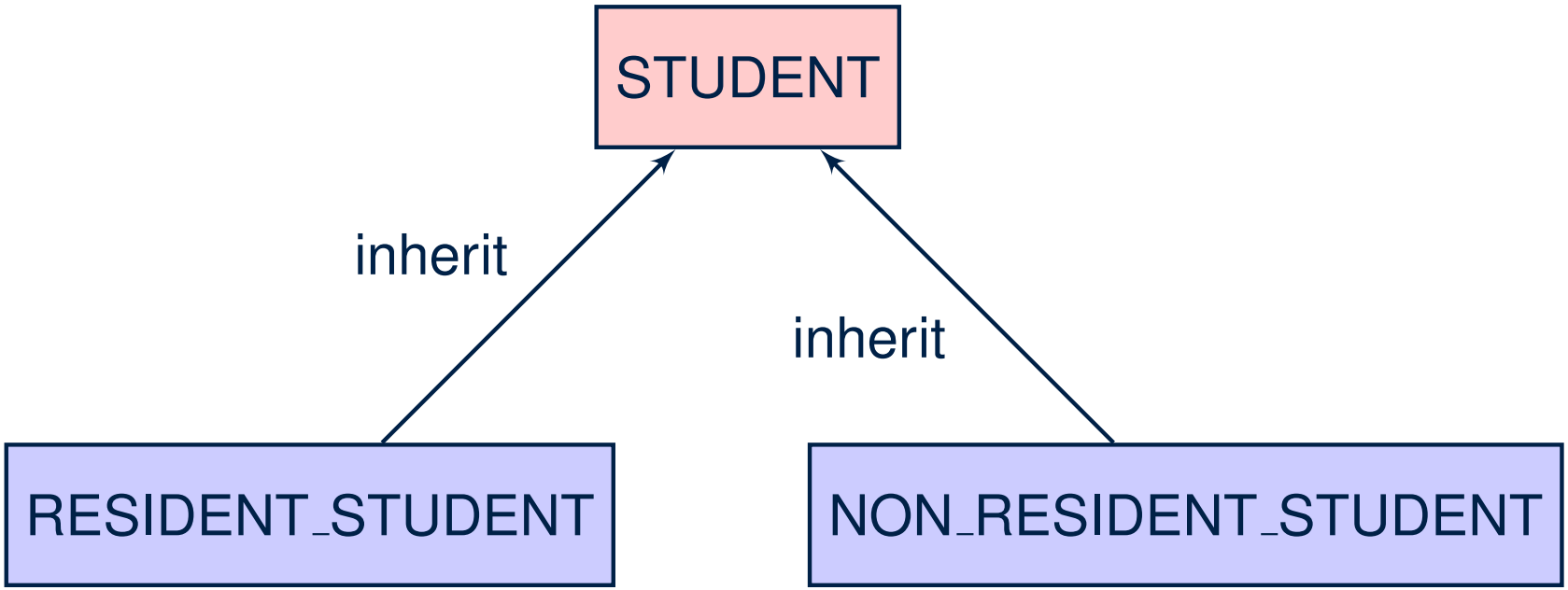
No Inheritance: A Collection of Various Kinds of Students

How do you define a class `StudentManagementSystem` that contains a list of *resident* and *non-resident* students?

```
class STUDENT_MANAGEMENT_SYSETM
  rs : LINKED_LIST[RESIDENT_STUDENT]
  nrs : LINKED_LIST[NON_RESIDENT_STUDENT]
  add_rs (rs: RESIDENT_STUDENT) do ... end
  add_nrs (nrs: NON_RESIDENT_STUDENT) do ... end
  register_all (Course c) -- Register a common course 'c'.
    do
      across rs as c loop c.item.register (c) end
      across nrs as c loop c.item.register (c) end
    end
end
```

But what if we later on introduce *more kinds of students*? *Inconvenient* to handle each list of students, in pretty much the ***same*** manner, *separately*!

Inheritance Architecture



Inheritance: The STUDENT Parent Class

```
1  class STUDENT
2  create make
3  feature -- Attributes
4    name: STRING
5    courses: LINKED_LIST[COURSE]
6  feature -- Commands that can be used as constructors.
7    make (n: STRING) do name := n ; create courses.make end
8  feature -- Commands
9    register (c: COURSE) do courses.extend (c) end
10 feature -- Queries
11   tuition: REAL
12     local base: REAL
13     do base := 0.0
14        across courses as c loop base := base + c.item.fee end
15        Result := base
16     end
17 end
```

Inheritance: The RESIDENT_STUDENT Child Class

```
1  class
2    RESIDENT_STUDENT
3  inherit
4    STUDENT
5      redefine tuition end
6  create make
7  feature -- Attributes
8    premium_rate : REAL
9  feature -- Commands
10   set_pr (r: REAL) do premium_rate := r end
11 feature -- Queries
12   tuition: REAL
13     local base: REAL
14     do base := Precursor ;  Result := base * premium_rate end
15 end
```

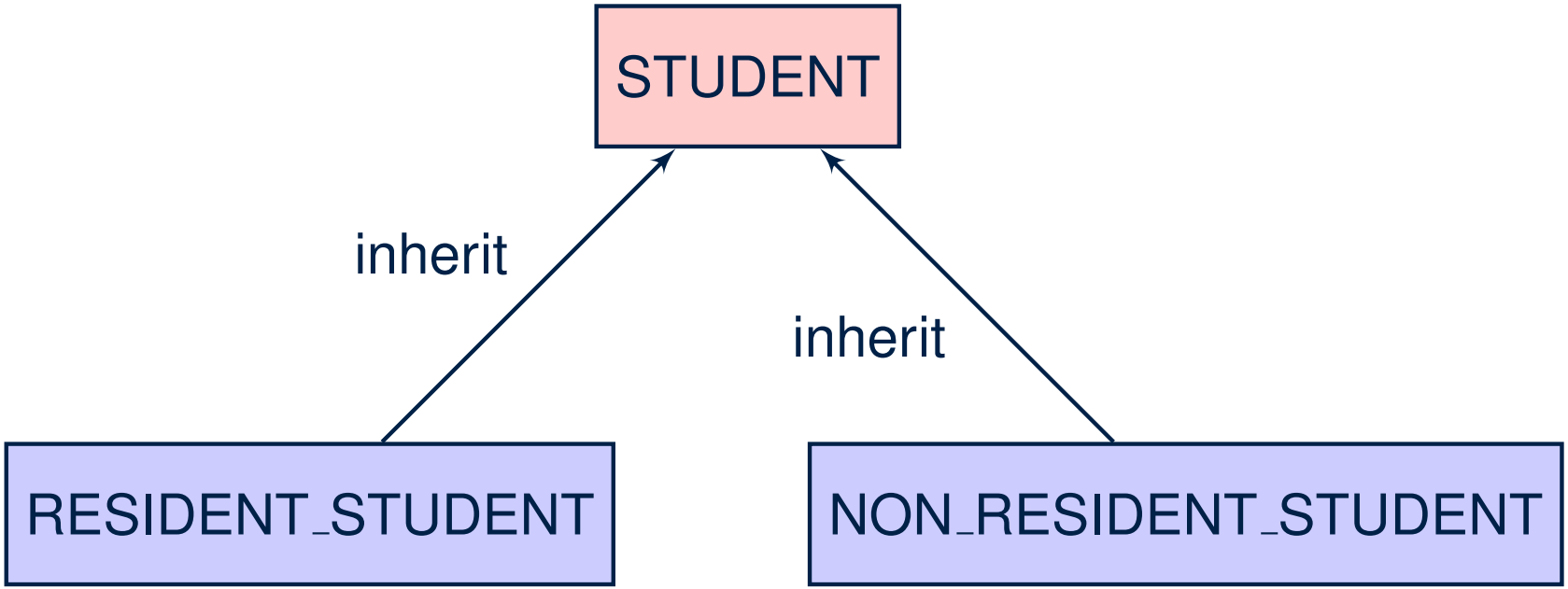
- **L3**: RESIDENT_STUDENT inherits all features from STUDENT.
- There is no need to repeat the register command
- **L14**: *Precursor* returns the value from query tuition in STUDENT.

Inheritance: The NON_RESIDENT_STUDENT Child Class

```
1  class
2    NON_RESIDENT_STUDENT
3  inherit
4    STUDENT
5      redefine tuition end
6  create make
7  feature -- Attributes
8    discount_rate : REAL
9  feature -- Commands
10   set_dr (r: REAL) do discount_rate := r end
11 feature -- Queries
12   tuition: REAL
13     local base: REAL
14     do base := Precursor ;  Result := base * discount_rate end
15 end
```

- **L3**: NON_RESIDENT_STUDENT inherits all features from STUDENT.
- There is no need to repeat the register command
- **L14**: *Precursor* returns the value from query tuition in STUDENT.

Inheritance Architecture Revisited



- The class that defines the common features (attributes, commands, queries) is called the *parent*, *super*, or *ancestor* class.
- Each "specialized" class is called a *child*, *sub*, or *descendent* class.

Using Inheritance for Code Reuse

Inheritance in Eiffel (or any OOP language) allows you to:

- Factor out *common features* (attributes, commands, queries) in a separate class.
  e.g., the `STUDENT` class
- Define an "specialized" version of the class which:
  - *inherits* definitions of all attributes, commands, and queries
    e.g., attributes `name`, `courses`
    e.g., command `register`
    e.g., query on base amount in `tuition`
    This means code reuse and elimination of code duplicates!
  - *defines* **new** features if necessary
    e.g., `set_pr` for `RESIDENT_STUDENT`
    e.g., `set_dr` for `NON_RESIDENT_STUDENT`
  - *redefines* features if necessary
    e.g., compounded tuition for `RESIDENT_STUDENT`
    e.g., discounted tuition for `NON_RESIDENT_STUDENT`

Testing the Two Student Sub-Classes

```
test_students: BOOLEAN
 local
  c1, c2: COURSE
  jim: RESIDENT_STUDENT ; jeremy: NON_RESIDENT_STUDENT
 do
  create c1.make ("EECS2030", 500.0); create c2.make ("EECS3311", 500.0)
  create jim.make ("J. Davis")
  jim.set_pr (1.25) ; jim.register (c1); jim.register (c2)
  Result := jim.tuition = 1250
  check Result end
  create jeremy.make ("J. Gibbons")
  jeremy.set_dr (0.75); jeremy.register (c1); jeremy.register (c2)
  Result := jeremy.tuition = 750
 end
```

- The software can be used in exactly the same way as before (because we did not modify *feature signatures*).
- But now the internal structure of code has been made *maintainable* using *inheritance*.

Static Type vs. Dynamic Type

- In *object orientation*, an entity has two kinds of types:
  - *static type* is declared at compile time [**unchangeable**]
    An entity's ***ST*** determines what features may be called upon it.
  - *dynamic type* is changeable at runtime
- In Java:

```
Student s = new Student("Alan");
Student rs = new ResidentStudent("Mark");
```

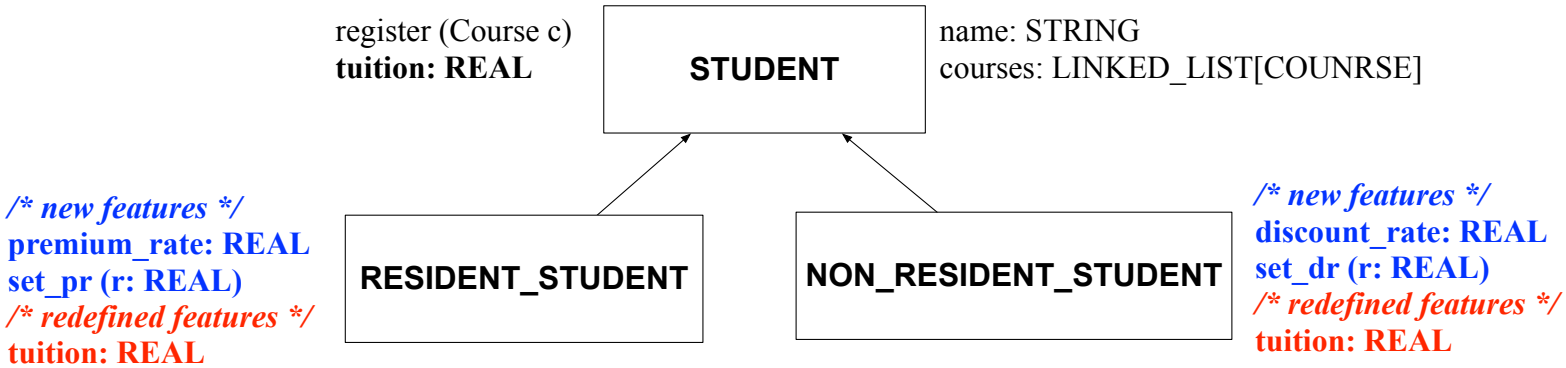
- In Eiffel:

```
local s: STUDENT
      rs: STUDENT
do create {STUDENT} s.make ("Alan")
   create {RESIDENT_STUDENT} rs.make ("Mark")
```

  - In Eiffel, the *dynamic type* can be ignored if it is meant to be the same as the *static type*:

```
local s: STUDENT
do create s.make ("Alan")
```

Inheritance Architecture Revisited




```
s1,s2,s3: STUDENT ; rs: RESIDENT_STUDENT ; nrs : NON_RESIDENT_STUDENT
create {STUDENT} s1.make ("S1")
create {RESIDENT_STUDENT} s2.make ("S2")
create {NON_RESIDENT_STUDENT} s3.make ("S3")
create {RESIDENT_STUDENT} rs.make ("RS")
create {NON_RESIDENT_STUDENT} nrs.make ("NRS")
```

	name	courses	reg	tuition	pr	set_pr	dr	set_dr
s1.	✓				×			
s2.	✓				×			
s3.	✓				×			
rs.	✓				✓		×	
nrs.	✓				×		✓	

Polymorphism: Intuition (1)

```
1  local
2    s: STUDENT
3    rs: RESIDENT_STUDENT
4  do
5    create s.make ("Stella")
6    create rs.make ("Rachael")
7    rs.set_pr (1.25)
8    s := rs /* Is this valid? */
9    rs := s /* Is this valid? */
```

- Which one of **L8** and **L9** is *valid*? Which one is *invalid*?
  - **L8**: What *kind* of address can `s` store? [`STUDENT`]

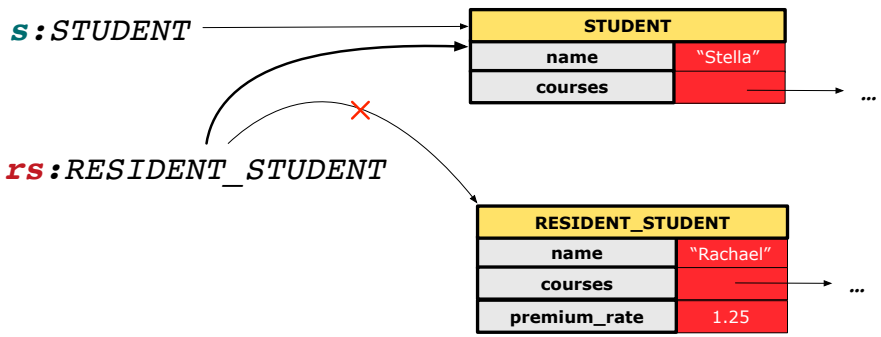
    ∴ The context object `s` is *expected* to be used as:
    - `s.register(eecs3311)` and `s.tuition`
  - **L9**: What *kind* of address can `rs` store? [`RESIDENT_STUDENT`]

    ∴ The context object `rs` is *expected* to be used as:
    - `rs.register(eecs3311)` and `rs.tuition`
    - `rs.set_pr (1.50)` [increase premium rate]

Polymorphism: Intuition (2)

```
1  local s: STUDENT ; rs: RESIDENT_STUDENT
2  do create {STUDENT} s.make ("Stella")
3     create {RESIDENT_STUDENT} rs.make ("Rachael")
4     rs.set_pr (1.25)
5     s := rs /* Is this valid? */
6     rs := s /* Is this valid? */
```

- **rs** := **s** (**L6**) should be *invalid*:



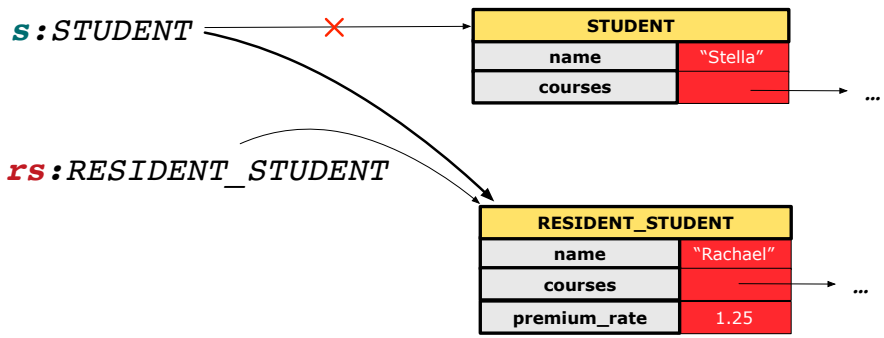
- **rs** declared of type `RESIDENT_STUDENT`
  ∴ calling **rs**.*set_pr(1.50)* can be expected.
- **rs** is now pointing to a `STUDENT` object.
- Then, what would happen to **rs**.*set_pr(1.50)*?

CRASH ∵ **rs**.`premium_rate` is *undefined*!!

Polymorphism: Intuition (3)

```
1  local s: STUDENT ; rs: RESIDENT_STUDENT
2  do create {STUDENT} s.make ("Stella")
3     create {RESIDENT_STUDENT} rs.make ("Rachael")
4     rs.set_pr (1.25)
5     s := rs /* Is this valid? */
6     rs := s /* Is this valid? */
```

- **s** := **rs** (**L5**) should be *valid*:



- Since **s** is declared of type `STUDENT`, a subsequent call **s**`.set_pr(1.50)` is *never* expected.
- **s** is now pointing to a `RESIDENT_STUDENT` object.
- Then, what would happen to **s**`.tuition`?

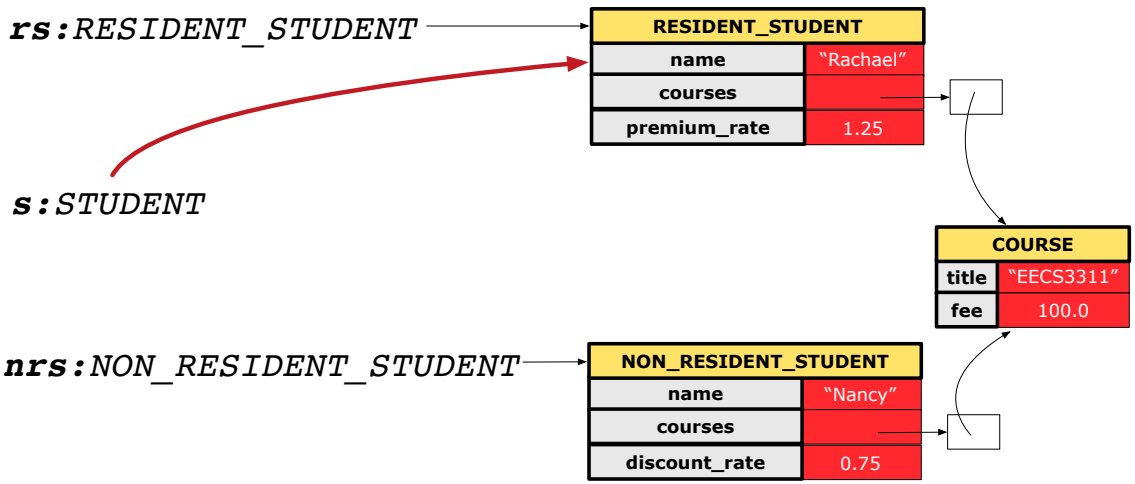
OK ∵ **s**`.premium_rate` is just *never used*!!

Dynamic Binding: Intuition (1)

```
1  local c : COURSE ; s : STUDENT
2  do crate c.make ("EECS3311", 100.0)
3     create {RESIDENT_STUDENT} rs.make("Rachael")
4     create {NON_RESIDENT_STUDENT} nrs.make("Nancy")
5     rs.set_pr(1.25); rs.register(c)
6     nrs.set_dr(0.75); nrs.register(c)
7     s := rs; ; check s.tuition = 125.0 end
8     s := nrs; ; check s.tuition = 75.0 end
```

After `s := rs` (**L7**), `s` points to a `RESIDENT_STUDENT` object.

⇒ Calling `s.tuition` applies the `premium_rate`.

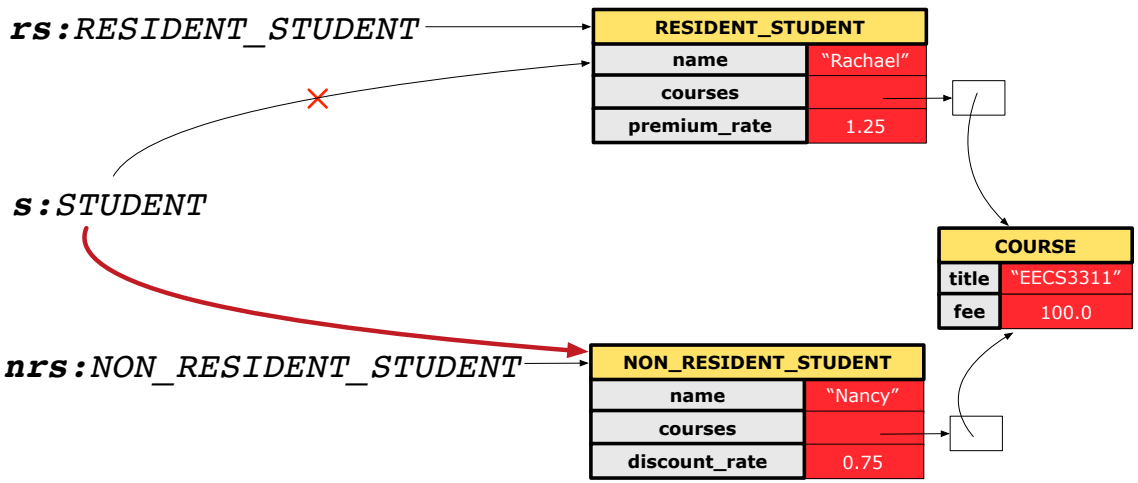


Dynamic Binding: Intuition (2)

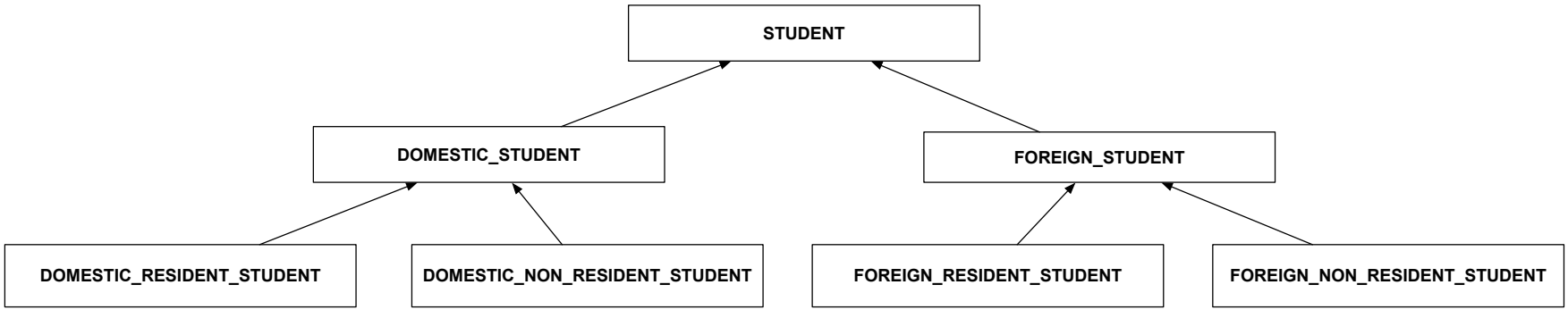
```
1  local c : COURSE ; s : STUDENT
2  do crate c.make ("EECS3311", 100.0)
3     create {RESIDENT_STUDENT} rs.make("Rachael")
4     create {NON_RESIDENT_STUDENT} nrs.make("Nancy")
5     rs.set_pr(1.25); rs.register(c)
6     nrs.set_dr(0.75); nrs.register(c)
7     s := rs; ; check s.tuition = 125.0 end
8     s := nrs; ; check s.tuition = 75.0 end
```

After `s:=nrs` (**L8**), `s` points to a `NON_RESIDENT_STUDENT` object.

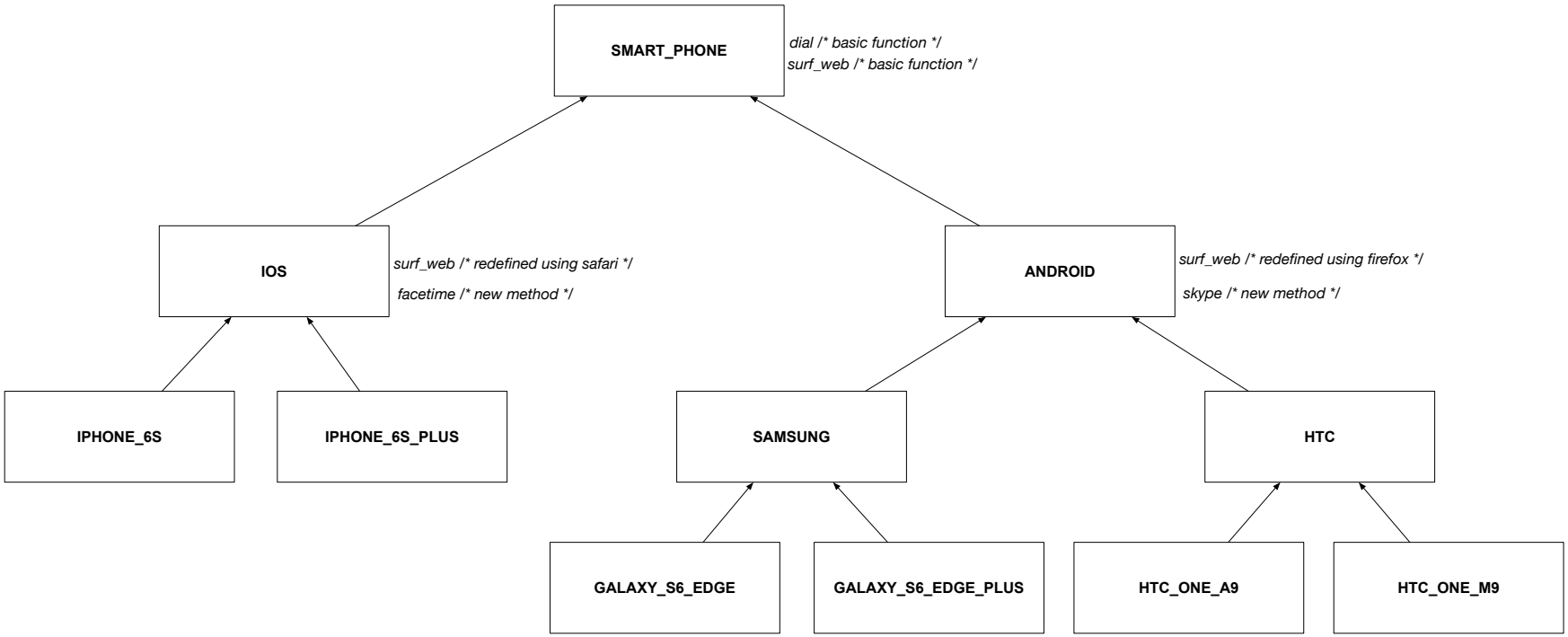
⇒ Calling `s.tuition` applies the `discount_rate`.



Multi-Level Inheritance Architecture (1)



Multi-Level Inheritance Architecture (2)



Inheritance Forms a Type Hierarchy

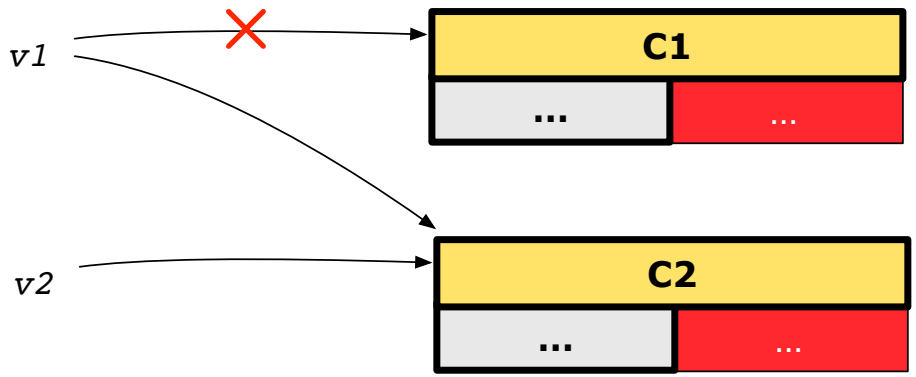
- A (data) *type* denotes a set of related *runtime values*.
  - Every *class* can be used as a type: the set of runtime *objects*.
- Use of *inheritance* creates a *hierarchy* of classes:
  - (Implicit) Root of the hierarchy is `ANY`.
  - Each `inherit` declaration corresponds to an upward arrow.
  - The `inherit` relationship is *transitive*: when `A` inherits `B` and `B` inherits C, we say `A` *indirectly* inherits `C`.
    e.g., Every class implicitly `inherits` the `ANY` class.
- *Ancestor* vs. *Descendant* classes:
  - The *ancestor classes* of a class `A` are: `A` itself and all classes that `A` directly, or indirectly, inherits.
    - `A` inherits all features from its *ancestor classes*.
      ∴ `A`'s instances have a ***wider range of expected usages*** (i.e., attributes, queries, commands) than instances of its *ancestor* classes.
  - The *descendant classes* of a class `A` are: `A` itself and all classes that directly, or indirectly, inherits `A`.
    - Code defined in `A` is inherited to all its *descendant classes*.

Inheritance Accumulates Code for Reuse

- The *lower* a class is in the type hierarchy, the *more code* it accumulates from its *ancestor classes*:
  - A *descendant class* inherits all code from its *ancestor classes*.
  - A *descendant class* may also:
    - Declare new attributes.
    - Define new queries or commands.
    - *Redefine* inherited queries or commands.
- Consequently:
  - When being used as *context objects*, instances of a class' *descendant classes* have a ***wider range of expected usages*** (i.e., attributes, commands, queries).
  - When expecting an object of a particular class, we may *substitute* it with an object of any of its *descendant classes*.
  - e.g., When expecting a `STUDENT` object, substitute it with either a `RESIDENT_STUDENT` or a `NON_RESIDENT_STUDENT` object.
  - **Justification**: A *descendant class* contains *at least as many* features as defined in its *ancestor classes* (but ***not vice versa***!).

Substitutions via Assignments

- By declaring `v1:C1`, *reference variable* `v1` will store the *address* of an object of class `C1` at runtime.
- By declaring `v2:C2`, *reference variable* `v2` will store the *address* of an object of class `C2` at runtime.
- Assignment `v1:=v2` *copies the address* stored in `v2` into `v1`.
  - `v1` will instead point to wherever `v2` is pointing to. [*object alias*]



- In such assignment `v1:=v2`, we say that we *substitute* an object of type `C1` with an object of type `C2`.
- *Substitutions* are subject to *rules*!

Rules of Substitution

Given an inheritance hierarchy:

1. When expecting an object of class `A`, it is *safe* to *substitute* it with an object of any *descendant class* of `A` (including `A`).
   - e.g., When expecting an `IOS` phone, you *can* substitute it with either an `IPhone6s` or `IPhone6sPlus`.
   - ∵ Each *descendant class* of `A` is guaranteed to contain all code of (non-private) attributes, commands, and queries defined in `A`.
   - ∴ All features defined in `A` are *guaranteed to be available* in the new substitute.
2. When expecting an object of class `A`, it is *unsafe* to *substitute* it with an object of any *ancestor class of `A`'s parent*.
   - e.g., When expecting an `IOS` phone, you *cannot* substitute it with just a `SmartPhone`, because the `facetime` feature is not supported in an `Android` phone.
   - ∵ Class `A` may have defined new features that do not exist in any of its *parent's ancestor classes*.

Reference Variable: Static Type

- A *reference variable*'s *static type* is what we declare it to be.
  - e.g., `jim:STUDENT` declares `jim`'s static type as `STUDENT`.
  - e.g., `my_phone:SMART_PHONE` declares a variable `my_phone` of static type `SmartPhone`.
  - The *static type* of a *reference variable never changes*.
- For a *reference variable* *v*, its *static type* *C* defines the *expected usages of v as a context object*.
- A feature call `v.m(...)` is ***compilable*** if ***m*** is defined in *C*.
  - e.g., After declaring `jim:STUDENT`, we
    - **may** call `register` and `tuition` on `jim`
    - **may *not*** call `set_pr` (specific to a resident student) or `set_dr` (specific to a non-resident student) on `jim`
  - e.g., After declaring `my_phone:SMART_PHONE`, we
    - **may** call `dial` and `surf_web` on `my_phone`
    - **may *not*** call `facetime` (specific to an IOS phone) or `skype` (specific to an Android phone) on `my_phone`

Reference Variable: Dynamic Type

A *reference variable*'s *dynamic type* is the type of object that it is currently pointing to at runtime.

- The *dynamic type* of a reference variable *may change* whenever we *re-assign* that variable to a different object.
- There are two ways to re-assigning a reference variable.

Reference Variable: Changing Dynamic Type (1)

Re-assigning a reference variable to a newly-created object:

- *Substitution Principle*: the new object's class must be a *descendant class* of the reference variable's *static type*.
- e.g., Given the declaration `jim: STUDENT`:
  - `create {RESIDENT_STUDENT} jim.make("Jim")` changes the *dynamic type* of `jim` to `RESIDENT_STUDENT`.
  - `create {NON_RESIDENT_STUDENT} jim.make("Jim")` changes the *dynamic type* of `jim` to `NON_RESIDENT_STUDENT`.
- e.g., Given an alternative declaration `jim: RESIDENT_STUDENT`:
  - e.g., `create {STUDENT} jim.make("Jim")` is illegal because `STUDENT` is not a *descendant class* of the *static type* of `jim` (i.e., `RESIDENT_STUDENT`).

Reference Variable: Changing Dynamic Type (2)

Re-assigning a reference variable `v` to an existing object that is referenced by another variable `other` (i.e., `v := other`):

- *Substitution Principle*: the static type of `other` must be a *descendant class* of `v`'s *static type*.
- e.g.,

```
jim: STUDENT ; rs: RESIDENT_STUDENT; nrs: NON_RESIDENT_STUDENT
create {STUDENT} jim.make (...)
create {RESIDENT_STUDENT} rs.make (...)
create {NON_RESIDENT_STUDENT} nrs.make (...)
```

- `rs := jim` ×
- `nrs := jim` ×
- `jim := rs` ✓
  changes the *dynamic type* of `jim` to the dynamic type of `rs`
- `jim := nrs` ✓
  changes the *dynamic type* of `jim` to the dynamic type of `nrs`

Polymorphism and Dynamic Binding (1)

- *Polymorphism*: An object variable may have *"multiple possible shapes"* (i.e., allowable *dynamic types*).
  - Consequently, there are *multiple possible versions* of each feature that may be called.
    - e.g., 3 possibilities of `tuition` on a ***STUDENT*** reference variable:
      In ***STUDENT***: base amount
      In ***RESIDENT_STUDENT***: base amount with `premium_rate`
      In ***NON_RESIDENT_STUDENT***: base amount with `discount_rate`
- *Dynamic binding*: When a feature `m` is called on an object variable, the version of `m` corresponding to its *"current shape"* (i.e., one defined in the *dynamic type* of *m*) will be called.

```
jim: STUDENT; rs: RESIDENT_STUDENT; nrs: NON_STUDENT
create {RESIDENT_STUDENT} rs.make (...)
create {NON_RESIDENT_STUDENT} nrs.nrs (...)
jim := rs
jim.tuitoion;  /* version in RESIDENT_STUDENT */
jim := nrs
jim.tuition;  /* version in NON_RESIDENT_STUDENT */
```

Polymorphism and Dynamic Binding (2.1)

```
1  test_polymorphism_students
2   local
3     jim: STUDENT
4     rs: RESIDENT_STUDENT
5     nrs: NON_RESIDENT_STUDENT
6   do
7     create {STUDENT} jim.make ("J. Davis")
8     create {RESIDENT_STUDENT} rs.make ("J. Davis")
9     create {NON_RESIDENT_STUDENT} nrs.make ("J. Davis")
10    jim := rs   ✓
11    rs := jim   ×
12    jim := nrs   ✓
13    rs := jim   ×
14  end
```

In (**L3**, **L7**), (**L4**, **L8**), (**L5**, **L9**), ***ST*** = ***DT***, so we may abbreviate:

L7:
```
create jim.make ("J. Davis")
```

L8:
```
create rs.make ("J. Davis")
```

L9:
```
create nrs.make ("J. Davis")
```

Polymorphism and Dynamic Binding (2.2)

```
test_dynamic_binding_students: BOOLEAN
  local
    jim: STUDENT
    rs: RESIDENT_STUDENT
    nrs: NON_RESIDENT_STUDENT
    c: COURSE
  do
    create c.make ("EECS3311", 500.0)
    create {STUDENT} jim.make ("J. Davis")
    create {RESIDENT_STUDENT} rs.make ("J. Davis")
    rs.register (c)
    rs.set_pr (1.5)
    jim := rs
    Result := jim.tuition = 750.0
    check Result end
    create {NON_RESIDENT_STUDENT} nrs.make ("J. Davis")
    nrs.register (c)
    nrs.set_dr (0.5)
    jim := nrs
    Result := jim.tuition = 250.0
end
```

Reference Type Casting: Motivation

```
1  local jim: STUDENT; rs: RESIDENT_STUDENT
2  do create {RESIDENT_STUDENT} jim.make ("J. Davis")
3     rs := jim
4     rs.setPremiumRate(1.5)
```

- **Line 2** is *legal*: `RESIDENT_STUDENT` is a *descendant class* of the static type of `jim` (i.e., `STUDENT`).
- **Line 3** is *illegal*: `jim`'s static type (i.e., `STUDENT`) is ***not*** a *descendant class* of `rs`'s static type (i.e., `RESIDENT_STUDENT`).
- Eiffel compiler is *unable to infer* that `jim`'s *dynamic type* in **Line 4** is `RESIDENT_STUDENT`. [*Undecidable*]
- Force the Eiffel compiler to believe so, by replacing **L3**, **L4** by a *type cast* (which *temporarily* changes the ***ST*** of `jim`):

```
check attached {RESIDENT_STUDENT} jim as rs_jim then
  rs := rs_jim
end
rs.set_pr (1.5)
```

Reference Type Casting: Syntax

```
1  check attached {RESIDENT_STUDENT} jim as rs_jim then
2    rs := rs_jim
3  end
4  rs.set_pr (1.5)
```

L1 is an assertion:

- `attached RESIDENT_STUDENT jim` is a Boolean expression that is to be evaluated at ***runtime***.
  - If it evaluates to ***true***, then the `as rs_jim` expression has the effect of assigning "the cast version" of `jim` to a new variable `rs_jim`.
  - If it evaluates to ***false***, then a runtime assertion violation occurs.
- *Dynamic Binding*: **Line 4** executes the correct version of `set_pr`.

- It is equivalent to the following Java code:

```
if(jim instanceof ResidentStudent) {
  ResidentStudent rs_jim = (ResidentStudent) jim; }
else { throw new Exception("Illegal Cast"); }
rs.set_pr (1.5)
```

Notes on Type Cast (1)

- Given *v* of static type *ST*, it is ***compilable*** to cast *v* to *C*, as long as *C* is a descendant or ancestor class of *ST*.
- Why Cast?
  - Without cast, we can **only** call features defined in *ST* on *v*.
  - By casting *v* to *C*, we *change* the *static type* of *v* from *ST* to *C*.
    ⇒ All features that are defined in *C* can be called.

```
my_phone: IOS
create {IPHONE_6S_PLUS} my_phone.make
-- can only call features defined in IOS on myPhone
-- dial, surf_web, facetime ✓  three_d_touch, skype ×
check attached {SMART_PHONE} my_phone as sp then
  -- can now call features defined in SMART_PHONE on sp
  -- dial, surf_web ✓  facetime, three_d_touch, skype ×
end
check attached {IPHONE_6S_PLUS} my_phone as ip6s_plus then
  -- can now call features defined in IPHONE_6S_PLUS on ip6s_plus
  -- dial, surf_web, facetime, three_d_touch ✓  skype ×
end
```

Notes on Type Cast (2)

- A cast being ***compilable*** is not necessarily ***runtime-error-free***!
- A cast `check attached {C} v as ...` triggers an assertion violation if `C` is ***not*** along the **ancestor path** of `v`'s *DT*.

```
test_smart_phone_type_cast_violation
  local mine: ANDROID
  do create {SAMSUNG} mine.make
    -- ST of mine is ANDROID; DT of mine is SAMSUNG
    check attached {SMART_PHONE} mine as sp then ... end
    -- ST of sp is SMART_PHONE; DT of sp is SAMSUNG
    check attached {SAMSUNG} mine as samsung then ... end
    -- ST of android is SAMSNG; DT of samsung is SAMSUNG
    check attached {HTC} mine as htc then ... end
    -- Compiles ∵ HTC is descendant of mine's ST (ANDROID)
    -- Assertion violation
    -- ∵ HTC is not ancestor of mine's DT (SAMSUNG)
    check attached {GALAXY_S6_EDGE} mine as galaxy then ... end
    -- Compiles ∵ GALAXY_S6_EDGE is descendant of mine's ST (ANDROID)
    -- Assertion violation
    -- ∵ GALAXY_S6_EDGE is not ancestor of mine's DT (SAMSUNG)
end
```

Why Inheritance: A Collection of Various Kinds of Students

How do you define a class `STUDENT_MANAGEMENT_SYSETM` that contains a list of *resident* and *non-resident* students?

```
class STUDENT_MANAGEMENT_SYSETM
  students: LINKED_LIST[STUDENT]
  add_student(s: STUDENT)
    do
      students.extend (s)
    end
  registerAll (c: COURSE)
    do
      across
        students as s
      loop
        s.item.register (c)
      end
    end
end
```

Polymorphism and Dynamic Binding: A Collection of Various Kinds of Students

```
test_sms_polymorphism: BOOLEAN
  local
    rs: RESIDENT_STUDENT
    nrs: NON_RESIDENT_STUDENT
    c: COURSE
    sms: STUDENT_MANAGEMENT_SYSTEM
  do
    create rs.make ("Jim")
    rs.set_pr (1.5)
    create nrs.make ("Jeremy")
    nrs.set_dr (0.5)
    create sms.make
    sms.add_s (rs)
    sms.add_s (nrs)
    create c.make ("EECS3311", 500)
    sms.register_all (c)
    Result := sms.ss[1].tuition = 750 and sms.ss[2].tuition = 250
  end
```

Polymorphism: Feature Call Arguments (1)

```
1  class STUDENT_MANAGEMENT_SYSTEM {
2    ss : ARRAY[STUDENT] -- ss[i] has static type Student
3    add_s (s: STUDENT) do ss[0] := s end
4    add_rs (rs: RESIDENT_STUDENT) do ss[0] := rs end
5    add_nrs (nrs: NON_RESIDENT_STUDENT) do ss[0] := nrs end
```

- **L4**: `ss[0]:=rs` is valid. ∵ RHS's ST ***RESIDENT_STUDENT*** is a *descendant class* of LHS's ST ***STUDENT***.
- Say we have a `STUDENT_MANAGEMENT_SYSETM` object `sms`:
  - ∵ *call by reference*, `sms.add_rs(o)` attempts the following assignment (i.e., replace parameter `rs` by a copy of argument `o`):

    ```
    rs := o
    ```

  - Whether this argument passing is valid depends on `o`'s *static type*.

Rule: In the signature of a feature `m`, if the type of a parameter is class `C`, then we may call feature `m` by passing objects whose *static types* are `C`'s *descendants*.

Polymorphism: Feature Call Arguments (2)

```
test_polymorphism_feature_arguments
  local
    s1, s2, s3: STUDENT
    rs: RESIDENT_STUDENT ; nrs: NON_RESIDENT_STUDENT
    sms: STUDENT_MANAGEMENT_SYSTEM
  do
    create sms.make
    create {STUDENT} s1.make ("s1")
    create {RESIDENT_STUDENT} s2.make ("s2")
    create {NON_RESIDENT_STUDENT} s3.make ("s3")
    create {RESIDENT_STUDENT} rs.make ("rs")
    create {NON_RESIDENT_STUDENT} nrs.make ("nrs")
    sms.add_s (s1) ✓ sms.add_s (s2) ✓ sms.add_s (s3) ✓
    sms.add_s (rs) ✓ sms.add_s (nrs) ✓
    sms.add_rs (s1) × sms.add_rs (s2) × sms.add_rs (s3) ×
    sms.add_rs (rs) ✓ sms.add_rs (nrs) ×
    sms.add_nrs (s1) × sms.add_nrs (s2) × sms.add_nrs (s3) ×
    sms.add_nrs (rs) × sms.add_nrs (nrs) ✓
  end
```

Polymorphism: Return Values (1)

```
1  class STUDENT_MANAGEMENT_SYSTEM {
2    ss: LINKED_LIST[STUDENT]
3    add_s (s: STUDENT)
4      do
5        ss.extend (s)
6      end
7    get_student(i: INTEGER): STUDENT
8      require 1 <= i and i <= ss.count
9      do
10       Result := ss[i]
11     end
12 end
```

- **L2**: ***ST*** of each stored item (`ss[i]`) in the list: [STUDENT]
- **L3**: ***ST*** of input parameter `s`: [STUDENT]
- **L7**: ***ST*** of return value (`Result`) of `get_student`: [STUDENT]
- **L11**: `ss[i]`'s ***ST*** is *descendant* of `Result`' ***ST***.
  Question: What can be the ***dynamic type*** of `s` after **Line 11**?
  Answer: All descendant classes of `Student`.

Polymorphism: Return Values (2)

```
1  test_sms_polymorphism: BOOLEAN
2  local
3    rs: RESIDENT_STUDENT ; nrs: NON_RESIDENT_STUDENT
4    c: COURSE ; sms: STUDENT_MANAGEMENT_SYSTEM
5  do
6    create rs.make ("Jim") ; rs.set_pr (1.5)
7    create nrs.make ("Jeremy") ; nrs.set_dr (0.5)
8    create sms.make ; sms.add_s (rs) ; sms.add_s (nrs)
9    create c.make ("EECS3311", 500) ; sms.register_all (c)
10   Result :=
11         get_student(1).tuition = 750
12     and get_student(2).tuition = 250
13 end
```

- **L11**: `get_student(1)`'s dynamic type? [*RESIDENT_STUDENT*]
- **L11**: Version of `tuition`? [*RESIDENT_STUDENT*]
- **L12**: `get_student(2)`'s dynamic type? [*NON_RESIDENT_STUDENT*]
- **L12**: Version of `tuition`? [*NON_RESIDENT_STUDENT*]

Design Principle: Polymorphism

- When declaring an attribute `a: T`
  ⇒ Choose ***static type*** `T` which "accumulates" all features that you predict you will want to call on `a`.
  e.g., Choose `s: STUDENT` if you do not intend to be specific about which kind of student `s` might be.
  ⇒ Let ***dynamic binding*** determine at runtime which version of `tuition` will be called.
- What if after declaring `s: STUDENT` you find yourself often needing to *cast* `s` to `RESIDENT_STUDENT` in order to access `premium_rate`?

```
check attached {RESIDENT_STUDENT} s as rs then rs.set_pr(...) end
```

  ⇒ Your design decision should have been: `s:RESIDENT_STUDENT`
- Same design principle applies to:
  - Type of feature parameters: `f(a: T)`
  - Type of queries: `q(...): T`

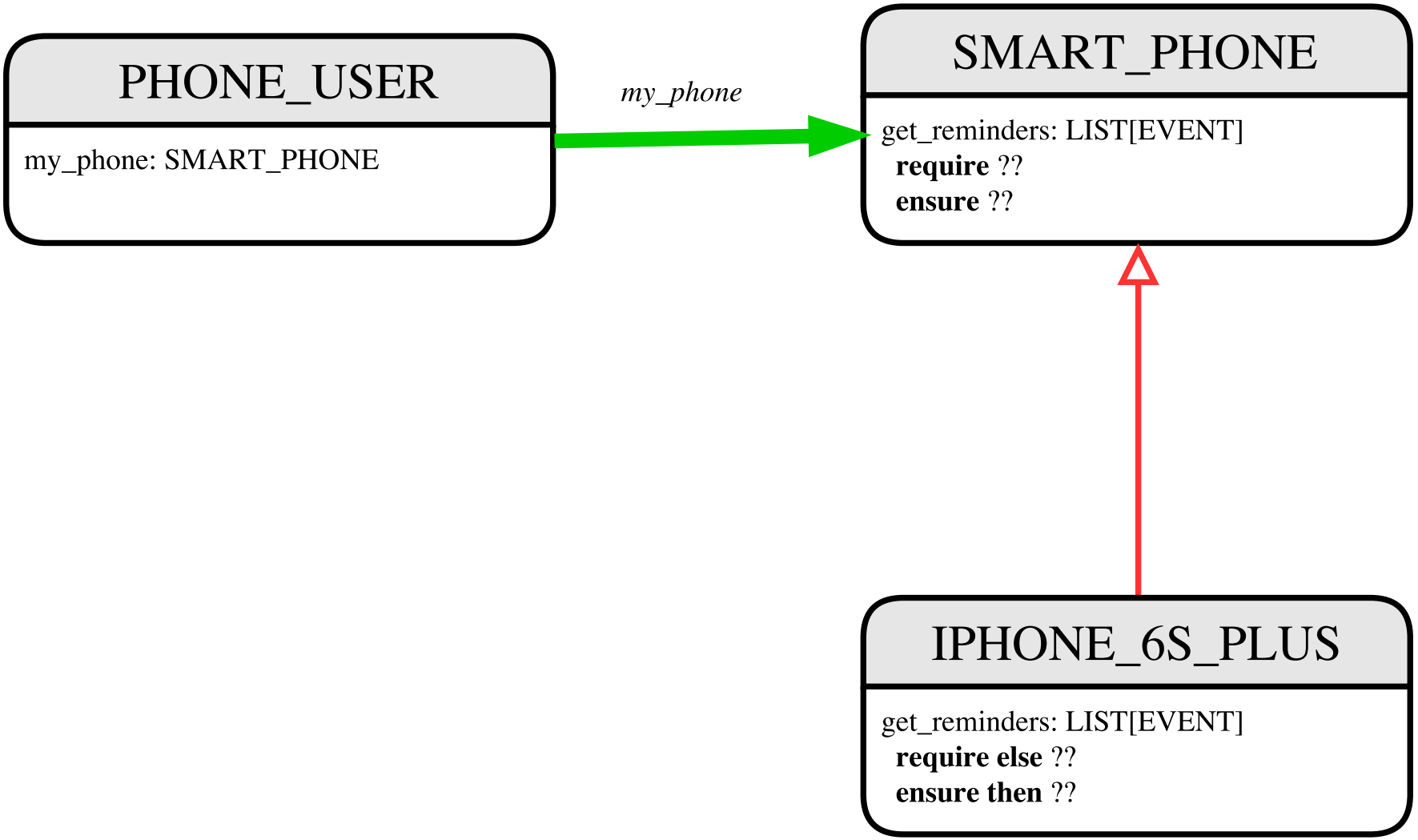
Inheritance and Contracts (1)

- The fact that we allow *polymorphism*:

```
local my_phone: SMART_PHONE
      i_phone: IPHONE_6S_PLUS
      samsung_phone: GALAXY_S6_EDGE
      htc_phone: HTC_ONE_A9
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.

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_6S_PLUS
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_6S_PLUS* 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_6S_PLUS*.

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_6S_PLUS
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_6S_PLUS* 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_6S_PLUS*.

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_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 or tomorrow
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.

e.g., A client able to get reminders on a *SMART_PHONE*, when batter level is 12%, will fail to do so on an *IPHONE_6S_PLUS*.

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_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 or tomorrow
end
```

Contracts in descendant class *IPHONE_6S_PLUS* 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_6S_PLUS*.

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*** ∨ *new_pre* ≡ ***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*** ∧ *new_post* ≡ ***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)

```
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`

$\because$ *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`

$\because$ *original_post* $\wedge$ ***true*** $\equiv$ *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:

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

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

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

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*:

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

(Dynamic) *Runtime*:

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

Index (1)

Index (2)

Index (3)

Index (4)

Index (5)

Generics

EECS3311: Software Design
Fall 2017

CHEN-WEI WANG

Motivating Example: A Book of Any Objects

```
class BOOK
  names: ARRAY[STRING]
  records: ARRAY[ANY]
  -- Create an empty book
  make do ... end
  -- Add a name-record pair to the book
  add (name: STRING; record: ANY) do ... end
  -- Return the record associated with a given name
  get (name: STRING): ANY do ... end
end
```

Question: Which line has a type error?

```
1  birthday: DATE; phone_number: STRING
2  b: BOOK; is_wednesday: BOOLEAN
3  create {BOOK} b.make
4  phone_number := "416-677-1010"
5  b.add ("SuYeon", phone_number)
6  create {DATE} birthday.make(1975, 4, 10)
7  b.add ("Yuna", birthday)
8  is_wednesday := b.get("Yuna").get_day_of_week = 4
```

Motivating Example: Observations (1)

- In the `BOOK` class:
  - In the attribute declaration

```
records: ARRAY[ANY]
```

    - ***ANY*** is the most general type of records.
    - Each book instance may store any object whose *static type* is a *descendant class* of ***ANY***.
  - Accordingly, from the return type of the `get` feature, we only know that the returned record has the static type ***ANY***, but not certain about its *dynamic type* (e.g., `DATE`, `STRING`, *etc.*).
    ∴ a record retrieved from the book, e.g., `b.get("Yuna")`, may only be called upon features defined in its *static type* (i.e,. ***ANY***).
- In the tester code of the `BOOK` class:
  - In **Line 1**, the *static types* of variables `birthday` (i.e., `DATE`) and `phone_number` (i.e., `STRING`) are *descendant classes* of `ANY`.
    ∴ **Line 5** and **Line 7** compile.

Motivating Example: Observations (2)

Due to *polymorphism*, in a collection, the *dynamic types* of stored objects (e.g., `phone_number` and `birthday`) need not be the same.

- Features specific to the *dynamic types* (e.g., `get_day_of_week` of class `Date`) may be new features that are not inherited from `ANY`.
- This is why **Line 8** would fail to compile, and may be fixed using an explicit *cast*:

```
check attached {DATE} b.get("Yuna") as yuna_bday then
  is_wednesday := yuna_bday.get_day_of_week = 4
end
```

- But what if the *dynamic type* of the returned object is not a `DATE`?

```
check attached {DATE} b.get("SuYeon") as suyeon_bday then
  is_wednesday := suyeon_bday.get_day_of_week = 4
end
```

⇒ An *assertion violation* at *runtime*!

Motivating Example: Observations (2.1)

- It seems that a combination of `attached` check (similar to an `instanceof` check in Java) and type cast can work.
- Can you see any potential problem(s)?
- **Hints:**
  - Extensibility and Maintainability
  - What happens when you have a large number of records of distinct *dynamic types* stored in the book
    (e.g., `DATE`, `STRING`, `PERSON`, `ACCOUNT`, `ARRAY_CONTAINER`, `DICTIONARY`, *etc.*)? [all classes are descendants of ***ANY***]

Motivating Example: Observations (2.2)

Imagine that the tester code (or an application) stores 100 different record objects into the book.

```
rec1: C1
... -- declarations of rec2 to rec99
rec100: C100
create {C1} rec1.make(...) ; b.add(..., rec1)
... -- additions of rec2 to rec99
create {C100} rec100.make(...) ; b.add(..., rec100)
```

where ***static types*** `C1` to `C100` are *descendant classes* of `ANY`.

- *Every time* you retrieve a record from the book, you need to check "exhaustively" on its *dynamic type* before calling some feature(s).

```
-- assumption: 'f1' specific to C1, 'f2' specific to C2, etc.
check attached {C1} b.get("Jim") as c1 then c1.f1 end
... -- casts for C2 to C99
check attached {C100} b.get("Jim") as c100 then c100.f100 end
```

- Writing out this list multiple times is tedious and error-prone!

Motivating Example: Observations (3)

We need a solution that:

- Eliminates runtime assertion violations due to wrong casts
- Saves us from explicit `attached` checks and type casts

As a sketch, this is how the solution looks like:

- When the user declares a `BOOK` object `b`, they must commit to the kind of record that `b` stores at runtime.
  e.g., `b` stores either `DATE` objects (and its *descendants*) only or `String` objects (and its *descendants*) only, but *not a mix*.
- When attempting to store a new record object `rec` into `b`, if `rec`'s *static type* is not a *descendant class* of the type of book that the user previously commits to, then:
  - It is considered as a ***compilation error***
  - Rather than triggering a ***runtime assertion violation***
- When attempting to retrieve a record object from `b`, there is *no longer a need* to check and cast.
  ∵ *Static types* of all records in `b` are guaranteed to be the same.

Parameters

- In mathematics:
  - The same *function* is applied with different *argument values*.
    e.g., `2 + 3`, `1 + 1`, `10 + 101`, *etc.*
  - We *generalize* these instance applications into a definition.
    e.g., $+ : (\mathbb{Z} \times \mathbb{Z}) \to \mathbb{Z}$ is a function that takes two integer *parameters* and returns an integer.
- In object-oriented programming:
  - We want to call a *feature*, with different *argument values*, to achieve a similar goal.
    e.g., `acc.deposit(100)`, `acc.deposit(23)`, *etc.*
  - We *generalize* these possible feature calls into a definition.
    e.g., In class `ACCOUNT`, a feature `deposit(amount: REAL)` takes a real-valued *parameter*.
- When you design a mathematical function or a class feature, always consider the list of *parameters*, each of which representing a set of possible *argument values*.

Generics: Design of a Generic Book

```
class BOOK[ G ]
  names: ARRAY[STRING]
  records: ARRAY[ G ]
  -- Create an empty book
  make do ... end
  /* Add a name-record pair to the book */
  add (name: STRING; record: G ) do ... end
  /* Return the record associated with a given name */
  get (name: STRING): G do ... end
end
```

Question: Which line has a type error?

```
1  birthday: DATE; phone_number: STRING
2  b: BOOK[DATE]; is_wednesday: BOOLEAN
3  create BOOK[DATE] b.make
4  phone_number = "416-67-1010"
5  b.add ("SuYeon", phone_number)
6  create {DATE} birthday.make (1975, 4, 10)
7  b.add ("Yuna", birthday)
8  is_wednesday := b.get("Yuna").get_day_of_week == 4
```

Generics: Observations

- In class BOOK:
  - At the class level, we *parameterize the type of records*:

```
class BOOK[G]
```

  - Every occurrence of ANY is replaced by E.
- As far as a client of BOOK is concerned, they must *instantiate* G.
  ⇒ This particular instance of book must consistently store items of that instantiating type.
- As soon as E instantiated to some known type (e.g., DATE, STRING), every occurrence of E will be replaced by that type.
- For example, in the tester code of BOOK:
  - In **Line 2**, we commit that the book b will store DATE objects only.
  - **Line 5** fails to compile. [∵ STRING not **descendant** of DATE]
  - **Line 7** still compiles. [∵ DATE is **descendant** of itself]
  - **Line 8** does *not need* any attached check and type cast, and does *not cause* any runtime assertion violation.
    ∵ All attempts to store non-DATE objects are caught at ***compile time***.

Bad Example of using Generics

Has the following client made an appropriate choice?

```
book: BOOK[ANY]
```

NO!!!!!!!!!!!!!!!!!!!!!!

- It allows **all** kinds of objects to be stored.
  ∴ All classes are descendants of ***ANY***.
- We can expect **very little** from an object retrieved from this book.
  ∴ The ***static type*** of `book`'s items are ***ANY***, root of the class hierarchy, has the *minimum* amount of features available for use.
  ∴ Exhaustive list of casts are unavoidable.

[***bad*** for extensibility and maintainability]

Instantiating Generic Parameters

- Say the **supplier** provides a generic `DICTIONARY` class:

```
class DICTIONARY[V, K] -- V type of values; K type of keys
  add_entry (v: V; k: K) do ... end
  remove_entry (k: K) do ... end
end
```

- **Clients** use `DICTIONARY` with different degrees of instantiations:

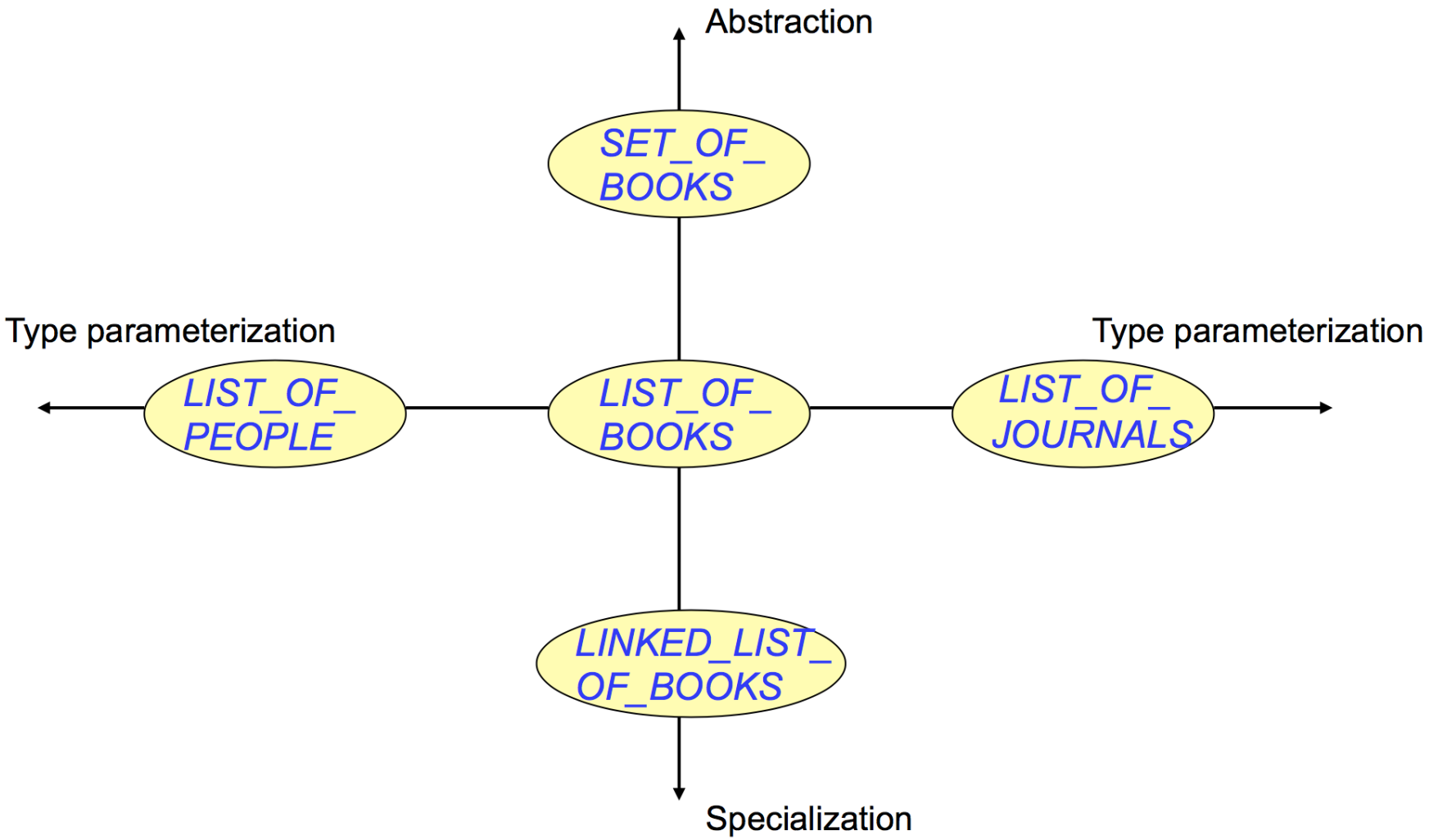
```
class DATABASE_TABLE[K, V]
  imp: DICTIONARY[V, K]
end
```

e.g., Declaring `DATABSE_TABLE[INTEGER, STRING]` instantiates `DICTIONARY[STRING, INTEGER]`.

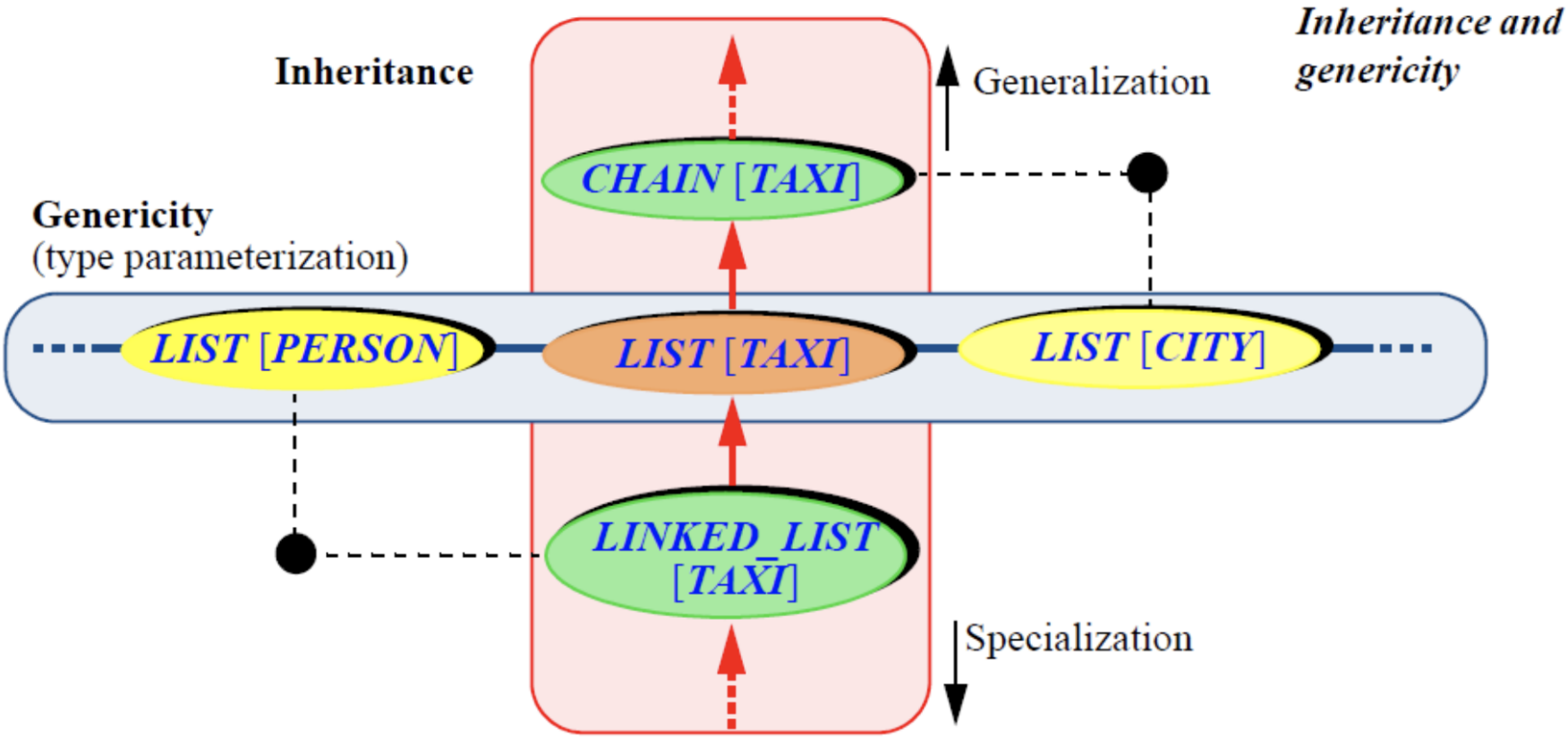
```
class STUDENT_BOOK[V]
  imp: DICTIONARY[V, STRING]
end
```

e.g., Declaring `STUDENT_BOOK[ARRAY[COURSE]]` instantiates `DICTIONARY[ARRAY[COURSE], STRING]`.

Generics vs. Inheritance (1)



Generics vs. Inheritance (2)



Beyond this lecture . . .

- Study the “Generic Parameters and the Iterator Pattern” Tutorial Videos.

Index (1)

The State Design Pattern

Readings: OOSC2 Chapter 20

EECS3311: Software Design
Fall 2017

CHEN-WEI WANG

Motivating Problem

Consider the reservation panel of an online booking system:

-- Enquiry on Flights --

Flight sought from: Toronto To: Zurich

Departure on or after: 23 June On or before: 24 June

Preferred airline (s):

Special requirements:

AVAILABLE FLIGHTS: 1

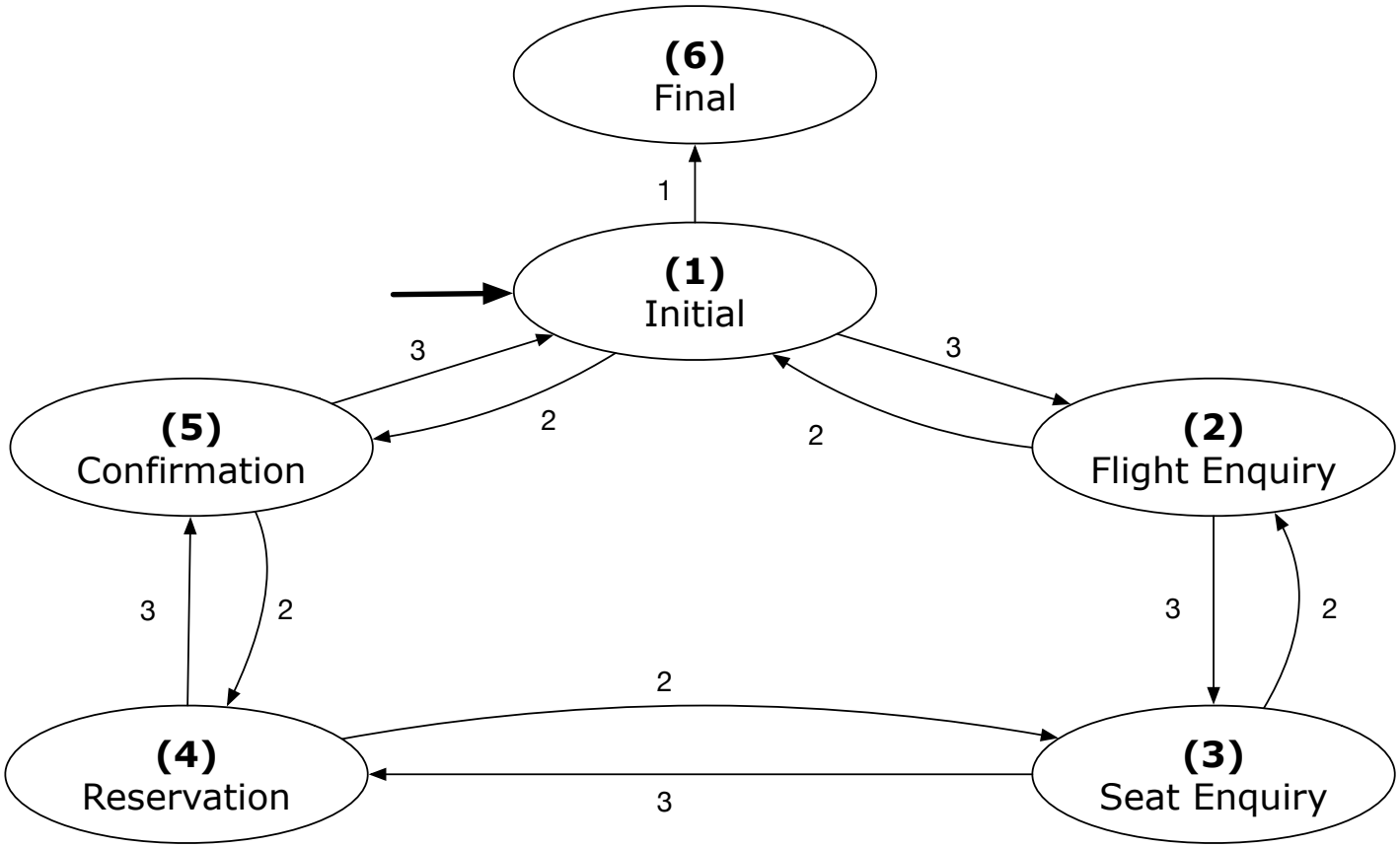
Flt#AA 42 Dep 8:25 Arr 7:45 Thru: Chicago

Choose next action:

0 – Exit
1 – Help
2 – Further enquiry
3 – Reserve a seat

State Transition Diagram

Characterize *interactive system* as: **1)** A set of *states*; and **2)** For each state, its list of *applicable transitions* (i.e., actions).
e.g., Above reservation system as a *finite state machine*:



Design Challenges

1. The state-transition graph may *large* and *sophisticated*.
   A large number N of states and number of transitions $\approx N^2$
2. The graph structure is subject to *extensions*/*modifications*.
   e.g., To merge "(2) Flight Enquiry" and "(3) Seat Enquiry":
   Delete the state "(3) Seat Enquiry".
   Delete its 4 incoming/outgoing transitions.
   e.g., Add a new state "Dietary Requirements"
3. A *general solution* is needed for such *interactive systems*.
   e.g., taobao, eBay, amazon, etc.

A First Attempt

```
1_Initial_panel:
  -- Actions for Label 1.
2_Flight_Enquiry_panel:
  -- Actions for Label 2.
3_Seat_Enquiry_panel:
  -- Actions for Label 3.
4_Reservation_panel:
  -- Actions for Label 4.
5_Confirmation_panel:
  -- Actions for Label 5.
6_Final_panel:
  -- Actions for Label 6.
```

```
3_Seat_Enquiry_panel:
  from
    Display Seat Enquiry Panel
  until
    not (wrong answer or wrong choice)
  do
    Read user's answer for current panel
    Read user's choice C for next step
    if wrong answer or wrong choice then
      Output error messages
    end
  end
  Process user's answer
  case C in
    2: goto 2_Flight_Enquiry_panel
    3: goto 4_Reservation_panel
  end
```

A First Attempt: Good Design?

- Runtime execution ≈ a ***"bowl of spaghetti"***.

  ⇒ The system's behaviour is hard to predict, trace, and debug.
- *Transitions* hardwired as system's ***central control structure***.

  ⇒ The system is vulnerable to changes/additions of states/transitions.
- All labelled blocks are largely similar in their code structures.

  ⇒ This design "***smells***" due to duplicates/repetitions!
- The branching structure of the design exactly corresponds to that of the specific *transition graph*.

  ⇒ The design is ***application-specific*** and *not reusable* for other interactive systems.

A Top-Down, Hierarchical Solution

- *Separation of Concern* Declare *transition graph* as a feature the system, rather than its central control structure:

```
transition (src: INTEGER; choice: INTEGER): INTEGER
    -- Return state by taking transition 'choice' from 'src' state.
  require valid_source_state: 1 ≤ src ≤ 6
          valid_choice: 1 ≤ choice ≤ 3
  ensure valid_target_state: 1 ≤ Result ≤ 6
```

- We may implement `transition` via a 2-D array.

SRC STATE \ CHOICE	1	2	3
1 (Initial)	6	5	2
2 (Flight Enquiry)	–	1	3
3 (Seat Enquiry)	–	2	4
4 (Reservation)	–	3	5
5 (Confirmation)	–	4	1
6 (Final)	–	–	–

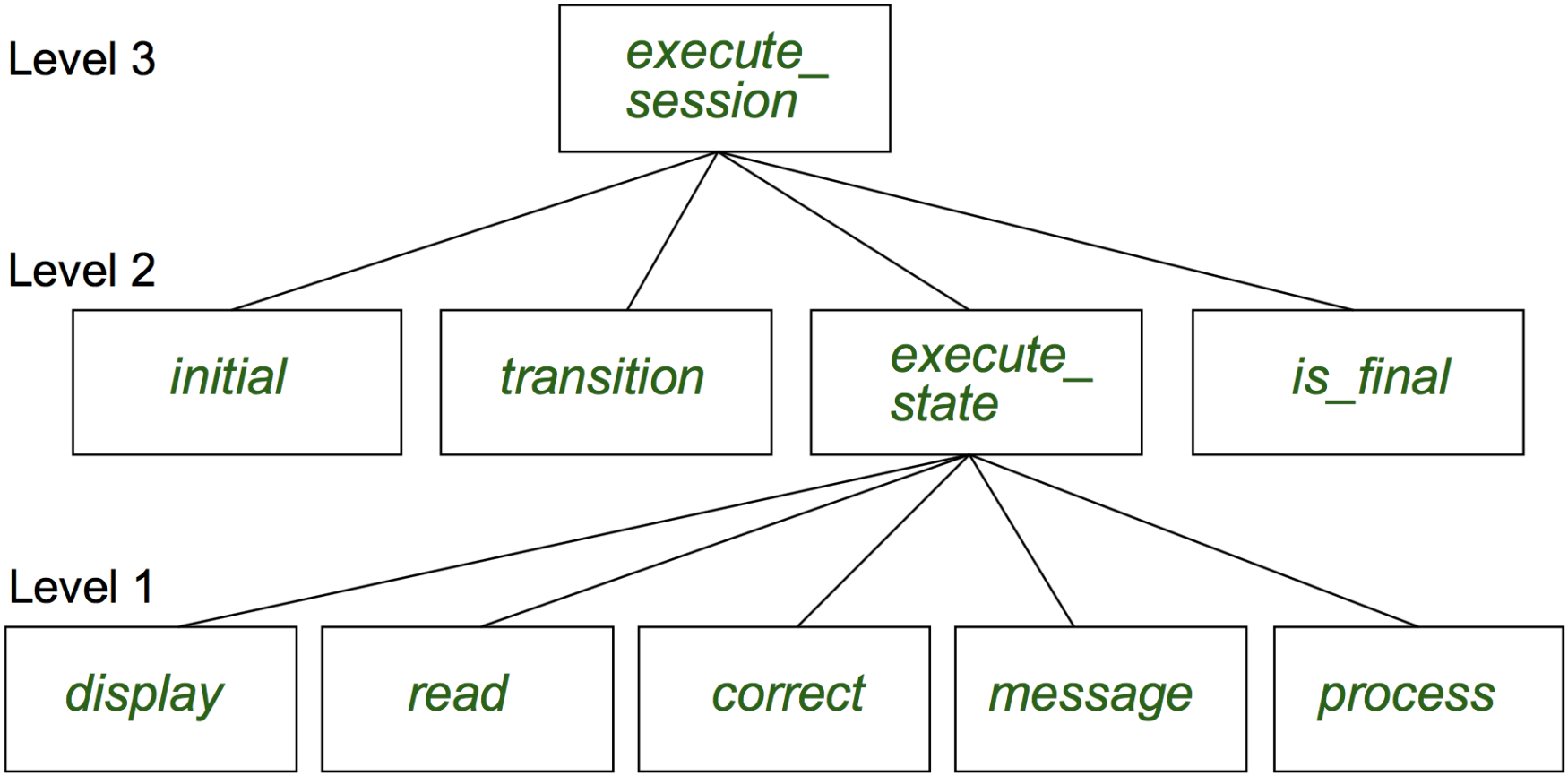
Hierarchical Solution: Good Design?

- This is a more general solution.

  ∵ *State transitions* are *separated* from the system's *central control structure*.

  ⇒ *Reusable* for another interactive system by making changes only to the `transition` feature.
- How does the *central control structure* look like in this design?

Hierarchical Solution: Top-Down Functional Decomposition



Modules of **execute_session** and **execute_state** are general enough on their *control structures*. ⇒ *reusable*

Hierarchical Solution: System Control

All interactive sessions **share** the following *control pattern*:

- Start with some *initial state*.
- Repeatedly make *state transitions* (based on *choices* read from the user) until the state is *final* (i.e., the user wants to exit).

```
execute_session
    -- Execute a full interactive session.
  local
    current_state, choice: INTEGER
  do
    from
      current_state := initial
    until
      is_final (current_state)
    do
      choice := execute_state (current_state)
      current_state := transition (current_state, choice)
    end
  end
```

Hierarchical Solution: State Handling (1)

The following *control pattern* handles **all** states:

```
execute_state ( current_state : INTEGER): INTEGER
    -- Handle interaction at the current state.
    -- Return user's exit choice.
  local
    answer: ANSWER; valid_answer: BOOLEAN; choice: INTEGER
  do
    from
    until
      valid_answer
    do
      display( current_state )
      answer := read_answer( current_state )
      choice := read_choice( current_state )
      valid_answer := correct( current_state , answer)
      if not valid_answer then message( current_state , answer)
    end
    process( current_state , answer)
    Result := choice
  end
```

Hierarchical Solution: State Handling (2)

FEATURE CALL	FUNCTIONALITY
`display`(**s**)	Display screen outputs associated with ***state*** *s*
`read_answer`(**s**)	Read user's input for answers associated with ***state*** *s*
`read_choice`(**s**)	Read user's input for exit choice associated with ***state*** *s*
`correct`(**s**, answer)	Is the user's *answer* valid w.r.t. ***state*** *s*?
`process`(**s**, answer)	Given that user's *answer* is valid w.r.t. ***state*** *s*, process it accordingly.
`message`(**s**, answer)	Given that user's *answer* is not valid w.r.t. ***state*** *s*, display an error message accordingly.

Q: How similar are the code structures of the above state-dependant commands or queries?

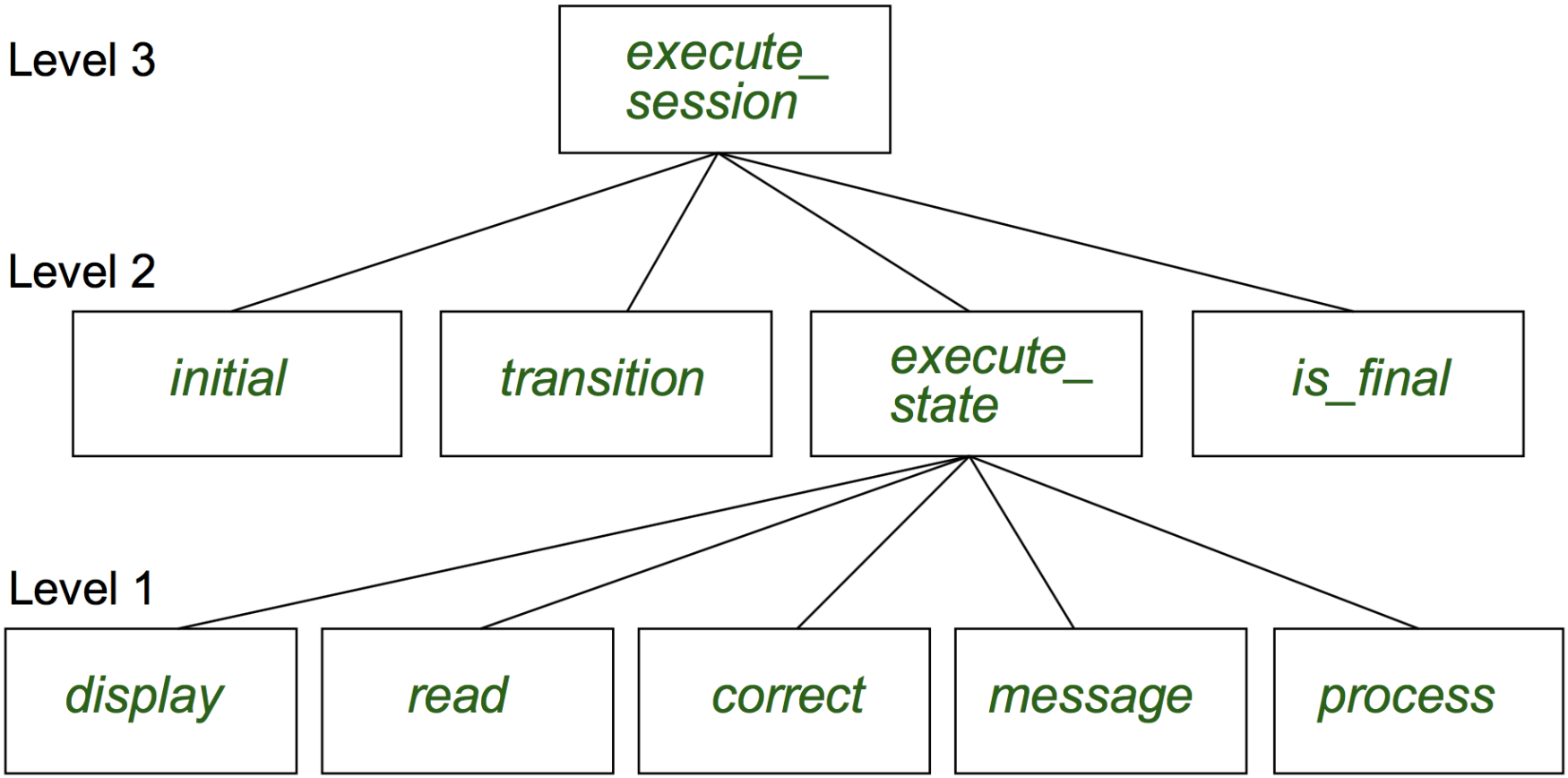
Hierarchical Solution: State Handling (3)

A: Actions of all such state-dependant features must **explicitly** *discriminate* on the input state argument.

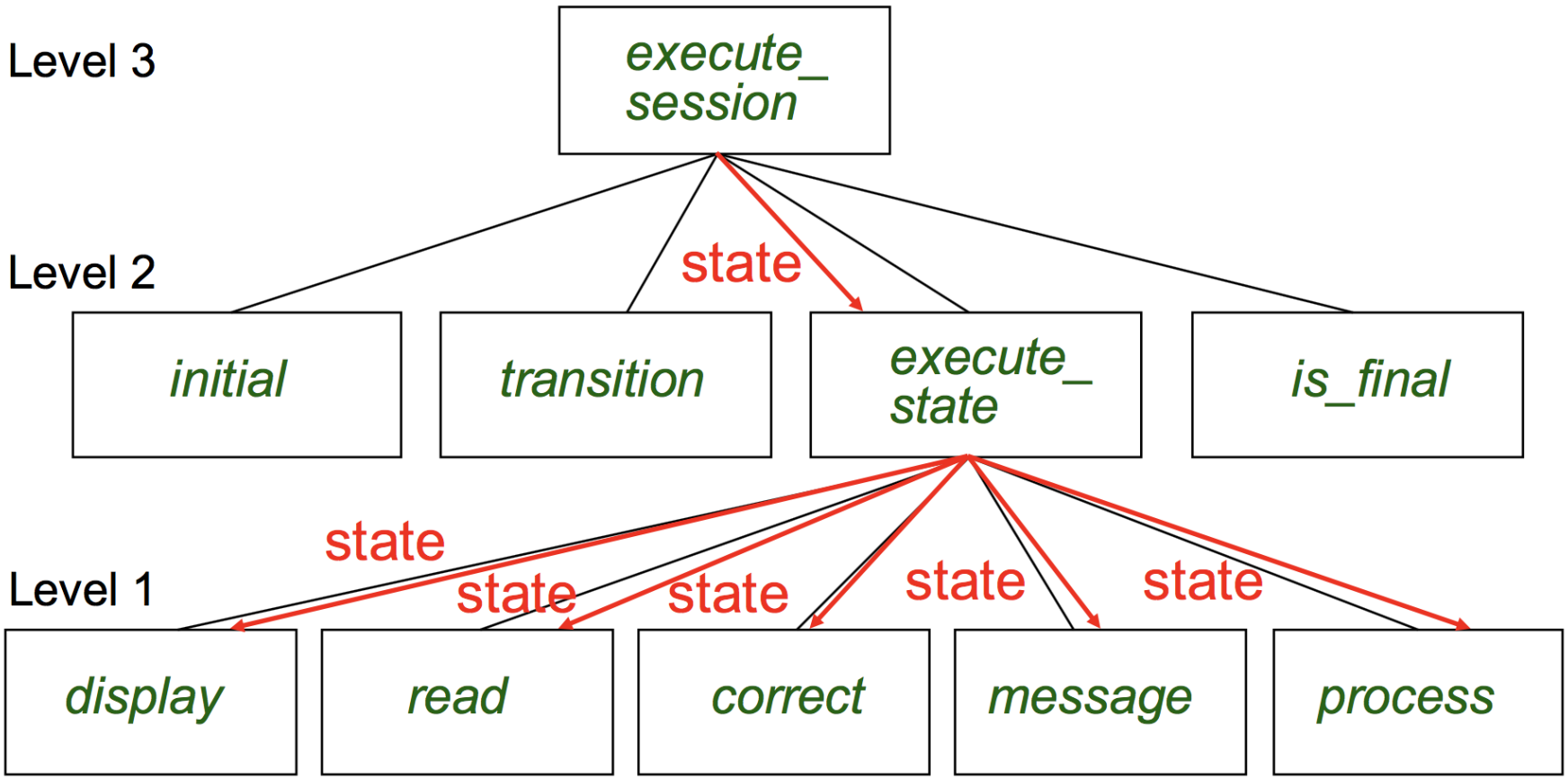
```
display(current_state: INTEGER)
  require
    valid_state: 1 ≤ current_state ≤ 6
  do
    if current_state = 1 then
      -- Display Initial Panel
    elseif current_state = 2 then
      -- Display Flight Enquiry Panel
    ...
    else
      -- Display Final Panel
    end
  end
```

- Such design *smells* !
  ∵ Same list of conditional repeats for **all** state-dependant features.
- Such design ***violates*** the *Single Choice Principle* .
  e.g., To add/delete a state ⇒ Add/delete a branch in all such features.

Hierarchical Solution: Visible Architecture



Hierarchical Solution: Pervasive States



Too much data transmission: `current_state` is passed

- From `execute_session` (**Level 3**) to `execute_state` (**Level 2**)
- From `execute_state` (**Level 2**) to all features at **Level 1**

Law of Inversion

If your routines exchange too many data, then put your routines in your data.

e.g.,

`execute_state` (**Level 2**) and all features at **Level 1**:

- Pass around (as ***inputs***) the notion of ***current_state***
- Build upon (via ***discriminations***) the notion of ***current_state***

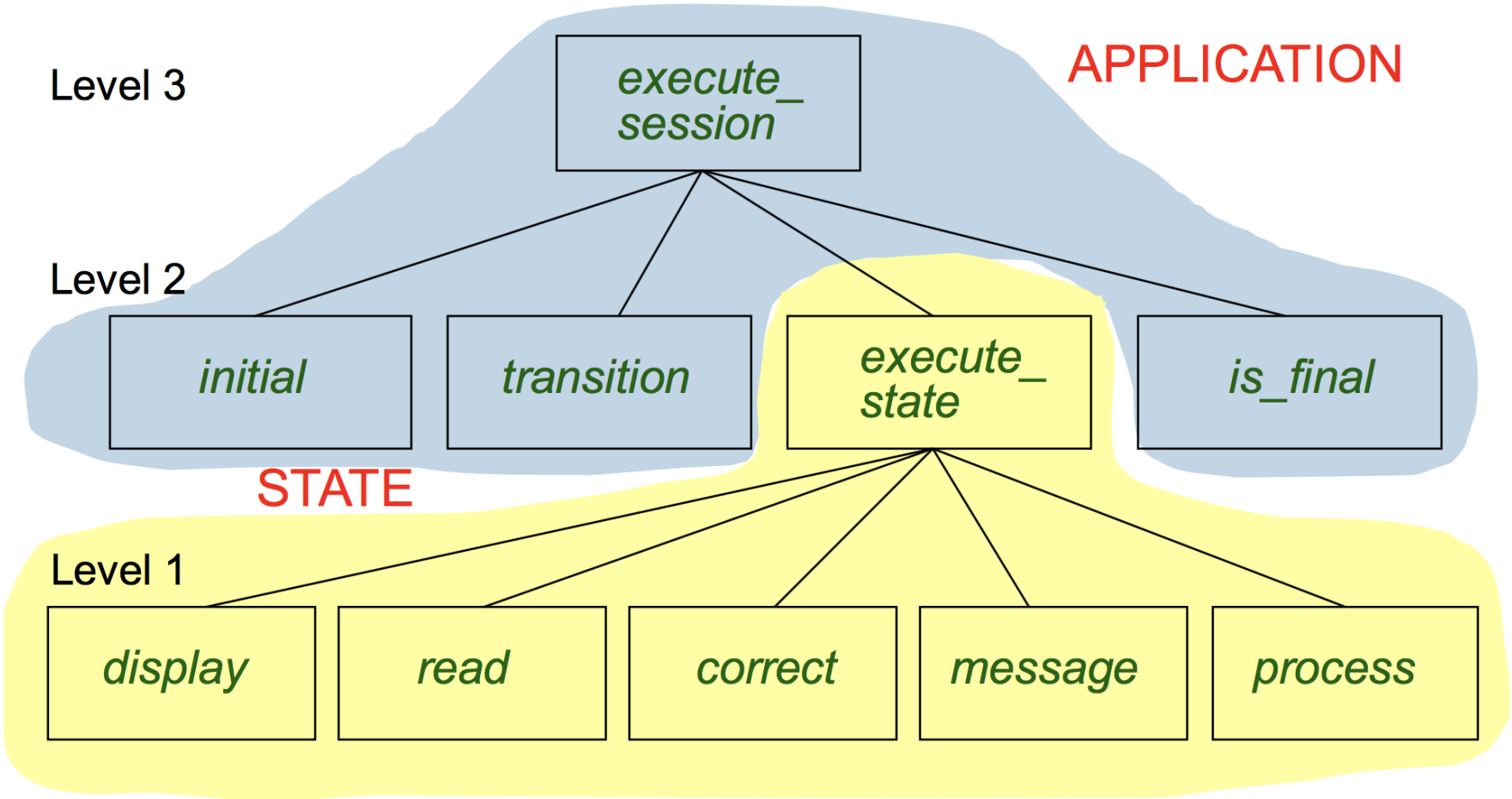
execute_state	(***s***: *INTEGER*)
display	(***s***: *INTEGER*)
read_answer	(***s***: *INTEGER*)
read_choice	(***s***: *INTEGER*)
correct	(***s***: *INTEGER* ; answer: ANSWER)
process	(***s***: *INTEGER* ; answer: ANSWER)
message	(***s***: *INTEGER* ; answer: ANSWER)

⇒ *Modularize* the notion of state as ***class STATE***.

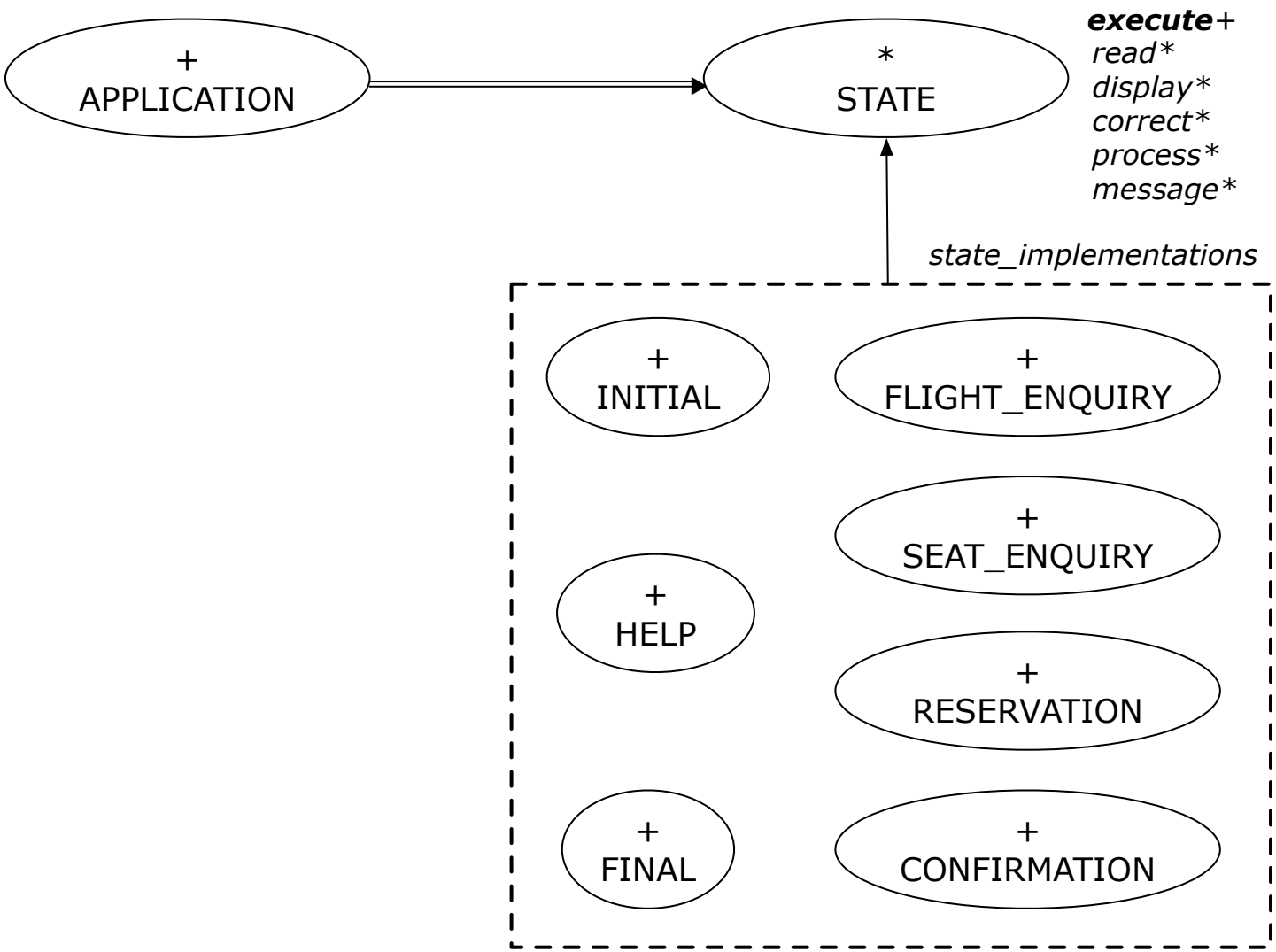
⇒ *Encapsulate* state-related information via a ***STATE*** interface.

⇒ Notion of ***current_state*** becomes ***implicit***: the `Current` class.

Grouping by Data Abstractions



Architecture of the State Pattern

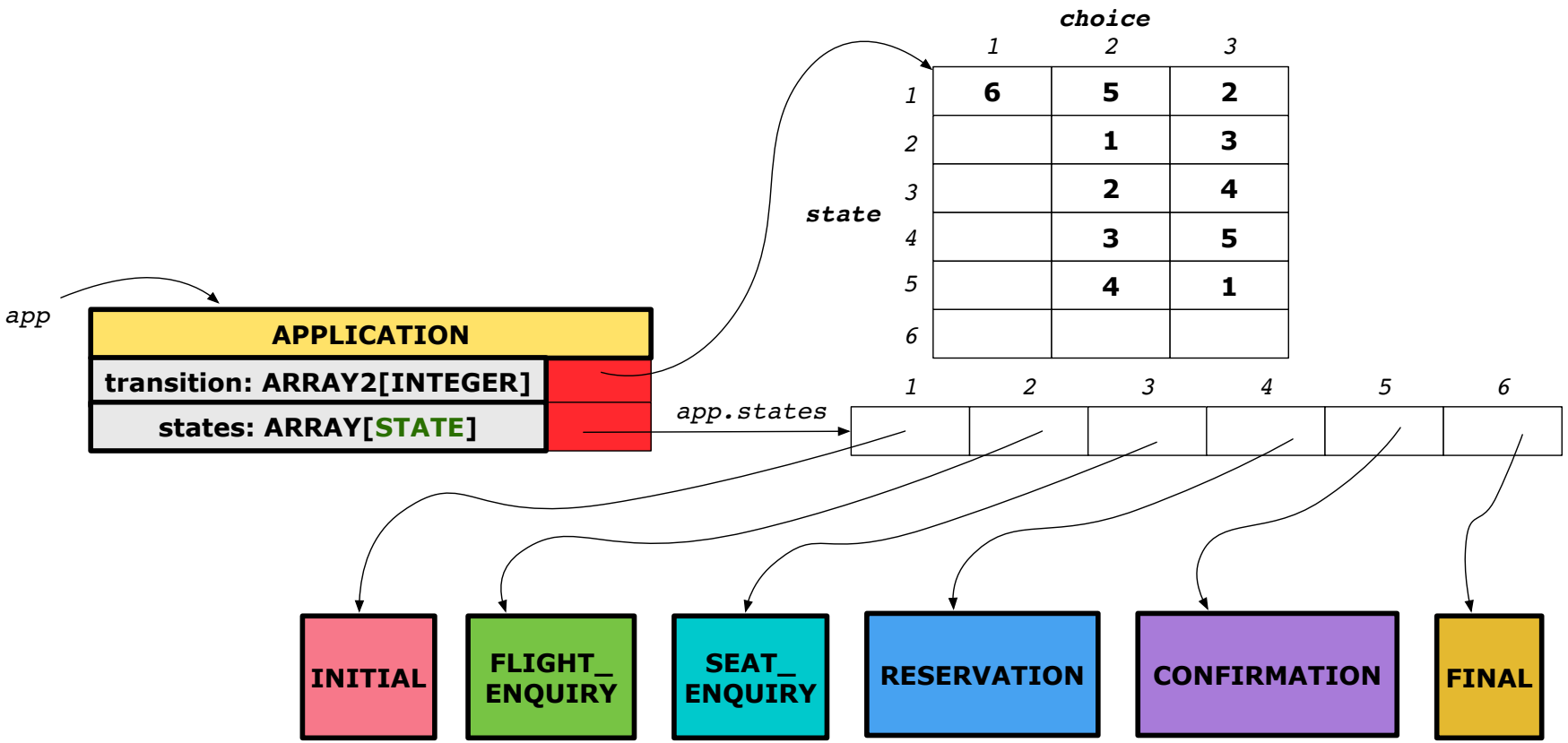


The STATE ADT

```
deferred class STATE
  read
    -- Read user's inputs
    -- Set 'answer' and 'choice'
    deferred end
  answer: ANSWER
    -- Answer for current state
  choice: INTEGER
    -- Choice for next step
  display
    -- Display current state
    deferred end
  correct: BOOLEAN
    deferred end
  process
    require correct
    deferred end
  message
    require not correct
    deferred end
```

```
  execute
    local
      good: BOOLEAN
    do
      from
      until
        good
      loop
        display
        -- set answer and choice
        read
        good := correct
        if not good then
          message
        end
      end
      process
  end
end
```

The APPLICATION Class: Array of STATE



The APPLICATION Class (1)

```
class APPLICATION create make
feature {NONE} -- Implementation of Transition Graph
  transition: ARRAY2[INTEGER]
    -- State transitions: transition[state, choice]
  states: ARRAY[STATE]
    -- State for each index, constrained by size of 'transition'
feature
  initial: INTEGER
  number_of_states: INTEGER
  number_of_choices: INTEGER
  make(n, m: INTEGER)
    do number_of_states := n
       number_of_choices := m
       create transition.make_filled(0, n, m)
       create states.make_empty
    end
invariant
   transition.height = number_of_states
   transition.width = number_of_choices
end
```

The APPLICATION Class (2)

```
class APPLICATION
feature {NONE} -- Implementation of Transition Graph
  transition: ARRAY2[INTEGER]
  states: ARRAY[STATE]
feature
  put_state(s: STATE; index: INTEGER)
    require 1 ≤ index ≤ number_of_states
    do states.force(s, index) end
  choose_initial(index: INTEGER)
    require 1 ≤ index ≤ number_of_states
    do initial := index end
  put_transition(tar, src, choice: INTEGER)
    require
      1 ≤ src ≤ number_of_states
      1 ≤ tar ≤ number_of_states
      1 ≤ choice ≤ number_of_choices
    do
      transition.put(tar, src, choice)
    end
end
```

The APPLICATION Class (3)

```
class APPLICATION
feature {NONE} -- Implementation of Transition Graph
  transition: ARRAY2[INTEGER]
  states: ARRAY[STATE]
feature
  execute_session
    local
      current_state: STATE
      index: INTEGER
    do
      from
        index := initial
      until
        is_final (index)
      loop
        current_state := states[index]  -- polymorphism
        current_state.execute  -- dynamic binding
        index := transition.item (index, current_state.choice)
      end
    end
end
```

Building an Application

- Create instances of `STATE`.

```
s1: STATE
create {INITIAL} s1.make
```

- Initialize an `APPLICATION`.

```
create app.make(number_of_states, number_of_choices)
```

- Perform polymorphic assignments on `app.states`.

```
app.put_state(initial, 1)
```

- Choose an initial state.

```
app.choose_initial(1)
```

- Build the transition table.

```
app.put_transition(6, 1, 1)
```

- Run the application.

```
app.execute_session
```

An Example Test

```
test_application: BOOLEAN
  local
    app: APPLICATION ; current_state: STATE ; index: INTEGER
  do
    create app.make (6, 3)
    app.put_state (create {INITIAL}.make, 1)
    -- Similarly for other 5 states.
    app.choose_initial (1)
    -- Transit to FINAL given current state INITIAL and choice 1.
    app.put_transition (6, 1, 1)
    -- Similarly for other 10 transitions.

    index := app.initial
    current_state := app.states [index]
    Result := attached {INITIAL} current_state
    check Result end
    -- Say user's choice is 3: transit from INITIAL to FLIGHT_STATUS
    index := app.transition.item (index, 3)
    current_state := app.states [index]
    Result := attached {FLIGHT_ENQUIRY} current_state
end
```

Top-Down, Hierarchical vs. OO Solutions

- In the second (top-down, hierarchy) solution, it is required for every state-related feature to *explicitly* and *manually* discriminate on the argument value, via a a list of conditionals.
  e.g., Given `display(current_state: INTEGER)`, the calls `display(1)` and `display(2)` behave differently.
- The third (OO) solution, called the State Pattern, makes such conditional *implicit* and *automatic*, by making `STATE` as a deferred class (whose descendants represent all types of states), and by delegating such conditional actions to *dynamic binding*.
  e.g., Given `s: STATE`, behaviour of the call `s.display` depends on the *dynamic type* of `s` (such as `INITIAL` vs. `FLIGHT_ENQUIRY`).

Index (1)

Index (2)

The Composite Design Pattern

EECS3311: Software Design
Fall 2017

CHEN-WEI WANG

Motivating Problem (1)

- Many manufactured systems, such as computer systems or stereo systems, are composed of ***individual components*** and ***sub-systems*** that contain components.

  e.g., A computer system is composed of:

  - Individual pieces of equipment (*hard drives*, *cd-rom drives*)
    Each equipment has *properties*: e.g., power consumption and cost.
  - Composites such as *cabinets*, *busses*, and *chassis*
    Each *cabinet* contains various types of *chassis*, each of which in turn containing components (*hard-drive*, *power-supply*) and *busses* that contain *cards*.

- Design a system that will allow us to easily *build* systems and *calculate* their total cost and power consumption.

Motivating Problem (2)

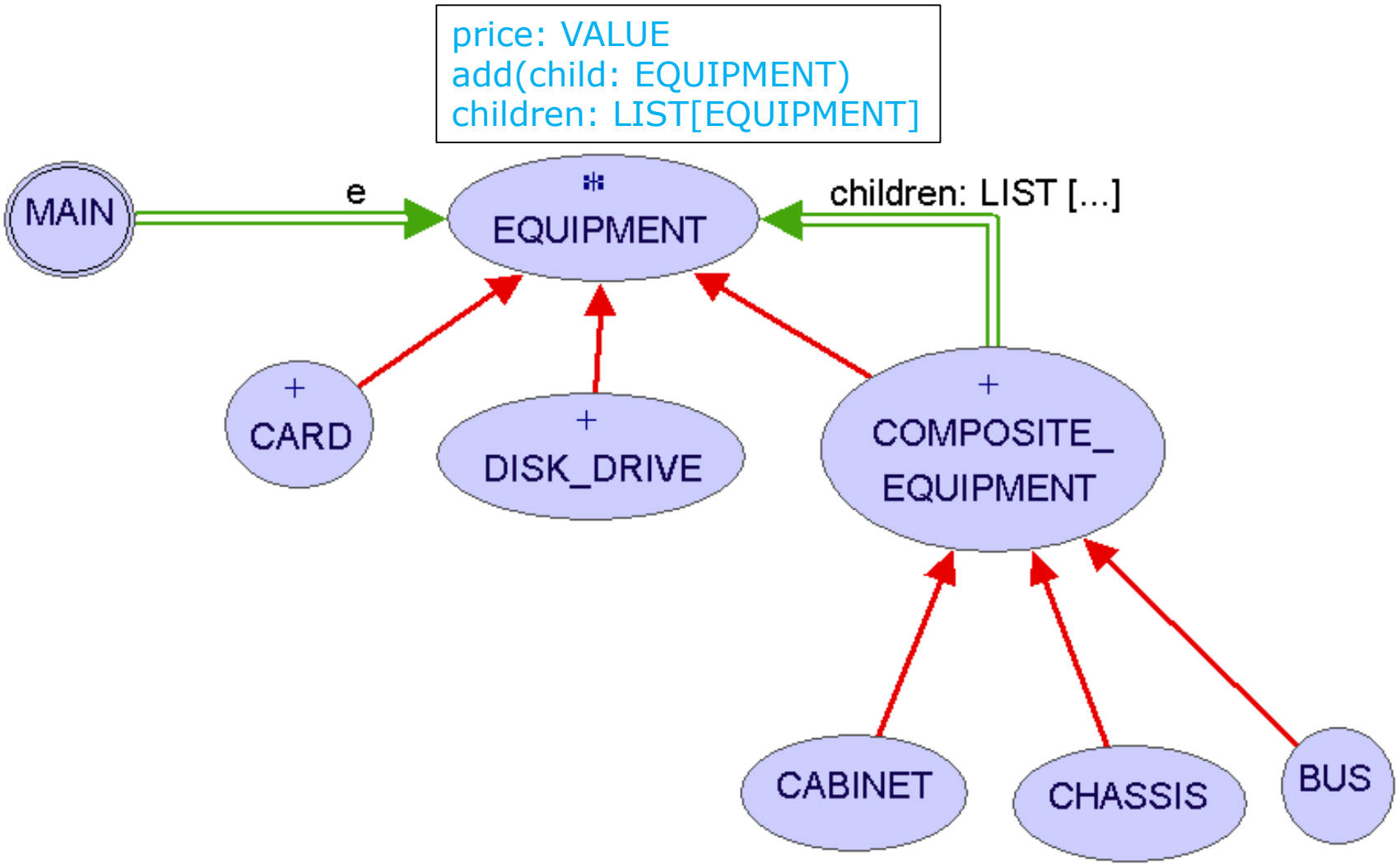
Design for *tree structures* with whole-part *hierarchies*.

Challenge: There are ***base*** and ***recursive*** modelling artifacts.

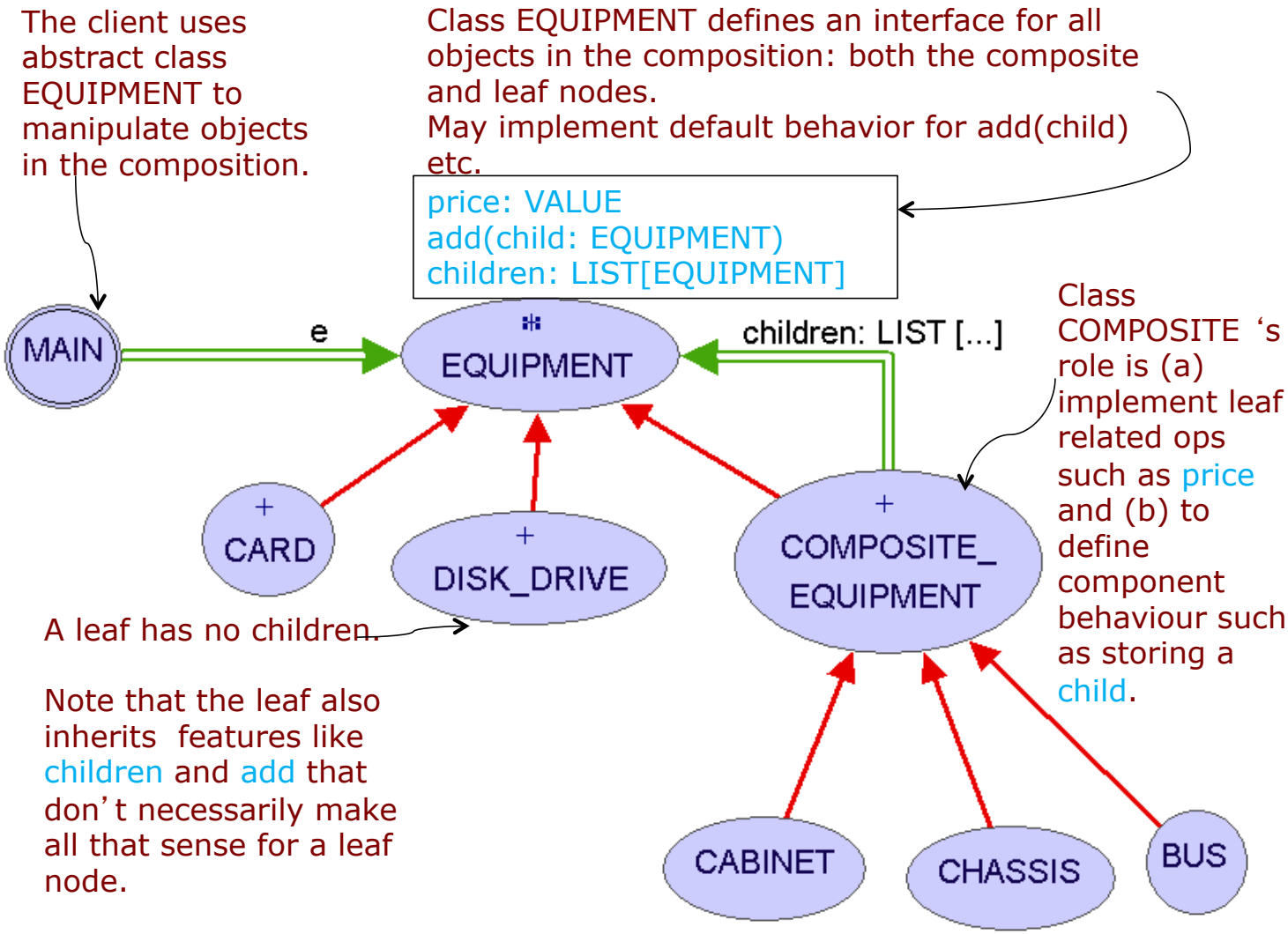
Solution: The Composite Pattern

- **Design**: Categorize into *base* artifacts or *recursive* artifacts.
- **Programming**:
  Build a *tree structure* representing the whole-part *hierarchy*.
- **Runtime**:
  Allow clients to treat *base* objects (leafs) and *recursive* compositions (nodes) *uniformly*.
  ⇒ *Polymorphism*: *leafs* and *nodes* are "substitutable".
  ⇒ *Dynamic Binding*: Different versions of the same operation is applied on *individual objects* and *composites*.
  e.g., Given e: ***EQUIPMENT***:
  - `e.price` may return the unit price of a ***DISK_DRIVE***.
  - `e.price` may sum prices of a ***CHASIS***' containing equipments.

Composite Architecture: Design (1.1)



Composite Architecture: Design (1.2)



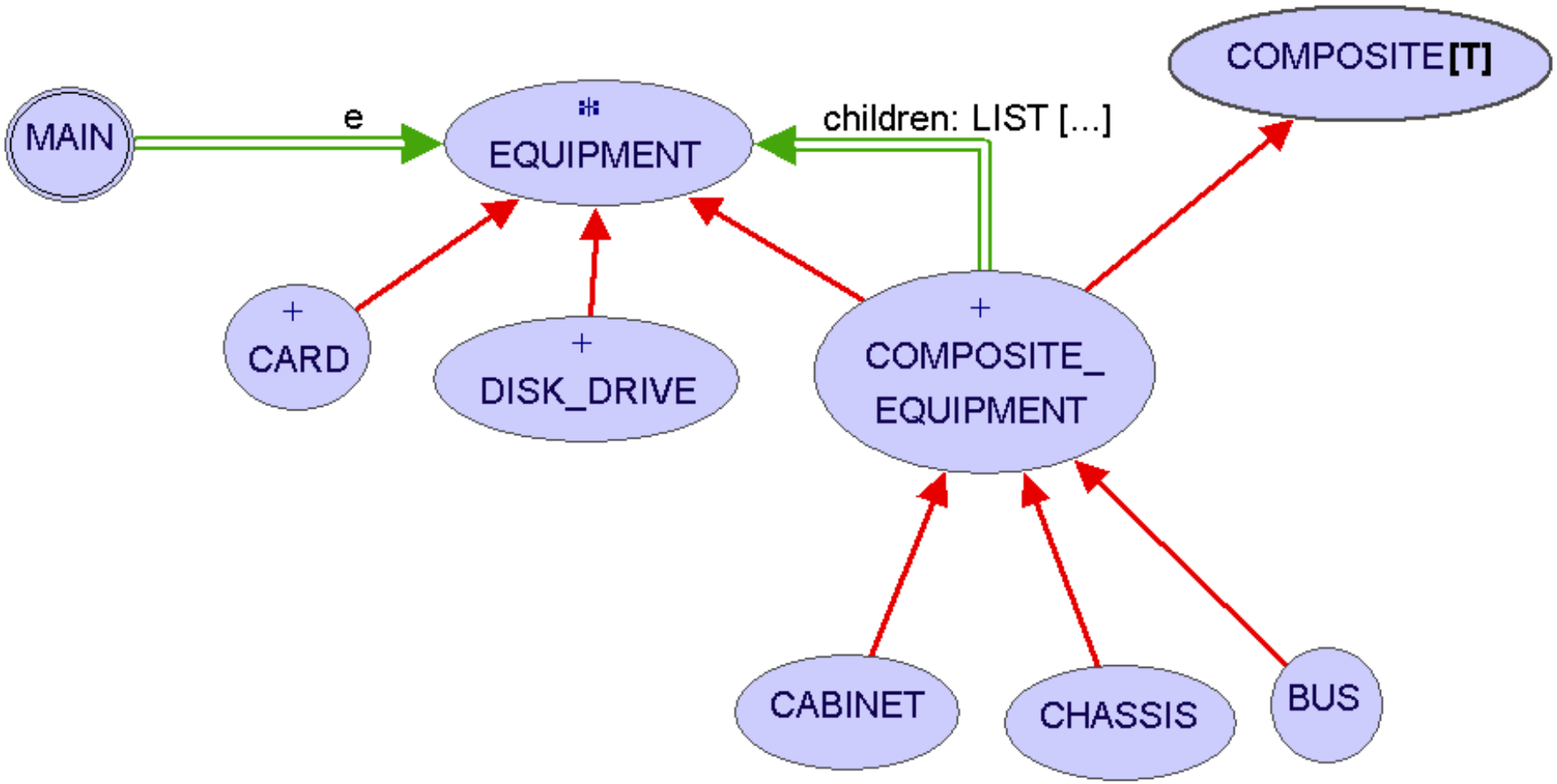
Composite Architecture: Design (1.3)

Q: Any flaw of this first design?

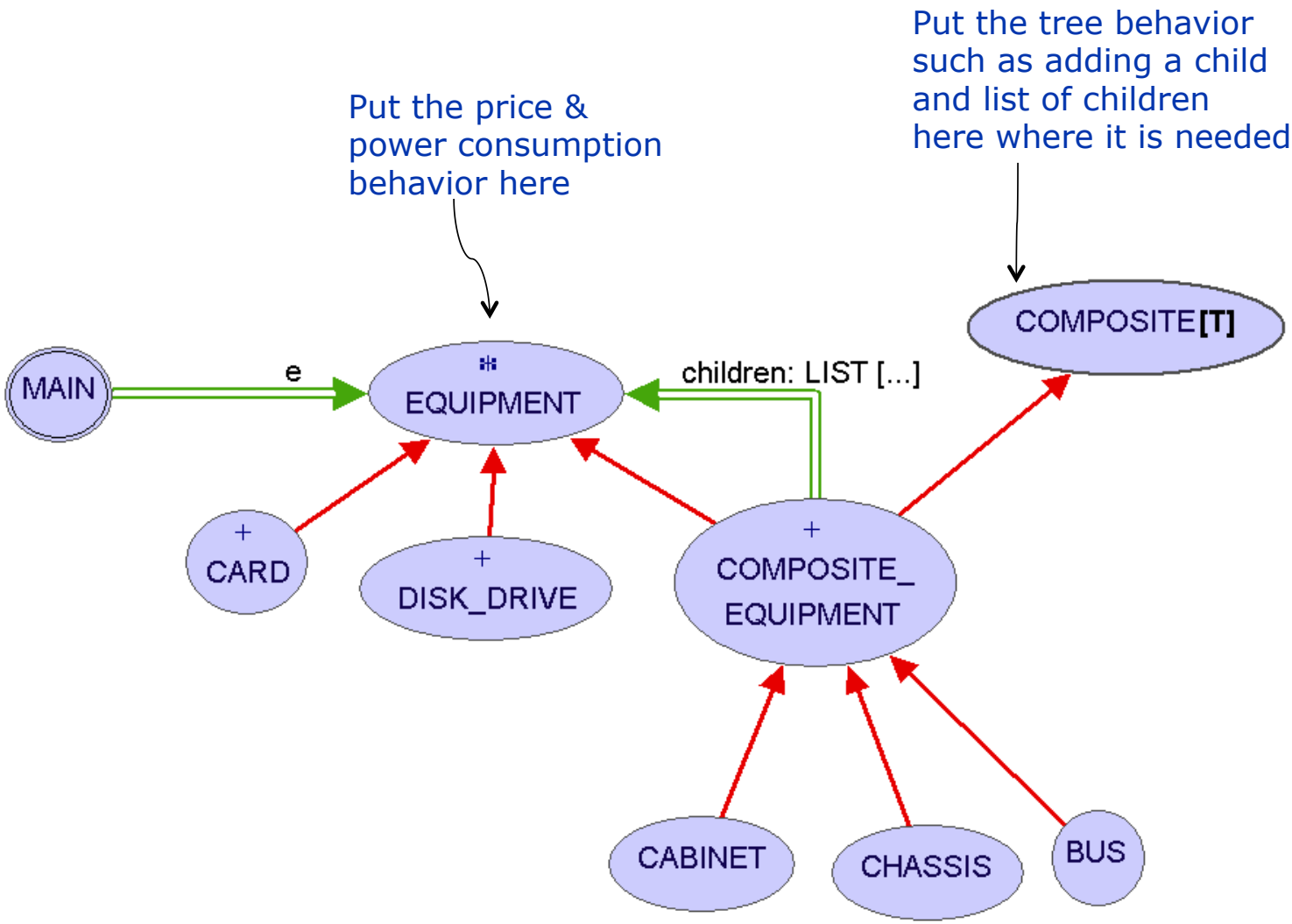
A:

The `add(child: EQUIPMENT)` and `children: LIST[EQUIPMENT]` features are defined at the `EQUIPMENT` level.
⇒ Inherited to all *base* equipments (e.g., `HARD_DRIVE`) that do not apply to such features.

Composite Architecture: Design (2.1)



Composite Architecture: Design (2.2)



Implementing the Composite Pattern (1)

```
deferred class
  EQUIPMENT
feature
  name: STRING
  price: REAL -- uniform access principle
end
```

```
class
  CARD
inherit
  EQUIPMENT
feature
  make (n: STRING; p: REAL)
    do
      name := n
      price := p -- price is an attribute
    end
end
```

Implementing the Composite Pattern (2.1)

```
deferred class
  COMPOSITE[T]
feature
  children: LINKED_LIST[T]

  add_child (c: T)
    do
      children.extend (c) -- Polymorphism
    end
end
```

Exercise: Make the `COMPOSITE` class *iterable*.

Implementing the Composite Pattern (2.2)

```
class
  COMPOSITE_EQUIPMENT
inherit
  EQUIPMENT
  COMPOSITE [EQUIPMENT]
create
  make
feature
  make (n: STRING)
    do name := n ; create children.make end
  price : REAL -- price is a query
      -- Sum the net prices of all sub-equipments
    do
      across
        children as cursor
      loop
        Result := Result + cursor.item.price -- dynamic binding
      end
    end
end
```

Testing the Composite Pattern

```
test_composite_equipment: BOOLEAN
  local
    card, drive: EQUIPMENT
    cabinet: CABINET -- holds a CHASSIS
    chassis: CHASSIS -- contains a BUS and a DISK_DRIVE
    bus: BUS -- holds a CARD
  do
    create {CARD} card.make("16Mbs Token Ring", 200)
    create {DISK_DRIVE} drive.make("500 GB harddrive", 500)
    create bus.make("MCA Bus")
    create chassis.make("PC Chassis")
    create cabinet.make("PC Cabinet")

    bus.add(card)
    chassis.add(bus)
    chassis.add(drive)
    cabinet.add(chassis)
    Result := cabinet.price = 700
  end
```

Index (1)

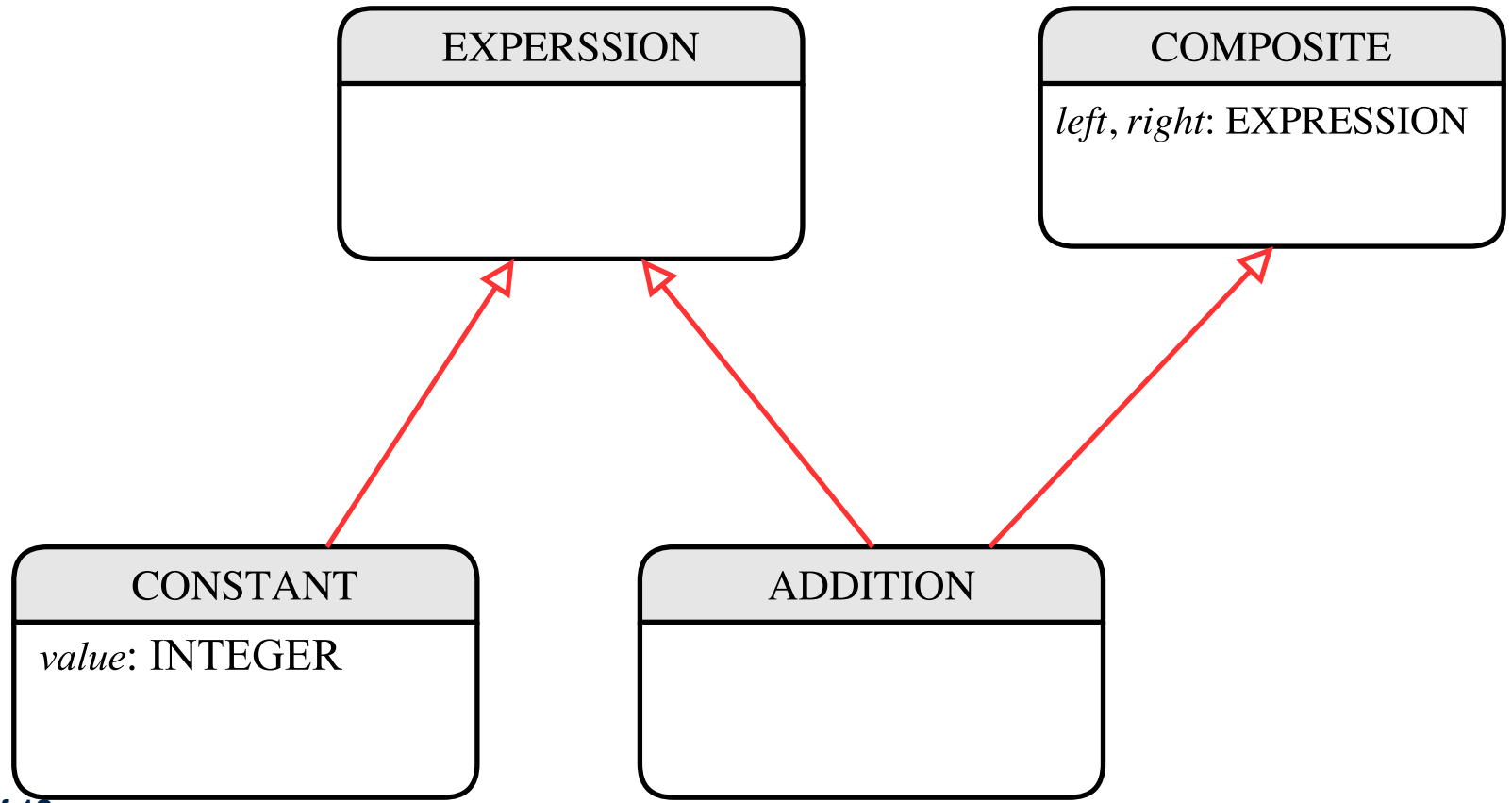
The Visitor Design Pattern

EECS3311: Software Design
Fall 2017

CHEN-WEI WANG

Motivating Problem (1)

Based on the *composite pattern* you learned, design classes to model *structures* of arithmetic expressions (e.g., `341`, `2`, `341 + 2`).



Open/Closed Principle

Software entities (classes, features, etc.) should be ***open*** for *extension*, but ***closed*** for *modification*.

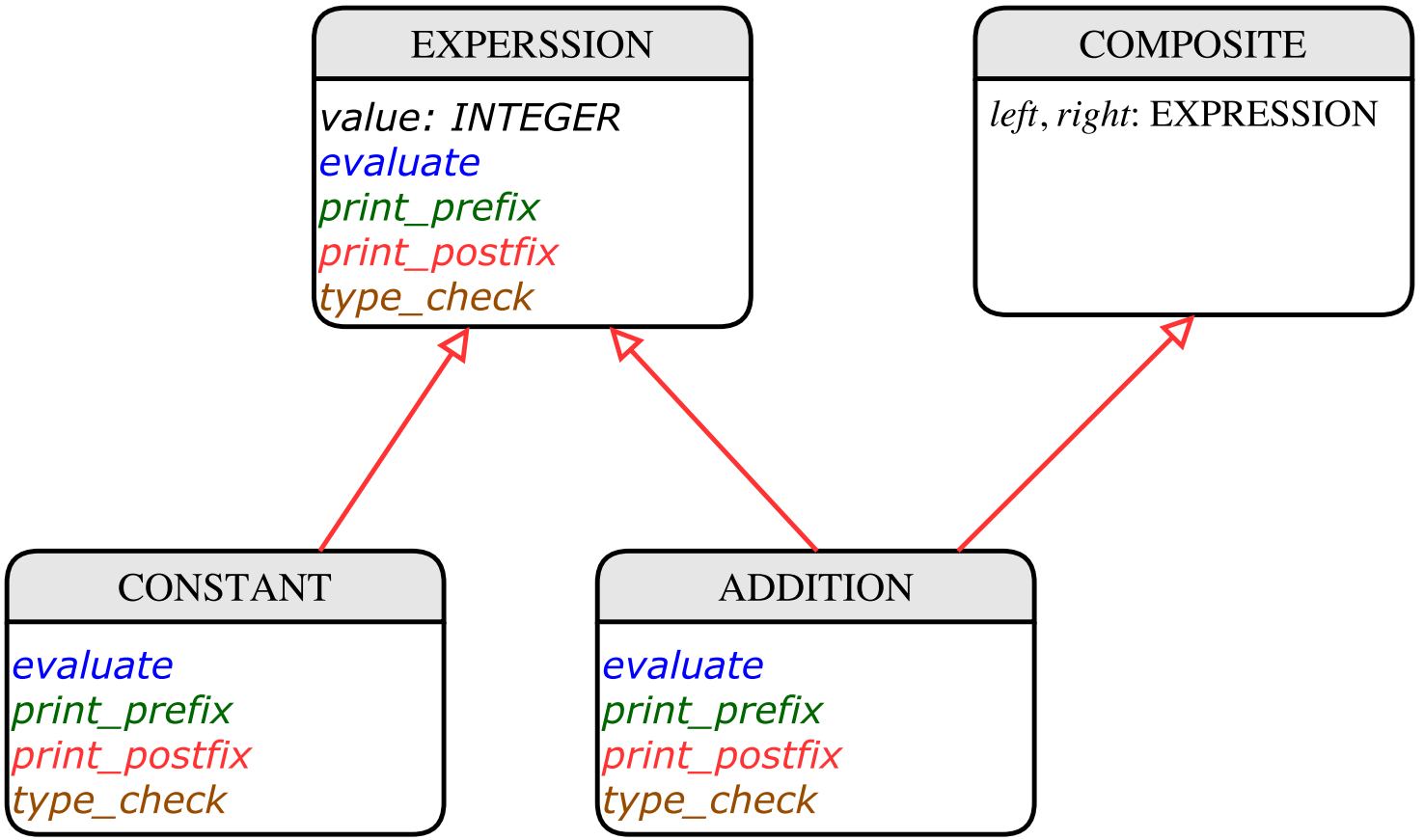
⇒ When *extending* the behaviour of a system, we may *add new code*, but we should *not modify the existing code*.

e.g., In the design for structures of expressions:

- ***Closed***: Syntactic constructs of the language [stable]
- ***Open***: New operations on the language [unstable]

Motivating Problem (2)

Extend the *composite pattern* to support *operations* such as `evaluate`, pretty printing (`print_prefix`, `print_postfix`), and `type_check`.



Problems of Extended Composite Pattern

- Distributing the various **unrelated** ***operations*** across nodes of the ***abstract syntax tree*** violates the *single-choice principle* :

  To add/delete/modify an operation
  ⇒ Change of all descendants of `EXPRESSION`

- Each node class lacks in *cohesion* :

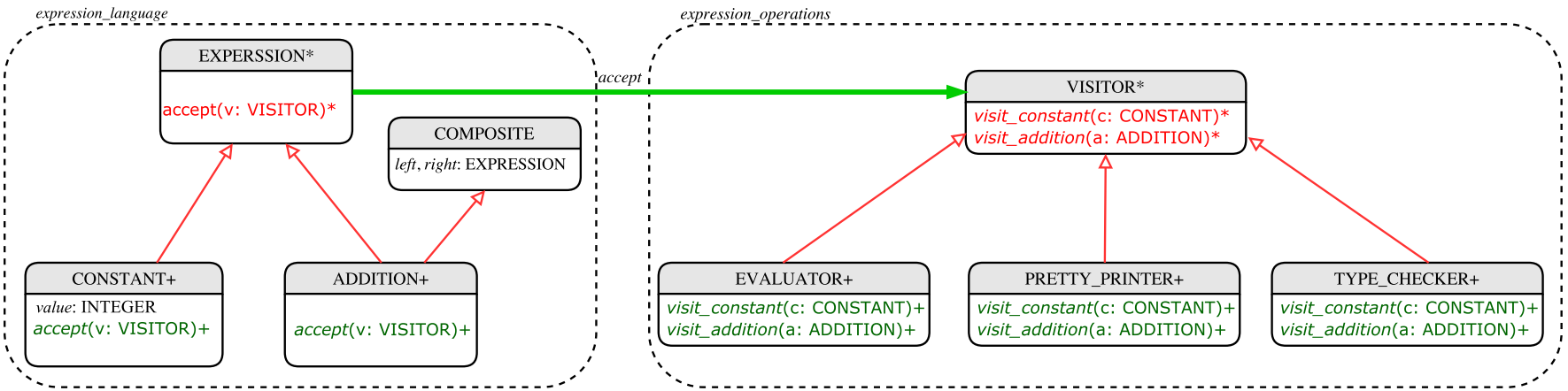
  A *class* is supposed to group *relevant* concepts in a *single* place.
  ⇒ Confusing to mix codes for evaluation, pretty printing, and type checking.
  ⇒ We want to avoid “polluting” the classes with these various unrelated operations.

Visitor Pattern

- *Separation of concerns* :
  - Set of language constructs [***closed***, stable]
  - Set of operations [***open***, unstable]

    ⇒ Classes from these two sets are *decoupled* and organized into two separate clusters.

Visitor Pattern: Architecture



Visitor Pattern Implementation: Structures

Cluster ***expression_language***

- Declare *deferred* feature `accept(v: VISITOR)` in `EXPRSSION`.
- Implement `accept` feature in each of the descendant classes.

```
class CONSTANT
...
  accept(v: VISITOR)
    do
      v.visit_constant (Current)
    end
end
```

```
class ADDITION
...
  accept(v: VISITOR)
    do
      v.visit_addition (Current)
    end
end
```

Visitor Pattern Implementation: Operations

Cluster ***expression_operations***

- For each descendant class `C` of `EXPRESSION`, declare a *deferred* feature `visit_c (e: C)` in the *deferred* class `VISITOR`.

```
class VISITOR
  visit_constant(c: CONSTANT) deferred end
  visit_addition(a: ADDITION) deferred end
end
```

- Each descendant of `VISITOR` denotes a kind of operation.

```
class EVALUATOR
  value: INTEGER
  visit_constant(c: CONSTANT)  do value := c.value end
  visit_addition(a: ADDITION)
    local eval_left, eval_right: EVALUATOR
    do a.left.accept(eval_left)
       a.right.accept(eval_right)
       value := eval_left.value + eval_right.value
    end
end
```

Testing the Visitor Pattern

```
1  test_expression_evaluation: BOOLEAN
2    local add, c1, c2: EXPRESSION ; v: VISITOR
3    do
4      create {CONSTANT} c1.make (1) ; create {CONSTANT} c2.make (2)
5      create {ADDITION} add.make (c1, c2)
6      create {EVALUATOR} v.make
7      add.accept(v)
8      check attached {EVALUATOR} v as eval then
9        Result := eval.value = 3
10     end
11   end
```

Double Dispatch in **Line 7**:

1. ***DT*** of `add` is *ADDITION* ⇒ Call `accept` in *ADDITION*

   `v.visit_addition (add)`

2. ***DT*** of `v` is *EVALUATOR* ⇒ Call `visit_addition` in *EVALUATOR*

   visiting result of `add.left` + visiting result of `add.right`

To Use or Not to Use the Visitor Pattern

- In the architecture of visitor pattern, what kind of *extensions* is easy and hard? Language structure? Language Operation?
  - Adding a new kind of ***operation*** element is easy.

    To introduce a new operation for generating C code, we only need to introduce a new descendant class `C_CODE_GENERATOR` of `VISITOR`, then implement how to handle each language element in that class.

    ⇒ *Single Choice Principle* is *obeyed*.
  - Adding a new kind of ***structure*** element is hard.

    After adding a descendant class `MULTIPLICATION` of `EXPRESSION`, every concrete visitor (i.e., descendant of `VISITOR`) must be amended to provide a new `visit_multiplication` operation.

    ⇒ *Single Choice Principle* is *violated*.
- The applicability of the visitor pattern depends on to what extent the ***structure*** will change.

  ⇒ Use visitor if ***operations*** applied to ***structure*** might change.

  ⇒ Do not use visitor if the ***structure*** might change.

Index (1)

Void Safety

EECS3311: Software Design
Fall 2017

CHEN-WEI WANG

Java Program: Example 1

```
1  class Point {
2    double x;
3    double y;
4    Point(double x, double y) {
5      this.x = x;
6      this.y = y;
7  }
```

```
1  class PointCollector {
2    ArrayList<Point> points;
3    PointCollector() { }
4    void addPoint(Point p) {
5      points.add(p); }
6    Point getPointAt(int i) {
7      return points.get(i); } }
```

The above Java code ***compiles***. But anything wrong?

```
1  @Test
2  public void test1() {
3    PointCollector pc = new PointCollector();
4    pc.addPoint(new Point(3, 4));
5    Point p = pc.getPointAt(0);
6    assertTrue(p.x == 3 && p.y == 4); }
```

L3 calls `PointCollector` constructor not initializing `points`.

∴ *NullPointerException* when **L4** calls **L5** of `PointCollector`.

Java Program: Example 2

```
1  class Point {
2    double x;
3    double y;
4    Point(double x, double y) {
5      this.x = x;
6      this.y = y;
7  }
```

```
1  class PointCollector {
2    ArrayList<Point> points;
3    PointCollector() {
4      points = new ArrayList<>(); }
5    void addPoint(Point p) {
6      points.add(p); }
7    Point getPointAt(int i) {
8      return points.get(i); } }
```

```
1  @Test
2  public void test2() {
3    PointCollector pc = new PointCollector();
4    Point p = null;
5    pc.addPoint(p);
6    p = pc.getPointAt(0);
7    assertTrue(p.x == 3 && p.y == 4); }
```

The above Java code ***compiles***. But anything wrong?

L5 adds `p` (which stores `null`).

∴ *NullPointerException* when **L7** calls `p.x`.

Java Program: Example 3

```
1  class Point {
2    double x;
3    double y;
4    Point(double x, double y) {
5      this.x = x;
6      this.y = y;
7  }
```

```
1  class PointCollector {
2    ArrayList<Point> points;
3    PointCollector() {
4      points = new ArrayList<>(); }
5    void addPoint(Point p) {
6      points.add(p); }
7    Point getPointAt(int i) {
8      return points.get(i); } }
```

```
1  public void test3() {
2    PointCollector pc = new PointCollector();
3    Scanner input = new Scanner(System.in);
4    System.out.println("Ener an integer:");
5    int i = input.nextInt();
6    if(i < 0) { pc = null; }
7    pc.addPoint(new Point(3, 4));
8    assertTrue(pc.getPointAt(0).x == 3 && pc.getPointAt(0).y == 4);
9  }
```

The above Java code ***compiles***. But anything wrong?

NullPointerException when user's input at **L5** is non-positive.

Limitation of Java's Type System

- A program that compiles does not guarantee that it is free from *NullPointerExceptions*:
  - Uninitialized attributes (in constructors).
  - Passing ***nullable*** variable as a method argument.
  - Calling methods on ***nullable*** local variables.
- The notion of `Null` references was back in 1965 in ALGO W.
- Tony Hoare (inventor of Quick Sort), introduced this notion of `Null` references "simply because *it was so easy to implement*".
- But he later considers it as his "***billion-dollar mistake***".
  - When your program manipulates reference/object variables whose types include the legitimate value of `Null` or `Void`, then there is always a possibility of having a *NullPointerExceptions*.
  - For undisciplined programmers, this means the final software product *crashes* often!

Eiffel's Type System for Void Safety

- By default, a reference variable is ***non-detachable***.

  e.g., `acc: ACCOUNT` means that `acc` is always ***attached*** to some valid `ACCOUNT` point.

- *VOID* is an illegal value for ***non-detachable*** variables.

  ⇒ Scenarios that might make a reference variable *detached* are considered as ***compile-time errors***:

  - Variables can not be assigned to `Void` directly.
  - ***Non-detachable*** variables can only be re-assigned to ***non-detachable*** variables.

    e.g., `acc2: ACCOUNT` ⇒ `acc := acc2` *compilable*

    e.g., `acc3: detachable ACCOUNT` ⇒ `acc := acc3` *non-compilable*

  - Creating variables (e.g., `create acc.make`) *compilable*
  - ***Non-detachable*** attribute not explicitly initialized (via creation or assignment) in all constructors is *non-compilable*.

Eiffel Program: Example 1

```
1  class
2    POINT
3  create
4    make
5  feature
6    x: REAL
7    y: REAL
8  feature
9    make (nx: REAL; ny: REAL)
10     do x := nx
11        y := ny
12     end
13 end
```

```
1  class
2    POINT_COLLECTOR_1
3  create
4    make
5  feature
6    points: LINKED_LIST[POINT]
7  feature
8    make do end
9  add_point (p: POINT)
10   do points.extend (p) end
11 get_point_at (i: INTEGER): POINT
12   do Result := points [i] end
13 end
```

- Above code is semantically equivalent to Example 1 Java code.
- Eiffel compiler won't allow you to run it.

  ∵ **L8** does ***non compile***

  ∴ It is *void safe* [Possibility of *NullPointerException* ruled out]

Eiffel Program: Example 2

```
1  class
2    POINT
3  create
4    make
5  feature
6    x: REAL
7    y: REAL
8  feature
9    make (nx: REAL; ny: REAL)
10     do x := nx
11        y := ny
12     end
13 end
```

```
1  class
2    POINT_COLLECTOR_2
3  create
4    make
5  feature
6    points: LINKED_LIST[POINT]
7  feature
8    make do create points.make end
9  add_point (p: POINT)
10   do points.extend (p) end
11 get_point_at (i: INTEGER): POINT
12   do Result := points [i] end
13 end
```

```
1  test_2: BOOLEAN
2    local
3      pc: POINT_COLLECTOR_2 ; p: POINT
4    do
5      create pc.make
6      p := Void
7      p.add_point (p)
8      p := pc.get_point_at (0)
9      Result := p.x = 3 and p.y = 4
10   end
```

- Above code is semantically equivalent to Example 2 Java code.
  L7 does ***non compile*** ∵ `pc` might be void. [*void safe*]

Eiffel Program: Example 3

```
1  class
2    POINT
3  create
4    make
5  feature
6    x: REAL
7    y: REAL
8  feature
9    make (nx: REAL; ny: REAL)
10     do x := nx
11        y := ny
12     end
13 end
```

```
1  class
2    POINT_COLLECTOR_2
3  create
4    make
5  feature
6    points: LINKED_LIST[POINT]
7  feature
8    make do create points.make end
9  add_point (p: POINT)
10   do points.extend (p) end
11 get_point_at (i: INTEGER): POINT
12   do Result := points [i] end
13 end
```

```
1  test_3: BOOLEAN
2    local pc: POINT_COLLECTOR_2 ; p: POINT ; i: INTEGER
3    do create pc.make
4       io.print ("Enter an integer:%N")
5       io.read_integer
6       if io.last_integer < 0 then pc := Void end
7       pc.add_point (create {POINT}.make (3, 4))
8       p := pc.get_point_at (0)
9       Result := p.x = 3 and p.y = 4
10   end
```

- Above code is semantically equivalent to Example 3 Java code.
  L7 and **L8** do ***non compile*** ∵ `pc` might be void. [*void safe*]

Lessons from Void Safety

- It is much more costly to recover from *crashing* programs (due to `NullPointerException`) than to fix *uncompilable* programs.

  e.g., You'd rather have a ***void-safe design*** for an airplane, rather than hoping that the plane won't crash after taking off.
- If you are used to the standard by which Eiffel compiler checks your code for *void safety*, then you are most likely to write Java/C/C++/C#/Python code that is *void-safe* (i.e., free from *NullPointerExceptions*).

Beyond this lecture. . .

- Tutorial Series on Void Safety by Bertrand Meyer (inventor of Eiffel):
  - The End of Null Pointer Dereferencing
  - The Object Test
  - The Type Rules
  - Final Rules
- Null Pointer as a Billion-Dollar Mistake by Tony Hoare
- More notes on void safety

Index (1)

Observer Design Pattern
Event-Driven Design

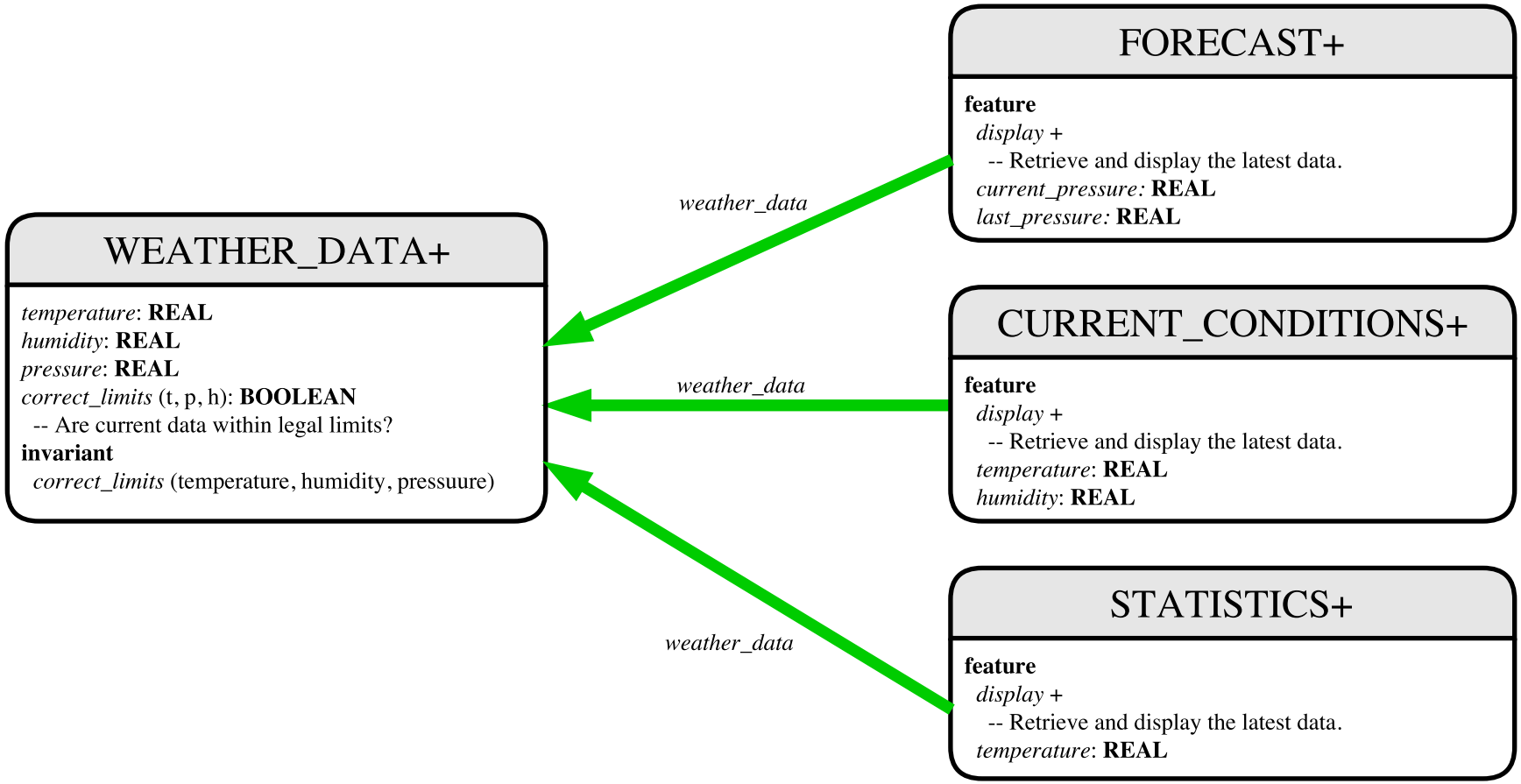
EECS3311: Software Design
Fall 2017

CHEN-WEI WANG

Motivating Problem

- A *weather station* maintains *weather data* such as *temperature*, *humidity*, and *pressure*.
- Various kinds of applications on these *weather data* should regularly update their *displays*:
  - *Condition*: *temperature* in celsius and *humidity* in percentages.
  - *Forecast*: if expecting for rainy weather due to reduced *pressure*.
  - *Statistics*: minimum/maximum/average measures of *temperature*.

First Design: Weather Station



Whenever the `display` feature is called, **retrieve** the current values of `temperature`, `humidity`, and/or `pressure` via the `weather_data` reference.

Implementing the First Design (1)

```
class WEATHER_DATA create make
feature -- Data
  temperature: REAL
  humidity: REAL
  pressure: REAL
feature -- Queries
  correct_limits(t,p,h: REAL): BOOLEAN
    ensure
      Result implies -36 <=t and t <= 60
      Result implies 50 <= p and p <= 110
      Result implies 0.8 <= h and h <= 100
feature -- Commands
  make (t, p, h: REAL)
    require
      correct_limits(temperature, pressure, humidity)
    ensure
      temperature = t and pressure = p and humidity = h
invariant
  correct_limits(temperature, pressure, humidity)
end
```

Implementing the First Design (2.1)

```
class CURRENT_CONDITIONS create make
feature -- Attributes
  temperature: REAL
  humidity: REAL
  weather_data: WEATHER_DATA
feature -- Commands
  make(wd: WEATHER_DATA)
    ensure weather_data = wd
  update
    do temperature := weather_data.temperature
       humidity := weather_data.humidity
    end
  display
    do update
       io.put_string("Current Conditions: ")
       io.put_real (temperature) ; io.put_string (" degrees C and ")
       io.put_real (humidity) ; io.put_string (" percent humidity%N")
    end
end
```

Implementing the First Design (2.2)

```
class FORECAST create make
feature -- Attributes
  current_pressure: REAL
  last_pressure: REAL
  weather_data: WEATHER_DATA
feature -- Commands
  make(wd: WEATHER_DATA) ensure weather_data = a_weather_data
  update
    do last_pressure := current_pressure
       current_pressure := weather_data.pressure
    end
  display
    do update
       if current_pressure > last_pressure then
         print("Improving weather on the way!%N")
       elseif current_pressure = last_pressure then
         print("More of the same%N")
       else print("Watch out for cooler, rainy weather%N") end
    end
end
```

Implementing the First Design (2.3)

```
class STATISTICS create make
feature -- Attributes
  weather_data: WEATHER_DATA
  current_temp: REAL
  max, min, sum_so_far: REAL
  num_readings: INTEGER
feature -- Commands
  make(wd: WEATHER_DATA)
    ensure weather_data = a_weather_data
  update
    do current_temp := weather_data.temperature
       -- Update min, max if necessary.
    end
  display
    do update
       print("Avg/Max/Min temperature = ")
       print(sum_so_far / num_readings + "/" + max + "/" min + "%N")
    end
end
```

Implementing the First Design (3)

```
1  class WEATHER_STATION create make
2  feature -- Attributes
3    cc: CURRENT_CONDITIONS ; fd: FORECAST ; sd: STATISTICS
4    wd: WEATHER_DATA
5  feature -- Commands
6    make
7      do create wd.make (9, 75, 25)
8         create cc.make (wd) ; create fd.make (wd) ; create sd.make(wd)
9
10        wd.set_measurements (15, 60, 30.4)
11        cc.display ; fd.display ; sd.display
12
13        cc.display ; fd.display ; sd.display
14
15        wd.set_measurements (11, 90, 20)
16        cc.display ; fd.display ; sd.display
17   end
18 end
```

L14: Updates occur on `cc`, `fd`, `sd` even with the same data.

First Design: Good Design?

- Each application (`CURRENT_CONDITION`, `FORECAST`, `STATISTICS`) *cannot know* when the weather data change.

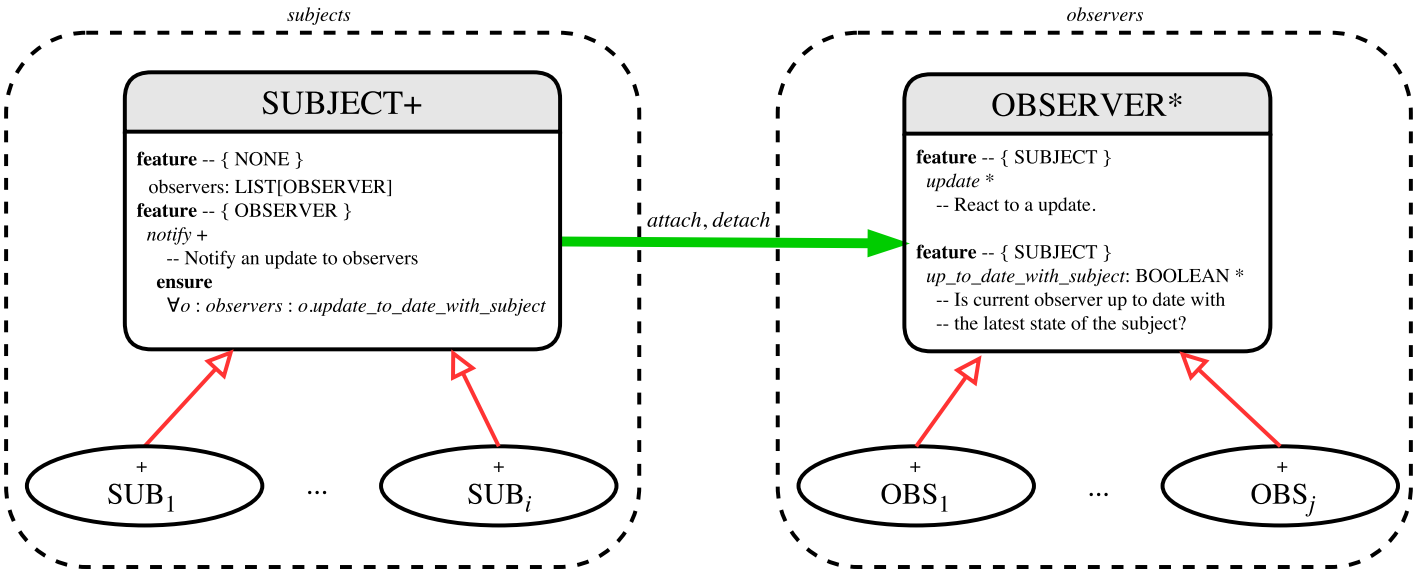
  $\Rightarrow$ All applications have to periodically initiate updates in order to keep the `display` results up to date.

  $\because$ Each inquiry of current weather data values is *a remote call*.

  $\therefore$ Waste of computing resources (e.g., network bandwidth) when there are actually no changes on the weather data.
- To avoid such overhead, it is better to let:
  - Each application ***subscribe*** the weather data.
  - The weather station ***publish/notify*** new changes.

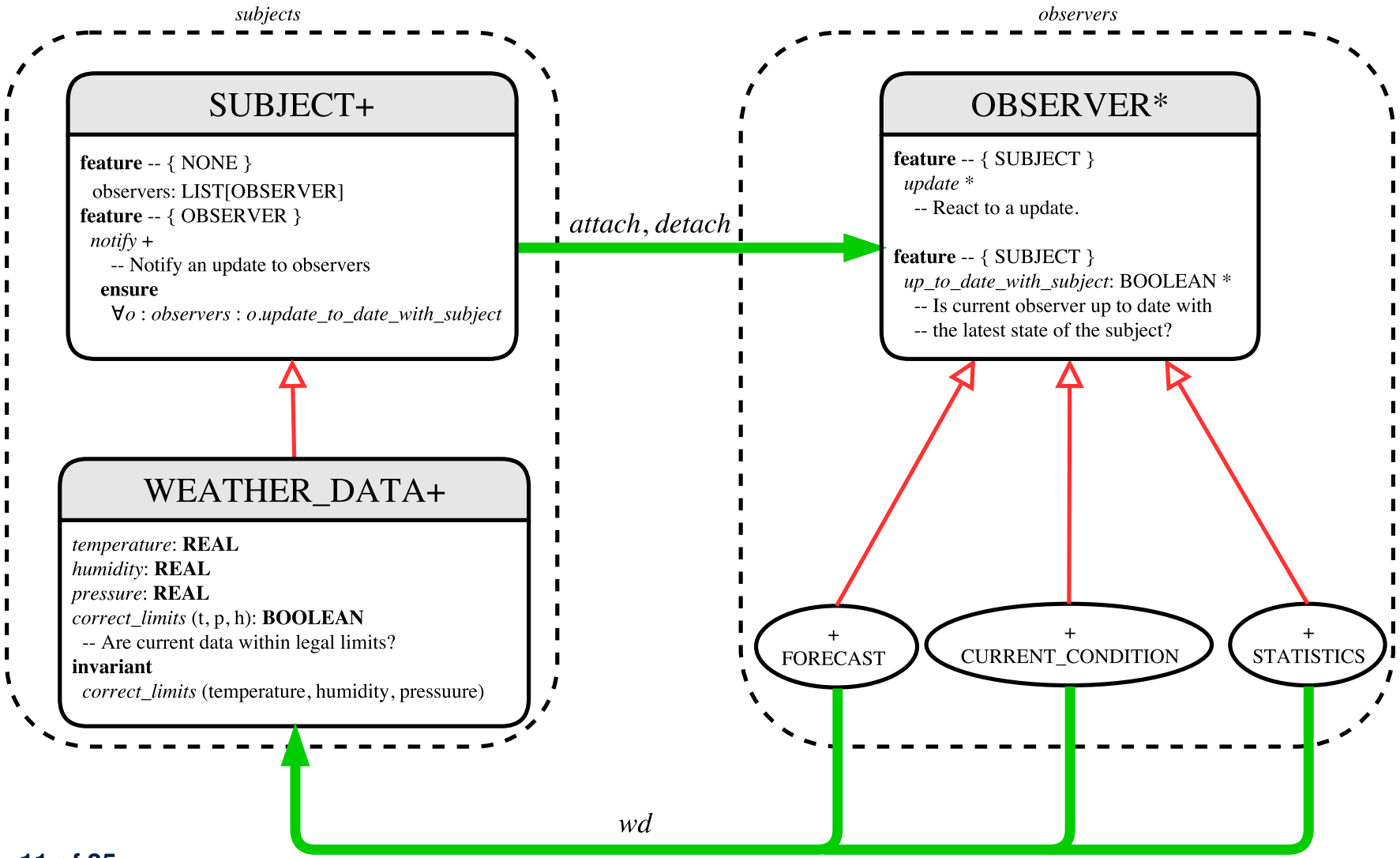
    $\Rightarrow$ Updates on the application side occur only *when necessary*.

Observer Pattern: Architecture



- Observer (publish-subscribe) pattern: *one-to-many* relation.
  - Observers (*subscribers*) are attached to a subject (*publisher*).
  - The subject notify its attached observers about changes.
- Some interchangeable vocabulary:
  - subscribe ≈ attach ≈ register
  - unsubscribe ≈ detach ≈ unregister
  - publish ≈ notify
  - handle ≈ update

Observer Pattern: Weather Station



Implementing the Observer Pattern (1.1)

```
deferred class
  OBSERVER
feature -- To be effected by a descendant
  up_to_date_with_subject: BOOLEAN
     -- Is this observer up to date with its subject?
    deferred
    end

  update
     -- Update the observer's view of 's'
    deferred
    ensure
      up_to_date_with_subject: up_to_date_with_subject
    end
end
```

Each effective descendant class of OBSERVDER should:

- Define what weather data are required to be up-to-date.
- Define how to update the required weather data.

Implementing the Observer Pattern (1.2)

```
class CURRENT_CONDITIONS
inherit OBSERVER
feature -- Commands
  make(a_weather_data: WEATHER_DATA)
    do weather_data := a_weather_data
       weather_data.attach (Current)
    ensure weather_data = a_weather_data
           weather_data.observers.has (Current)
    end
feature -- Queries
  up_to_date_with_subject: BOOLEAN
    ensure then Result = temperature = weather_data.temperature and
                         humidity = weather_data.humidity
  update
    do -- Same as 1st design; Called only on demand
    end
  display
    do -- No need to update; Display contents same as in 1st design
    end
end
```

Implementing the Observer Pattern (1.3)

```
class FORECAST
inherit OBSERVER
feature -- Commands
  make(a_weather_data: WEATHER_DATA)
    do weather_data := a_weather_data
        weather_data.attach (Current)
    ensure weather_data = a_weather_data
           weather_data.observers.has (Current)
    end
feature -- Queries
  up_to_date_with_subject: BOOLEAN
    ensure then
      Result = current_pressure = weather_data.pressure
  update
    do -- Same as 1st design; Called only on demand
    end
  display
    do -- No need to update; Display contents same as in 1st design
    end
end
```

Implementing the Observer Pattern (1.4)

```
class STATISTICS
inherit OBSERVER
feature -- Commands
  make(a_weather_data: WEATHER_DATA)
    do weather_data := a_weather_data
        weather_data.attach (Current)
    ensure weather_data = a_weather_data
           weather_data.observers.has (Current)
    end
feature -- Queries
  up_to_date_with_subject: BOOLEAN
    ensure then
      Result = current_temperature = weather_data.temperature
  update
    do -- Same as 1st design; Called only on demand
    end
  display
    do -- No need to update; Display contents same as in 1st design
    end
end
```

Implementing the Observer Pattern (2.1)

```
class SUBJECT create make
feature -- Attributes
  observers : LIST[OBSERVER]
feature -- Commands
  make
   do create {LINKED_LIST[OBSERVER]} observers.make
   ensure no_observers:  observers.count = 0 end
feature -- Invoked by an OBSERVER
  attach (o: OBSERVER) -- Add 'o' to the observers
    require not_yet_attached: not observers.has (o)
    ensure is_attached: observers.has (o) end
  detach (o: OBSERVER) -- Add 'o' to the observers
    require currently_attached: observers.has (o)
    ensure is_attached: not observers.has (o) end
feature -- invoked by a SUBJECT
  notify -- Notify each attached observer about the update.
    do across observers as cursor loop cursor.item.update end
    ensure all_views_updated:
      across observers as o all o.item.up_to_date_with_subject end
    end
end
```

Implementing the Observer Pattern (2.2)

```
class WEATHER_DATA
inherit SUBJECT  rename make as make_subject end
create make
feature -- data available to observers
  temperature: REAL
  humidity: REAL
  pressure: REAL
  correct_limits(t,p,h: REAL): BOOLEAN
feature -- Initialization
  make (t, p, h: REAL)
    do
      make_subject -- initialize empty observers
      set_measurements (t, p, h)
    end
feature -- Called by weather station
  set_measurements(t, p, h: REAL)
    require correct_limits(t,p,h)
invariant
   correct_limits(temperature, pressure, humidity)
end
```

Implementing the Observer Pattern (3)

```
1  class WEATHER_STATION create make
2  feature -- Attributes
3    cc: CURRENT_CONDITIONS ; fd: FORECAST ; sd: STATISTICS
4    wd: WEATHER_DATA
5  feature -- Commands
6    make
7      do create wd.make (9, 75, 25)
8         create cc.make (wd) ; create fd.make (wd) ; create sd.make(wd)
9
10        wd.set_measurements (15, 60, 30.4)
11        wd.notify
12
13        cc.display ; fd.display ; sd.display
14
15        wd.set_measurements (11, 90, 20)
16        wd.notify
17   end
18 end
```

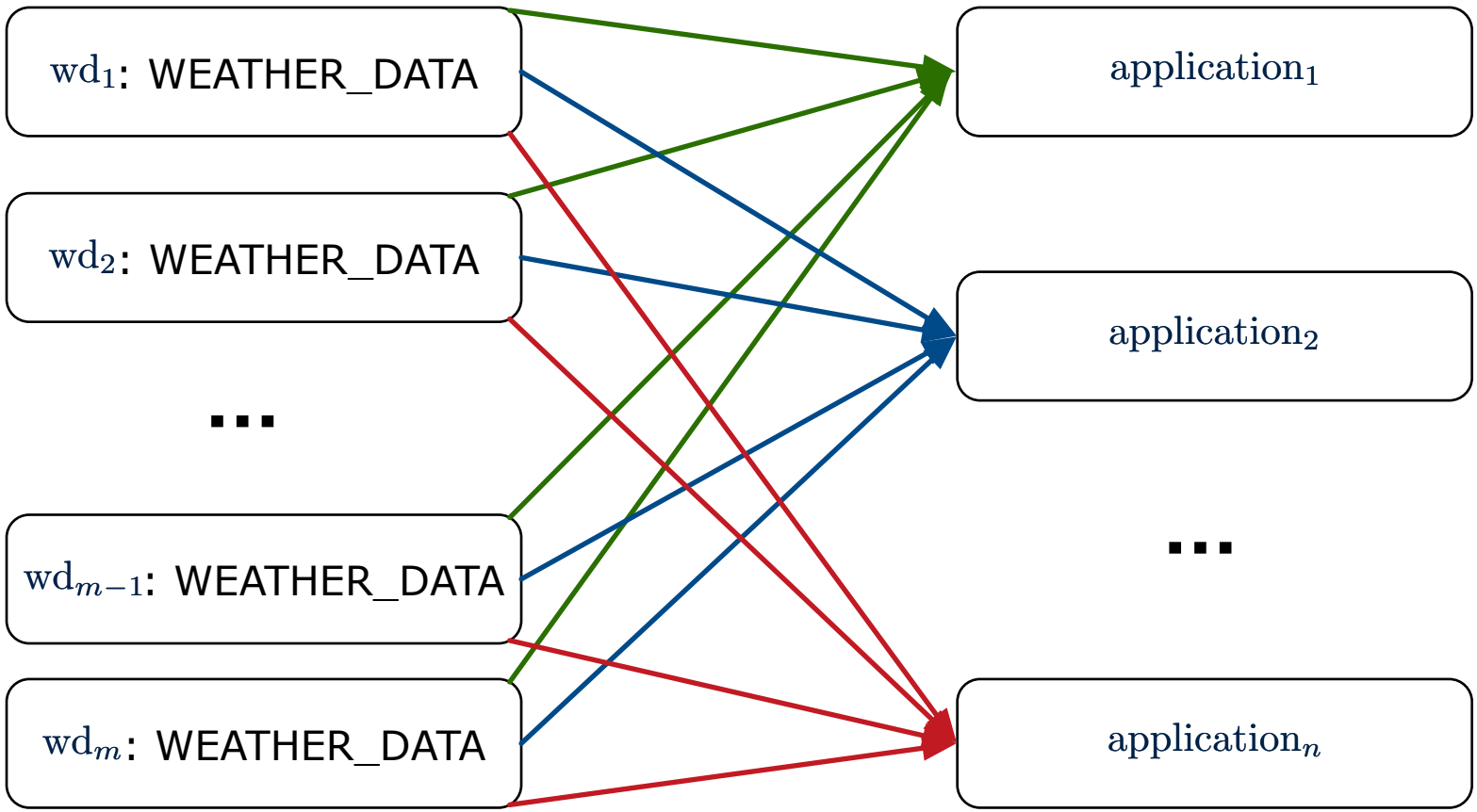
L13: `cc`, `fd`, `sd` make use of "cached" data values.

Observer Pattern: Limitation? (1)

- The *observer design pattern* is a reasonable solution to building a *one-to-many* relationship: one subject (publisher) and multiple observers (subscribers).
- But what if a *many-to-many* relationship is required for the application under development?
  - *Multiple* *weather data* are maintained by weather stations.
  - Each application observes *all* these *weather data*.
  - But, each application still stores the *latest* measure only.
    e.g., the statistics app stores one copy of `temperature`
  - Whenever some weather station updates the `temperature` of its associated *weather data*, all **relevant** subscribed applications (i.e., current conditions, statistics) should update their temperatures.
- How can the observer pattern solve this general problem?
  - Each *weather data* maintains a list of subscribed *applications*.
  - Each *application* is subscribed to *multiple* *weather data*.

Observer Pattern: Limitation? (2)

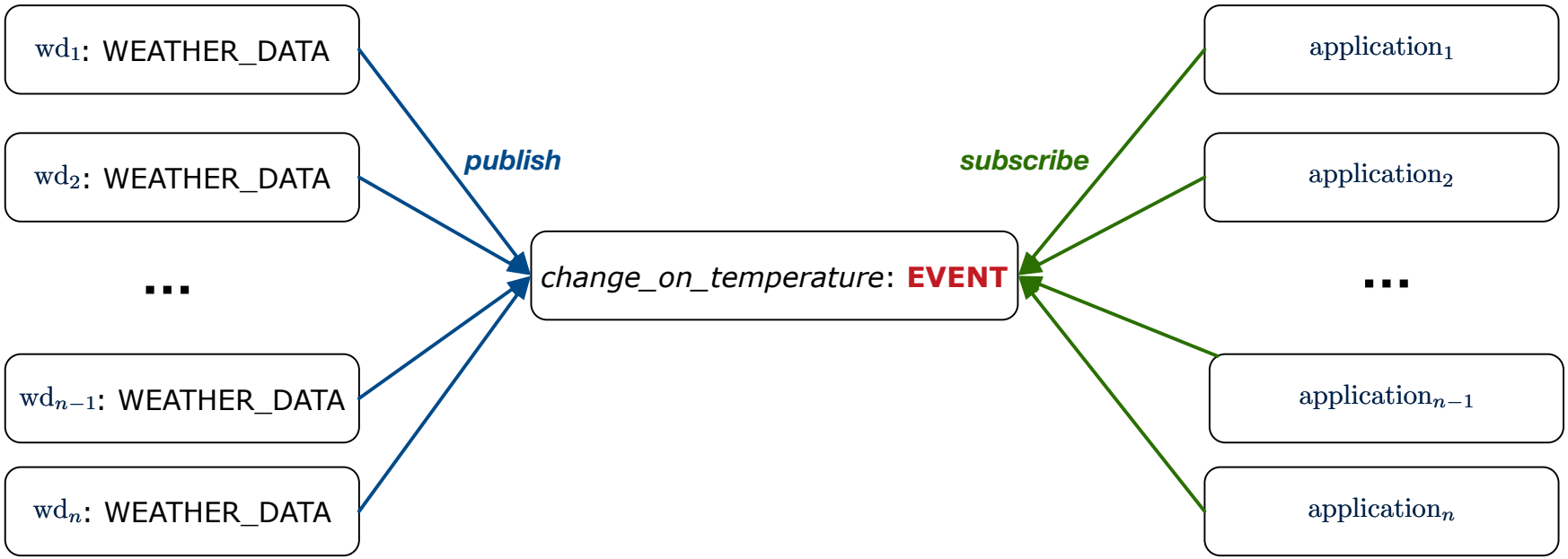
What happens at runtime when building a *many-to-many* relationship using the *observer pattern*?



Graph complexity, with *m* subjects and *n* observers? [$O(m \cdot n)$]

Event-Driven Design (1)

Here is what happens at runtime when building a *many-to-many* relationship using the *event-driven design*.



Graph complexity, with m subjects and n observers? $[O(m+n)]$

Additional cost by adding a new subject? $[O(1)]$

Additional cost by adding a new observer? $[O(1)]$

Additional cost by adding a new event type? $[O(m+n)]$

Event-Driven Design (2)

In an *event-driven design*:

- Each variable being observed (e.g., `temperature`, `humidity`, `pressure`) is called a *monitored variable*.

  e.g., A nuclear power plant (i.e., the ***subject***) has its `temperature` and `pressure` being *monitored* by a shutdown system (i.e., an ***observer***): as soon as values of these *monitored variables* exceed the normal threshold, the SDS will be notified and react by shutting down the plant.

- Each *monitored variable* is declared as an *event*:
  - An ***observer*** is ***attached***/***subscribed*** to the relevant events.
    - `CURRENT_CONDITION` attached to events for `temperature`, `humidity`.
    - `FORECAST` only subscribed to the event for `pressure`.
    - `STATISTICS` only subscribed to the event for `temperature`.
  - A ***subject*** ***notifies***/***publishes*** changes to the relevant events.

Event-Driven Design: Implementation

- Requirements for implementing an *event-driven design* are:
  1. When an ***observer*** object is *subscribed to* an ***event***, it attaches:
     - **1.1** The **reference/pointer** to an update operation
       Such reference/pointer is used for delayed executions.
     - **1.2** Itself (i.e., the **context object** for invoking the update operation)
  2. For the ***subject*** object to *publish* an update to the ***event***, it:
     - **2.1** Iterates through all its observers (or listeners)
     - **2.2** Uses the operation reference/pointer (attached earlier) to update the corresponding observer.
- Both requirements can be satisfied by Eiffel and Java.
- We will compare how an *event-driven design* for the weather station problems is implemented in Eiffel and Java.

  $\Rightarrow$ It's much more convenient to do such design in Eiffel.

Event-Driven Design in Java (1)

```
1  public class Event {
2    Hashtable<Object, MethodHandle> listenersActions;
3    Event() { listenersActions = new Hashtable<>(); }
4    void subscribe(Object listener, MethodHandle action) {
5      listenersActions.put( listener , action );
6    }
7    void publish(Object arg) {
8      for (Object listener : listenersActions.keySet()) {
9        MethodHandle action = listenersActions.get(listener);
10       try {
11         action .invokeWithArguments( listener , arg);
12       } catch (Throwable e) { }
13     }
14   }
15 }
```

- **L5**: Both the delayed `action` reference and its context object (or call target) `listener` are stored into the table.
- **L11**: An invocation is made from retrieved `listener` and `action`.

Event-Driven Design in Java (2)

```
1  public class WeatherData {
2    private double temperature;
3    private double pressure;
4    private double humidity;
5    public WeatherData(double t, double p, double h) {
6      setMeasurements(t, h, p);
7    }
8    public static Event changeOnTemperature = new Event();
9    public static Event changeOnHumidity = new Event();
10   public static Event changeOnPressure = new Event();
11   public void setMeasurements(double t, double h, double p) {
12     temperature = t;
13     humidity = h;
14     pressure = p;
15     changeOnTemperature.publish(temperature);
16     changeOnHumidity.publish(humidity);
17     changeOnPressure.publish(pressure);
18   }
19 }
```

Event-Driven Design in Java (3)

```
1  public class CurrentConditions {
2    private double temperature; private double humidity;
3    public void updateTemperature(double t) { temperature = t; }
4    public void updateHumidity(double h) { humidity = h; }
5    public CurrentConditions() {
6      MethodHandles.Lookup lookup = MethodHandles.lookup();
7      try {
8        MethodHandle ut = lookup.findVirtual(
9          this.getClass(), "updateTemperature",
10         MethodType.methodType(void.class, double.class));
11       WeatherData.changeOnTemperature.subscribe(this, ut);
12       MethodHandle uh = lookup.findVirtual(
13         this.getClass(), "updateHumidity",
14         MethodType.methodType(void.class, double.class));
15       WeatherData.changeOnHumidity.subscribe(this, uh);
16     } catch (Exception e) { e.printStackTrace(); }
17   }
18   public void display() {
19     System.out.println("Temperature: " + temperature);
20     System.out.println("Humidity: " + humidity); } }
```

Event-Driven Design in Java (4)

```
1  public class WeatherStation {
2    public static void main(String[] args) {
3      WeatherData wd = new WeatherData(9, 75, 25);
4      CurrentConditions cc = new CurrentConditions();
5      System.out.println("=======");
6      wd.setMeasurements(15, 60, 30.4);
7      cc.display();
8      System.out.println("=======");
9      wd.setMeasurements(11, 90, 20);
10     cc.display();
11   } }
```

L4 invokes

```
WeatherData.changeOnTemperature.subscribe(
        cc, ``updateTemperature handle'')
```

L6 invokes

```
WeatherData.changeOnTemperature.publish(15)
```

which in turn invokes

```
``updateTemperature handle''.invokeWithArguments(cc, 15)
```

Event-Driven Design in Eiffel (1)

```
1  class EVENT [ARGUMENTS -> TUPLE]
2  create make
3  feature -- Initialization
4    actions: LINKED_LIST[PROCEDURE[ARGUMENTS]]
5    make do create actions.make end
6  feature
7    subscribe (an_action: PROCEDURE[ARGUMENTS])
8      require action_not_already_subscribed: not actions.has(an_action)
9      do actions.extend (an_action)
10     ensure action_subscribed: action.has(an_action) end
11   publish (args: G)
12     do from actions.start until actions.after
13        loop actions.item.call (args) ; actions.forth end
14     end
15 end
```

- **L1** constrains the generic parameter `ARGUMENTS`: any class that instantiates `ARGUMENTS` must be a ***descendant*** of `TUPLE`.
- **L4**: The type ***PROCEDURE*** encapsulates both the context object and the reference/pointer to some update operation.

Event-Driven Design in Eiffel (2)

```
1  class WEATHER_DATA
2  create make
3  feature -- Measurements
4    temperature: REAL ; humidity: REAL ; pressure: REAL
5    correct_limits(t,p,h: REAL): BOOLEAN do ... end
6    make (t, p, h: REAL) do ... end
7  feature -- Event for data changes
8    change_on_temperature : EVENT[TUPLE[REAL]]once create Result end
9    change_on_humidity : EVENT[TUPLE[REAL]]once create Result end
10   change_on_pressure : EVENT[TUPLE[REAL]]once create Result end
11 feature -- Command
12   set_measurements(t, p, h: REAL)
13     require correct_limits(t,p,h)
14     do temperature := t ; pressure := p ; humidity := h
15        change_on_temperature .publish ([t])
16        change_on_humidity .publish ([p])
17        change_on_pressure .publish ([h])
18     end
19 invariant correct_limits(temperature, pressure, humidity) end
```

Event-Driven Design in Eiffel (3)

```
1  class CURRENT_CONDITIONS
2  create make
3  feature -- Initialization
4    make(wd: WEATHER_DATA)
5      do
6        wd.change_on_temperature.subscribe (agent update_temperature)
7        wd.change_on_temperature.subscribe (agent update_humidity)
8      end
9  feature
10   temperature: REAL
11   humidity: REAL
12   update_temperature (t: REAL) do temperature := t end
13   update_humidity (h: REAL) do humidity := h end
14   display do ... end
15 end
```

- **agent** `cmd` retrieves the pointer to `cmd` and its context object.
- **L6** ≈ ... (**agent** ***Current***.update_temperature)
- Contrast **L6** with **L8–11** in Java class `CurrentConditions`.

Event-Driven Design in Eiffel (4)

```
1  class WEATHER_STATION create make
2  feature
3    cc: CURRENT_CONDITIONS
4    make
5      do create wd.make (9, 75, 25)
6         create cc.make (wd)
7         wd.set_measurements (15, 60, 30.4)
8         cc.display
9         wd.set_measurements (11, 90, 20)
10        cc.display
11     end
12 end
```

L6 invokes

```
wd.change_on_temperature.subscribe(
        agent cc.update_temperature)
```

L7 invokes

```
wd.change_on_temperature.publish([15])
```

which in turn invokes `cc.update_temperature(15)`

Event-Driven Design: Eiffel vs. Java

- *Storing observers/listeners of an event*
  - Java, in the `Event` class:

```
Hashtable<Object, MethodHandle> listenersActions;
```

  - Eiffel, in the `EVENT` class:

```
actions: LINKED_LIST[PROCEDURE[ARGUMENTS]]
```

- *Creating and passing function pointers*
  - Java, in the `CurrentConditions` class constructor:

```
MethodHandle ut = lookup.findVirtual(
  this.getClass(), "updateTemperature",
  MethodType.methodType(void.class, double.class));
WeatherData.changeOnTemperature.subscribe(this, ut);
```

  - Eiffel, in the `CURRENT_CONDITIONS` class construction:

```
wd.change_on_temperature.subscribe (agent update_temperature)
```

⇒ Eiffel's type system has been better thought-out for *design*.

Index (1)

Index (2)

Index (3)

Abstractions via Mathematical Models

EECS3311: Software Design
Fall 2017

CHEN-WEI WANG

Motivating Problem: Complete Contracts

- Recall what we learned in the *Complete Contracts* lecture:
  - In *post-condition*, for *each attribute*, specify the relationship between its ***pre-state*** value and its ***post-state*** value.
  - Use the **old** keyword to refer to ***post-state*** values of expressions.
  - For a ***composite***-structured attribute (e.g., arrays, linked-lists, hash-tables, *etc.*), we should specify that after the update:
    1. The intended change is present; **and**
    2. *The rest of the structure is unchanged*.
- Let's now revisit this technique by specifying a *LIFO stack*.

Motivating Problem: LIFO Stack (1)

- Let's consider three different implementation strategies:

Stack Feature	Array	Linked List	
	Strategy 1	Strategy 2	Strategy 3
count	imp.count		
top	imp[imp.count]	imp.first	imp.last
push(g)	imp.force(g, imp.count + 1)	imp.put_font(g)	imp.extend(g)
pop	imp.list.remove_tail (1)	list.start list.remove	imp.finish imp.remove

- Given that all strategies are meant for implementing the ***same ADT***, will they have *identical* contracts?

Motivating Problem: LIFO Stack (2.1)

```
class LIFO_STACK[G] create make
feature {NONE} -- Strategy 1: array
  imp: ARRAY[G]
feature -- Initialization
  make do create imp.make_empty ensure imp.count = 0 end
feature -- Commands
  push(g: G)
    do imp.force(g, imp.count + 1)
    ensure
      changed: imp[count] ~ g
      unchanged: across 1 |..| count - 1 as i all
                   imp[i.item] ~ (old imp.deep_twin)[i.item] end
    end
  pop
    do imp.remove_tail(1)
    ensure
      changed: count = old count - 1
      unchanged: across 1 |..| count as i all
                   imp[i.item] ~ (old imp.deep_twin)[i.item] end
    end
```

Motivating Problem: LIFO Stack (2.2)

```
class LIFO_STACK[G] create make
feature {NONE} -- Strategy 2: linked-list first item as top
  imp: LINKED_LIST[G]
feature -- Initialization
  make do create imp.make ensure imp.count = 0 end
feature -- Commands
  push(g: G)
    do imp.put_front(g)
    ensure
      changed: imp.first ~ g
      unchanged: across 2 |..| count as i all
                   imp[i.item] ~ (old imp.deep_twin)[i.item] end
    end
  pop
    do imp.start ; imp.remove
    ensure
      changed: count = old count - 1
      unchanged: across 1 |..| count as i all
                   imp[i.item] ~ (old imp.deep_twin)[i.item + 1] end
    end
```

Motivating Problem: LIFO Stack (2.3)

```
class LIFO_STACK[G] create make
feature {NONE} -- Strategy 3: linked-list last item as top
  imp: LINKED_LIST[G]
feature -- Initialization
  make do create imp.make ensure imp.count = 0 end
feature -- Commands
  push(g: G)
    do imp.extend(g)
    ensure
      changed: imp.last ~ g
      unchanged: across 1 |..| count - 1 as i all
                   imp[i.item] ~ (old imp.deep_twin)[i.item] end
    end
  pop
    do imp.finish ; imp.remove
    ensure
      changed: count = old count - 1
      unchanged: across 1 |..| count as i all
                   imp[i.item] ~ (old imp.deep_twin)[i.item] end
    end
```

Motivating Problem: LIFO Stack (3)

- *Postconditions* of all 3 versions of stack are *complete*.
  i.e., Not only the new item is ***pushed/popped***, but also the remaining part of the stack is ***unchanged***.
- But they violate the principle of *information hiding*:
  Changing the ***secret***, internal workings of data structures should not affect any existing clients.
- How so?

  The private attribute `imp` is referenced in the *postconditions*, exposing the implementation strategy not relevant to clients:
  - Top of stack may be `imp[count]`, `imp.first`, or `imp.last`.
  - Remaining part of stack may be `across 1 |..| count - 1` or `across 2 |..| count`.

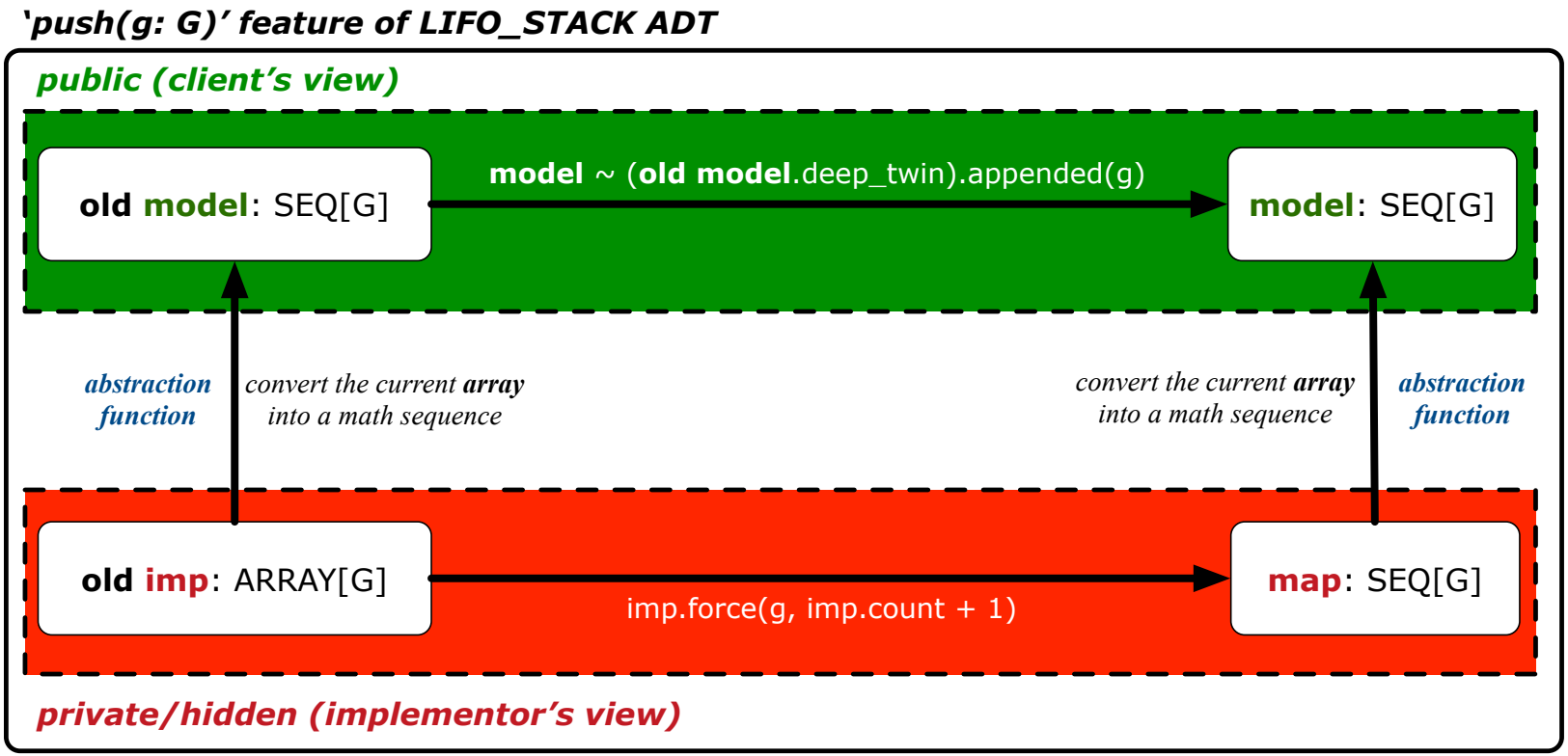
  ⇒ *Changing the implementation strategy* from one to another will also *change the contracts for **all** features*.

  ⇒ This also violates the *Single Choice Principle*.

Implementing an Abstraction Function (1)

```
class LIFO_STACK[G -> attached ANY] create make
feature {NONE} -- Implementation Strategy 1
  imp: ARRAY[G]
feature -- Abstraction function of the stack ADT
  model: SEQ[G]
    do create Result.make_from_array (imp)
    ensure
      counts: imp.count = Result.count
      contents: across 1 |..| Result.count as i all
                  Result[i.item] ~ imp[i.item]
    end
feature -- Commands
  make do create imp.make_empty ensure model.count = 0 end
  push (g: G) do imp.force(g, imp.count + 1)
    ensure pushed: model ~ (old model.deep_twin).appended(g) end
  pop do imp.remove_tail(1)
    ensure popped: model ~ (old model.deep_twin).front end
end
```

Abstracting ADTs as Math Models (1)



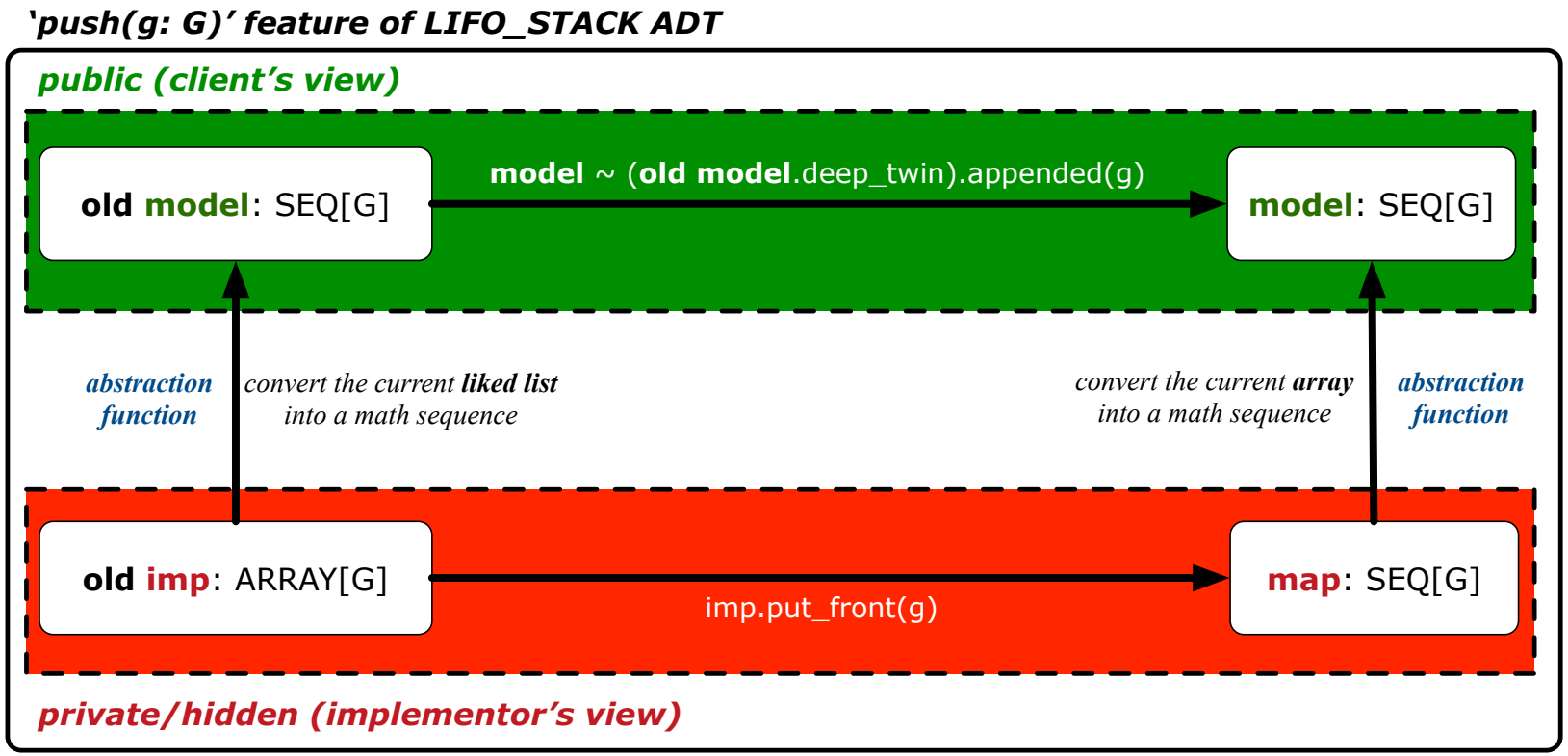
- **Strategy 1** *Abstraction function*: Convert the *implementation array* to its corresponding *model sequence*.
- *Contract* for the `put(g: G)` feature remains the ***same***:

```
model ~ (old model.deep_twin).appended(g)
```

Implementing an Abstraction Function (2)

```
class LIFO_STACK[G -> attached ANY] create make
feature {NONE} -- Implementation Strategy 2 (first as top)
  imp: LINKED_LIST[G]
feature -- Abstraction function of the stack ADT
  model: SEQ[G]
    do create Result.make_empty
       across imp as cursor loop Result.prepend(cursor.item) end
    ensure
      counts: imp.count = Result.count
      contents: across 1 |..| Result.count as i all
                    Result[i.item] ~ imp[count - i.item + 1]
    end
feature -- Commands
  make do create imp.make ensure model.count = 0 end
  push (g: G) do imp.put_front(g)
    ensure pushed: model ~ (old model.deep_twin).appended(g) end
  pop do imp.start ; imp.remove
    ensure popped: model ~ (old model.deep_twin).front end
end
```

Abstracting ADTs as Math Models (2)



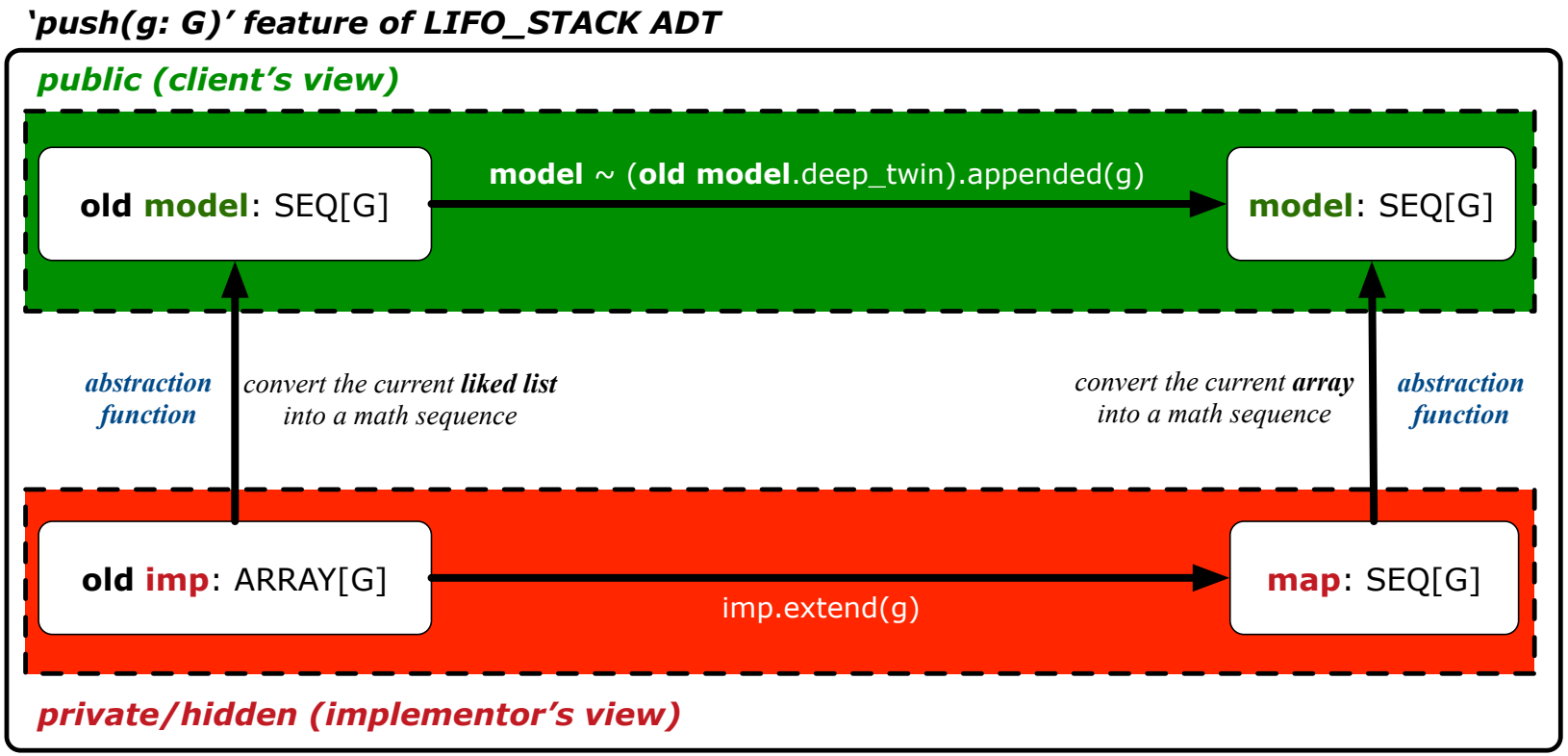
- **Strategy 2** *Abstraction function*: Convert the *implementation list* (first item is top) to its corresponding *model sequence*.
- *Contract* for the `put(g: G)` feature remains the ***same***:

```
model ~ (old model.deep_twin).appended(g)
```

Implementing an Abstraction Function (3)

```
class LIFO_STACK[G -> attached ANY] create make
feature {NONE} -- Implementation Strategy 3 (last as top)
  imp: LINKED_LIST[G]
feature -- Abstraction function of the stack ADT
  model: SEQ[G]
    do create Result.make_empty
       across imp as cursor loop Result.append(cursor.item) end
    ensure
      counts: imp.count = Result.count
      contents: across 1 |..| Result.count as i all
                    Result[i.item] ~ imp[i.item]
    end
feature -- Commands
  make do create imp.make ensure model.count = 0 end
  push (g: G) do imp.extend(g)
    ensure pushed: model ~ (old model.deep_twin).appended(g) end
  pop do imp.finish ; imp.remove
    ensure popped: model ~ (old model.deep_twin).front end
end
```

Abstracting ADTs as Math Models (3)



- **Strategy 3** *Abstraction function*: Convert the *implementation list* (last item is top) to its corresponding *model sequence*.
- *Contract* for the `put(g: G)` feature remains the ***same***:

```
model ~ (old model.deep_twin).appended(g)
```

Solution: Abstracting ADTs as Math Models

- Writing contracts in terms of *implementation attributes* (arrays, LL's, hash tables, *etc.*) violates *information hiding* principle.
- Instead:
  - For each ADT, create an *abstraction* via a ***mathematical model***.
    e.g., Abstract a `LIFO_STACK` as a mathematical sequence.
  - For each ADT, define an *abstraction function* (i.e., a query) whose return type is a kind of ***mathematical model***.
    e.g., Convert *implementation array* to *mathematical sequence*
  - Write contracts in terms of the ***abstract math model***.
    e.g., When pushing an item *g* onto the stack, specify it as appending *g* into its model sequence.
  - Upon *changing the implementation*:
    - **No** change on **what** the abstraction is, hence *no change on contracts*.
    - **Only** change **how** the abstraction is constructed, hence *changes on the body of the abstraction function*.
      e.g., Convert *implementation linked-list* to *mathematical sequence*
      ⇒ The *Single Choice Principle* is obeyed.

Math Review: Set Definitions and Membership

- A *set* is a collection of objects.
  - Objects in a set are called its *elements* or *members*.
  - *Order* in which elements are arranged does not matter.
  - An element can appear *at most once* in the set.
- We may define a set using:
  - *Set Enumeration*: Explicitly list all members in a set.
    e.g., $\{1, 3, 5, 7, 9\}$
  - *Set Comprehension*: Implicitly specify the condition that all members satisfy.
    e.g., $\{x \mid 1 \leq x \leq 10 \wedge x \texttt{ is an odd number}\}$
- An empty set (denoted as $\{\}$ or $\varnothing$) has no members.
- We may check if an element is a *member* of a set:
  e.g., $5 \in \{1, 3, 5, 7, 9\}$ [*true*]
  e.g., $4 \notin \{x \mid x \leq 1 \leq 10, x \texttt{ is an odd number}\}$ [*true*]
- The number of elements in a set is called its *cardinality*.
  e.g., $|\varnothing| = 0$, $|\{x \mid x \leq 1 \leq 10, x \texttt{ is an odd number}\}| = 5$

Math Review: Set Relations

Given two sets S_1 and S_2:

- S_1 is a *subset* of S_2 if every member of S_1 is a member of S_2.

$$S_1 \subseteq S_2 \iff (\forall x \bullet x \in S_1 \Rightarrow x \in S_2)$$

- S_1 and S_2 are *equal* iff they are the subset of each other.

$$S_1 = S_2 \iff S_1 \subseteq S_2 \land S_2 \subseteq S_1$$

- S_1 is a *proper subset* of S_2 if it is a strictly smaller subset.

$$S_1 \subset S_2 \iff S_1 \subseteq S_2 \land |S1| < |S2|$$

Math Review: Set Operations

Given two sets S_1 and S_2:

- *Union* of S_1 and S_2 is a set whose members are in either.

$$S_1 \cup S_2 = \{x \mid x \in S_1 \vee x \in S_2\}$$

- *Intersection* of S_1 and S_2 is a set whose members are in both.

$$S_1 \cap S_2 = \{x \mid x \in S_1 \wedge x \in S_2\}$$

- *Difference* of S_1 and S_2 is a set whose members are in S_1 but not S_2.

$$S_1 \setminus S_2 = \{x \mid x \in S_1 \wedge x \notin S_2\}$$

Math Review: Power Sets

The *power set* of a set S is a *set* of all S' *subsets*.

$$\mathbb{P}(S) = \{s \mid s \subseteq S\}$$

The power set contains subsets of *cardinalities* 0, 1, 2, ..., $|S|$.

e.g., $\mathbb{P}(\{1,2,3\})$ is a set of sets, where each member set s has cardinality 0, 1, 2, or 3:

$$\left\{ \begin{array}{l} \varnothing, \\ \{1\},\ \{2\},\ \{3\}, \\ \{1,2\},\ \{2,3\},\ \{3,1\}, \\ \{1,2,3\} \end{array} \right\}$$

Math Review: Set of Tuples

Given n sets $S_1, S_2, \ldots, S_n$, a *cross product* of theses sets is a set of n-tuples.

Each *n-tuple* $(e_1, e_2, \ldots, e_n)$ contains n elements, each of which a member of the corresponding set.

$$S_1 \times S_2 \times \cdots \times S_n = \{(e_1, e_2, \ldots, e_n) \mid e_i \in S_i \wedge 1 \leq i \leq n\}$$

e.g., $\{a, b\} \times \{2, 4\} \times \{\$, \&\}$ is a set of triples:

$$\begin{aligned} & \{a, b\} \times \{2, 4\} \times \{\$, \&\} \\ = \; & \{ (e_1, e_2, e_3) \mid e_1 \in \{a, b\} \wedge e_2 \in \{2, 4\} \wedge e_3 \in \{\$, \&\} \} \\ = \; & \{(a, 2, \$), (a, 2, \&), (a, 4, \$), (a, 4, \&), \\ & \; (b, 2, \$), (b, 2, \&), (b, 4, \$), (b, 4, \&)\} \end{aligned}$$

Math Models: Relations (1)

- A *relation* is a collection of mappings, each being an *ordered pair* that maps a member of set S to a member of set T.
  e.g., Say $S = \{1, 2, 3\}$ and $T = \{a, b\}$
  - $\varnothing$ is an empty relation.
  - $S \times T$ is a relation (say r_1) that maps from each member of S to each member in T: $\{(1, a), (1, b), (2, a), (2, b), (3, a), (3, b)\}$
  - $\{(x, y) : S \times T \mid x \neq 1\}$ is a relation (say r_2) that maps only some members in S to every member in T: $\{(2, a), (2, b), (3, a), (3, b)\}$.
- Given a relation r:
  - *Domain* of r is the set of S members that r maps from.

    $$\mathrm{dom}(r) = \{s : S \mid (\exists t \bullet (s, t) \in r)\}$$

    e.g., $\mathrm{dom}(r_1) = \{1, 2, 3\}$, $\mathrm{dom}(r_2) = \{2, 3\}$
  - *Range* of r is the set of T members that r maps to.

    $$\mathrm{ran}(r) = \{t : T \mid (\exists s \bullet (s, t) \in r)\}$$

    e.g., $\mathrm{ran}(r_1) = \{a, b\} = \mathrm{ran}(r_2)$

Math Models: Relations (2)

- We use the power set operator to express the set of *all possible relations* on S and T:

$$\mathbb{P}(S \times T)$$

- To declare a relation variable r, we use the colon ($:$) symbol to mean *set membership*:

$$r : \mathbb{P}(S \times T)$$

- Or alternatively, we write:

$$r : S \leftrightarrow T$$

where the set $S \leftrightarrow T$ is synonymous to the set $\mathbb{P}(S \times T)$

Math Models: Relations (3.1)

Say $r = \{(a,1),(b,2),(c,3),(a,4),(b,5),(c,6),(d,1),(e,2),(f,3)\}$

- r.***domain***: set of first-elements from r
  - r.**domain** = $\{\ d \mid (d,r) \in r\ \}$
  - e.g., r.**domain** = $\{a,b,c,d,e,f\}$
- r.***range***: set of second-elements from r
  - r.**range** = $\{\ r \mid (d,r) \in r\ \}$
  - e.g., r.**range** = $\{1,2,3,4,5,6\}$
- r.***inverse***: a relation like r except elements are in reverse order
  - r.**inverse** = $\{\ (r,d) \mid (d,r) \in r\ \}$
  - e.g., r.**inverse** = $\{(1,a),(2,b),(3,c),(4,a),(5,b),(6,c),(1,d),(2,e),(3,f)\}$

Math Models: Relations (3.2)

Say $r = \{(a,1),(b,2),(c,3),(a,4),(b,5),(c,6),(d,1),(e,2),(f,3)\}$

- r.***domain_restricted***(ds) : sub-relation of r with domain *ds*.
  - r.**domain_restricted**(ds) = $\{ (d,r) \mid (d,r) \in r \wedge d \in ds \}$
  - e.g., r.**domain_restricted**({a, b}) = $\{(\mathbf{a},1),(\mathbf{b},2),(\mathbf{a},4),(\mathbf{b},5)\}$
- r.***domain_subtracted***(ds) : sub-relation of r with domain not *ds*.
  - r.**domain_subtracted**(ds) = $\{ (d,r) \mid (d,r) \in r \wedge d \notin ds \}$
  - e.g., r.**domain_subtracted**({a, b}) = $\{(\mathbf{c},6),(\mathbf{d},1),(\mathbf{e},2),(\mathbf{f},3)\}$
- r.***range_restricted***(rs) : sub-relation of r with range *rs*.
  - r.**range_restricted**(rs) = $\{ (d,r) \mid (d,r) \in r \wedge r \in rs \}$
  - e.g., r.**range_restricted**({1, 2}) = $\{(a,\mathbf{1}),(b,\mathbf{2}),(d,\mathbf{1}),(e,\mathbf{2})\}$
- r.***range_subtracted***(ds) : sub-relation of r with range not *ds*.
  - r.**range_subtracted**(rs) = $\{ (d,r) \mid (d,r) \in r \wedge r \notin rs \}$
  - e.g., r.**range_subtracted**({1, 2}) = $\{(c,\mathbf{3}),(a,\mathbf{4}),(b,\mathbf{5}),(c,\mathbf{6})\}$

Math Models: Relations (3.3)

Say $r = \{(a,1),(b,2),(c,3),(a,4),(b,5),(c,6),(d,1),(e,2),(f,3)\}$

- r.***overridden***(t) : a relation which agrees on *r* outside domain of *t*.*domain*, and agrees on *t* within domain of *t*.*domain*
  - r.**overridden**(t) $t \cup r.\textbf{domain\_subtracted}(t.\textbf{domain})$
  - $$r.\textbf{overridden}(\underbrace{\{(a,3),(c,4)\}}_{t}) = \underbrace{\{(a,3),(c,4)\}}_{t} \cup \underbrace{\{(b,2),(b,5),(d,1),(e,2),(f,3)\}}_{r.\textbf{domain\_subtracted}(\underbrace{t.\textbf{domain}}_{\{a,c\}})} = \{(a,3),(c,4),(b,2),(b,5),(d,1),(e,2),(f,3)\}$$

Math Review: Functions (1)

A *function* f on sets S and T is a *specialized form* of relation: it is forbidden for a member of S to map to more than one members of T.

$$\forall s : S; t_1 : T; t_2 : T \bullet (s, t_1) \in f \land (s, t_2) \in f \Rightarrow t_1 = t_2$$

e.g., Say $S = \{1, 2, 3\}$ and $T = \{a, b\}$, which of the following relations are also functions?

- $S \times T$ [No]
- $(S \times T) - \{(x, y) \mid (x, y) \in S \times T \land x = 1\}$ [No]
- $\{(1, a), (2, b), (3, a)\}$ [Yes]
- $\{(1, a), (2, b)\}$ [Yes]

Math Review: Functions (2)

- We use *set comprehension* to express the set of all possible functions on S and T as those relations that satisfy the *functional property*:

$$\{r : S \leftrightarrow T \mid (\forall s : S; t_1 : T; t_2 : T \bullet (s, t_1) \in r \land (s, t_2) \in r \Rightarrow t_1 = t_2)\}$$

- This set (of possible functions) is a subset of the set (of possible relations): $\mathbb{P}(S \times T)$ and $S \leftrightarrow T$.
- We abbreviate this set of possible functions as $S \to T$ and use it to declare a function variable f:

$$f : S \to T$$

Math Review: Functions (3.1)

Given a function $f : S \to T$:

- f is *injective* (or an injection) if f does not map a member of S to more than one members of T.

$$f \texttt{ is injective} \iff (\forall s_1 : S; s_2 : S; t : T \bullet (s_1, t) \in r \land (s_2, t) \in r \Rightarrow s_1 = s_2)$$

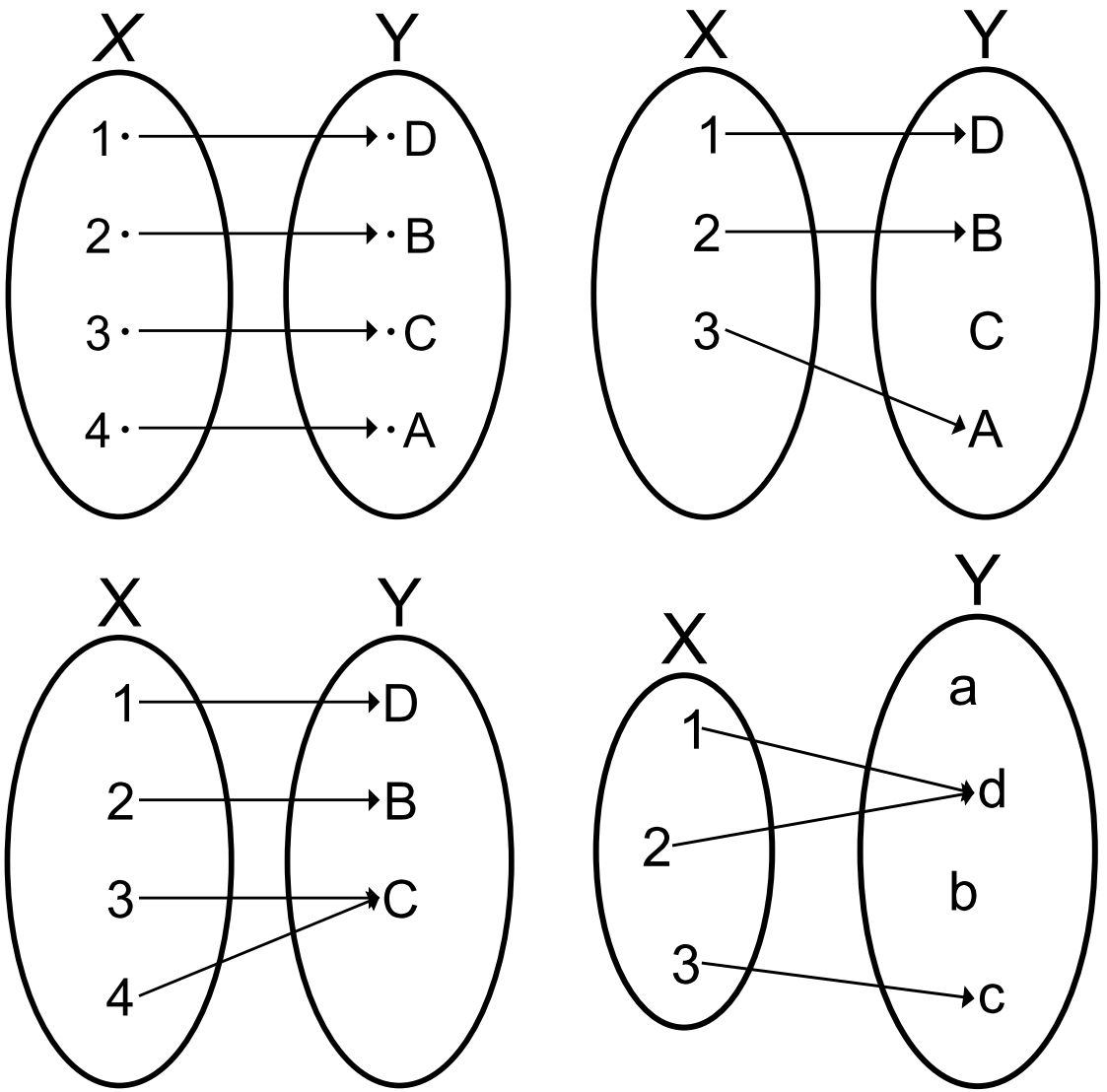
  e.g., Considering an array as a function from integers to objects, being injective means that the array does not contain any duplicates.

- f is *surjective* (or a surjection) if f maps to all members of T.

$$f \texttt{ is surjective} \iff \mathrm{ran}(f) = T$$

- f is *bijective* (or a bijection) if f is both injective and surjective.

Math Review: Functions (3.2)



Math Models: Command-Query Separation

Command	*Query*
`domain_restrict`	`domain_restrict`**`ed`**
`domain_restrict_by`	`domain_restrict`**`ed`**`_by`
`domain_subtract`	`domain_subtract`**`ed`**
`domain_subtract_by`	`domain_subtract`**`ed`**`_by`
`range_restrict`	`range_restrict`**`ed`**
`range_restrict_by`	`range_restrict`**`ed`**`_by`
`range_subtract`	`range_subtract`**`ed`**
`range_subtract_by`	`range_subtract`**`ed`**`_by`
`override`	`overrid`**`den`**
`override_by`	`overrid`**`den`**`_by`

Say $r = \{(a,1),(b,2),(c,3),(a,4),(b,5),(c,6),(d,1),(e,2),(f,3)\}$

- ***Commands*** modify the context relation objects.

  `r.`***`domain_restrict`***`({a})` changes r to $\{(a,1),(a,4)\}$

- ***Queries*** return new relations without modifying context objects.

  `r.`***`domain_restricted`***`({a})` returns $\{(a,1),(a,4)\}$ with r untouched

Math Models: Example Test

```
test_rel: BOOLEAN
  local
    r, t: REL[STRING, INTEGER]
    ds: SET[STRING]
  do
    create r.make_from_tuple_array (
      <<["a", 1], ["b", 2], ["c", 3],
        ["a", 4], ["b", 5], ["c", 6],
        ["d", 1], ["e", 2], ["f", 3]>>)
    create ds.make_from_array (<<"a">>)
    -- r is not changed by the query 'domain_subtracted'
    t := r.domain_subtracted (ds)
    Result :=
      t /~ r and not t.domain.has ("a") and r.domain.has ("a")
    check Result end
    -- r is changed by the command 'domain_subtract'
    r.domain_subtract (ds)
    Result :=
      t ~ r and not t.domain.has ("a") and not r.domain.has ("a")
end
```

Math Models: Command or Query

- Use the state-changing ***commands*** to define the body of an *abstraction function*.

```
class LIFO_STACK[G -> attached ANY] create make
feature {NONE} -- Implementation
  imp: LINKED_LIST[G]
feature -- Abstraction function of the stack ADT
  model: SEQ[G]
    do create Result.make_empty
      across imp as cursor loop Result.append(cursor.item) end
    end
```

- Use the side-effect-free ***queries*** to write contracts.

```
class LIFO_STACK[G -> attached ANY] create make
feature -- Abstraction function of the stack ADT
  model: SEQ[G]
feature -- Commands
  push (g: G)
    ensure pushed: model ~ (old model.deep_twin).appended(g) end
```

Beyond this lecture . . .

Familiarize yourself with the features of classes `REL` and `SET` for the exam.

Index (1)

Index (2)

Index (3)

Eiffel Testing Framework (ETF): Acceptance Tests via Abstract User Interface

EECS3311: Software Design
Fall 2017

CHEN-WEI WANG

Bank ATM

The ATM application has a variety of *concrete* user interfaces.

Separation of Concerns

- The (Concrete) User Interface
  - The executable of your application *hides* the implementing classes and features.
  - Users typically interact with your application via some GUI.
    e.g., web app, mobile app, or desktop app
- The *Business Logic (Model)*
  - When you develop your application software, you implement classes and features.
    e.g., How the bank stores, processes, retrieves information about accounts and transactions

In practice:

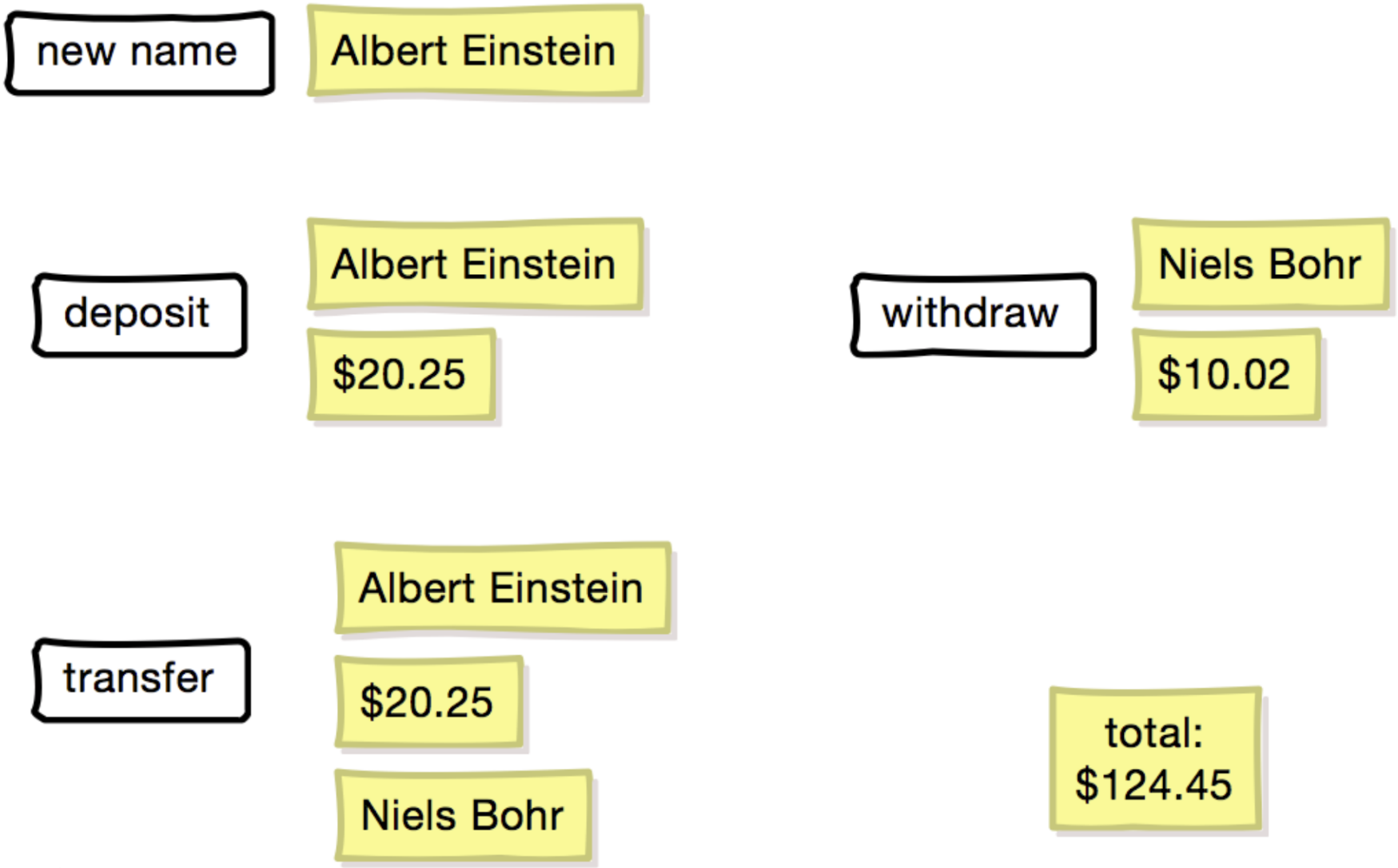
- You need to test your software as if it were a real app ***way before*** dedicating to the design of an actual GUI.
- The model should be *independent* of the View, Input and Output.

Prototyping System with Abstract UI

- For you to quickly prototype a working system, you do not need to spend time on developing a fancy GUI.
- The *Eiffel Testing Framework (ETF)* allows you to:
  - Focus on developing the business model;
  - Test your business model as if it were a real app.
- In ETF, observable interactions with the application GUI (e.g., “button clicks”) are ***abstracted*** as monitored events.

Events	Features
interactions	computations
external	internal
observable	hidden
acceptance tests	unit tests
users, customers	programmers, developers

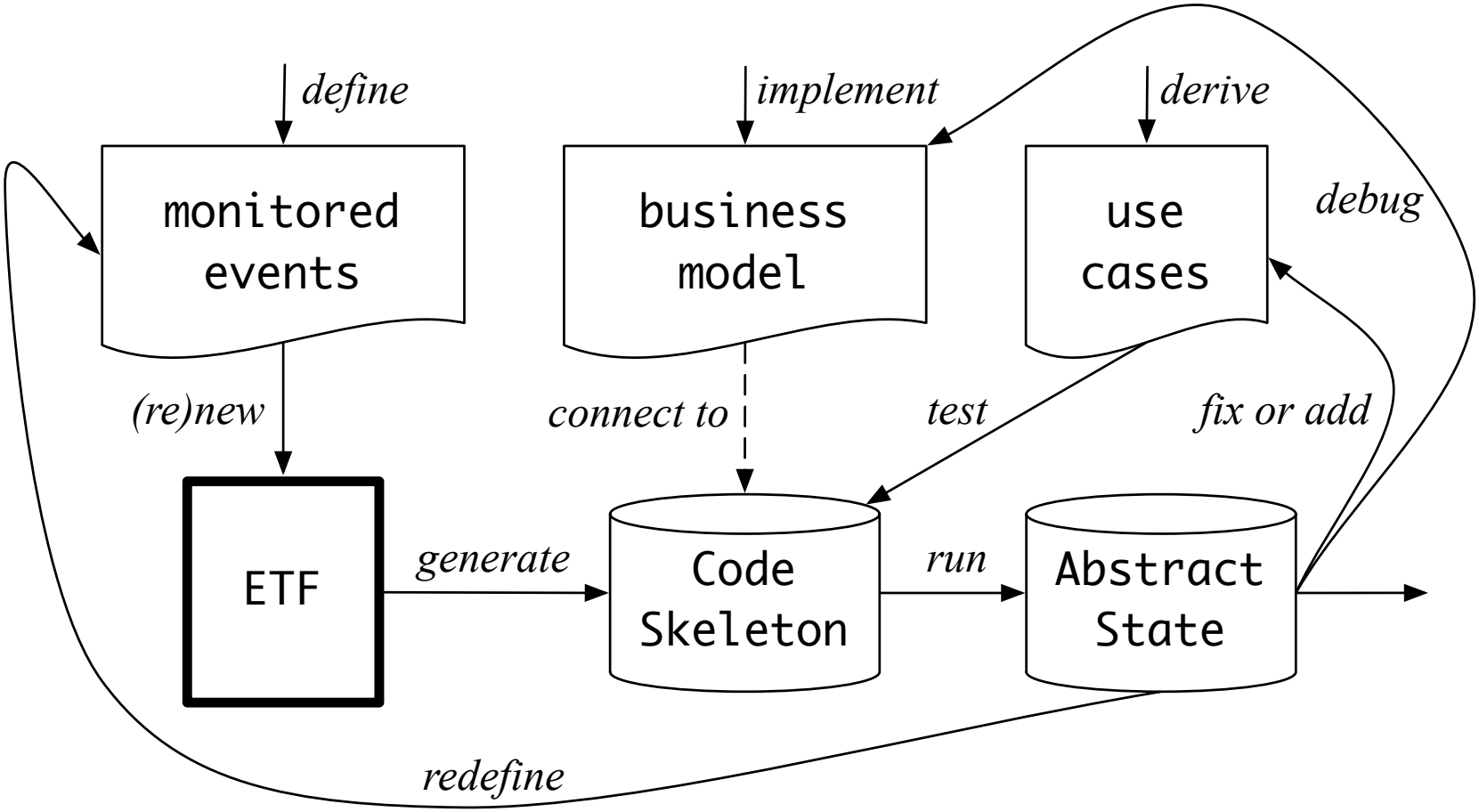
Abstract Events: Bank ATM



ETF in a Nutshell

- *Eiffel Testing Framework (ETF)* facilitates engineers to write and execute ***input-output-based acceptance tests***.
  - ***Inputs*** are specified as traces of events (or sequences).
  - The ***boundary*** of the system under development (SUD) is defined by declaring the list of input events that might occur.
  - ***Outputs*** (from executing events in the input trace) are by default logged onto the terminal, and their formats may be customized.
- An executable ETF that is tailored for the SUD can already be generated, using these event declarations (documented documented in a plain text file), with a default *business model*.
- Once the *business model* is implemented, there is only a small number of steps to follow for the developers to connect it to the generated ETF.
- Once connected, developers may re-run all use cases and observe if the expected state effects take place.

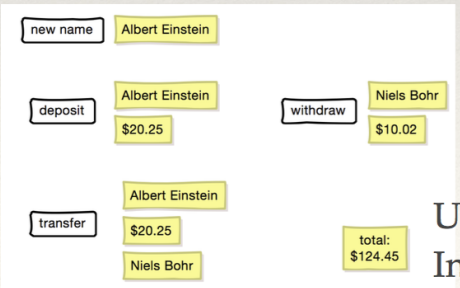
Workflow: Develop-Connect-Test



ETF: Abstract User Interface

Input Grammar

```
system bank
type NAME = STRING

new(name1: NAME)
  -- create a new bank account for "id"

deposit(name1: NAME; amount: VALUE)
  -- deposit "amount" into the account of "id"

withdraw(name1: NAME; amount: VALUE)
  -- withdraw "amount" from the account of "id"

transfer(name1: NAME; name2: NAME; amount: VALUE)
  -- transfer "amount" from "id1" to "id2"
```



User Interface

```
%bank -b at1.txt
 init
->new("Steve")
name: Steve, balance: 0.00
->new("Bill")
name: Bill, balance: 0.00
name: Steve, balance: 0.00
->deposit("Steve",520)
name: Bill, balance: 0.00
name: Steve, balance: 520.00
->new("Pam")
name: Bill, balance: 0.00
name: Pam, balance: 0.00
name: Steve, balance: 520.00
->deposit("Bill",100)
name: Bill, balance: 100.00
name: Pam, balance: 0.00
name: Steve, balance: 520.00
->withdraw("Steve",20)
name: Bill, balance: 100.00
name: Pam, balance: 0.00
name: Steve, balance: 500.00
```

ETF: Generating a New Project

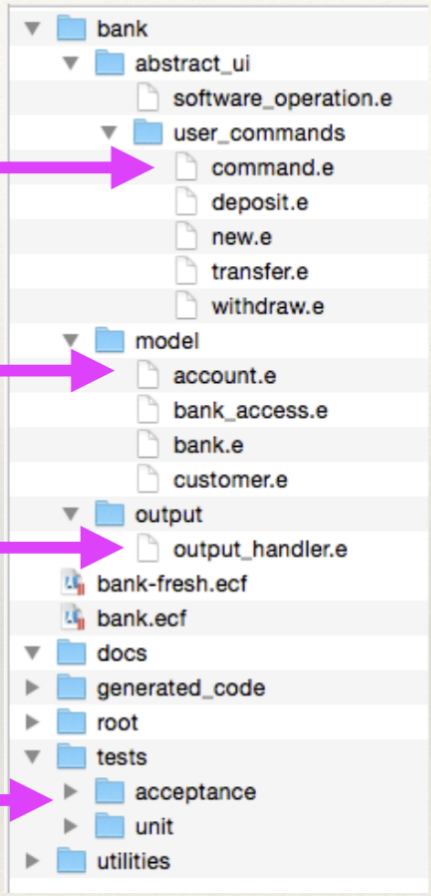
etf -new bank.input.txt <directory>

User Input
(from command line)

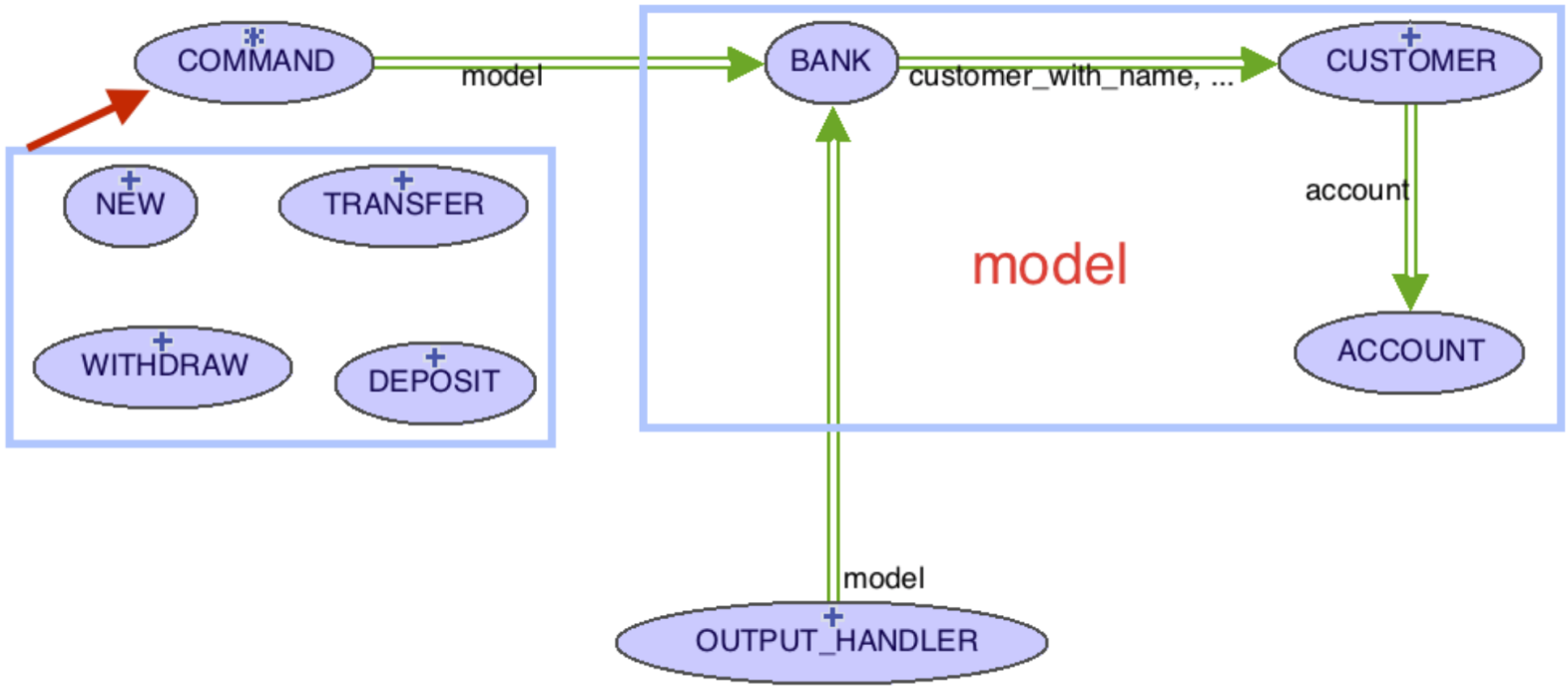
Model
(business logic)

Output

Unit Tests
Acceptance Tests



ETF: Architecture



- Classes in the `model` cluster are hidden from the users.
- All commands reference to the same model (bank) instance.
- When a user's request is made:
  - A ***command object*** of the corresponding type is created, which invokes relevant feature(s) in the `model` cluster.
  - Updates to the model are published to the output handler.

ETF: Input Errors

```
class
    ETF_DEPOSIT
inherit
    ETF_DEPOSIT_INTERFACE
        redefine deposit end
create
    make
feature -- command
    deposit(id: STRING ; amount: REAL_64)
        do
            if not model.has_user (id) then
                -- Set some error message
            elseif not amount <= model.get_balance (id)  then
                -- Set some other error message
            else
                -- perform some update on the model state
                model.deposit (id, amount)
            end
            -- Publish model update
            etf_cmd_container.on_change.notify ([Current])
        end
end
```

Index (1)

Program Correctness

OOSC2 Chapter 11

EECS3311: Software Design
Fall 2017

CHEN-WEI WANG

Weak vs. Strong Assertions

- Describe each assertion as ***a set of satisfying value***.
  $x > 3$ has satisfying values $\{4, 5, 6, 7, \ldots\}$
  $x > 4$ has satisfying values $\{5, 6, 7, \ldots\}$
- An assertion p is ***stronger*** than an assertion q if p's set of satisfying values is a subset of q's set of satisfying values.
  - Logically speaking, p being stronger than q (or, q being weaker than p) means $p \Rightarrow q$.
  - e.g., $x > 4 \Rightarrow x > 3$
- What's the weakest assertion? [**TRUE**]
- What's the strongest assertion? [**FALSE**]
- In *Design by Contract*:
  - A weaker *invariant* has more acceptable object states
    e.g., *balance* > 0 vs. *balance* > 100 as an invariant for `ACCOUNT`
  - A weaker *precondition* has more acceptable input values
  - A weaker *postcondition* has more acceptable output values

Motivating Examples (1)

Is this feature correct?

```
class FOO
  i: INTEGER
  increment_by_9
    require
      i > 3
    do
      i := i + 9
    ensure
      i > 13
    end
end
```

Q: Is $i > 3$ is too weak or too strong?

A: Too weak

$\because$ assertion $i > 3$ allows value 4 which would fail postcondition.

Motivating Examples (2)

Is this feature correct?

```
class FOO
  i: INTEGER
  increment_by_9
    require
      i > 5
    do
      i := i + 9
    ensure
      i > 13
    end
end
```

Q: Is $i > 5$ too weak or too strong?

A: Maybe too strong

$\because$ assertion $i > 5$ disallows 5 which would not fail postcondition.

Whether 5 should be allowed depends on the requirements.

Software Correctness

- Correctness is a *relative* notion:

  consistency of *implementation* with respect to *specification*.

  $\Rightarrow$ This assumes there is a specification!
- We introduce a formal and systematic way for formalizing a program **S** and its *specification* (pre-condition ***Q*** and post-condition ***R***) as a *Boolean predicate*: $\boxed{\{Q\}\ \mathtt{S}\ \{R\}}$
  - e.g., $\{i > 3\}$ `i := i + 9` $\{i > 13\}$
  - e.g., $\{i > 5\}$ `i := i + 9` $\{i > 13\}$
  - If $\boxed{\{Q\}\ \mathtt{S}\ \{R\}}$ **can** be proved **TRUE**, then the **S** is correct.

    e.g., $\{i > 5\}$ `i := i + 9` $\{i > 13\}$ can be proved TRUE.
  - If $\boxed{\{Q\}\ \mathtt{S}\ \{R\}}$ **cannot** be proved **TRUE**, then the **S** is incorrect.

    e.g., $\{i > 3\}$ `i := i + 9` $\{i > 13\}$ cannot be proved TRUE.

Hoare Logic

- Consider a program **S** with precondition ***Q*** and postcondition ***R***.
  - $\{Q\}\ S\ \{R\}$ is a *correctness predicate* for program **S**
  - $\{Q\}\ S\ \{R\}$ is TRUE if program **S** starts executing in a state satisfying the precondition ***Q***, and then:
    (a) The program **S** terminates.
    (b) Given that program **S** terminates, then it terminates in a state satisfying the postcondition ***R***.
- Separation of concerns

  (a) requires a proof of *termination*.

  (b) requires a proof of ***partial*** *correctness*.

  Proofs of (a) + (b) imply ***total*** *correctness*.

Hoare Logic and Software Correctness

Consider the *contract view* of a feature *f* (whose body of implementation is **S**) as a Hoare Triple:

$$\{Q\}\ S\ \{R\}$$

Q is the *precondition* of *f*.
S is the implementation of *f*.
R is the *postcondition* of *f*.

- $\{true\}\ S\ \{R\}$
  All input values are valid [Most-user friendly]
- $\{false\}\ S\ \{R\}$
  All input values are invalid [Most useless for clients]
- $\{Q\}\ S\ \{true\}$
  All output values are valid [Most risky for clients; Easiest for suppliers]
- $\{Q\}\ S\ \{false\}$
  All output values are invalid [Most challenging coding task]
- $\{true\}\ S\ \{true\}$
  All inputs/outputs are valid (No contracts) [Least informative]

Hoare Logic A Simple Example

Given $\{??\}n := n + 9\{n > 13\}$:

- $\boxed{n > 4}$ is the *weakest precondition (wp)* for the given implementation (n := n + 9) to start and establish the postcondition ($n > 13$).
- Any precondition that is ***equal to or stronger than*** the *wp* ($n > 4$) will result in a correct program.

  e.g., $\{n > 5\}n := n + 9\{n > 13\}$ can be proved **TRUE**.
- Any precondition that is ***weaker than*** the *wp* ($n > 4$) will result in an incorrect program.

  e.g., $\{n > 3\}n := n + 9\{n > 13\}$ cannot be proved **TRUE**.

  Counterexample: $n = 4$ satisfies precondition $n > 3$ but the output $n = 13$ fails postcondition $n > 13$.

Proof of Hoare Triple using *wp*

$$\{Q\}\ S\ \{R\} \equiv Q \Rightarrow wp(S, R)$$

- $wp(S, R)$ is the *weakest precondition for S to establish R*.
- S can be:
  - Assignments (`x := y`)
  - Alternations (**`if`** ... **`then`** ... **`else`** ... **`end`**)
  - Sequential compositions (S_1 `;` S_2)
  - Loops (**`from`** ... **`until`** ... **`loop`** ... **`end`**)
- We now show how to calculate the *wp* for the above programming constructs.

Denoting New and Old Values

In the *postcondition*, for a program variable x:

- We write $\boxed{x_0}$ to denote its ***pre-state (old)*** value.
- We write $\boxed{x}$ to denote its ***post-state (new)*** value.

  Implicitly, in the *precondition*, all program variables have their ***pre-state*** values.

e.g., $\{b_0 > a\}$ `b := b - a` $\{b = b_0 - a\}$

- Notice that:
  - We don't write b_0 in preconditions
    ∵ All variables are pre-state values in preconditions
  - We don't write b_0 in program
    ∵ there might be *multiple intermediate values* of a variable due to sequential composition

wp Rule: Assignments (1)

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

$\boldsymbol{R}[x := e]$ means to substitute all *free occurrences* of variable x in postcondition $\boldsymbol{R}$ by expression e.

wp Rule: Assignments (2)

How do we prove $\{Q\}\ \mathtt{x}\ :=\ \mathtt{e}\ \{R\}$?

$$\{Q\}\ \mathtt{x}\ :=\ \mathtt{e}\ \{R\} \iff Q \Rightarrow \underbrace{R[x := e]}_{wp(\mathtt{x}\ :=\ \mathtt{e}, R)}$$

wp Rule: Assignments (3) Exercise

What is the weakest precondition for a program `x := x + 1` to establish the postcondition $x > x_0$?

$$\{??\}\ \texttt{x := x + 1}\ \{x > x_0\}$$

For the above Hoare triple to be ***TRUE***, it must be that $?? \Rightarrow wp(\texttt{x := x + 1}, x > x_0)$.

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

Any precondition is OK. ***False*** is valid but not useful.

wp Rule: Assignments (4) Exercise

What is the weakest precondition for a program `x := x + 1` to establish the postcondition $x > x_0$?

$$\{??\}\ \texttt{x := x + 1}\ \{x = 23\}$$

For the above Hoare triple to be ***TRUE***, it must be that $?? \Rightarrow wp(\texttt{x := x + 1}, x = 23)$.

$$\begin{aligned}
& wp(\texttt{x := x + 1}, x = 23) \\
= \quad & \{\textit{Rule of wp: Assignments}\} \\
& x = 23[x := x_0 + 1] \\
= \quad & \{\textit{Replacing } x \textit{ by } x_0 + 1\} \\
& x_0 + 1 = 23 \\
= \quad & \{\textit{arithmetic}\} \\
& x_0 = 22
\end{aligned}$$

Any precondition weaker than $x = 22$ is not OK.

wp Rule: Alternations (1)

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

The *wp* of an alternation is such that ***all branches*** are able to establish the postcondition ***R***.

wp Rule: Alternations (2)

How do we prove that $\{Q\}$ `if` B `then` S_1 `else` S_2 `end` $\{R\}$?

$\{Q\}$
`if` B `then`
  $\{Q \wedge B\}\ S_1\ \{R\}$
`else`
  $\{Q \wedge \neg B\}\ S_2\ \{R\}$
`end`
$\{R\}$

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

wp Rule: Alternations (3) Exercise

Is this program correct?

$\{x > 0 \wedge y > 0\}$
```
if x > y then
  bigger := x ; smaller := y
else
  bigger := y ; smaller := x
end
```
$\{bigger \geq smaller\}$

$$\left(\begin{array}{l} \{(x > 0 \wedge y > 0) \wedge (x > y)\} \\ \quad \texttt{bigger := x ; smaller := y} \\ \{bigger \geq smaller\} \end{array} \right)$$
$$\wedge$$
$$\left(\begin{array}{l} \{(x > 0 \wedge y > 0) \wedge \neg(x > y)\} \\ \quad \texttt{bigger := y ; smaller := x} \\ \{bigger \geq smaller\} \end{array} \right)$$

wp Rule: Sequential Composition (1)

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

The *wp* of a sequential composition is such that the first phase establishes the *wp* for the second phase to establish the postcondition ***R***.

wp Rule: Sequential Composition (2)

How do we prove $\{Q\}\ S_1\ ;\ S_2\ \{R\}$?

$$\{Q\}\ S_1\ ;\ S_2\ \{R\} \iff Q \Rightarrow \underbrace{wp(S_1, wp(S_2, R))}_{wp(S_1\ ;\ S_2, R)}$$

wp Rule: Sequential Composition (3) Exercise

Is { ***True*** } `tmp := x; x := y; y := tmp` { $x > y$ } correct?

If and only if ***True*** $\Rightarrow$ *wp*(`tmp := x ; x := y ; y := tmp`, $x > y$)

wp(`tmp := x ;` `x := y ; y := tmp`, $x > y$)

= {*wp rule for seq. comp.*}

wp(`tmp := x`, *wp*(`x := y ;` `y := tmp`, $x > y$))

= {*wp rule for seq. comp.*}

wp(`tmp := x`, *wp*(`x := y`, *wp*(`y := tmp`, $x >$ `y`)))

= {*wp rule for assignment*}

wp(`tmp := x`, *wp*(`x := y`, `x` $> tmp$))

= {*wp rule for assignment*}

wp(`tmp := x`, $y >$ `tmp`)

= {*wp rule for assignment*}

$y > x$

$\because$ ***True*** $\Rightarrow y > x$ does not hold in general.

$\therefore$ The above program is not correct.

Loops

- A loop is a way to compute a certain result by *successive approximations*.

  e.g. computing the maximum value of an array of integers
- Loops are needed and powerful
- But loops *very hard* to get right:
  - Infinite loops [termination]
  - “off-by-one” error [partial correctness]
  - Improper handling of borderline cases [partial correctness]
  - Not establishing the desired condition [partial correctness]

Loops: Binary Search

BS1

```
from
        i := 1; j := n
until i = j loop
        m := (i + j) // 2
        if t @ m <= x then
                i := m
        else
                j := m
        end
end
Result := (x = t @ i)
```

BS2

```
from
        i := 1; j := n; found := false
until i = j and not found loop
        m := (i + j) // 2
        if t @ m < x then
                i := m + 1
        elseif t @ m = x then
                found := true
        else
                j := m – 1
        end
end
Result := found
```

BS3

```
from
i := 0; j := n
until i = j loop
        m := (i + j + 1) // 2
        if t @ m <= x then
                i := m + 1
        else
                j := m
        end
end
if i >= 1 and i <= n then
        Result := (x = t @ i)
else
        Result := false
end
```

BS4

```
from
        i := 0; j := n + 1
until i = j loop
        m := (i + j) // 2
        if t @ m <= x then
                i := m + 1
        else
                j := m
        end
end
if i >= 1 and i <= n then
        Result := (x = t @ i)
else
        Result := false
end
```

4 implementations for binary search: published, but *wrong*!

See page 381 in *Object Oriented Software Construction*

Correctness of Loops

How do we prove that the following loops are correct?

```
{Q}
from
  S_init
until
  B
loop
  S_body
end
{R}
```

```
{Q}
S_init
while (¬ B) {
  S_body
}
{R}
```

- In case of C/Java, $\boxed{\neg B}$ denotes the ***stay condition***.
- In case of Eiffel, $\boxed{B}$ denotes the ***exit condition***.

  There is native, syntactic support for checking/proving the *total correctness* of loops.

Contracts for Loops: Syntax

```
from
  S_init
invariant
  invariant_tag: I -- Boolean expression for partial correctness
until
  B
loop
  S_body
variant
  variant_tag: V -- Integer expression for termination
end
```

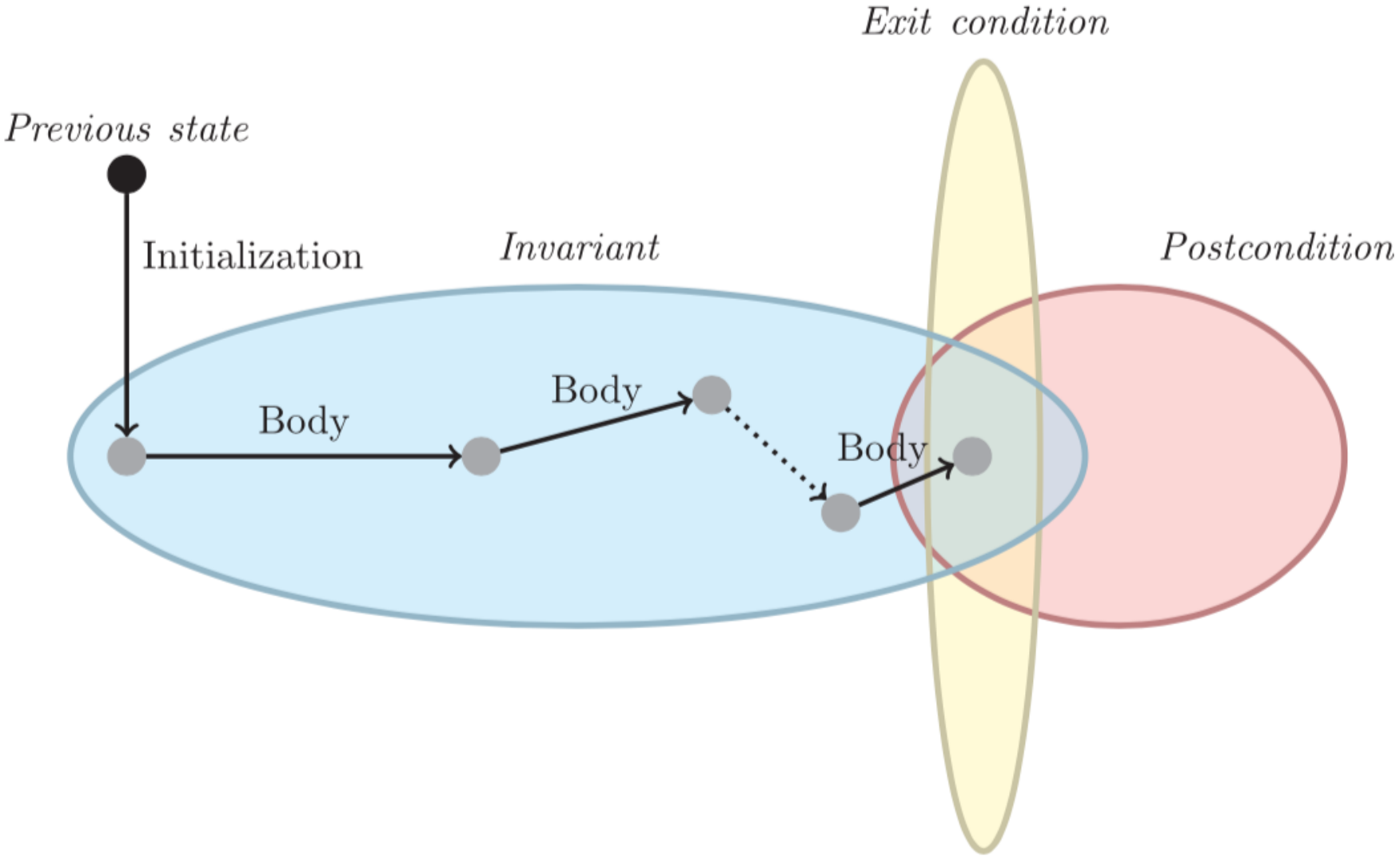
Contracts for Loops

- Use of ***loop invariants (LI)*** and ***loop variants (LV)***.
  - ***Invariants***: Boolean expressions for ***partial*** *correctness*.
    - Typically a special case of the postcondition.
      e.g., Given postcondition " *Result is maximum of the array* ":

      LI can be " *Result is maximum of the* ***part of array scanned so far*** ".
    - Established before the very first iteration.
    - Maintained TRUE after each iteration.
  - ***Variants***: Integer expressions for ***termination***
    - Denotes the *number of iterations remaining*
    - *Decreased* at the end of each subsequent iteration
    - Maintained *positive* in all iterations
    - As soon as value of ***LV*** reaches *zero*, meaning that no more iterations remaining, the loop must exit.
- Remember:

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

Contracts for Loops: Visualization



Digram Source: page 5 in *Loop Invariants: Analysis, Classification, and Examples*

Contracts for Loops: Example 1.1

```
find_max (a: ARRAY [INTEGER]): INTEGER
  local i: INTEGER
  do
    from
      i := a.lower ; Result := a[i]
    invariant
      loop_invariant: -- ∀j | a.lower ≤ j < i • Result ≥ a[j]
        across a.lower |..| (i - 1) as j all Result >= a [j.item] end
    until
      i > a.upper
    loop
      if a [i] > Result then Result := a [i] end
      i := i + 1
    variant
      loop_variant: a.upper - i + 1
    end
  ensure
    correct_result: -- ∀j | a.lower ≤ j ≤ a.upper • Result ≥ a[j]
      across a.lower |..| a.upper as j all Result >= a [j.item]
  end
end
```

Contracts for Loops: Example 1.2

Consider the feature call find_max(⟨⟨20, 10, 40, 30⟩⟩), given:

- ***Loop Invariant***: $\forall j \mid a.lower \leq j < i \bullet Result \geq a[j]$
- ***Loop Variant***: $a.upper - i + 1$
- *Postcondition*: $\forall j \mid a.lower \leq j \leq a.upper \bullet Result \geq a[j]$

After Iteration	i	Result	*LI*	Exit ($i > a.upper$)?	*LV*
Initialization	1	20	✓	×	–
1st	2	20	✓	×	3
2nd	3	20	✓	×	2
3rd	4	40	✓	×	1
4th	5	40	✓	✓	0

Contracts for Loops: Example 2.1

```
find_max (a: ARRAY [INTEGER]): INTEGER
  local i: INTEGER
  do
    from
      i := a.lower ; Result := a[i]
    invariant
      loop_invariant: -- ∀j | a.lower ≤ j ≤ i • Result ≥ a[j]
        across a.lower |..| i as j all Result >= a [j.item] end
    until
      i > a.upper
    loop
      if a [i] > Result then Result := a [i] end
      i := i + 1
    variant
      loop_variant: a.upper - i + 1
    end
  ensure
    correct_result: -- ∀j | a.lower ≤ j ≤ a.upper • Result ≥ a[j]
      across a.lower |..| a.upper as j all Result >= a [j.item]
  end
end
```

Contracts for Loops: Example 2.2

Consider the feature call `find_max( ⟨⟨20, 10, 40, 30⟩⟩ )`, given:

- ***Loop Invariant***: $\forall j \mid a.lower \leq j \leq i \bullet Result \geq a[j]$
- ***Loop Variant***: $a.upper - i + 1$

After Iteration	`i`	`Result`	***LI***	Exit ($i > a.upper$)?	***LV***
Initialization	1	20	✓	×	–
1st	2	20	✓	×	3
2nd	***3***	20	×	–	–

Loop invariant violation at the end of the 2nd iteration:

$$\forall j \mid a.lower \leq j \leq \boxed{3} \bullet \boxed{20} \geq a[j]$$

evaluates to ***false*** $\because 20 \not\geq a[3] = 40$

Contracts for Loops: Example 3.1

```
find_max (a: ARRAY [INTEGER]): INTEGER
  local i: INTEGER
  do
    from
      i := a.lower ; Result := a[i]
    invariant
      loop_invariant: -- ∀j | a.lower ≤ j < i • Result ≥ a[j]
        across a.lower |..| (i - 1) as j all Result >= a [j.item] end
    until
      i > a.upper
    loop
      if a [i] > Result then Result := a [i] end
      i := i + 1
    variant
      loop_variant: a.upper - i
    end
  ensure
    correct_result: -- ∀j | a.lower ≤ j ≤ a.upper • Result ≥ a[j]
      across a.lower |..| a.upper as j all Result >= a [j.item]
  end
end
```

Contracts for Loops: Example 3.2

Consider the feature call `find_max( ⟨⟨20, 10, 40, 30⟩⟩ )`, given:

- ***Loop Invariant***: $\forall j \mid a.lower \leq j < i \bullet Result \geq a[j]$
- ***Loop Variant***: $a.upper - i$

After Iteration	i	Result	*LI*	Exit ($i > a.upper$)?	*LV*
Initialization	1	20	✓	×	–
1st	2	20	✓	×	2
2nd	3	20	✓	×	1
3rd	4	40	✓	×	0
4th	5	40	✓	✓	***-1***

Loop variant violation at the end of the 2nd iteration

$\because a.upper - i = 4 - 5$ evaluates to ***non-zero***.

Contracts for Loops: Exercise

```
class DICTIONARY[V, K]
feature {NONE} -- Implementations
  values: ARRAY[K]
  keys: ARRAY[K]
feature -- Abstraction Function
  model: FUN[K, V]
feature -- Queries
  get_keys(v: V): ITERABLE[K]
    local i: INTEGER; ks: LINKED_LIST[K]
    do
      from i := keys.lower ; create ks.make_empty
      invariant  ??
      until i > keys.upper
      do if values[i] ~ v then ks.extend(keys[i]) end
      end
      Result := ks.new_cursor
    ensure
      result_valid: ∀k | k ∈ Result • model.item(k) ~ v
      no_missing_keys: ∀k | k ∈ model.domain • model.item(k) ~ v ⇒ k ∈ Result
    end
```

Proving Correctness of Loops (1)

```
{Q}     from
            S_init
        invariant
            I
        until
            B
        loop
            S_body
        variant
            V
        end     {R}
```

- A loop is ***partially** correct* if:
  - Given precondition ***Q***, the initialization step S_{init} establishes ***LI*** *I*.
  - At the end of S_{body}, if not yet to exit, ***LI*** *I* is maintained.
  - If ready to exit and ***LI*** *I* maintained, postcondition ***R*** is established.
- A loop *terminates* if:
  - Given ***LI*** *I*, and not yet to exit, S_{body} maintains ***LV*** *V* as positive.
  - Given ***LI*** *I*, and not yet to exit, S_{body} decrements ***LV*** *V*.

Proving Correctness of Loops (2)

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

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

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

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

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

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

$$I \wedge B \Rightarrow R$$

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

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

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

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

Proving Correctness of Loops: Exercise (1.1)

Prove that the following program is correct:

```
find_max (a: ARRAY [INTEGER]): INTEGER
  local i: INTEGER
  do
    from
      i := a.lower ; Result := a[i]
    invariant
      loop_invariant: ∀j | a.lower ≤ j < i • Result ≥ a[j]
    until
      i > a.upper
    loop
      if a [i] > Result then Result := a [i] end
      i := i + 1
    variant
      loop_variant: a.upper − i + 1
    end
  ensure
    correct_result: ∀j | a.lower ≤ j ≤ a.upper • Result ≥ a[j]
  end
end
```

Proving Correctness of Loops: Exercise (1.2)

Prove that each of the following *Hoare Triples* is TRUE.

1. Establishment of Loop Invariant:

$\{\ True\ \}$
```
  i := a.lower
  Result := a[i]
```
$\{\ \forall j \mid a.lower \leq j < i \bullet Result \geq a[j]\ \}$

2. Maintenance of Loop Invariant:

$\{\ \forall j \mid a.lower \leq j < i \bullet Result \geq a[j] \wedge \neg(i > a.upper)\ \}$
```
  if a [i] > Result then Result := a [i] end
  i := i + 1
```
$\{\ \forall j \mid a.lower \leq j < i \bullet Result \geq a[j]\ \}$

3. Establishment of Postcondition upon Termination:

$$\forall j \mid a.lower \leq j < i \bullet Result \geq a[j] \wedge i > a.upper$$
$$\Rightarrow \forall j \mid a.lower \leq j \leq a.upper \bullet Result \geq a[j]$$

Proving Correctness of Loops: Exercise (1.3)

Prove that each of the following *Hoare Triples* is TRUE.

4. Loop Variant Stays Positive Before Exit:

$\{\ \forall j \mid a.lower \le j < i \bullet Result \ge a[j] \wedge \neg(i > a.upper)\ \}$

```
if a [i] > Result then Result := a [i] end
i := i + 1
```

$\{\ a.upper - i + 1 > 0\ \}$

5. Loop Variant Keeps Decrementing before Exit:

$\{\ \forall j \mid a.lower \le j < i \bullet Result \ge a[j] \wedge \neg(i > a.upper)\ \}$

```
if a [i] > Result then Result := a [i] end
i := i + 1
```

$\{\ a.upper - i + 1 < (a.upper - i + 1)_0\ \}$

where $(a.upper - i + 1)_0 \equiv a.upper_0 - i_0 + 1$

Proof Tips (1)

$$\{Q\}\ \mathtt{S}\ \{R\} \Rightarrow \{Q \wedge P\}\ \mathtt{S}\ \{R\}$$

In order to prove $\{Q \wedge P\}\ \mathtt{S}\ \{R\}$, it is sufficient to prove a version with a *weaker* precondition: $\{Q\}\ \mathtt{S}\ \{R\}$.

Proof:

- Assume: $\{Q\}\ \mathtt{S}\ \{R\}$

  It's equivalent to assuming: $\boxed{\mathrm{Q}} \Rightarrow wp(\mathtt{S}, R)$ **(A1)**
- To prove: $\{Q \wedge P\}\ \mathtt{S}\ \{R\}$
  - It's equivalent to proving: $Q \wedge P \Rightarrow wp(\mathtt{S}, R)$
  - Assume: $Q \wedge P$, which implies $\boxed{\mathrm{Q}}$
  - According to **(A1)**, we have $wp(\mathtt{S}, R)$. ■

Proof Tips (2)

When calculating $wp(\texttt{S}, R)$, if either program S or postcondition R involves array indexing, then R should be augmented accordingly.

e.g., Before calculating $wp(\texttt{S}, a[i] > 0)$, augment it as

$$wp(\texttt{S}, a.lower \leq i \leq a.upper \wedge a[i] > 0)$$

e.g., Before calculating $wp(\texttt{x := a[i]}, R)$, augment it as

$$wp(\texttt{x := a[i]}, a.lower \leq i \leq a.upper \wedge R)$$

Index (1)

Index (2)

Index (3)

Wrap-Up

EECS3311: Software Design
Fall 2017

CHEN-WEI WANG

What You Learned

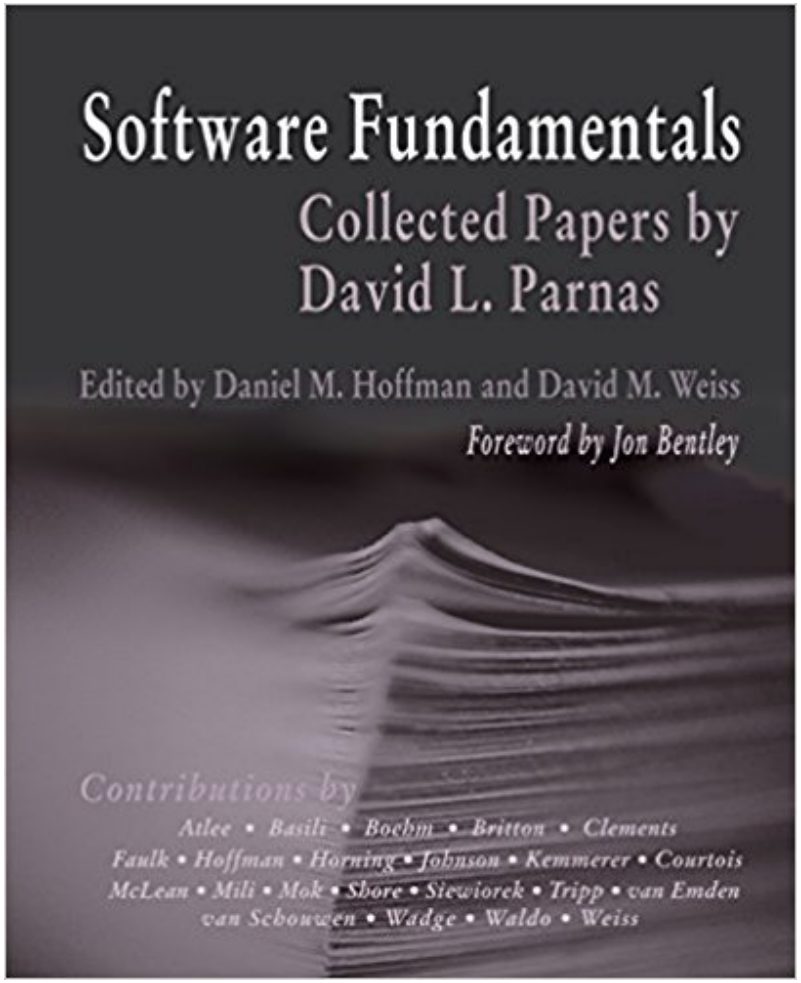
- **Design Principles**:
  - *Abstraction* [contracts, architecture, math models]
    Think ***above the code level***
  - Information Hiding
  - Single Choice Principle
  - Open-Closed Principle
  - Uniform Access Principle
- **Design Patterns**:
  - Singleton
  - Iterator
  - State
  - Composite
  - Visitor
  - Observer
  - Event-Driven Design
  - Undo/Redo, Command [lab 4]
  - Model-View-Controller [project]

Beyond this course. . . (1)

- How do I program in a language not supporting *DbC* natively?
  - Document your *contracts* (e.g., JavaDoc)
  - But, it's critical to ensure (manually) that contracts are ***in sync*** with your latest implementations.
  - Incorporate contracts into your Unit and Regression *tests*
- How do I program in a language without a *math library*?
  - Again, before diving into coding, always start by *thinking above the code level*.
  - Plan ahead how you intend for your system to behaviour at runtime, in terms of interactions among *mathematical objects*.

    A ***mathematical relation***, a formal model of the ***graph data structure***, suffices to cover all the common problems.
  - Use efficient data structures to support the math operations.
  - Document your code with *contracts* specified in terms of the math models.
  - Test!

Beyond this course. . . (2)



- *Software fundamentals: collected papers by David L. Parnas*
- Design Techniques:
  - Tabular Expressions
  - Information Hiding