

Design-by-Contract (DbC)

Readings: OOSC2 Chapters 6, 7, 8, 11



EECS3311 A & E: Software Design
Fall 2020

CHEN-WEI WANG

Learning Objectives

Upon completing this lecture, you are expected to understand:

1. *Design by Contract* (*DbC*): Motivation & Terminology
2. Supporting *DbC* (Java vs. Eiffel):
Preconditions, Postconditions, Class Invariants
3. *Runtime Assertion Checking* of Contracts

Part 1



Design by Contract (DbC): Motivation & Terminology

Motivation: Catching Defects – When?

- To minimize **development costs**, minimize *software defects*.
- Software Development Cycle:

Requirements → *Design* → *Implementation* → Release

Q. Design or Implementation Phase?

Catch defects **as early as possible**.

Design and architecture	Implementation	Integration testing	Customer beta test	Postproduct release
1X*	5X	10X	15X	30X

∴ The cost of fixing defects *increases exponentially* as software progresses through the development lifecycle.

- Discovering *defects* after **release** costs up to 30 times more than catching them in the **design** phase.
- Choice of **design language** for your project is therefore of paramount importance.

Source: **IBM Report**

What this Course Is About (1)

Design

Abstract Data types (ADTs)

Cohesion Principle
Single Choice Principle
Open-Closed Principle

Design Document

Justified Design Decisions

Architecture: Client-Supplier Relation
Architecture: Inheritance Relation
Program to Interface,
Not to Implementation

Modularity: Classes

Design Patterns

(Iterator, Singleton, State, Template,
Composite, Visitor, Strategy,
Observer, Event-Driven Design)
Anti-Patterns

Code Reuse via Inheritance

Substitutability

Polymorphism (esp. *Polymorphic Collections*)
Type Casting
Static Typing, Dynamic Binding
Unit Testing

OOP

Design by Contract (DbC):

Class Invariant, Pre-/Post-condition
Information Hiding Principle
Eiffel Testing Framework (ETF)
Abstraction (via Mathematical Models)

Regression Testing

Acceptance Testing

Void Safety

Generics

Multiple Inheritance

Sub-Contracting

Architectural Design Diagrams

Eiffel

Syntax: Implementation vs. Specification

agent expression, **across** constructs

expanded types, **export** status

Runtime Contract Checking

Debugger

Specification: *Predicates*

Contracts of Loops: Invariant & Variant
Program Correctness

Weakest Precondition (WP)

Hoare Triples

Specification: Higher-Order Functions

Axioms, Lemmas, Theorems

Equational Proofs

Proof by Contradiction (*witness*)

Logic

What this Course Is About (2)

- Focus is *design*
 - **Architecture**: (many) *inter-related* modules
 - **Specification**: *precise* (functional) interface of each module
- For this course, having a prototypical, *working* implementation for your design suffices.
- A later *refinement* into more efficient data structures and algorithms is beyond the scope of this course.

[assumed from EECS2011, EECS3101]

∴ Having a suitable language for **design** matters the most.

Q: Is Java also a “good” **design** language?

A: Let’s first understand what a “good” **design** is.

Terminology: Contract, Client, Supplier

- A **supplier** implements/provides a service (e.g., microwave).
- A **client** uses a service provided by some supplier.
 - The client is required to 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 guaranteed (e.g., a lunch box is heated).
 - The client does not care how the supplier implements it.
- What then are the *benefits* and *obligations* of the two parties?

	<i>benefits</i>	<i>obligations</i>
CLIENT	obtain a service	follow instructions
SUPPLIER	assume instructions followed	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 call 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)**.

Part 2.1



Supporting DbC in Java: Problem & 1st Attempt (No Contracts)

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 Java project archive (a zip file) here:

<https://www.eecs.yorku.ca/~jackie/teaching/lectures/2020/F/ECS3311/codes/DbCIntro.zip>

- Follow this tutorial to learn how to **import** an project archive into your workspace in Eclipse:

<https://youtu.be/h-rad0Zq2qY>

- Follow this tutorial to learn how to **enable** assertions in Eclipse:

<https://youtu.be/OEgRV4a5Dzd>

V1: 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.

V1: 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

V1: 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.

V1: 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.

Supporting DbC in Java: 2nd Attempt (Method Preconditions)

V1: How Should We Improve it? (1)

Preconditions of a method specify the precise circumstances under which that method can be executed.

- Precond. of `divide(int x, int y)`? $[y \neq 0]$
- Precond. of `binSearch(int x, int[] xs)`? $[xs \text{ is sorted}]$
- Precond. of `topoSort(Graph g)`? $[g \text{ is a DAG}]$

V1: How Should We Improve it? (2)

- 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.
 - `topoSort(Graph g)`
throws `NotDAGException` when `g` is *not* directed and acyclic.
- Design your method by specifying the *preconditions* (i.e., *service* conditions for *valid* inputs) it requires, not the *exceptions* (i.e., *error* conditions for *invalid* inputs) for it to fail.
- Create `V2` by adding *exceptional conditions* (an *approximation* of *preconditions*) to the constructor and `withdraw` method of the `Account` class.

V2: Preconditions \approx Exceptions

```
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    }
```

V2: Why Better than V1? (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.*

V2: Why Better than V1? (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        }
20    }
21 }
```

V2: Why Better than V1? (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.
```

- **L8:** 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 *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!!

V2: Why Better than V1? (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        }
20    }
21 }
```

V2: Why Better than V1? (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.
```

- **L8:** 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 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!!

V2: 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 (\neg *preconditions*) appropriate?
- What if amount == balance when calling withdraw?

V2: 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        }
20    }
21 }
```

V2: 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**).
- \therefore **L13** of `AccountV2` should be `balance <= amount`.

***Supporting DbC in Java:
3rd Attempt (Class Invariants)***

V2: 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 `CSMajorStudent`? [`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 **V3** by adding **assertions** to the end of constructor and `withdraw` method of the `Account` class.

V3: Class Invariants \approx Assertions

```
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    }
```


V3: Why Better than V2?

```
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        * V2: 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.*

V3: Why Still Not a Good Design?

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

- From **V2**: *exceptions* encoding **negated preconditions**
- From **V3**: *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 **V4**, where deliberately mistakenly implement `withdraw`.

Supporting DbC in Java: 4th Attempt (Faulty Implementation)

V4: withdraw implemented incorrectly? (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 { /* WRONG 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**.

V4: withdraw implemented incorrectly? (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        * V2: 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: Resulting balance of Jeremy is valid ($150 > 0$), but withdrawal was done via an *mistaken* increase. ⇒ Violation of **REQ2**

***Supporting DbC in Java:
5th Attempt (Method Postconditions)***

V4: 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 `double divide(int x, int y)`?

$$[\text{Result} \times y == x]$$
 - Postcondition of `boolean binSearch(int x, int[] xs)`?

$$[x \in xs \iff \text{Result}]$$
- 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 **V5** by adding *assertions* to the end of `withdraw` method of the `Account` class.

V5: Postconditions \approx Assertions

```
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**.

V5: Why Better than V4?

```
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        * V2: 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.*

Supporting DbC: Java vs. Eiffel

Evolving from V1 to V5

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

- 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.

V5: Contract between Client and Supplier



	<i>benefits</i>	<i>obligations</i>
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

	<i>benefits</i>	<i>obligations</i>
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 -- problematic, why?
        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 -- problematic, why?
    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 Command

```
some_command (x: SOME_TYPE_1; y: SOME_TYPE_2)
  -- 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`

DbC in Eiffel: Anatomy of a Query

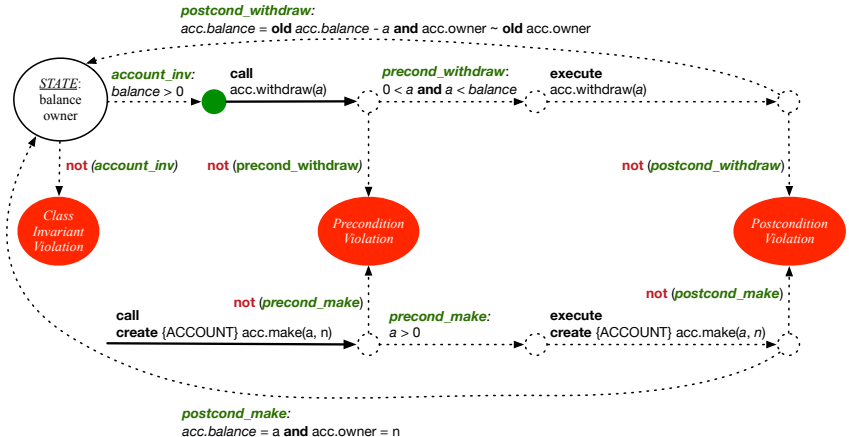
```
some_query (x: SOME_TYPE_1; y: SOME_TYPE_2): SOME_RT
  -- Description of the query.
  require
    -- List of tagged boolean expressions for preconditions
  local
    -- List of local variable declarations
  do
    -- List of instructions as implementation
    Result := ...
  ensure
    -- List of tagged boolean expressions for postconditions
  end
```

- Each query has a predefined variable **Result**.
- Implicitly, you may think of:
 - First line of the query declares **Result: SOME_RT**
 - Last line of the query return the value of **Result**.
⇒ Manipulate **Result** so that its last value is the desired result.

DbC in Eiffel: Runtime Checking

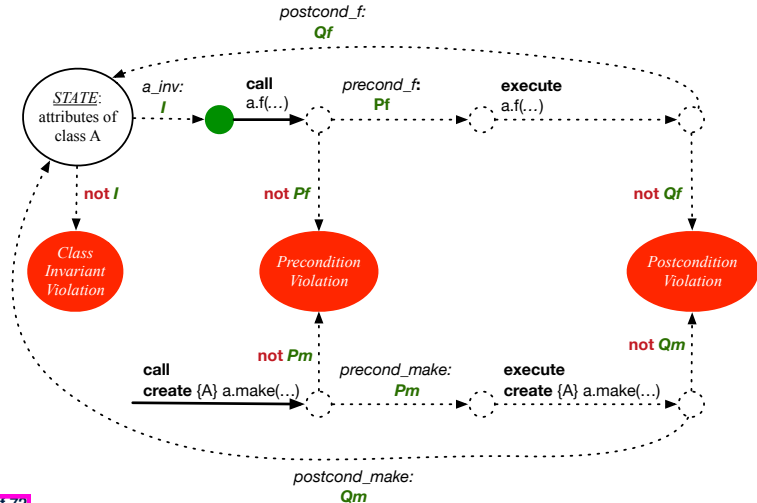
Runtime Monitoring of Contracts (1)

In the specific case of `ACCOUNT` class with creation procedure `make` and command `withdraw`:



Runtime Monitoring of Contracts (2)

In general, class C with creation procedure cp and any feature f :



Runtime Monitoring of Contracts (3)

- 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.

Experimenting Contract Violations in Eiffel



- **Download** the Eiffel project archive (a zip file) here:

<https://www.eecs.vorku.ca/~jackie/teaching/lectures/2020/F/ECS3311/codes/DbCIntroEiffel.zip>

- Unzip and compile the project in Eiffel Studio.
- Follow the in-code comments to re-produce the various **contract violations** and understand from the **stack trace** how they occur.

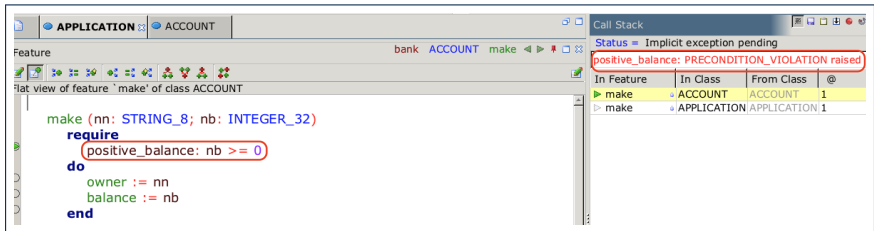
DbC in Eiffel: Precondition Violation (1.1)

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)



Feature bank ACCOUNT make

Flat view of feature 'make' of class ACCOUNT

```
make (nn: STRING_8; nb: INTEGER_32)
  require
    positive_balance: nb >= 0
  do
    owner := nn
    balance := nb
  end
```

Call Stack

Status = Implicit exception pending

positive_balance: PRECONDITION_VIOLATION raised

In Feature	In Class	From Class	@
▶ make	ACCOUNT	ACCOUNT	1
▶ make	APPLICATION	APPLICATION	1

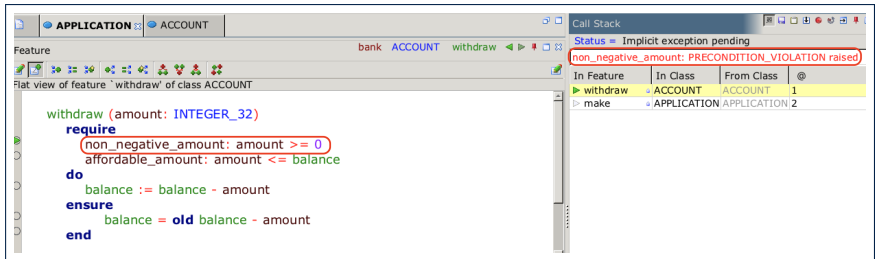
DbC in Eiffel: Precondition Violation (2.1)

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
  create {ACCOUNT} mark.make ("Mark", 100)
  -- A precondition violation with tag "non_negative_amount"
  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)



APPLICATION ACCOUNT

Feature bank ACCOUNT withdraw

Flat view of feature 'withdraw' of class ACCOUNT

```
withdraw (amount: INTEGER_32)
  require
    non_negative_amount: amount >= 0
    affordable_amount: amount <= balance
  do
    balance := balance - amount
  ensure
    balance = old balance - amount
end
```

Call Stack

Status = Implicit exception pending

non_negative_amount: PRECONDITION_VIOLATION raised

In Feature	In Class	From Class	@
withdraw	ACCOUNT	ACCOUNT	1
make	APPLICATION	APPLICATION	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
  create {ACCOUNT} tom.make ("Tom", 100)
  -- A precondition violation with tag "affordable_amount"
  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)



Feature bank ACCOUNT withdraw

Flat view of feature 'withdraw' of class ACCOUNT

```
withdraw (amount: INTEGER_32)
  require
    non_negative_amount: amount >= 0
    affordable_amount: amount <= balance
  do
    balance := balance - amount
  ensure
    balance = old balance - amount
end
```

Call Stack

Status = Implicit exception pending

affordable_amount: PRECONDITION_VIOLATION raised

In Feature	In Class	From Class	@
withdraw	ACCOUNT	ACCOUNT	2
make	APPLICATION	APPLICATION	2

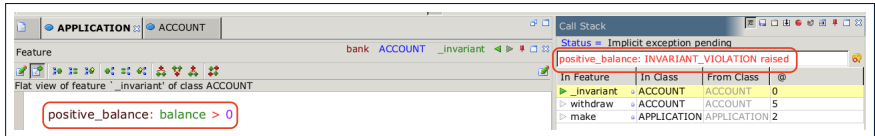
DbC in Eiffel: Class Invariant Violation (4.1)

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
    create {ACCOUNT} tom.make ("Jim", 100)
    jim.withdraw(100)
    -- A class invariant violation with tag "positive_balance"
  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)



APPLICATION ACCOUNT

Feature bank ACCOUNT _invariant

Flat view of feature '_invariant' of class ACCOUNT

```
positive_balance: balance > 0
```

Call Stack

Status = Implicit exception pending

positive_balance: INVARIANT_VIOLATION raised

In Feature	In Class	From Class	@
▶ _invariant	ACCOUNT	ACCOUNT	0
▶ withdraw	ACCOUNT	ACCOUNT	5
▶ make	APPLICATION	APPLICATION	2

DbC in Eiffel: Postcondition 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
    -- Faulty implementation of withdraw in ACCOUNT:
    -- balance := balance + amount
    create {ACCOUNT} jeremy.make ("Jeremy", 100)
    jeremy.withdraw(150)
    -- A postcondition violation with tag "balance_deducted"
  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: Postcondition Violation (5.2)



The screenshot displays the Eiffel IDE interface. The main editor shows the flat view of the 'withdraw' feature of the 'ACCOUNT' class. The code is as follows:

```
affordable_amount: amount <= balance
do
  balance := balance + amount
ensure
  balance_deducted: balance = old balance - amount
end
```

The 'ensure' clause is highlighted with a red circle. The 'Call Stack' window on the right shows the current status: 'Status = Implicit exception pending' and 'balance_deducted: POSTCONDITION_VIOLATION raised'. Below this, a table lists the call stack entries:

In Feature	In Class	From Class	@
withdraw	ACCOUNT	ACCOUNT	4
make	APPLICATION	APPLICATION	2

Beyond this lecture...

1. Review your Lab0 tutorial about how DbC is supported in Eiffel.
2. Explore in Eclipse how *contract* checks are *manually-coded*:

<https://www.eecs.yorku.ca/~jackie/teaching/lectures/2020/F/EECS3311/codes/DbCIntro.zip>

3. Recall the 4th requirement of the bank problem (see [here](#)):

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

Design the header of this `add` method, implement it, and encode proper pre-condition and *post-condition* for it.

Q. What postcondition can you think of? Does it require any skill from EECS1090? What attribute value(s) do you need to manually store in the *pre-state*?

4. 3 short courses which will help your labs and project:
 - Eiffel Syntax: [here](#).
 - Common Syntax/Type Errors in Eiffel: [here](#).
 - Drawing Design Diagrams: [here](#).

Index (1)

Learning Objectives

Part 1

Motivation: Catching Defects – When?

What this Course Is About (1)

What this Course Is About (2)

Terminology: Contract, Client, Supplier

Client, Supplier, Contract in OOP (1)

Client, Supplier, Contract in OOP (2)

What is a Good Design?

Part 2.1

A Simple Problem: Bank Accounts

Index (2)

Playing with the Various Versions in Java

V1: An Account Class

V1: Why Not a Good Design? (1)

V1: Why Not a Good Design? (2)

V1: Why Not a Good Design? (3)

Part 2.2

V1: How Should We Improve it? (1)

V1: How Should We Improve it? (2)

V2: Preconditions \approx Exceptions

V2: Why Better than V1? (1)

V2: Why Better than V1? (2.1)

Index (3)

V2: Why Better than V1? (2.2)

V2: Why Better than V1? (3.1)

V2: Why Better than V1? (3.2)

V2: Why Still Not a Good Design? (1)

V2: Why Still Not a Good Design? (2.1)

V2: Why Still Not a Good Design? (2.2)

Part 2.3

V2: How Should We Improve it?

V3: Class Invariants \approx Assertions

V3: Why Better than V2?

V3: Why Still Not a Good Design?

Index (4)

Part 2.4

V4: withdraw implemented incorrectly? (1)

V4: withdraw implemented incorrectly? (2)

Part 2.5

V4: How Should We Improve it?

V5: Postconditions \approx Assertions

V5: Why Better than V4?

Part 2.6

Evolving from V1 to V5

V5: Contract between Client and Supplier

DbC in Java

Index (5)

DbC in Eiffel: Supplier

DbC in Eiffel: Contract View of Supplier

DbC in Eiffel: Anatomy of a Class

DbC in Eiffel: Anatomy of a Command

DbC in Eiffel: Anatomy of a Query

Part 3

Runtime Monitoring of Contracts (1)

Runtime Monitoring of Contracts (2)

Runtime Monitoring of Contracts (3)

Experimenting Contract Violations in Eiffel

DbC in Eiffel: Precondition Violation (1.1)

Index (6)

DbC in Eiffel: Precondition Violation (1.2)

DbC in Eiffel: Precondition Violation (2.1)

DbC in Eiffel: Precondition Violation (2.2)

DbC in Eiffel: Precondition Violation (3.1)

DbC in Eiffel: Precondition Violation (3.2)

DbC in Eiffel: Class Invariant Violation (4.1)

DbC in Eiffel: Class Invariant Violation (4.2)

DbC in Eiffel: Postcondition Violation (5.1)

DbC in Eiffel: Postcondition Violation (5.2)

Beyond this lecture...

Modularity

Abstract Data Types (ADTs)



EECS3311 A & E: Software Design
Fall 2020

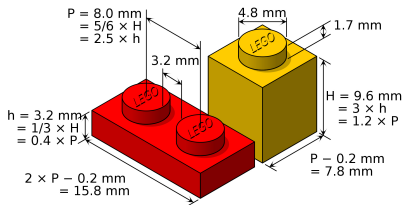
CHEN-WEI WANG

Learning Objectives

Upon completing this lecture, you are expected to understand:

1. Criterion of *Modularity*, Modular Design
2. *Abstract Data Types* (*ADTs*)

Modularity (1): Childhood Activity

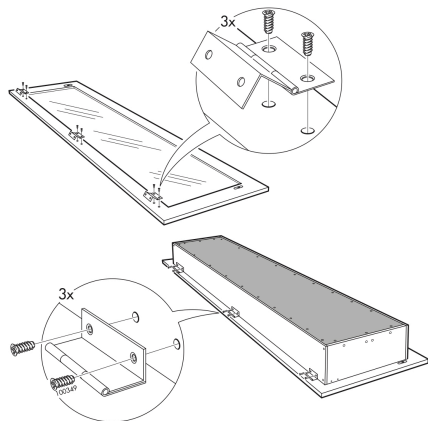
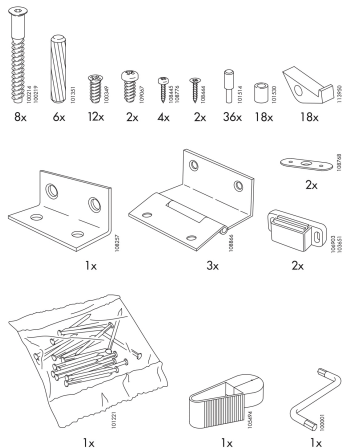


(INTERFACE) SPECIFICATION

(ASSEMBLY) ARCHITECTURE

Sources: <https://commons.wikimedia.org> and <https://www.wish.com>

Modularity (2): Daily Construction



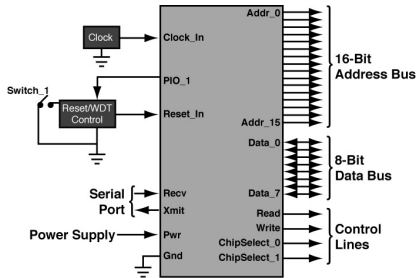
(INTERFACE) SPECIFICATION

(ASSEMBLY) ARCHITECTURE

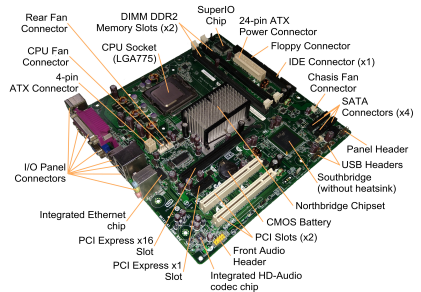
Source: <https://usermanual.wiki/>

Modularity (3): Computer Architecture

Motherboards are built from functioning units (e.g., *CPUs*).



(INTERFACE) SPECIFICATION

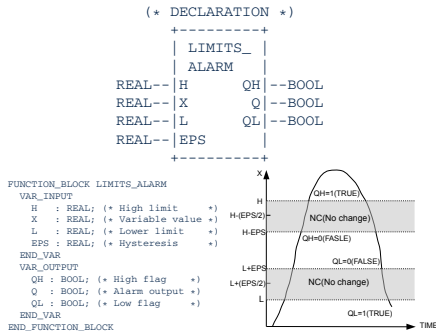


(ASSEMBLY) ARCHITECTURE

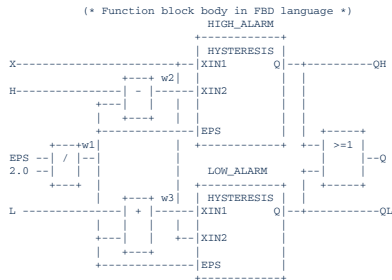
Sources: www.embeddedlinux.org.cn and <https://en.wikipedia.org>

Modularity (4): System Development

Safety-critical systems (e.g., *nuclear shutdown systems*) are built from *function blocks*.



(INTERFACE) SPECIFICATION

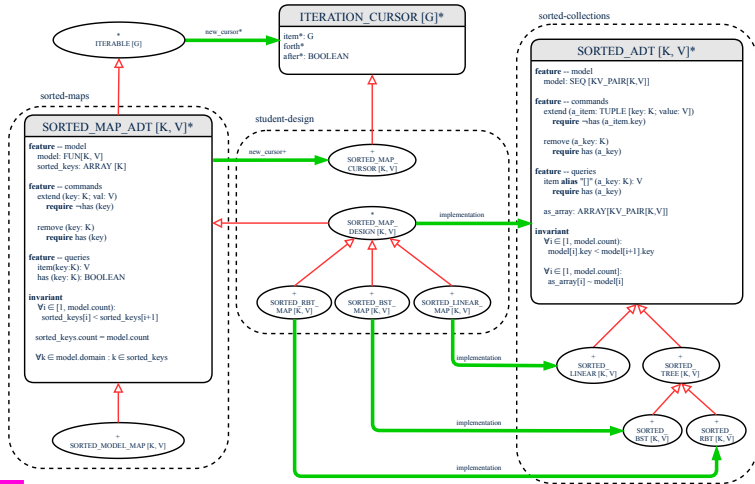


(ASSEMBLY) ARCHITECTURE

Sources: <https://plcopen.org/iec-61131-3>

Modularity (5): Software Design

Software systems are composed of well-specified *classes*.

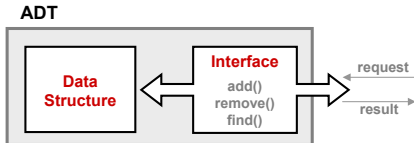


Design Principle: Modularity

- **Modularity** refers to a sound quality of your design:
 1. **Divide** a given complex **problem** into inter-related **sub-problems** via a logical/justifiable functional decomposition.
e.g., In designing a game, solve sub-problems of: 1) rules of the game; 2) actor characterizations; and 3) presentation.
 2. **Specify** each **sub-solution** as a **module** with a clear **interface**: inputs, outputs, and input-output relations.
 - The UNIX principle: Each command does one thing and does it well.
 - In object-oriented design (OOD), each class serves as a module.
 3. **Conquer** original **problem** by assembling **sub-solutions**.
 - In OOD, classes are assembled via client-supplier relations (aggregations or compositions) or inheritance relations.
- A **modular design** satisfies the criterion of modularity and is:
 - **Maintainable**: fix issues by changing the relevant modules only.
 - **Extensible**: introduce new functionalities by adding new modules.
 - **Reusable**: a module may be used in different compositions
- Opposite of modularity: A **superman module** doing everything.

Abstract Data Types (ADTs)

- Given a problem, decompose its solution into **modules**.
- Each **module** implements an **abstract data type (ADT)**:
 - filters out *irrelevant* details
 - contains a list of declared data and *well-specified* 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  $1 \leq i \leq \text{count}$ 
  ensure
    size_unchanged: imp.count = (old imp.twin).count
    item_assigned: imp[i] ~ s
    others_unchanged  $\forall j : 1 \leq j \leq \text{imp.count} : j \neq i \Rightarrow \text{imp}[j] \sim (\text{old imp.twin})[j]$ 

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

invariant
  consistency: imp.count = count
  
```

Beyond this lecture...

1. **Q.** Can you think of more real-life examples of leveraging the power of *modularity*?
2. Visit the Java API page:

<https://docs.oracle.com/javase/8/docs/api>

Visit collection classes which you used in EECS2030 (e.g., `ArrayList`, `HashMap`) and EECS2011.

- Q.** Can you identify/justify some example methods which illustrate that these Java collection classes are not true **ADTs** (i.e., ones with well-specified interfaces)?
3. Contrast with the corresponding library classes and features in EiffelStudio (e.g., `ARRAYED_LIST`, `HASH_TABLE`).
Q. Are these Eiffel features *better specified* w.r.t. obligations/benefits of clients/suppliers?

Index (1)

Learning Objectives

Modularity (1): Childhood Activity

Modularity (2): Daily Construction

Modularity (3): Computer Architecture

Modularity (4): System Development

Modularity (5): Software Design

Design Principle: Modularity

Abstract Data Types (ADTs)

Building ADTs for Reusability

Why Java Interfaces Unacceptable ADTs (1)

Why Java Interfaces Unacceptable ADTs (2)

Index (2)

Why Eiffel Contract Views are ADTs (1)

Why Eiffel Contract Views are ADTs (2)

Beyond this lecture...

Copying Objects

Writing Complete Postconditions



EECS3311 A & E: Software Design
Fall 2020

CHEN-WEI WANG

Learning Objectives



Upon completing this lecture, you are expected to understand:

1. 3 Levels of *Copying Objects*:
Reference vs. Shallow vs. Deep
2. Use of the *old keyword* in Postconditions
3. Writing *Complete Postconditions* using logical quantifications:
Universal (\forall) vs. Existential (\exists)

Part 1

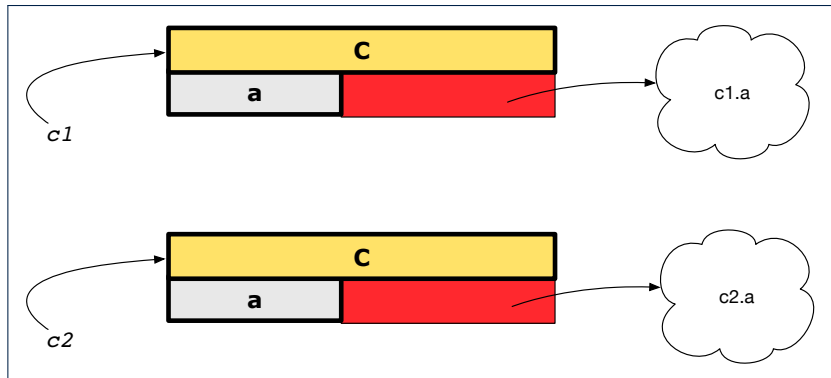


Copying Objects

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` are references to objects.



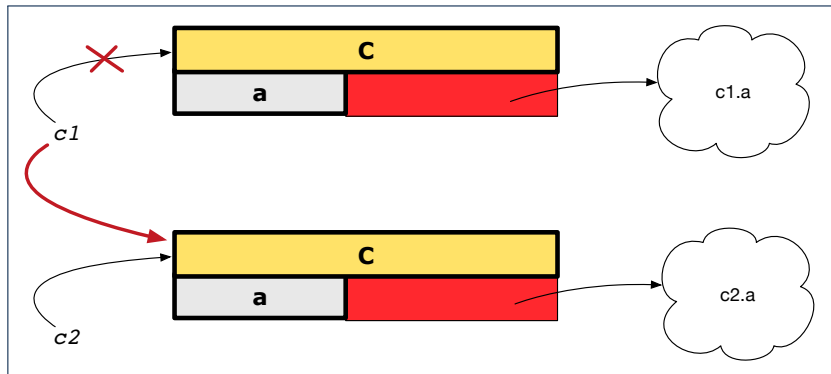
Copying Objects: Reference Copy

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*]

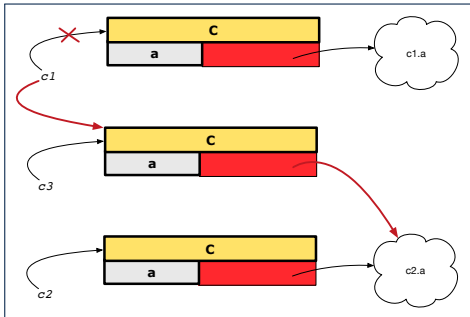


Copying Objects: Shallow Copy

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 $c3$: $c1 := c3$
 $\Rightarrow c1$ and $c2$ **are not** pointing to the same object. $[c1 \neq c2]$
 $\Rightarrow c1.a$ and $c2.a$ **are** pointing to the same object.
 \Rightarrow **Aliasing** still occurs: at 1st level (i.e., attributes of $c1$ and $c2$)

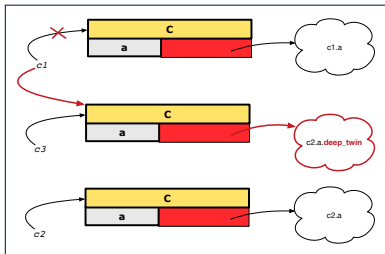


Copying Objects: Deep Copy

Deep Copy

```
c1 := c2.deep_twin
```

- Create a temporary, behind-the-scenes object $c3$ of type C .
- **Recursively** initialize each attribute a of $c3$ as follows:
 - Base Case:** a is primitive (e.g., INTEGER). $\Rightarrow c3.a := c2.a$.
 - Recursive Case:** a is referenced. $\Rightarrow c3.a := c2.a.deep_twin$
- Make a **reference copy** of $c3$: $c1 := c3$
 - $\Rightarrow c1$ and $c2$ **are not** pointing to the same object.
 - $\Rightarrow c1.a$ and $c2.a$ **are not** pointing to the same object.
 - \Rightarrow **No aliasing** occurs at any levels.



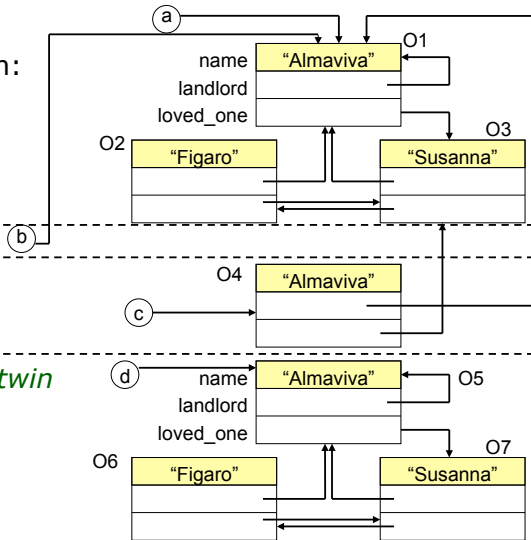
Copying Objects

- Initial situation:
- Result of:

$b := a$

$c := a.twin$

$d := a.deep_twin$



Example: Collection Objects (1)

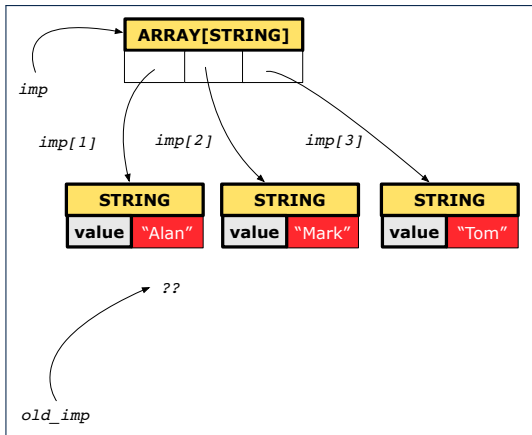
- In any OOPL, when a variable is declared of a **type** that corresponds to a **known class** (e.g., STRING, ARRAY, LINKED_LIST, etc.):
 - At **runtime**, that variable stores the **address** of an object of that type (as opposed to storing the object in its entirety).
- Assume the following variables of the same type:

```
local
  imp : ARRAY[STRING]
  old_imp: ARRAY[STRING]
do
  create {ARRAY[STRING]} imp.make_empty
  imp.force("Alan", 1)
  imp.force("Mark", 2)
  imp.force("Tom", 3)
```

- **Before** we undergo a change on `imp`, we **copy** it to `old_imp`.
- **After** the change is completed, we compare `imp` vs. `old_imp`.
- Can a change always be **visible** between **“old”** and **“new”** `imp`?

Example: Collection Objects (2)

- Variables `imp` and `old_imp` store address(es) of some array(s).
- Each “slot” of these arrays stores a `STRING` object’s address.

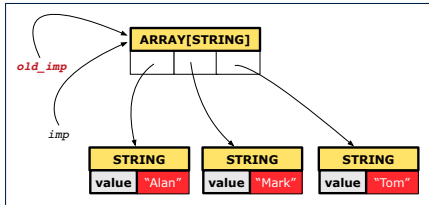


Reference Copy of Collection Object

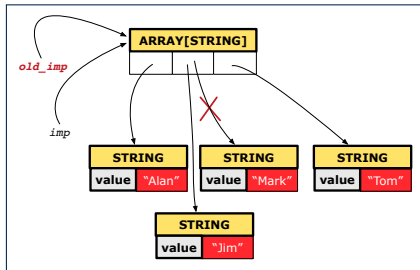
```

1  old_imp := imp
2  Result := old_imp = imp -- Result = true
3  imp[2] := "Jim"
4  Result :=
5  across 1 |..| imp.count is j
6  all imp [j] ~ old_imp [j]
7  end -- Result = true
  
```

Before Executing L3



After Executing L3



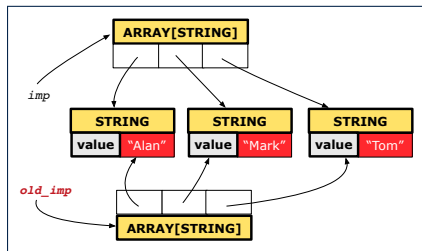
Shallow Copy of Collection Object (1)

```

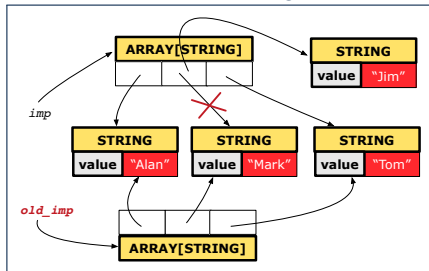
1  old_imp := imp.twin
2  Result := old_imp = imp  -- Result = false
3  imp[2] := "Jim"
4  Result :=
5  across 1 |..| imp.count is j
6  all imp [j] ~ old_imp [j]
7  end  -- Result = false

```

Before Executing L3



After Executing L3



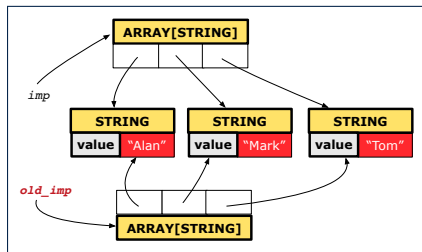
Shallow Copy of Collection Object (2)

```

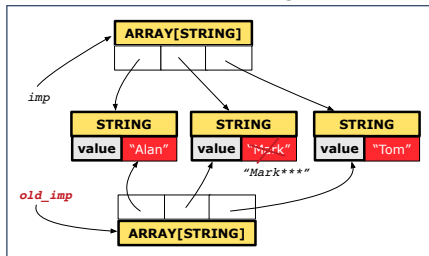
1  old_imp := imp.twin
2  Result := old_imp = imp  -- Result = false
3  imp[2].append ("****")
4  Result :=
5  across 1 |..| imp.count is j
6  all imp [j] ~ old_imp [j]
7  end  -- Result = true

```

Before Executing L3



After Executing L3



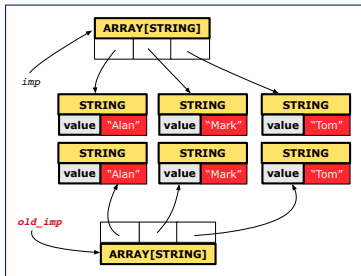
Deep Copy of Collection Object (1)

```

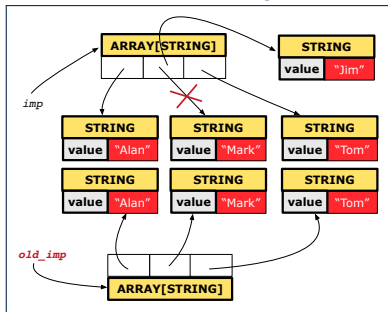
1  old_imp := imp.deep_twin
2  Result := old_imp = imp  -- Result = false
3  imp[2] := "Jim"
4  Result :=
5  across 1 |..| imp.count is j
6  all imp [j] ~ old_imp [j] end  -- Result = false

```

Before Executing L3



After Executing L3



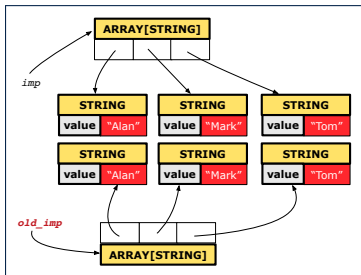
Deep Copy of Collection Object (2)

```

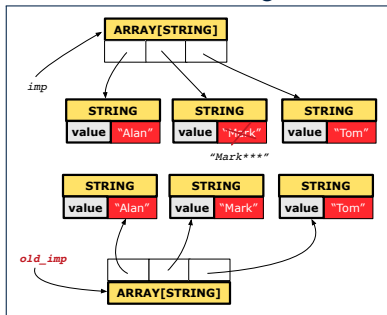
1  old_imp := imp.deep_twin
2  Result := old_imp = imp -- Result = false
3  imp[2].append ("***")
4  Result :=
5  across 1 |..| imp.count is j
6  all imp [j] ~ old_imp [j] end -- Result = false

```

Before Executing L3



After Executing L3



Experiment: Copying Objects

- **Download** the Eiffel project archive (a zip file) here:

https://www.eecs.vorku.ca/~jackie/teaching/lectures/2020/F/EECS3311/codes/copying_objects.zip

- Unzip and compile the project in Eiffel Studio.
- Reproduce the illustrations explained in lectures.

Part 2



Writing Complete Postconditions

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, via `:=`, of **old expressions** in the **post-condition**.

e.g., <code>old accounts[i].id</code>	<code>[old_accounts.i.id := accounts[i].id]</code>
e.g., <code>(old accounts[i]).id</code>	<code>[old_accounts.i := accounts[i]]</code>
e.g., <code>(old accounts[i].twin).id</code>	<code>[old_accounts.i.twin := accounts[i].twin]</code>
e.g., <code>(old accounts)[i].id</code>	<code>[old_accounts := accounts]</code>
e.g., <code>(old accounts.twin)[i].id</code>	<code>[old_accounts.twin := accounts.twin]</code>
e.g., <code>(old Current).accounts[i].id</code>	<code>[old_current := Current]</code>
e.g., <code>(old Current.twin).accounts[i].id</code>	<code>[old_current.twin := Current.twin]</code>

- Right **after** the feature call:
 - The current state of `acc` is called its **post-state**.
 - Evaluate **post-condition** using both **current values** and **“cached” values** of attributes and queries.
 - Evaluate **invariant** using **current values** of attributes and queries.

When are contracts complete?

- 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 -- Attributes
  owner: STRING
  balance: INTEGER

feature -- Commands
  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 -- the input name exists
      existing: across accounts is acc some acc.owner ~ n end
      -- not (across accounts is acc all acc.owner /~ n end)
    do ... ensure Result.owner ~ n end
  add (n: STRING)
    require -- the input name does not exist
      non_existing: across accounts is acc all acc.owner /~ n end
      -- not (across accounts is acc some acc.owner ~ n end)
    local new_account: ACCOUNT
    do
      create new_account.make (n)
      accounts.force (new_account, accounts.upper + 1)
    end
  end
end
```

Roadmap of Illustrations

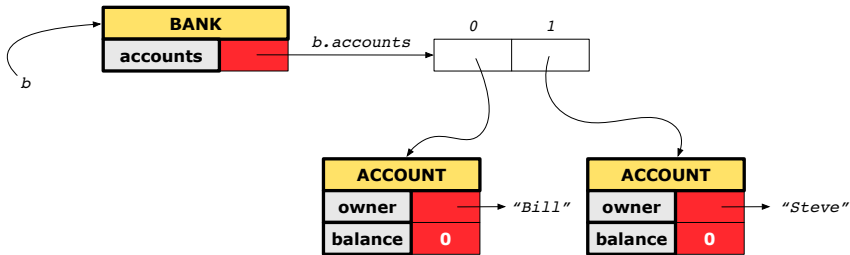
We examine 5 different versions of a command

deposit_on (*n* : *STRING*; *a* : *INTEGER*)

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

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 is acc some acc.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:
        Current.account_of(n).balance =
          old Current.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 is acc some acc.owner ~ n end
    local i: INTEGER
    do ...
      -- imp. of version 1, followed by a deposit into 1st account
      accounts[accounts.lower].deposit(a)
    ensure
      num_of_accounts_unchanged:
        accounts.count = old accounts.count
      balance_of_n_increased:
        Current.account_of(n).balance =
          old Current.account_of(n).balance + a
    end
  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 is acc some acc.owner ~ n end
    local i: INTEGER
    do ...
      -- imp. of version 1, followed by a deposit into 1st account
      accounts[accounts.lower].deposit(a)
    ensure
      num_of_accounts_unchanged: accounts.count = old accounts.count
      balance_of_n_increased:
        Current.account_of(n).balance =
          old Current.account_of(n).balance + a
      others_unchanged :
        across old accounts is acc
          all
            acc.owner /~ n implies acc ~ Current.account_of(acc.owner)
          end
    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 is acc some acc.owner ~ n end
    local i: INTEGER
    do ...
      -- imp. of version 1, followed by a deposit into 1st account
      accounts[accounts.lower].deposit(a)
    ensure
      num_of_accounts_unchanged: accounts.count = old accounts.count
      balance_of_n_increased:
        Current.account_of(n).balance =
          old Current.account_of(n).balance + a
      others_unchanged :
        across old accounts.twin is acc
        all
          acc.owner /~ n implies acc ~ Current.account_of(acc.owner)
        end
    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 is acc some acc.owner ~ n end
    local i: INTEGER
    do ...
      -- imp. of version 1, followed by a deposit into 1st account
      accounts[accounts.lower].deposit(a)
    ensure
      num_of_accounts_unchanged: accounts.count = old accounts.count
      balance_of_n_increased:
        Current.account_of(n).balance =
          old Current.account_of(n).balance + a
      others_unchanged :
        across old accounts.deep_twin is acc
        all
          acc.owner /~ n implies acc ~ Current.account_of(acc.owner)
        end
    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

Experiment: Complete Postconditions



- **Download** the Eiffel project archive (a zip file) here:

https://www.eecs.vorku.ca/~jackie/teaching/lectures/2020/F/EECS3311/codes/array_math_contract.zip

- Unzip and compile the project in Eiffel Studio.
- Reproduce the illustrations explained in lectures.

Beyond this lecture

- 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)

Learning Objectives

Part 1

Copying Objects

Copying Objects: Reference Copy

Copying Objects: Shallow Copy

Copying Objects: Deep Copy

Example: Copying Objects

Example: Collection Objects (1)

Example: Collection Objects (2)

Reference Copy of Collection Object

Shallow Copy of Collection Object (1)

Index (2)

Shallow Copy of Collection Object (2)

Deep Copy of Collection Object (1)

Deep Copy of Collection Object (2)

Experiment: Copying Objects

Part 2

How are contracts checked at runtime?

When are contracts complete?

Account

Bank

Roadmap of Illustrations

Object Structure for Illustration

Index (3)

Version 1:

Incomplete Contracts, Correct Implementation

Test of Version 1

Test of Version 1: Result

Version 2:

Incomplete Contracts, Wrong Implementation

Test of Version 2

Test of Version 2: Result

Version 3:

Complete Contracts with Reference Copy

Test of Version 3

Test of Version 3: Result

Index (4)

Version 4:

Complete Contracts with Shallow Object Copy

Test of Version 4

Test of Version 4: Result

Version 5:

Complete Contracts with Deep Object Copy

Test of Version 5

Test of Version 5: Result

Experiment: Complete Postconditions

Beyond this lecture

Use of Generics



EECS3311 A & E: Software Design
Fall 2020

CHEN-WEI WANG

Learning Objectives



Upon completing this lecture, you are expected to understand:

1. How to **write** a generic class (as a **supplier**)
2. How to **use** a generic class (as a **client**)

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 string 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 account stack operations (e.g., top, push, pop) depends on features specific to element type ACCOUNT (e.g., deposit, withdraw)? [**NO!**]
- A **collection** (e.g., table, tree, graph) is meant for the **storage** and **retrieval** of elements, not how those elements are manipulated.

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 [STRING]  
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
```

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*.

Index (1)

Learning Objectives

Generic Collection Class: Motivation (1)

Generic Collection Class: Motivation (2)

Generic Collection Class: Supplier

Generic Collection Class: Client (1.1)

Generic Collection Class: Client (1.2)

Generic Collection Class: Client (2)

Abstractions via Mathematical Models



EECS3311 A & E: Software Design
Fall 2020

CHEN-WEI WANG

Learning Objectives



Upon completing this lecture, you are expected to understand:

1. Creating a *mathematical abstraction* for alternative *implementations*
2. Two design principles: *Information Hiding* and *Single Choice*
3. Review of the basic discrete math (self-guided)

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
<i>count</i>	imp.count		
<i>top</i>	imp[imp.count]	imp.first	imp.last
<i>push(g)</i>	imp.force(g, imp.count + 1)	imp.put_front(g)	imp.extend(g)
<i>pop</i>	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
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 - 1] 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
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
end

```

Design Principles: Information Hiding & Single Choice

- **Information Hiding** (IH):
 - Hide supplier's **design decisions** that are *likely to change*.
 - Violation of IH means that your design's public API is **unstable**.
 - *Change of supplier's secrets* should not affect clients relying upon the existing API.
- **Single Choice Principle** (SCP):
 - When a **change** is needed, there should be **a single place** (or **a minimal number of places**) where you need to make that change.
 - Violation of SCP means that your design contains **redundancies**.

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*.

Math Models: Command vs Query

- Use MATHMODELS library to create math objects (SET, REL, SEQ).
- State-changing **commands**: Implement 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
```

- Side-effect-free **queries**: Write Complete 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 model ~ (old model.deep_twin).appended(g) end
```

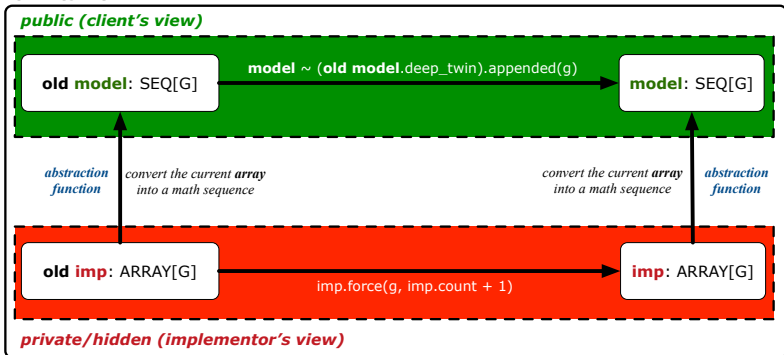

Implementing an Abstraction Function (1)



```
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)

'push(g: G)' feature of LIFO_STACK ADT



- **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)`

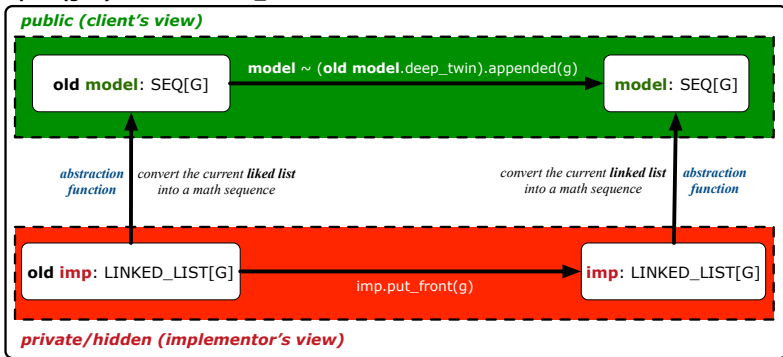
Implementing an Abstraction Function (2)



```
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)

'push(g: G)' feature of LIFO_STACK ADT



- **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)

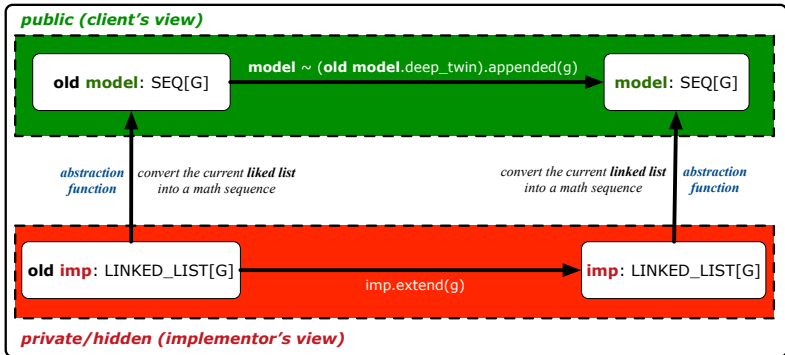
```

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)

'push(g: G)' feature of LIFO_STACK ADT



- **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**:

$model \sim (old\ model.deep_twin).appended(g)$

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.

Beyond this lecture ...



- Familiarize yourself with the features of class SEQ.

Index (1)

Learning Objectives

Motivating Problem: Complete Contracts

Motivating Problem: LIFO Stack (1)

Motivating Problem: LIFO Stack (2.1)

Motivating Problem: LIFO Stack (2.2)

Motivating Problem: LIFO Stack (2.3)

Design Principles:

Information Hiding & Single Choice

Motivating Problem: LIFO Stack (3)

Math Models: Command vs Query

Implementing an Abstraction Function (1)

Index (2)

Abstracting ADTs as Math Models (1)

Implementing an Abstraction Function (2)

Abstracting ADTs as Math Models (2)

Implementing an Abstraction Function (3)

Abstracting ADTs as Math Models (3)

Solution: Abstracting ADTs as Math Models

Beyond this lecture ...

Drawing a Design Diagram using the Business Object Notation (BON)



EECS3311 A & E: Software Design
Fall 2020

CHEN-WEI WANG

Learning Objectives



- Purpose of a **Design Diagram**: an **Abstraction** of Your Design
- Architectural Relation: **Client-Supplier** vs. **Inheritance**
- Presenting a class: Compact vs. Detailed
- Denoting a Class or Feature: Deferred vs. Effective

Why a Design Diagram?

- **SOURCE CODE** is **not** an appropriate form for communication.
- Use a **DESIGN DIAGRAM** showing **selective** sets of important:
 - clusters (i.e., packages)
 - [deferred vs. effective]
 - [generic vs. non-generic]
 - architectural relations
 - [client-supplier vs. inheritance]
 - routines (queries and commands)
 - [deferred vs. effective vs. redefined]
 - **contracts**
 - [precondition vs. postcondition vs. class invariant]
- Your design diagram is called an **abstraction** of your system:
 - Being **selective** on what to show, filtering out **irrelevant details**
 - Presenting **contractual specification** in a **mathematical form** (e.g., \forall instead of **across ... all ... end**).

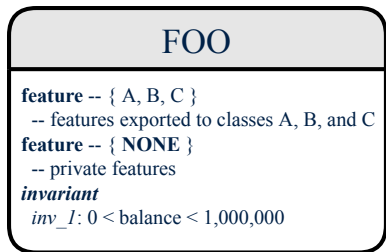
Classes:

Detailed View vs. Compact View (1)

- **Detailed view** shows a selection of:
 - **features** (queries and/or commands)
 - **contracts** (class invariant and feature pre-post-conditions)
 - Use the detailed view if readers of your design diagram **should know** such details of a class.
e.g., Classes critical to your design or implementation
- **Compact view** shows only the class name.
 - Use the compact view if readers **should not be bothered with** such details of a class.
e.g., Minor “helper” classes of your design or implementation
e.g., Library classes (e.g., ARRAY, LINKED_LIST, HASH_TABLE)

Classes: Detailed View vs. Compact View (2)

Detailed View



Compact View



Contracts: Mathematical vs. Programming

- When presenting the detailed view of a class, you should include **contracts** of features which you judge as **important**.
- Consider an array-based linear container:

```

ARRAYED_CONTAINER+

feature -- Queries
count+: INTEGER
    -- Number of items stored in the 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 }
imp+: ARRAY[STRING]
    -- Implementation of an arrayed-container

invariant
    consistency: imp.count = count
  
```

- A **tag** should be included for each contract.
- Use **mathematical** symbols (e.g., \forall , \exists , \leq) instead of **programming** symbols (e.g., **across ... all ...**, **across ... some ...**, \leq).

Classes: Generic vs. Non-Generic

- A class is **generic** if it declares at least one type parameters.
 - Collection classes are generic: `ARRAY[G]`, `HASH_TABLE[G, H]`, *etc.*
 - Type parameter(s) of a class may or may not be **instantiated**:

`HASH_TABLE[G, H]`

`MY_TABLE_1[STRING, INTEGER]`

`MY_TABLE_2[PERSON, INTEGER]`

- If necessary, present a generic class in the detailed form:

`DATABASE[G]+`

```
feature
-- some public features here
feature -- { NONE }
-- imp: ARRAY[G]
invariant
-- some class invariant here
```

`MY_DB_1[STRING]+`

```
feature
-- some public features here
feature -- { NONE }
-- imp: ARRAY[STRING]
invariant
-- some class invariant here
```

`MY_DB_2[PERSON]+`

```
feature
-- some public features here
feature -- { NONE }
-- imp: ARRAY[PERSON]
invariant
-- some class invariant here
```

- A class is **non-generic** if it declares no type parameters.

Deferred vs. Effective



Deferred means *unimplemented* (\approx **abstract** in Java)

Effective means *implemented*

Classes: Deferred vs. Effective

- A **deferred class** has at least one feature **unimplemented**.
 - A **deferred class** may only be used as a **static** type (for declaration), but cannot be used as a **dynamic** type.
 - e.g., By declaring `list: LIST[INTEGER]` (where `LIST` is a **deferred** class), it is invalid to write:
 - `create list.make`
 - `create {LIST[INTEGER]} list.make`
- An **effective class** has all features **implemented**.
 - An **effective class** may be used as both **static** and **dynamic** types.
 - e.g., By declaring `list: LIST[INTEGER]`, it is valid to write:
 - `create {LINKED_LIST[INTEGER]} list.make`
 - `create {ARRAYED_LIST[INTEGER]} list.make`

where `LINKED_LIST` and `ARRAYED_LIST` are both **effective** descendants of `LIST`.

Features: Deferred, Effective, Redefined (1)



A **deferred feature** is declared with its *header* only (i.e., name, parameters, return type).

- The word “**deferred**” means a descendant class would later implement this feature.
- The resident class of the **deferred** feature must also be **deferred**.

```
deferred class
  DATABASE[G]
feature -- Queries
  search (g: G): BOOLEAN
    -- Does item 'g' exist in database?
  deferred end
end
```

Features: Deferred, Effective, Redefined (2)

- An **effective feature** *implements* some inherited deferred feature.

```
class
  DATABASE_V1[G]
inherit
  DATABASE[G]
feature -- Queries
  search (g: G): BOOLEAN
    -- Perform a linear search on the database.
  do end
end
```

- A descendant class may still later *re-implement* this feature.

Features: Deferred, Effective, Redefined (3)



- A **redefined feature** *re-implements* some inherited effective feature.

```
class
  DATABASE_V2 [G]
inherit
  DATABASE_V1 [G]
    redefine search end
feature -- Queries
  search (g: G): BOOLEAN
    -- Perform a binary search on the database.
    do end
end
```

- A descendant class may still later *re-implement* this feature.

Classes: Deferred vs. Effective (2.1)

Append a star * to the name of a **deferred** class or feature.

Append a plus + to the name of an **effective** class or feature.

Append two pluses ++ to the name of a **redefined** feature.

- Deferred or effective classes may be in the compact form:

LIST[G]*

LINKED_LIST[G]+

ARRAYED_LIST[G]+

LIST[LIST[PERSON]]*

LINKED_LIST[INTEGER]+

ARRAYED_LIST[G]+

DATABASE[G]*

DATABASE_V1[G]+

DATABASE_V2[G]+

Classes: Deferred vs. Effective (2.2)

- Append a star * to the name of a **deferred** class or feature.
- Append a plus + to the name of an **effective** class or feature.
- Append two pluses ++ to the name of a **redefined** feature.
- Deferred or effective classes may be in the detailed form:

DATABASE[G]*

```

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

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

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

exists* (g: G): BOOLEAN
  -- Does item 'g' exist in database?
ensure
  correct_result: Result = ( $\exists i : 1 \leq i \leq \text{count} : \text{data}[i] \sim g$ )
  
```

DATABASE_V1[G]+

```

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

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

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

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

DATABASE_V2[G]++

```

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

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

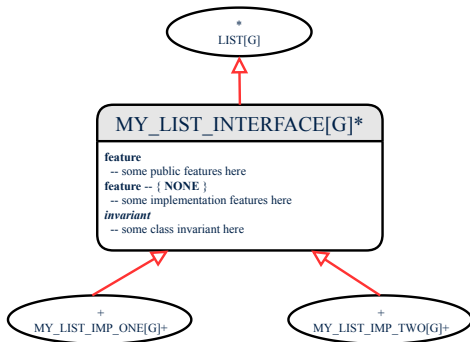
feature -- Queries
count+: INTEGER
  -- Number of items stored in database

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

invariant
sorted_data:  $\forall i : 1 \leq i < \text{count} : \text{data}[i] < \text{data}[i + 1]$ 
  
```


Class Relations: Inheritance (1)

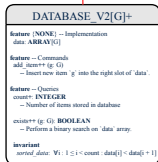
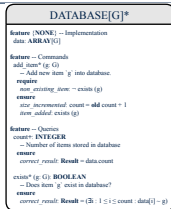
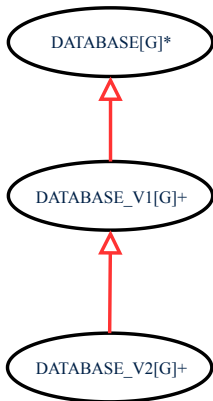
- An **inheritance hierarchy** is formed using **red arrows**.
 - Arrow's **origin** indicates the **child/descendant** class.
 - Arrow's **destination** indicates the **parent/ancestor** class.
- You may choose to present each class in an inheritance hierarchy in either the detailed form or the compact form:



Class Relations: Inheritance (2)

More examples (emphasizing different aspects of DATABASE):

Inheritance Hierarchy | Features being (Re-)Implemented



Class Relations: Client-Supplier (1)

- A `client-supplier (CS) relation` exists between two classes: one (the *client*) uses the service of another (the *supplier*).
- Programmatically, there is CS relation if in class `CLIENT` there is a variable declaration `s1: SUPPLIER`.
 - A variable may be an attribute, a parameter, or a local variable.
- A *green arrow* is drawn between the two classes.
 - Arrow's *origin* indicates the *client* class.
 - Arrow's *destination* indicates the *supplier* class.
 - Above the arrow there should be a *label* indicating the **supplier name** (i.e., variable name).
 - In the case where supplier is a routine, indicate after the label name if it is deferred (*), effective (+), or redefined (++)).

Class Relations: Client-Supplier (2.1)

```

class DATABASE
feature {NONE} -- implementation
  data: ARRAY[STRING]
feature -- Commands
  add_name (nn: STRING)
    -- Add name 'nn' to database.
    require ... do ... ensure ... end

  name_exists (n: STRING): BOOLEAN
    -- Does name 'n' exist in database?
    require ...
    local
      u: UTILITIES
    do ... ensure ... end
invariant
  ...
end
  
```

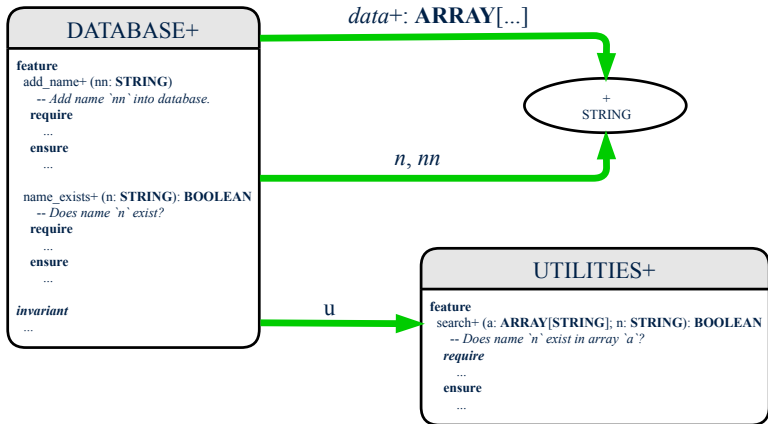
```

class UTILITIES
feature -- Queries
  search (a: ARRAY[STRING]; n: STRING): BOOLEAN
    -- Does name 'n' exist in array 'a'?
    require ... do ... ensure ... end
end
  
```

- Query `data: ARRAY[STRING]` indicates two suppliers: `STRING` and `ARRAY`.
- Parameters `nn` and `n` may have an arrow with label `nn, n`, pointing to the `STRING` class.
- Local variable `u` may have an arrow with label `u`, pointing to the `UTILITIES` class.

Class Relations: Client-Supplier (2.2.1)

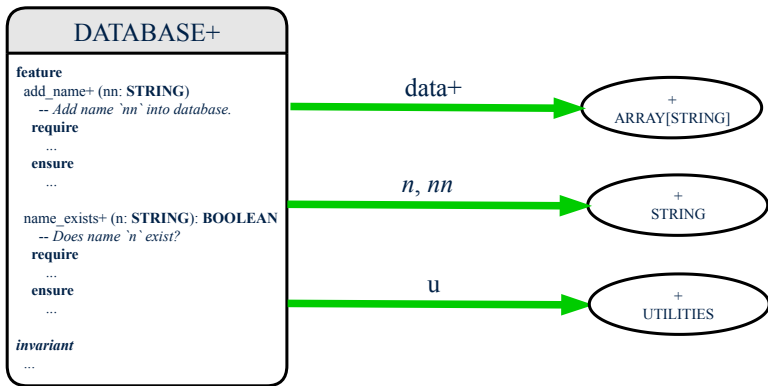
If `STRING` is to be emphasized, label is `data: ARRAY[...]`, where `...` denotes the supplier class `STRING` being pointed to.



Class Relations: Client-Supplier (2.2.2)

If ARRAY is to be emphasized, label is `data`.

The supplier's name should be complete: ARRAY [STRING]



Class Relations: Client-Supplier (3.1)

Known: The *deferred* class LIST has two *effective* descendants ARRAY_LIST and LINKED_LIST).

- DESIGN ONE:

```
class DATABASE_V1
feature {NONE} -- implementation
  imp: ARRAYED_LIST[PERSON]
... -- more features and contracts
end
```

- DESIGN TWO:

```
class DATABASE_V2
feature {NONE} -- implementation
  imp: LIST[PERSON]
... -- more features and contracts
end
```

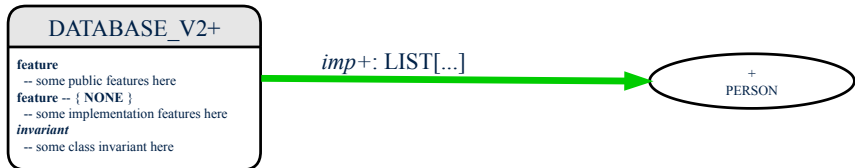
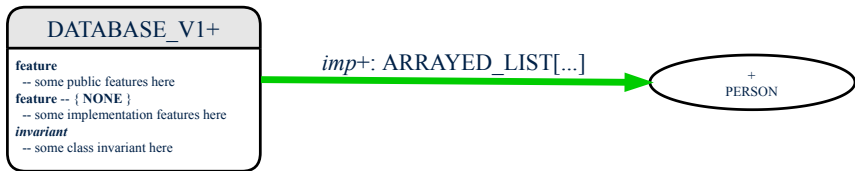
Question: Which design is better?

[DESIGN TWO]

Rationale: Program to the *interface*, not the *implementation*.

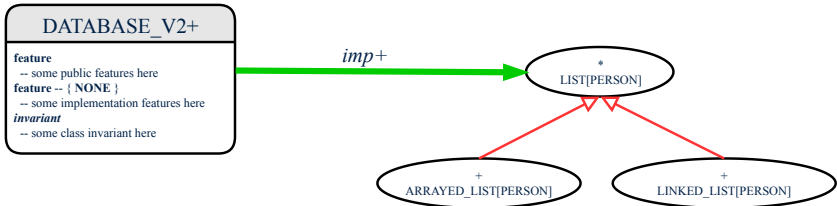
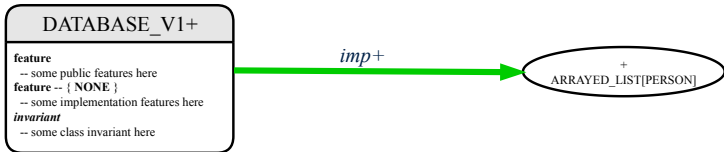
Class Relations: Client-Supplier (3.2.1)

We may focus on the `PERSON` supplier class, which may not help judge which design is better.



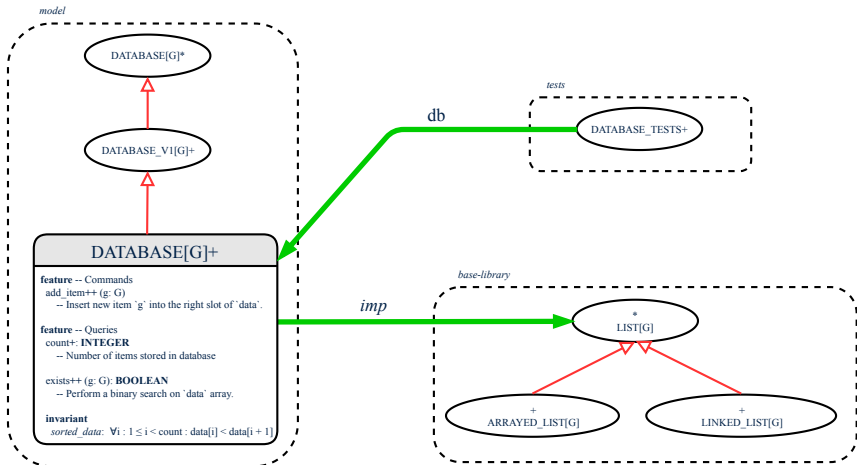
Class Relations: Client-Supplier (3.2.2)

Alternatively, we may focus on the `LIST` supplier class, which in this case helps us judge which design is better.



Clusters: Grouping Classes

Use *clusters* to group classes into logical units.



Beyond this lecture



- Your Lab0 introductory tutorial series contains the following classes:
 - BIRTHDAY
 - BIRTHDAY_BOOK
 - TEST_BIRTHDAY
 - TEST_BIRTHDAY_BOOK
 - TEST_LIBRARY
 - BAD_BIRTHDAY_VIOLATING_DAY_SET
 - BIRTHDAY_BOOK_VIOLATING_NAME_ADDED_TO_END

Draw a *design diagram* showing the *architectural relations* among the above classes.

Index (1)

Learning Objectives

Why a Design Diagram?

Classes:

Detailed View vs. Compact View (1)

Classes:

Detailed View vs. Compact View (2)

Contracts: Mathematical vs. Programming

Classes: Generic vs. Non-Generic

Deferred vs. Effective

Classes: Deferred vs. Effective

Features: Deferred, Effective, Redefined (1)

Index (2)

Features: Deferred, Effective, Redefined (2)

Features: Deferred, Effective, Redefined (3)

Classes: Deferred vs. Effective (2.1)

Classes: Deferred vs. Effective (2.2)

Class Relations: Inheritance (1)

Class Relations: Inheritance (2)

Class Relations: Client-Supplier (1)

Class Relations: Client-Supplier (2.1)

Class Relations: Client-Supplier (2.2.1)

Class Relations: Client-Supplier (2.2.2)

Class Relations: Client-Supplier (3.1)

Index (3)

Class Relations: Client-Supplier (3.2.1)

Class Relations: Client-Supplier (3.2.2)

Clusters: Grouping Classes

Beyond this lecture

Case Study: Abstraction of a Birthday Book



EECS3311 A & E: Software Design
Fall 2020

CHEN-WEI WANG

Learning Objectives



Upon completing this lecture, you are expected to understand:

1. Asserting Set Equality in Postconditions (Exercise)
2. The basics of discrete math (Self-Guided Study)
FUN is a REL, but not vice versa.
3. Creating a *mathematical abstraction* for a birthday book
4. Using commands and queries from two `mathmodels` classes:
REL and FUN

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., $\{x \mid 1 \leq x \leq 10 \wedge x \text{ is an odd number}\}$
- An empty set (denoted as $\{\}$ or \emptyset) 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., $4 \notin \{x \mid x \leq 1 \leq 10, x \text{ is an odd number}\}$ [true]
- The number of elements in a set is called its *cardinality*.
e.g., $|\emptyset| = 0$, $|\{x \mid x \leq 1 \leq 10, x \text{ is an odd number}\}| = 5$

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.

$$S_1 = S_2 \iff S_1 \subseteq S_2 \wedge S_2 \subseteq S_1$$

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

$$S_1 \subset S_2 \iff S_1 \subseteq S_2 \wedge |S_1| < |S_2|$$

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

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

Math Review: Set of Tuples

Given n sets S_1, S_2, \dots, S_n , a **cross product** of these sets is a set of n -tuples.

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

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

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 $T = \{a, b\}$
 - \emptyset 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.

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

e.g., $\text{dom}(r_1) = \{1, 2, 3\}$, $\text{dom}(r_2) = \{2, 3\}$

- **Range** of r is the set of T members that r maps to.

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

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

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

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

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
 - $r.overridden(t) = t \cup r.domain_subtracted(t.domain)$
 -

$$\begin{aligned}
 & r.overridden(\{(a, 3), (c, 4)\}) \\
 = & \underbrace{\{(a, 3), (c, 4)\}}_t \cup \underbrace{\{(b, 2), (b, 5), (d, 1), (e, 2), (f, 3)\}}_{r.domain_subtracted(\underbrace{t.domain}_{\{a,c\}})} \\
 = & \{(a, 3), (c, 4), (b, 2), (b, 5), (d, 1), (e, 2), (f, 3)\}
 \end{aligned}$$

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 .

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

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

- $S \times T$ [No]
- $(S \times T) - \{(x, y) \mid (x, y) \in S \times T \wedge x = 1\}$ [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*:

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

- 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 $S \rightarrow T$ and use it to declare a function variable f :

$$f : S \rightarrow T$$

Math Review: Functions (3.1)

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

- f is *injective* (or an injection) if f does not map two members of S to the same member of T .

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

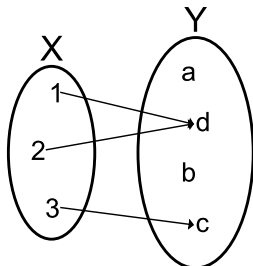
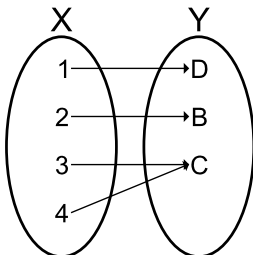
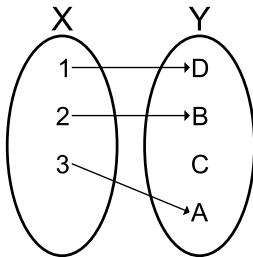
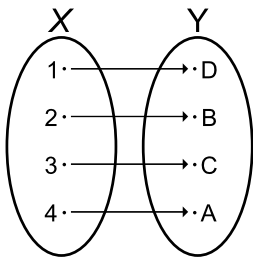
e.g., Considering an array as a function from integers to objects, being injective means that the array does not contain any duplicates.

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

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

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

Math Review: Functions (3.2)



Math Models: Command-Query Separation

<i>Command</i>	<i>Query</i>
domain_restrict	domain_restricted ed
domain_restrict_by	domain_restricted ed .by
domain_subtract	domain_subtracted ed
domain_subtract_by	domain_subtracted ed .by
range_restrict	range_restricted ed
range_restrict_by	range_restricted ed .by
range_subtract	range_subtracted ed
range_subtract_by	range_subtracted ed .by
override	overridden ed
override_by	overridden ed .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
end

```


Case Study: A Birthday Book

- A birthday book stores a collection of entries, where each entry is a pair of a person's name and their birthday.
- No two entries stored in the book are allowed to have the same name.
- Each birthday is characterized by a month and a day.
- A birthday book is first created to contain an empty collection of entries.
- Given a birthday book, we may:
 - Inquire about the number of entries currently stored in the book
 - Add a new entry by supplying its name and the associated birthday
 - Remove the entry associated with a particular person
 - Find the birthday of a particular person
 - Get a reminder list of names of people who share a given birthday

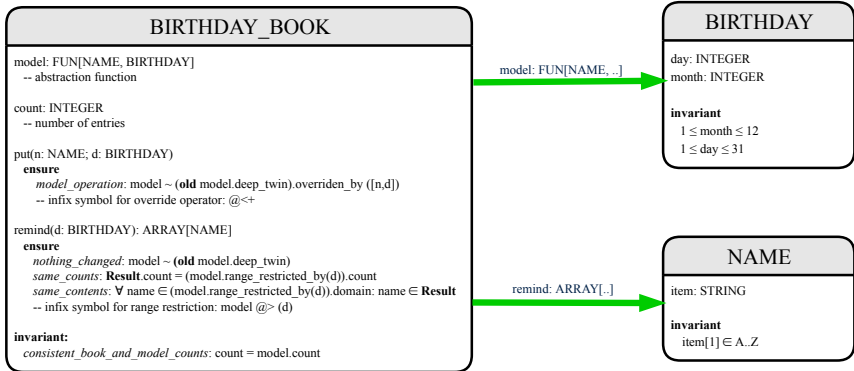
Birthday Book: Decisions

- **Design** Decision
 - Classes
 - Client Supplier vs. Inheritance
 - Mathematical Model? [e.g., REL or FUN]
 - Contracts

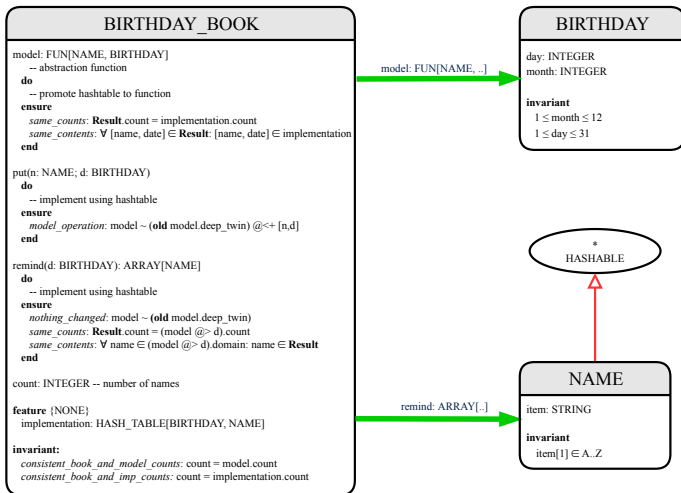
- **Implementation** Decision
 - Two linear structures (e.g., arrays, lists) [$O(n)$]
 - A balanced search tree (e.g., AVL tree) [$O(\log \cdot n)$]
 - A hash table [$O(1)$]

- Implement an **abstraction function** that maps implementation to the math model.

Birthday Book: Design



Birthday Book: Implementation



Beyond this lecture . . .



- Familiarize yourself with the features of class `REL`, `FUN`, and `SET`.
- **Exercise:**
 - Consider an alternative implementation using two linear structures (e.g., [here in Java](#)).
 - Implement the design of birthday book covered in lectures.
 - Create another `LINEAR_BIRTHDAY_BOOK` class and modify the implementation of abstraction function accordingly.
Do all contracts still pass? What should change? What remain unchanged?

Index (1)

Learning Objectives

Math Review: Set Definitions and Membership

Math Review: Set Relations

Math Review: Set Operations

Math Review: Power Sets

Math Review: Set of Tuples

Math Models: Relations (1)

Math Models: Relations (2)

Math Models: Relations (3.1)

Math Models: Relations (3.2)

Math Models: Relations (3.3)

Index (2)

Math Review: Functions (1)

Math Review: Functions (2)

Math Review: Functions (3.1)

Math Review: Functions (3.2)

Math Models: Command-Query Separation

Math Models: Example Test

Case Study: A Birthday Book

Birthday Book: Decisions

Birthday Book: Design

Birthday Book: Implementation

Beyond this lecture ...

Design Pattern: Iterator



EECS3311 A & E: Software Design
Fall 2020

CHEN-WEI WANG

Learning Objectives



Upon completing this lecture, you are expected to understand:

1. Motivating Problem of the Iterator Design Pattern
2. Supplier: Implementing the Iterator Design Pattern
3. Client: Using the Iterator Design Pattern
4. A Challenging Exercise (architecture & generics)

What are design patterns?



- Solutions to *recurring 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*!

Iterator Pattern: Motivation (1)

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
end
```

Iterator Pattern: Motivation (2)

Supplier:

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

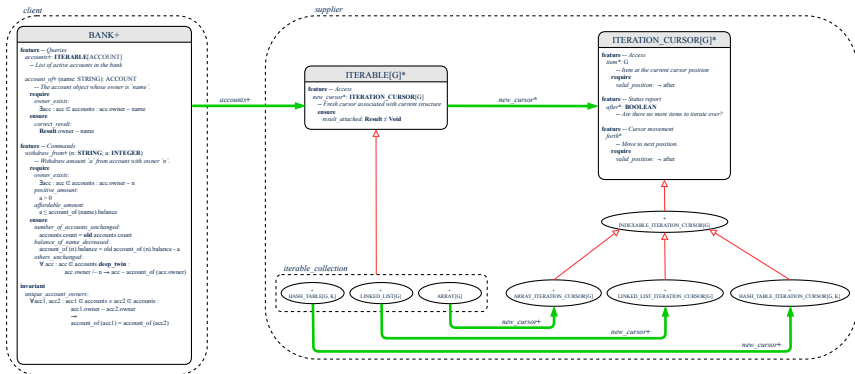
class
  ORDER
feature
  price: INTEGER
  quantity: INTEGER
end
```

Client's code must be modified to adapt to the supplier's *change on implementation*.

Client:

```
class
  SHOP
feature
  cart: CART
  checkout: INTEGER
do
  from
    cart.orders.start
  until
    cart.orders.after
do
  Result := Result +
    cart.orders.item.price
    *
    cart.orders.item.quantity
end
end
end
```

Iterator Pattern: Architecture



Iterator Pattern: Supplier's Side

- **Information Hiding Principle** :
 - Hide design decisions that are *likely to change* (i.e., *stable* API).
 - *Change of secrets* does not affect clients using the existing API.
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!

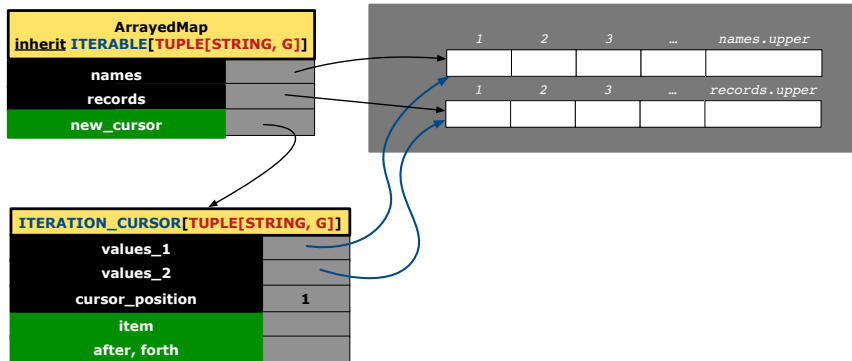
Iterator Pattern: Supplier's Imp. (2.2)

```
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
  cursor_position: INTEGER
  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*.

Iterator Pattern: Supplier's Imp. (2.3)

Visualizing iterator pattern at runtime:



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*).

Resources



- [Tutorial Videos on Generic Parameters and the Iterator Pattern](#)
- [Tutorial Videos on Information Hiding and the Iterator Pattern](#)
- [Tutorial on Making a Birthday Book \(implemented using HASH_TABLE\) ITERABLE](#)

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 **interface** type **ITERABLE[G]** (rather than **implementation** type **ARRAY**, **LINKED_LIST**, or **MY_TREE**).
e.g., `cart: CART`, where **CART** inherits **ITERABLE[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 is item
      all
        item > 0
      end
    end
end
```

- Using **all** corresponds to a universal quantification (i.e., \forall).
- Using **some** corresponds to an existential quantification (i.e., \exists).

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) is i
      all
        accounts [i].id <= accounts [i + 1].id
      end
    end
  do
    ...
  ensure
    Result.id = acc_id
  end
```

This precondition corresponds to:

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

Iterator Pattern: Clients using across for Contracts (3)

```
class BANK
...
  accounts: LIST [ACCOUNT]
  contains_duplicate: BOOLEAN
    -- Does the account list contain duplicate?
  do
    ...
  ensure
     $\forall i, j: \text{INTEGER} \mid$ 
       $1 \leq i \leq \text{accounts.count} \wedge 1 \leq j \leq \text{accounts.count} \bullet$ 
       $\text{accounts}[i] \sim \text{accounts}[j] \Rightarrow i = j$ 
  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 cursor := accounts.new_cursor; max := cursor.item
    until cursor.after
    do
      if cursor.item.balance > max.balance then
        max := cursor.item
      end
      cursor.forth
    end
  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   do
7     across
8       cart is order
9     loop
10      Result := Result + order.price * order.quantity
11    end
12  ensure ??
13 end
```

- Class *CART* should inherit from *ITERABLE[ORDER]*.
- **L10** implicitly declares `cursor: ITERATION_CURSOR[ORDER]` and does `cursor := cart.new_cursor`

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

```
class BANK
  accounts: LIST[ACCOUNT] -- Q: Can ITERABLE[ACCOUNT] work?
  max_balance: ACCOUNT
    -- Account with the maximum balance value.
  require ??
  local
    max: ACCOUNT
  do
    max := accounts [1]
  across
    accounts is acc
  loop
    if acc.balance > max.balance then
      max := acc
    end
  end
ensure ??
end
```

Beyond this lecture ...



- Tutorial Videos on Iterator Pattern
- Exercise: Architecture & Generics

Index (1)

Learning Objectives

What are design patterns?

Iterator Pattern: Motivation (1)

Iterator Pattern: Motivation (2)

Iterator Pattern: Architecture

Iterator Pattern: Supplier's Side

Iterator Pattern: Supplier's Implementation (1)

Iterator Pattern: Supplier's Imp. (2.1)

Iterator Pattern: Supplier's Imp. (2.2)

Iterator Pattern: Supplier's Imp. (2.3)

Exercises

Index (2)



Resources

Iterator Pattern: Client's Side

Iterator Pattern:

Clients using across for Contracts (1)

Iterator Pattern:

Clients using across for Contracts (2)

Iterator Pattern:

Clients using across for Contracts (3)

Iterator Pattern:

Clients using Iterable in Imp. (1)

Iterator Pattern:

Clients using Iterable in Imp. (2)

Index (3)

Iterator Pattern:

Clients using Iterable in Imp. (3)

Beyond this lecture ...

Singleton Design Pattern



EECS3311 A & E: Software Design
Fall 2020

CHEN-WEI WANG

Learning Objectives

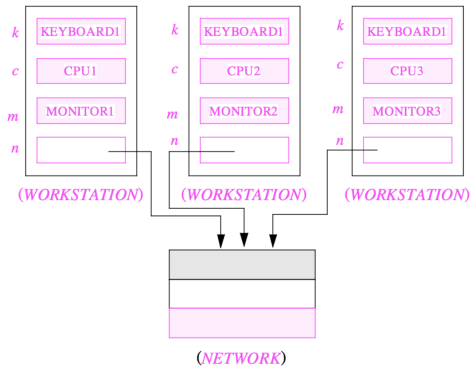
Upon completing this lecture, you are expected to understand:

1. Modeling Concept of *Expanded Types* (Compositions)
2. *Once Routines* in Eiffel vs. Static Methods in Java
3. Export Status
4. Sharing via *Inheritance* (w.r.t. *SCP* and *Cohesion*)
5. *Singleton* Design Pattern

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 keyboard.
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
end

```

```

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

```

- **L5**: object of expanded type is automatically initialized.
- **L10,L12,L13**: no sharing among objects of expanded type.
- **L6,L9,L14**: = compares contents between expanded objects.

Reference vs. Expanded (1)

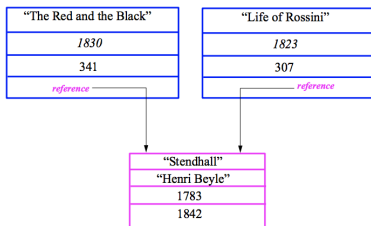
- Every entity must be declared to be of a certain type (based on a class).
- Every type is either *referenced* or *expanded*.
- In *reference* types:
 - y denotes *a reference* to some object
 - $x := y$ attaches x to same object as does y
 - $x = y$ compares references
- In *expanded* types:
 - y denotes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

$[x \sim y]$

Reference vs. Expanded (2)

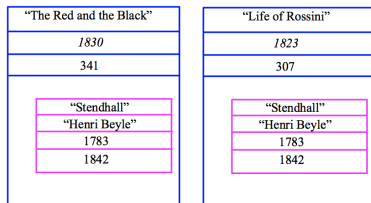
Problem: Every published book has an author. Every author may publish more than one books. Should the author field of a book *reference*-typed or *expanded*-typed?

reference-typed author



Hyperlinked author page

expanded-typed author



Physical printed copies

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.
e.g., printers

Shared Data via Inheritance

Descendant:

```
class DEPOSIT inherit SHARED_DATA
  -- 'maximum_balance' relevant
end

class WITHDRAW inherit SHARED_DATA
  -- 'minimum_balance' relevant
end

class INT_TRANSFER inherit SHARED_DATA
  -- 'exchange_rate' relevant
end

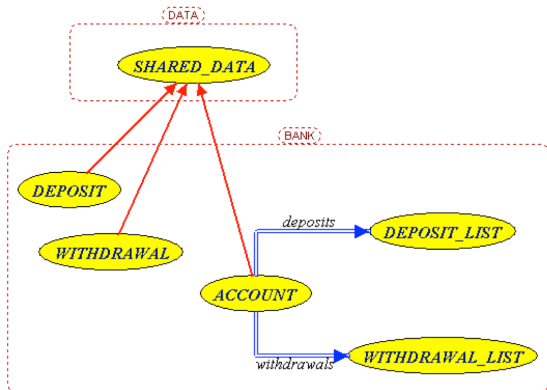
class ACCOUNT inherit SHARED_DATA
feature
  -- 'interest_rate' relevant
  deposits: DEPOSIT_LIST
  withdraws: WITHDRAW_LIST
end
```

Ancestor:

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

Problems?

Sharing Data via Inheritance: Architecture



- *Irreverent* features are inherited.
⇒ Descendants' **cohesion** is broken.
- Same set of data is *duplicated* as instances are created.
⇒ Updates on these data may result in **inconsistency**.

Sharing Data via Inheritance: Limitation

- Each descendant 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 !

Approximating Once Routine in Java (1)

We may encode Eiffel once routines in Java:

```
class BankData {  
    BankData() { }  
    double interestRate;  
    void setIR(double r);  
    ...  
}
```

```
class Account {  
    BankData data;  
    Account() {  
        data = BankDataAccess.getData();  
    }  
}
```

```
class BankDataAccess {  
    static boolean initOnce;  
    static BankData data;  
    static BankData getData() {  
        if(!initOnce) {  
            data = new BankData();  
            initOnce = true;  
        }  
        return data;  
    }  
}
```

Problem?

Multiple **BankData** objects may be created in Account, breaking the singleton!

```
Account() {  
    data = new BankData();  
}
```

Approximating Once Routine in Java (2)



We may encode Eiffel once routines in Java:

```
class BankData {
    private BankData() { }
    double interestRate;
    void setIR(double r);
    static boolean initOnce;
    static BankData data;
    static BankData getData() {
        if(!initOnce) {
            data = new BankData();
            initOnce = true;
        }
        return data;
    }
}
```

Problem?

Loss of Cohesion: **Data** and **Access to Data** are two separate concerns, so should be decoupled into two different classes!

Singleton Pattern in Eiffel (1)

Supplier:

```
class DATA
create {DATA_ACCESS} make
feature {DATA_ACCESS}
  make do v := 10 end
feature -- Data Attributes
  v: INTEGER
  change_v (nv: INTEGER)
    do v := nv end
end
```

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

Client:

```
test: BOOLEAN
  local
    access: DATA_ACCESS
    d1, d2: DATA
  do
    d1 := access.data
    d2 := access.data
    Result := d1 = d2
    and d1.v = 10 and d2.v = 10
  check Result end
  d1.change_v (15)
  Result := d1 = d2
  and d1.v = 15 and d2.v = 15
end
end
```

Writing `create d1.make` in test feature does not compile. Why?

Singleton Pattern in Eiffel (2)

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:

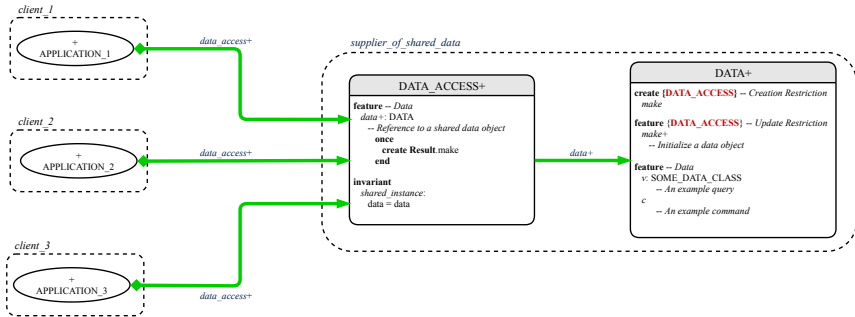
```
class
  ACCOUNT
  feature
    data: BANK_DATA
    make (...)
    -- Init. access to bank data.
  local
    data_access: BANK_DATA_ACCESS
  do
    data := data_access.data
    ...
  end
end
```

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
      and acc1.data.interest_rate = 3.11
    check Result end
    acc2.data.set_interest_rate (2.98)
    Result :=
      acc1.data.interest_rate = acc2.data.interest_rate
      and acc1.data.interest_rate = 2.98
  end
```

Singleton Pattern: Architecture



Important Exercises: Instantiate this architecture to the problem of shared bank data.

Draw it in `draw.io`.

Beyond this lecture

The *singleton* pattern is instantiated in the ETF framework:

- ETF_MODEL (*shared data*)
- ETF_MODEL_ACCESS (*exclusive once access*)
- ETF_COMMAND and its effective descendants:

```

deferred class
  ETF_COMMAND
  feature -- Attributes
    model: ETF_MODEL
  feature {NONE}
    make(...)
    local
      ma: ETF_MODEL_ACCESS
    do
      ...
      model := ma.m
    end
  end
end
  
```

```

class
  ETF_MOVE
  inherit
    ETF_MOVE_INTERFACE
  -- which inherits ETF_COMMAND
  feature -- command
    move(...)
  do
    ...
    model.some_routine (...)
    ...
  end
end
end
  
```

Index (1)

Learning Objectives

Expanded Class: Modelling

Expanded Class: Programming (2)

Expanded Class: Programming (3)

Reference vs. Expanded (1)

Reference vs. Expanded (2)

Singleton Pattern: Motivation

Shared Data via Inheritance

Sharing Data via Inheritance: Architecture

Sharing Data via Inheritance: Limitation

Introducing the Once Routine in Eiffel (1.1)

Index (2)

Introducing the Once Routine in Eiffel (1.2)

Introducing the Once Routine in Eiffel (1.3)

Introducing the Once Routine in Eiffel (2)

Approximating Once Routines in Java (1)

Approximating Once Routines in Java (2)

Singleton Pattern in Eiffel (1)

Singleton Pattern in Eiffel (2)

Testing Singleton Pattern in Eiffel

Singleton Pattern: Architecture

Beyond this lecture

Eiffel Testing Framework (ETF): Automated Regression & Acceptance Testing



EECS3311 A & E: Software Design
Fall 2020

CHEN-WEI WANG

Learning Objectives



Upon completing this lecture, you are expected to understand:

1. **User Interface**: Concrete vs. **Abstract**
2. **Use Case**: Interleaving Model, Events & **(Abstract) States**
3. **Acceptance Tests** vs. Unit Tests
4. **Regression Tests**

Required Tutorial



All technical details of ETF are discussed in this tutorial series:

<https://www.youtube.com/playlist?list=PL5dxAmCmjv>

[5unIqLB9XiLwBev105v3kI](https://www.youtube.com/playlist?list=PL5dxAmCmjv5unIqLB9XiLwBev105v3kI)

Take-Home Message

- Your remaining assignments are related to ETF: Lab3 & Project.
- You are no longer just given **partially** implemented classes:
 - Design decisions have already been made for you.
 - You are just to fill in the blanks (to-do's).
- ETF is in Eiffel, but try to see beyond what it allows you do:
 1. Design **your own classes and routines**.
 2. Practice **design principles**:
e.g., DbC, modularity, information hiding, single-choice, cohesion.
 3. Practice **design patterns**:
e.g., iterator, singleton.
 4. Practice **acceptance** testing and **regression** testing.

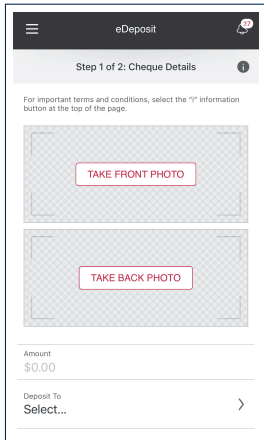
Bank ATM: Concrete User Interfaces

An ATM app has many **concrete** (implemented, functioning) UIs.

PHYSICAL INTERFACE



MOBILE INTERFACE



UI, Model, TDD

- **Separation of Concerns**

- The (**Concrete**) User Interface

Users typically interact with your application via some GUI.

e.g., web app, mobile app, or desktop app

- The **Model** (Business Logic)

Develop an application via classes and features.

e.g., a bank storing, processing, retrieving accounts & transactions

- **Test Driven Development (TDD)** In practice:

- The model should be **independent** of the UI or View.

- Do **not** wait to test the **model** when the **concrete** UI is built.

⇒ Test your software as if it was a real app

way before dedicating to the design of an actual GUI.

⇒ Use an **abstract** UI (e.g., a cmd-line UI) for this purpose.

Prototyping System with Abstract UI

- For you to quickly **prototype** a working system, you do not need to spend time on developing an elaborate, full-fledged GUI.
- The **Eiffel Testing Framework (ETF)** allows you to:
 - Generate a starter project from the specification of an **abstract UI**.
 - Focus on developing the business **model**.
 - Test your business model as if it were a real app.
- **Q.** What is an **abstract UI**?
- **Events abstracting** observable interactions with the concrete GUI (e.g., button clicks, text entering).
- **Q.** Events vs. Features (attributes & routines)?

Events	Features
interactions	computations
external	internal
observable	hidden
acceptance tests	unit tests
users, customers	programmers, developers

Bank ATM: Abstract UI

Abstract UI is the list of **events** *abstracting* observable interactions with the concrete GUI (e.g., button clicks, text entering).

```
system bank

new(id: STRING)
  -- create a new bank account for "id"
deposit(id: STRING; amount: INTEGER)
  -- deposit "amount" into the account of "id"
withdraw(id: STRING; amount: INTEGER)
  -- withdraw "amount" from the account of "id"
transfer(id1: STRING; id2: STRING; amount: INTEGER)
  -- transfer "amount" from "id1" to "id2"
```

Bank ATM: Abstract States

Abstract State is a representation of the system:

- **Including** relevant details of functionalities under **testing**
- **Excluding** other irrelevant details

e.g., An **abstract state** may show each account's owner:

```
{alan, mark, tom}
```

e.g., An **abstract state** may also show each account's balance:

```
{alan: 200, mark: 300, tom: 700}
```

e.g., An **abstract state** may show account's transactions:

```
Account Owner: alan  
  List of transactions:  
    + deposit (Oct 15): $100  
    - withdraw (Oct 18): $50  
Account Owner: mark  
  List of transactions:
```


Bank ATM: Inputs of Acceptance Tests



An **acceptance test** is a **use case** of the system under test, characterized by sequential occurrences of **abstract events**.

For example:

```
new("alan")
new("mark")
deposit("alan", 200)
deposit("mark", 100)
transfer("alan", "mark", 50)
```

Bank ATM: Outputs of Acceptance Tests (1)



Output from running an *acceptance test* is a sequence interleaving *abstract states* and *abstract events*:

$$S_0 \rightarrow e_1 \rightarrow S_1 \rightarrow e_2 \rightarrow S_2 \rightarrow \dots$$

where:

- S_0 is the *initial state*.
- S_i is the *pre-state* of event e_{i+1} [$i \geq 0$]
e.g., S_0 is the pre-state of e_1 , S_1 is the pre-state of e_2
- S_i is the *post-state* of event e_i [$i \geq 1$]
e.g., S_1 is the post-state of e_1 , S_2 is the post-state of e_2

Bank ATM: Outputs of Acceptance Tests (2)



Consider an example acceptance test output:

```
{  
->new("alan")  
  {alan: 0}  
->new("mark")  
  {alan: 0, mark: 0}  
->deposit("alan", 200)  
  {alan: 200, mark: 0}  
->deposit("mark", 100)  
  {alan: 200, mark: 100}  
->transfer("alan", "mark", 50)  
  {alan: 150, mark: 150}
```

- **Initial State?** {}
- What role does the state {alan: 200, mark: 0} play?
 - **Post-State** of deposit("alan", 200)
 - **Pre-State** of deposit("mark", 100)

Bank ATM: Acceptance Tests vs. Unit Tests



Q. Difference between an *acceptance test* and a *unit test*?

```
}  
->new("alan")  
  {alan: 0}  
->deposit("alan", 200)  
  {alan: 200}
```

```
test: BOOLEAN  
  local acc: ACCOUNT  
  do create acc.make("alan")  
    acc.add(200)  
    Result := acc.balance = 200  
  end
```

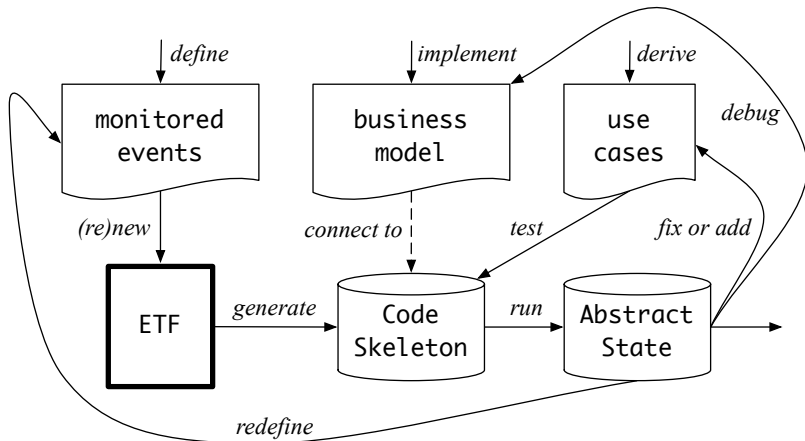
A.

- Writing a *unit test* requires knowledge about the *programming language* and details of *implementation*.
⇒ Written and run by developers
- Writing an *acceptance test* only requires familiarity with the *abstract UI* and *abstract state*.
⇒ Written and run by customers [for communication]
⇒ Written and run by developers [for testing]

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 *abstract UI* of the system under development (SUD) is defined by declaring the list of input events that might occur.
 - *Outputs* are interleaved states and events logged to the terminal, and their formats may be customized.
- An *executable* ETF project tailored for the SUD can already be generated, using these *event declarations* (specified in a plain text file), with a default *business model*.
 - Once the *business model* is implemented, there is a small number of steps to follow for developers to connect it to the generated ETF.
 - Once connected, developers may **re-run** all *acceptance tests* and observe if the expected state effects occur.

Workflow: Develop-Connect-Test



ETF: Abstract UI and Acceptance Test

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"
```



```
%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)
```

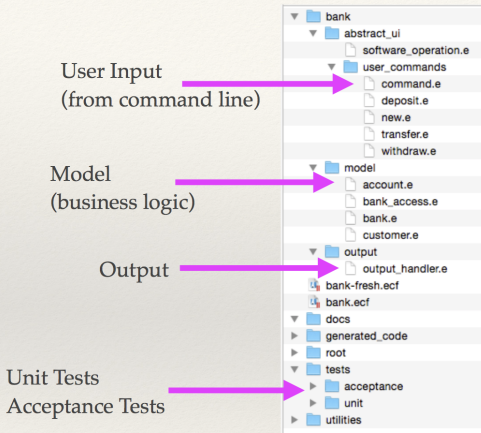
```
name: Bill, balance: 100.00
```

```
name: Pam, balance: 0.00
```

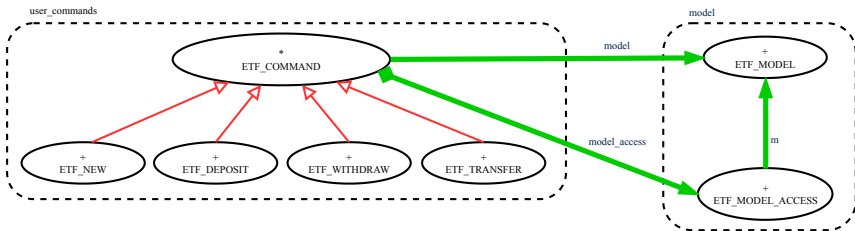
```
name: Steve, balance: 500.00
```

ETF: Generating a New Project

`etf -new bank.input.txt <directory>`



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: Implementing an Abstract Command



```
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
```

Beyond this lecture

The *singleton* pattern is instantiated in the ETF framework:

- ETF_MODEL (*shared data*)
- ETF_MODEL_ACCESS (*exclusive once access*)
- ETF_COMMAND and its effective descendants:

```

deferred class
  ETF_COMMAND
  feature -- Attributes
    model: ETF_MODEL
  feature {NONE}
    make(...)
    local
      ma: ETF_MODEL_ACCESS
    do
      ...
      model := ma.m
    end
  end
end
  
```

```

class
  ETF_DEPOSIT
  inherit
    ETF_DEPOSIT_INTERFACE
    -- which inherits ETF_COMMAND
  feature -- command
    deposit(...)
  do
    ...
    model.some_routine (...)
    ...
  end
end
end
  
```

Index (1)

Learning Objectives

Required Tutorial

Take-Home Message

Bank ATM: Concrete User Interfaces

UI, Model, TDD

Prototyping System with Abstract UI

Bank ATM: Abstract UI

Bank ATM: Abstract States

Bank ATM: Inputs of Acceptance Tests

Bank ATM: Outputs of Acceptance Tests (1)

Bank ATM: Outputs of Acceptance Tests (2)

Index (2)

Bank ATM: Acceptance Tests vs. Unit Tests

ETF in a Nutshell

Workflow: Develop-Connect-Test

ETF: Abstract UI and Acceptance Test

ETF: Generating a New Project

ETF: Architecture

ETF: Implementing an Abstract Command

Beyond this lecture

Inheritance

Readings: OOSCS2 Chapters 14 – 16



EECS3311 A & E: Software Design
Fall 2020

CHEN-WEI WANG

Learning Objectives

Upon completing this lecture, you are expected to understand:

1. Design Attempts without Inheritance (w.r.t. Cohesion, SCP)
2. Using Inheritance for Code Reuse
3. Static Type & Polymorphism
4. Dynamic Type & Dynamic Binding
5. Type Casting
6. Polymorphism & Dynamic Binding:
Routine Arguments, Routine Return Values, Collections

Aspects of Inheritance



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

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: NON_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*.
⇒ This violates the *Single Choice Principle* :
when a *change* is needed, there should be *a single place* (or *a minimal number of places*) where you need to make that change.

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
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**

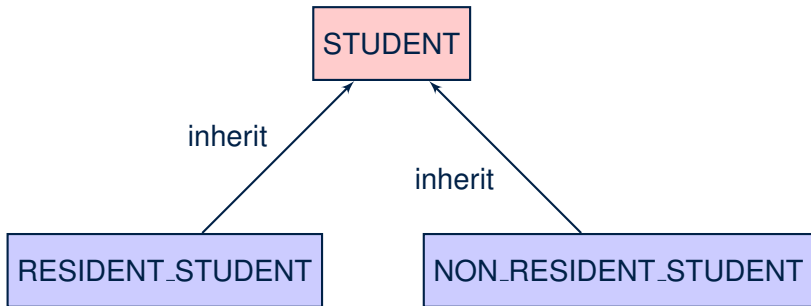
No Inheritance: A Collection of Various Kinds of Students

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.

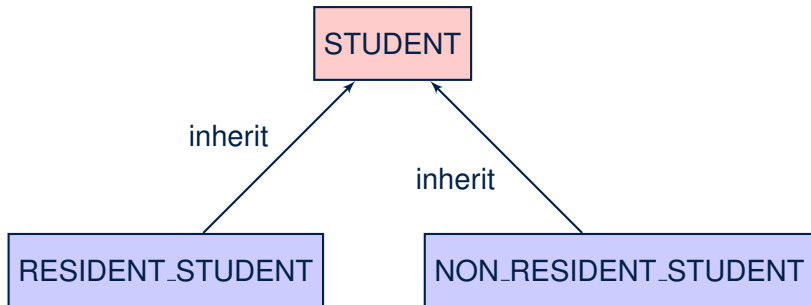
Inheritance:

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");
```

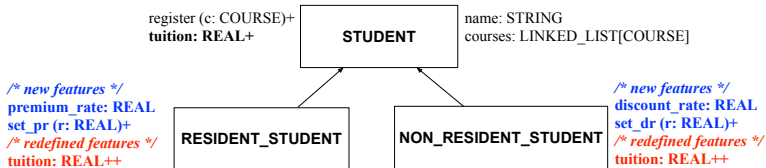
- In Eiffel:

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

- In Eiffel, the *dynamic type* can be omitted 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]

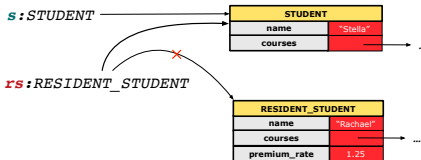
Polymorphism: Intuition (2)

```

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*!!

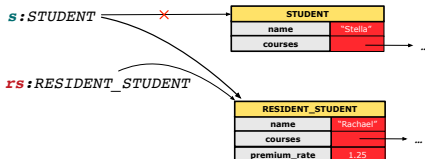
Polymorphism: Intuition (3)

```

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

\therefore `s.premium_rate` is just *never used*!!

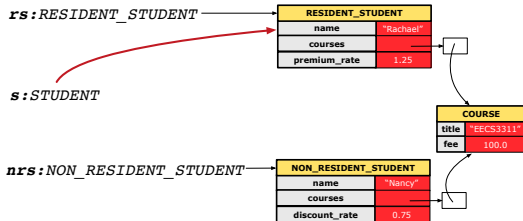
Dynamic Binding: Intuition (1)

```

1  local c : COURSE ; s : STUDENT
2      rs : RESIDENT_STUDENT ; nrs : NON_RESIDENT_STUDENT
3  do create c.make ("EECS3311", 100.0)
4      create {RESIDENT_STUDENT} rs.make("Rachael")
5      create {NON_RESIDENT_STUDENT} nrs.make("Nancy")
6      rs.set_pr(1.25); rs.register(c)
7      nrs.set_dr(0.75); nrs.register(c)
8      s := rs; ; check s.tuition = 125.0 end
9      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`.



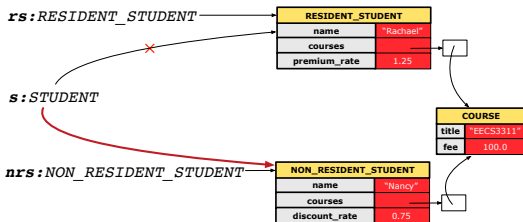
Dynamic Binding: Intuition (2)

```

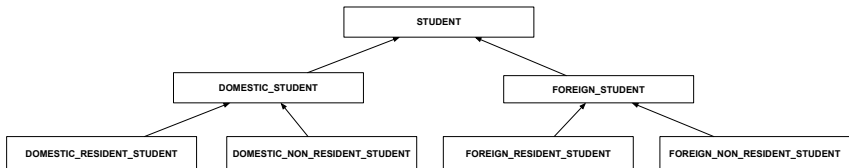
1 local c : COURSE ; s : STUDENT
2     rs : RESIDENT_STUDENT ; nrs : NON_RESIDENT_STUDENT
3 do create c.make ("EECS3311", 100.0)
4     create {RESIDENT_STUDENT} rs.make("Rachael")
5     create {NON_RESIDENT_STUDENT} nrs.make("Nancy")
6     rs.set_pr(1.25); rs.register(c)
7     nrs.set_dr(0.75); nrs.register(c)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

```

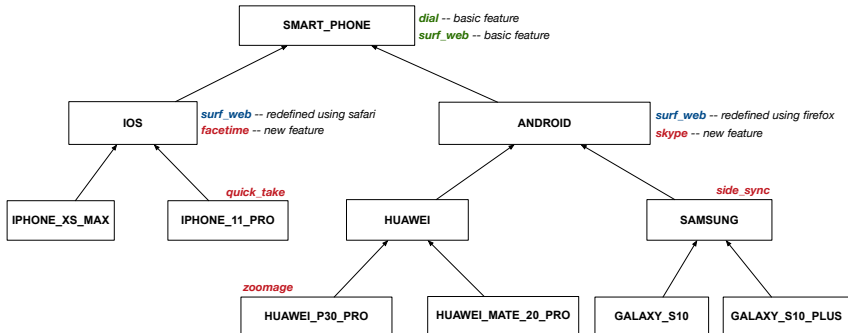
After `s := nrs` (L8), `s` points to a `NON_RESIDENT_STUDENT` object.
 ⇒ Calling `s.tuition` applies the `discount_rate`.



Multi-Level Inheritance Architecture (1)



Multi-Level Inheritance Architecture (2)



Inheritance Forms a Type Hierarchy

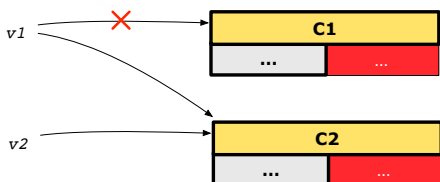
- A (data) **type** denotes a set of related *runtime values*.
 - Every *class* can be used as a type: the set of runtime *objects*.
- Use of *inheritance* creates a **hierarchy** of classes:
 - (Implicit) Root of the hierarchy is ANY.
 - Each `inherit` declaration corresponds to an upward arrow.
 - 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 `IPHONE_XS_MAX` or `IPHONE_11_PRO`.
 - \therefore Each **descendant class** of A is guaranteed to contain all code of (non-private) attributes, commands, and queries defined in A .
 - \therefore 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 `SMART_PHONE`, because the `facetime` feature is not supported in an `ANDROID` phone.
 - \therefore 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.tuioion; /* 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
  rs.set_pr (1.5)
end

```

Reference Type Casting: Syntax

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

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 approximately the same as following Java code:

```
if(jim instanceof ResidentStudent) {
    ResidentStudent rs = (ResidentStudent) jim;
    rs.set_pr(1.5);
}
else { throw new Exception("Cast Not Done."); }
```

Notes on Type Cast (1)

- `check attached {C} y then ... end` *always compiles*
- What if `C` is not an **ancestor** of `y`'s **DT**?
 - ⇒ A **runtime** assertion violation occurs!
 - ∴ `y`'s **DT** cannot fulfill the expectation of `C`.

Notes on Type Cast (2)

- Given **v** of static type **ST**, it is **violation-free** 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 create an **alias** of the object pointed by **v**, with the new **static type C**.
 ⇒ All features that are defined in **C** can be called.

```

my_phone: IOS
create {IPHONE_11_PRO} my_phone.make
  -- can only call features defined in IOS on myPhone
  -- dial, surf_web, facetime ✓ quick_take, skype, side_sync, zoomage ✗
check attached {SMART_PHONE} my_phone as sp then
  -- can now call features defined in SMART_PHONE on sp
  -- dial, surf_web ✓ facetime, quick_take, skype, side_sync, zoomage ✗
end
check attached {IPHONE_11_PRO} my_phone as ip11_pro then
  -- can now call features defined in IPHONE_11_PRO on ip11_pro
  -- dial, surf_web, facetime, quick_take ✓ skype, side_sync, zoomage ✗
end
  
```


Notes on Type Cast (3)

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 {HUAWEI} mine.make
  -- ST of mine is ANDROID; DT of mine is HUAWEI
  check attached {SMART_PHONE} mine as sp then ... end
  -- ST of sp is SMART_PHONE; DT of sp is HUAWEI
  check attached {HUAWEI} mine as huawei then ... end
  -- ST of huawei is HUAWEI; DT of huawei is HUAWEI
  check attached {SAMSUNG} mine as samsung then ... end
  -- Assertion violation
  -- ∴ SAMSUNG is not ancestor of mine's DT (HUAWEI)
  check attached {HUAWEI_P30_PRO} mine as p30_pro then ... end
  -- Assertion violation
  -- ∴ HUAWEI_P30_PRO is not ancestor of mine's DT (HUAWEI)
end
```

Polymorphism: Routine Call Parameters

```

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 value*, `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: Routine Call Arguments

```

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

```

Why Inheritance: A Polymorphic Collection 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 Polymorphic Collection 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: 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     sms.get_student(1).tuition = 750
12   and sms.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:
 - Type of queries:

```
f(a: T)
```

```
q(...): T
```


Static Type vs. Dynamic Type: When to consider which?

- *Whether or not an OOP code compiles* depends only on the **static types** of relevant variables.
 - ∴ Inferring the **dynamic type** statically is an **undecidable** problem that is inherently impossible to solve.
- *The behaviour of Eiffel code being executed at runtime*
 - e.g., which version of the routine is called
 - e.g., if a **check attached {...} as ... then ... end** assertion error will occurdepends on the **dynamic types** of relevant variables.
 - ⇒ Best practice is to visualize how objects are created (by drawing boxes) and variables are re-assigned (by drawing arrows).

Summary: Type Checking Rules

CODE	CONDITION TO BE TYPE CORRECT
<code>x := y</code>	y's ST a descendant of x's ST
<code>x.f(y)</code>	Feature <i>f</i> defined in x's ST y's ST a descendant of <i>f</i> 's parameter's ST
<code>z := x.f(y)</code>	Feature <i>f</i> defined in x's ST y's ST a descendant of <i>f</i> 's parameter's ST ST of <i>m</i> 's return value a descendant of z's ST
<code>check attached {C} y</code>	Always compiles
<code>check attached {C} y as temp then x := temp end</code>	C a descendant of x's ST
<code>check attached {C} y as temp then x.f(temp) end</code>	Feature <i>f</i> defined in x's ST C a descendant of <i>f</i> 's parameter's ST

Even if `check attached {C} y then ... end` always compiles,
a runtime assertion error occurs if C is not an **ancestor** of y's **DT!**

Beyond this lecture ...



- **Written Notes:** Static Types, Dynamic Types, Type Casts

https://www.eecs.yorku.ca/~jackie/teaching/lectures/2020/F/EECS3311/notes/EECS3311_F20_Notes_Static_Types_Cast.pdf

- **Recommended Exercise 1:**

Expand the student inheritance design ([here](#)) to reproduce the various fragments of polymorphism and dynamic binding.

- **Recommended Exercise 2:**

Create a new project (using [eiffel-new](#)) to reproduce the various fragments related to the running example of smart phones.

Index (1)

Learning Objectives

Aspects of Inheritance

Why Inheritance: A Motivating Example

The COURSE Class

No Inheritance: RESIDENT STUDENT Class

No Inheritance: NON RESIDENT STUDENT Class

No Inheritance: Testing Student Classes

No Inheritance:

Issues with the Student Classes

No Inheritance: Maintainability of Code (1)

No Inheritance: Maintainability of Code (2)

Index (2)

No Inheritance:

A Collection of Various Kinds of Students

Inheritance Architecture

Inheritance: The STUDENT Parent Class

Inheritance:

The RESIDENT STUDENT Child Class

Inheritance:

The NON RESIDENT STUDENT Child Class

Inheritance Architecture Revisited

Using Inheritance for Code Reuse

Testing the Two Student Sub-Classes

Static Type vs. Dynamic Type

Index (3)

Inheritance Architecture Revisited

Polymorphism: Intuition (1)

Polymorphism: Intuition (2)

Polymorphism: Intuition (3)

Dynamic Binding: Intuition (1)

Dynamic Binding: Intuition (2)

Multi-Level Inheritance Architecture (1)

Multi-Level Inheritance Architecture (2)

Inheritance Forms a Type Hierarchy

Inheritance Accumulates Code for Reuse

Substitutions via Assignments

Index (4)

Rules of Substitution

Reference Variable: Static Type

Reference Variable: Dynamic Type

Reference Variable:

Changing Dynamic Type (1)

Reference Variable:

Changing Dynamic Type (2)

Polymorphism and Dynamic Binding (1)

Polymorphism and Dynamic Binding (2.1)

Polymorphism and Dynamic Binding (2.2)

Reference Type Casting: Motivation

Reference Type Casting: Syntax

Index (5)

Notes on Type Cast (1)

Notes on Type Cast (2)

Notes on Type Cast (3)

Polymorphism: Routine Call Parameters

Polymorphism: Routine Call Arguments

Why Inheritance:

A Polymorphic Collection of Students

Polymorphism and Dynamic Binding:

A Polymorphic Collection of Students

Polymorphism: Return Values (1)

Polymorphism: Return Values (2)

Design Principle: Polymorphism

Index (6)

Static Type vs. Dynamic Type:

When to consider which?

Summary: Type Checking Rules

Beyond this lecture ...

Generics



EECS3311 A & E: Software Design
Fall 2020

CHEN-WEI WANG

Learning Objectives



Upon completing this lecture, you are expected to understand:

1. A **general** collection `ARRAY [ANY]` : storage vs. retrieval
2. A **generic** collection `ARRAY [G]` : storage vs. retrieval
3. Generics vs. Inheritance

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)

Say a client stores 100 distinct 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.
if attached {C1} b.get("Jim") as c1 then
  c1.f1
... -- cases for C2 to C99
elseif 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., $+: (\mathbb{Z} \times \mathbb{Z}) \rightarrow \mathbb{Z}$ 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:

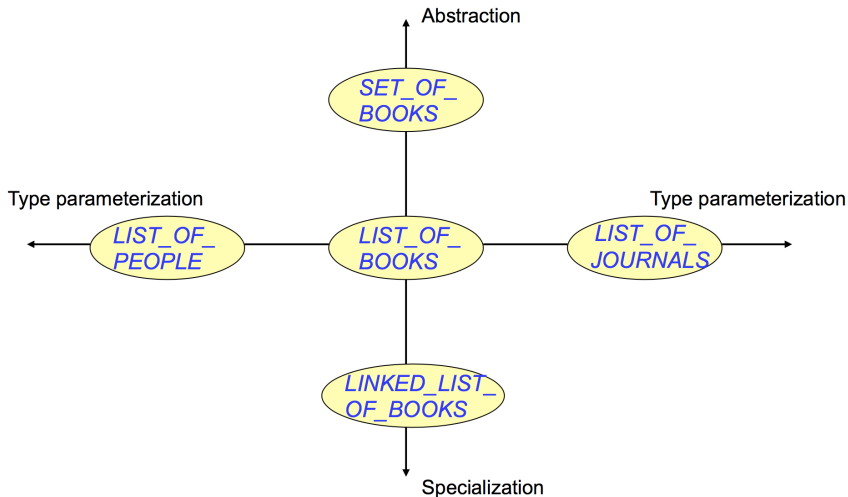
```
class DATABASE_TABLE[K, V]
  imp: DICTIONARY[V, K]
end
```

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

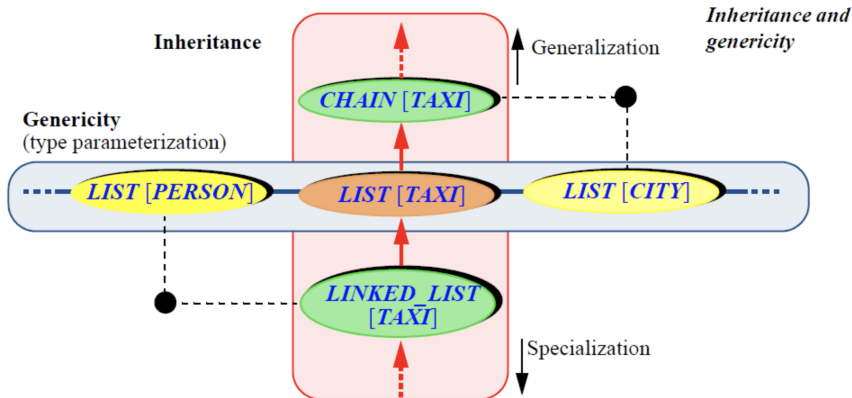
```
class STUDENT_BOOK[V]
  imp: DICTIONARY[V, STRING]
end
```

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)

Learning Objectives

Motivating Example: A Book of Any Objects

Motivating Example: Observations (1)

Motivating Example: Observations (2)

Motivating Example: Observations (2.1)

Motivating Example: Observations (2.2)

Motivating Example: Observations (3)

Parameters

Generics: Design of a Generic Book

Generics: Observations

Bad Example of using Generics

Index (2)

Instantiating Generic Parameters

Generics vs. Inheritance (1)

Generics vs. Inheritance (2)

Beyond this lecture ...

The State Design Pattern

Readings: OOSC2 Chapter 20



EECS3311 A & E: Software Design
Fall 2020

CHEN-WEI WANG

Learning Objectives



Upon completing this lecture, you are expected to understand:

1. Motivating Problem: *Interactive* Systems
2. First Design Attempt: Assembly Style
3. Second Design Attempt: *Hierarchical*, Procedural Style
4. *Template* & *State* Design Patterns: OO, *Polymorphic*

Motivating Problem

Consider the reservation panel of an **online booking system**:

-- Enquiry on Flights --

Flight sought from: To:

Departure on or after:

On or before:

Preferred airline (s):

Special requirements:

AVAILABLE FLIGHTS: 1

Flt#AA 42

Dep 8:25

Arr 7:45

Thru: Chicago

Choose next action:

0 - Exit

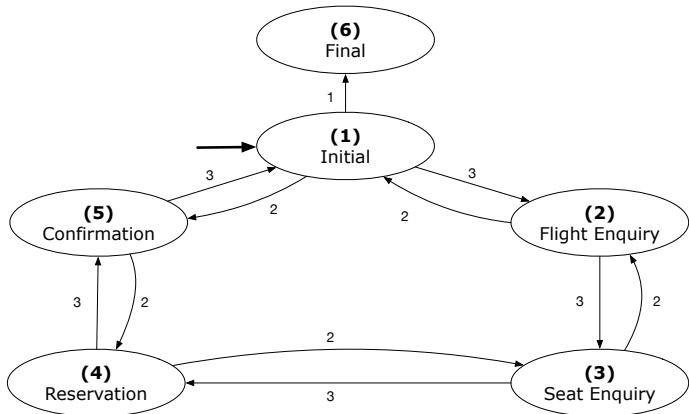
1 - Help

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 has $O(N^2)$ transitions
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  for next step  
    if wrong answer or wrong choice then  
      Output error messages  
    end  
  end  
  Process user's answer  
  case  in  
    2: goto 2_Flight_Enquiry_panel  
    3: goto 4_Reservation_panel  
  end
```

A First Attempt: Good Design?

- Runtime execution \approx 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 the *transition table* 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	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)	–	–	–

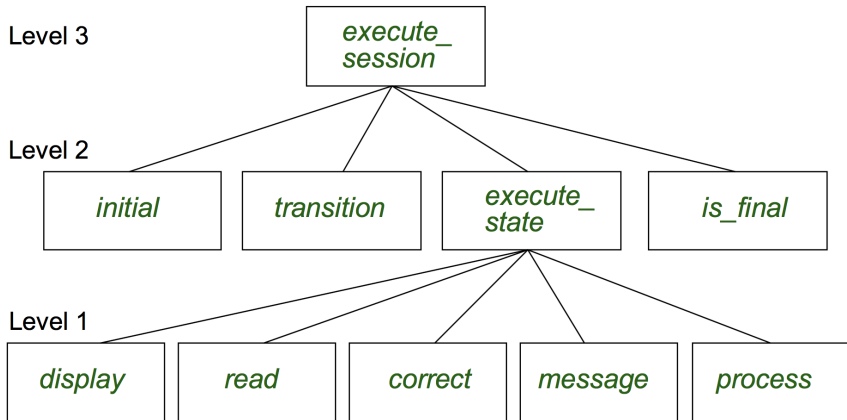
		choice		
		1	2	3
state	1	6	5	2
	2		1	3
	3		2	4
	4		3	5
	5		4	1
	6			

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
<i>display</i> (s)	Display screen outputs associated with state s
<i>read_answer</i> (s)	Read user's input for answers associated with state s
<i>read_choice</i> (s)	Read user's input for exit choice associated with state s
<i>correct</i> (s , answer)	Is the user's <i>answer</i> valid w.r.t. state s ?
<i>process</i> (s , answer)	Given that user's <i>answer</i> is valid w.r.t. state s , process it accordingly.
<i>message</i> (s , answer)	Given that user's <i>answer</i> 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?

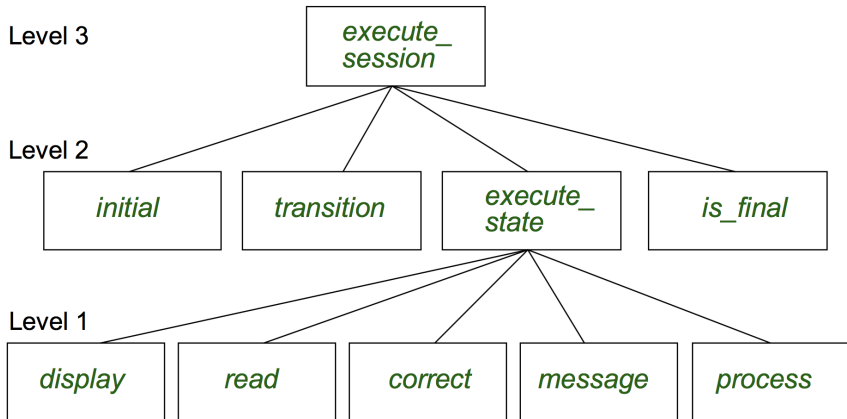
Hierarchical Solution: State Handling (3)

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

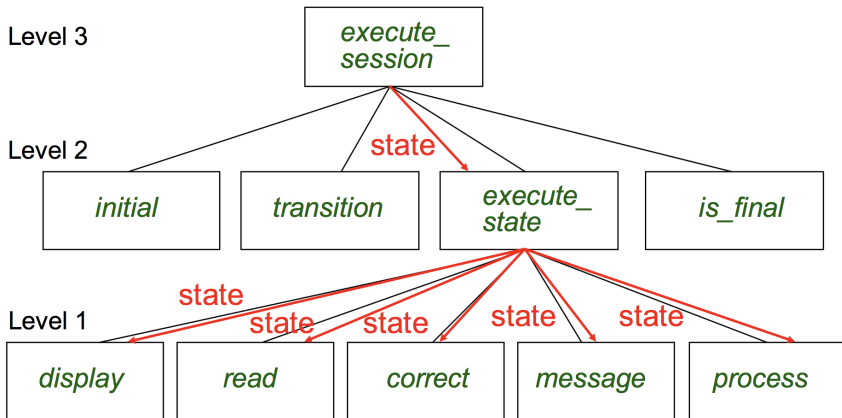
```
display(current_state: INTEGER)
  require
    valid_state: 1 ≤ current_state ≤ 6
  do
    if current_state = 1 then
      -- Display Initial Panel
    elseif current_state = 2 then
      -- Display Flight Enquiry Panel
    ...
    else
      -- Display Final Panel
    end
  end
end
```

- 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**:

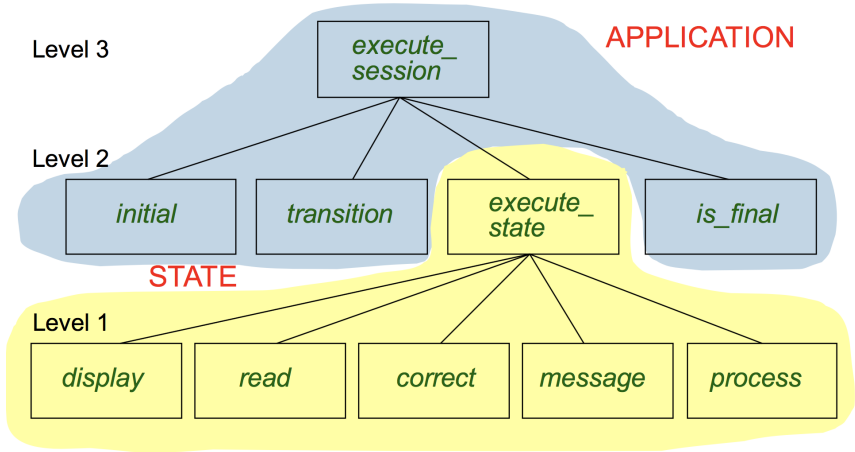
- Pass around (as *inputs*) the notion of *current_state*
- Build upon (via *discriminations*) the notion of *current_state*

```

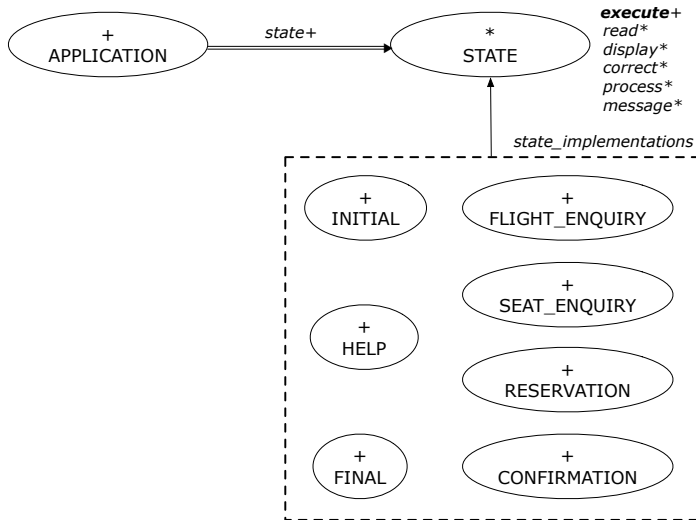
execute_state ( s: INTEGER )
display       ( s: INTEGER )
read_answer   ( s: INTEGER )
read_choice   ( s: INTEGER )
correct       ( s: INTEGER ; answer: ANSWER)
process       ( s: INTEGER ; answer: ANSWER)
message       ( s: INTEGER ; answer: ANSWER)
  
```

- ⇒ *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 Template Design Pattern

Consider the following fragment of Eiffel code:

```

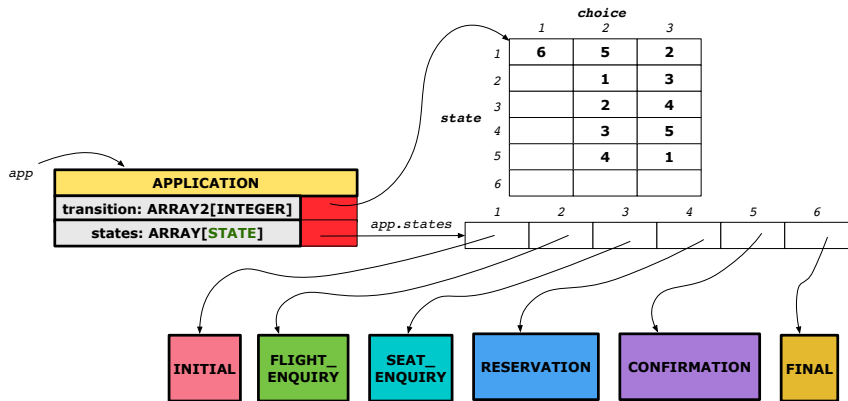
1  s: STATE
2  create {SEAT_ENQUIRY} s.make
3  s.execute
4  create {CONFIRMATION} s.make
5  s.execute
  
```

L2 and **L4**: the same version of effective feature `execute` (from the deferred class `STATE`) is called. [*template*]

L2: specific version of effective features `display`, `process`, *etc.*, (from the effective descendant class `SEAT_ENQUIRY`) is called. [*template instantiated for SEAT_ENQUIRY*]

L4: specific version of effective features `display`, `process`, *etc.*, (from the effective descendant class `CONFIRMATION`) is called. [*template instantiated for CONFIRMATION*]

APPLICATION Class: Array of STATE



APPLICATION Class (1)

```
class APPLICATION create make
feature {TEST_APPLICATION} -- 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
```

APPLICATION Class (2)

```
class APPLICATION
feature {TEST_APPLICATION} -- 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
```

Example Test: Non-Interactive Session

```
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
```

APPLICATION Class (3): Interactive Session



```
class APPLICATION
feature {TEST_APPLICATION} -- 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(create {INITIAL}.make, 4)
```

- Choose an initial state.

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

- Build the transition table.

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

- Run the application.

```
app.execute_session
```

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 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)

Learning Objectives

Motivating Problem

State Transition Diagram

Design Challenges

A First Attempt

A First Attempt: Good Design?

A Top-Down, Hierarchical Solution

Hierarchical Solution: Good Design?

Hierarchical Solution:

Top-Down Functional Decomposition

Hierarchical Solution: System Control

Index (2)

Hierarchical Solution: State Handling (1)

Hierarchical Solution: State Handling (2)

Hierarchical Solution: State Handling (3)

Hierarchical Solution: Visible Architecture

Hierarchical Solution: Pervasive States

Law of Inversion

Grouping by Data Abstractions

Architecture of the State Pattern

The STATE ADT

The Template Design Pattern

APPLICATION Class: Array of STATE

Index (3)

APPLICATION Class (1)

APPLICATION Class (2)

Example Test: Non-Interactive Session

APPLICATION Class (3): Interactive Session

Building an Application

Top-Down, Hierarchical vs. OO Solutions

Observer Design Pattern

Event-Driven Design



EECS3311 A & E: Software Design
Fall 2020

CHEN-WEI WANG

Learning Objectives

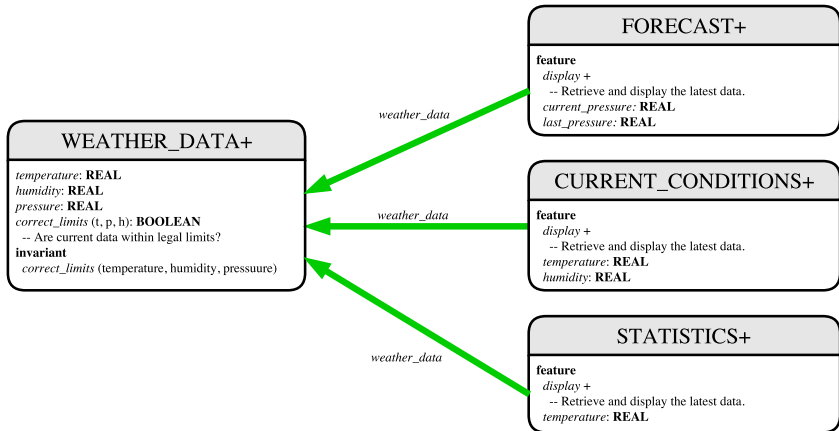
1. Motivating Problem: *Distributed* Clients and Servers
2. First Design Attempt: Remote Procedure Calls
3. Second Design Attempt: *Observer Design Pattern*
4. Third Design Attempt: **Event-Driven Design** (Java vs. Eiffel)
5. Use of agent
[\approx C function pointers \approx C# delegates \approx Java lambda]

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*:
 - *Forecast*: if expecting for rainy weather due to reduced *pressure*.
 - *Condition*: *temperature* in celsius and *humidity* in percentages.
 - *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(t, p, h)
  ensure
    temperature = t and pressure = p and humidity = h
invariant
  correct_limits(temperature, pressure, humidity)
end
```

Implementing the First Design (2.1)

```
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 = wd
  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 (2.2)

```
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 (2.3)

```
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 = wd
  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)

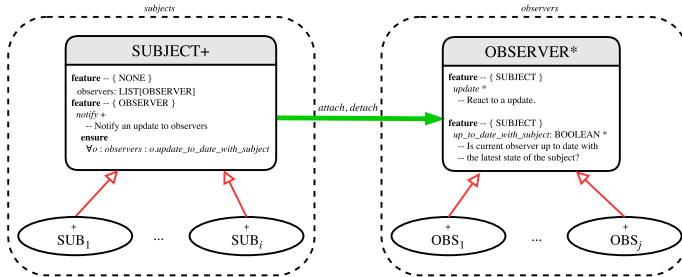
```
1 class WEATHER_STATION create make
2 feature -- Attributes
3   cc: CURRENT_CONDITIONS ; fd: FORECAST ; sd: STATISTICS
4   wd: WEATHER_DATA
5 feature -- Commands
6   make
7     do create wd.make (9, 75, 25)
8       create cc.make (wd) ; create fd.make (wd) ; create sd.make(wd)
9
10      wd.set_measurements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

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

First Design: Good Design?

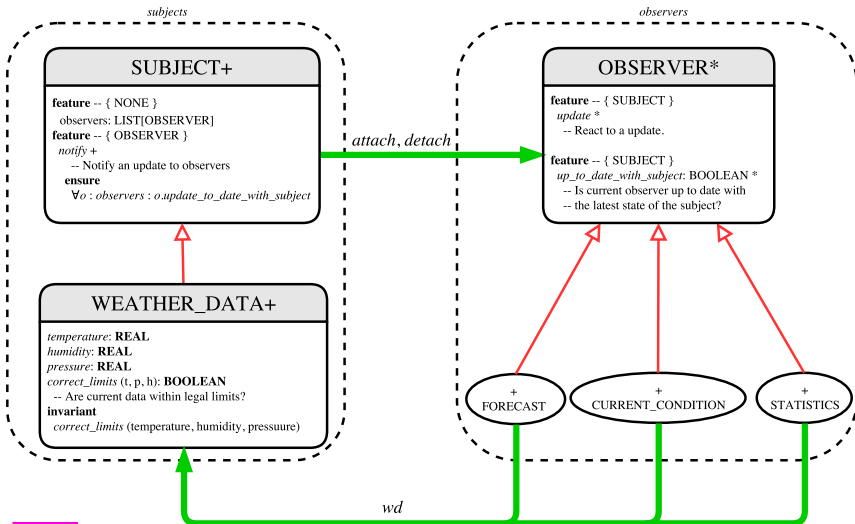
- Each application (CURRENT_CONDITION, FORECAST, STATISTICS) *cannot know* when the weather data change.
⇒ All applications have to periodically initiate updates in order to keep the `display` results up to date.
 - ∴ Each inquiry of current weather data values is *a remote call*.
 - ∴ Waste of computing 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 is *subscribed/attached/registered* to the weather data.
 - The weather data *publish/notify* new changes.
⇒ 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)



```
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 (1.2)

```
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 (2.1)



```
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
end
```

Each effective descendant class of `OBSERVER` should:

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

Implementing the Observer Pattern (2.2)



```
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 (2.3)



```
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 (2.4)



```
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 (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    cc.display ; fd.display ; sd.display
13    cc.display ; fd.display ; sd.display
14
15    wd.set_measurements (11, 90, 20)
16    wd.notify
17    cc.display ; fd.display ; sd.display
18 end
19 end
```

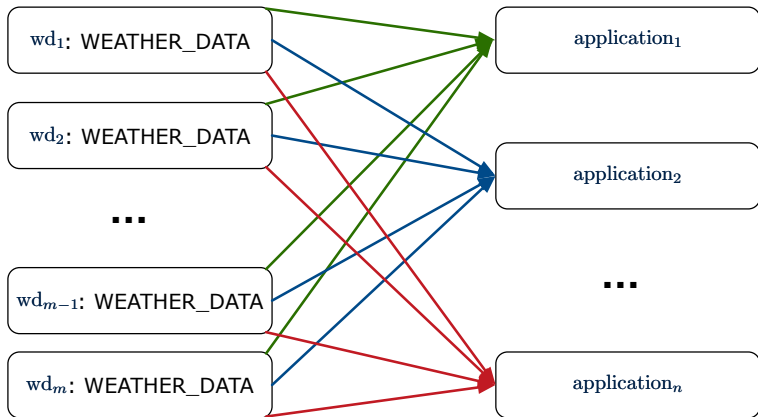
L13: cc, fd, sd make use of “cached” data values.

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)

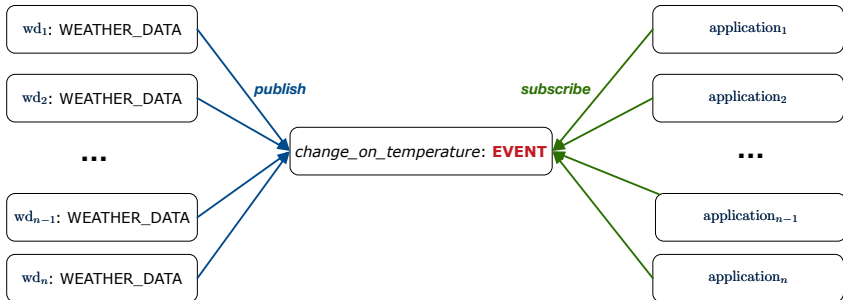
What happens at runtime when building a *many-to-many* relationship using the *observer pattern*?



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.
 - ⇒ 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 [ARGUMENT -> TUPLE ]
2 create make
3 feature -- Initialization
4   actions: LINKED_LIST[PROCEDURE[ARGUMENT] ]
5   make do create actions.make end
6 feature
7   subscribe (an_action: PROCEDURE[ARGUMENT])
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: ARGUMENT)
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 ARGUMENT: any class that instantiates ARGUMENT 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_humidity.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` \approx `... (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 [ARGUMENT ]]
```

- **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**.

Beyond this lecture...



Play with the source code of with the various designs (with an IDE debugger):

- `non_observer.zip`
- `observer.zip`
- `JavaObserverEvent.zip`
- `observer_event.zip`

[1st Design Attempt]
[Observer Design Pattern]
[Event-Driven Design in Java]
[Event-Driven Design in Eiffel]

Index (1)

Learning Objectives

Motivating Problem

First Design: Weather Station

Implementing the First Design (1)

Implementing the First Design (2.1)

Implementing the First Design (2.2)

Implementing the First Design (2.3)

Implementing the First Design (3)

First Design: Good Design?

Observer Pattern: Architecture

Observer Pattern: Weather Station

Index (2)

Implementing the Observer Pattern (1.1)

Implementing the Observer Pattern (1.2)

Implementing the Observer Pattern (2.1)

Implementing the Observer Pattern (2.2)

Implementing the Observer Pattern (2.3)

Implementing the Observer Pattern (2.4)

Implementing the Observer Pattern (3)

Observer Pattern: Limitation? (1)

Observer Pattern: Limitation? (2)

Event-Driven Design (1)

Event-Driven Design (2)

Index (3)

Event-Driven Design: Implementation

Event-Driven Design in Java (1)

Event-Driven Design in Java (2)

Event-Driven Design in Java (3)

Event-Driven Design in Java (4)

Event-Driven Design in Eiffel (1)

Event-Driven Design in Eiffel (2)

Event-Driven Design in Eiffel (3)

Event-Driven Design in Eiffel (4)

Event-Driven Design: Eiffel vs. Java

Beyond this lecture...

Subcontracting

Readings: OOSCS2 Chapters 14 – 16



EECS3311 A & E: Software Design
Fall 2020

CHEN-WEI WANG

Aspects of Inheritance



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

Learning Objectives



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

Background of Logic (1)

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

P_2 **requires less** than P_1 if

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

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

More concisely:

$$P_1 \Rightarrow P_2$$

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

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

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

Background of Logic (2)

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

Q_2 *ensures more* than Q_1 if

Q_2 is *stricter* on (thus *allowing less*) outputs than Q_1 does.

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

More concisely:

$$Q_2 \Rightarrow Q_1$$

e.g., For query $q(i: \text{INTEGER}) : \text{BOOLEAN}$,

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

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

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

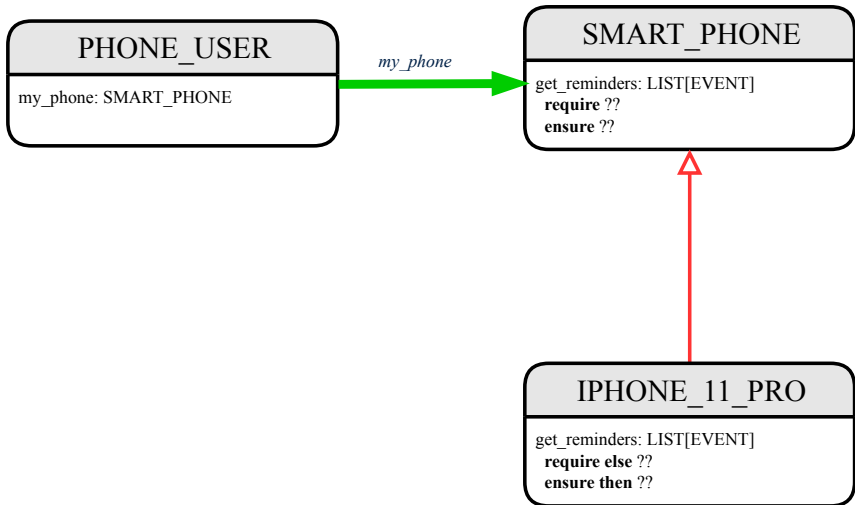
Inheritance and Contracts (1)

- The fact that we allow **polymorphism**:

```
local my_phone: SMART_PHONE
      i_phone: IPHONE_11_PRO
      samsung_phone: GALAXY_S10_PLUS
      huawei_phone: HUAWAI_P30_PRO
do my_phone := i_phone
   my_phone := samsung_phone
   my_phone := huawei_phone
```

- suggests that these instances may **substitute** for each other.
- Intuitively, when expecting SMART_PHONE, we can substitute it by instances of any of its **descendant** classes.
 - ∴ Descendants **accumulate code** from its ancestors and can thus **meet expectations** on their ancestors.
 - Such **substitutability** can be reflected on contracts, where a **substitutable instance** will:
 - Not** require more from clients for using the services.
 - Not** ensure less to clients for using the services.

Inheritance and Contracts (2.1)



Inheritance and Contracts (2.2)

```
class SMART_PHONE
  get_reminders: LIST[EVENT]
  require
     $\alpha$ : battery_level  $\geq$  0.1 -- 10%
  ensure
     $\beta$ :  $\forall e$ :Result | e happens today
end
```

```
class IPHONE_11_PRO
inherit SMART_PHONE redefine get_reminders end
get_reminders: LIST[EVENT]
require else
   $\gamma$ : battery_level  $\geq$  0.15 -- 15%
ensure then
   $\delta$ :  $\forall e$ :Result | e happens today or tomorrow
end
```

Contracts in descendant class *IPHONE_11_PRO* are *not suitable*.
 ($\textit{battery_level} \geq 0.1 \Rightarrow \textit{battery_level} \geq 0.15$) is not a tautology.
 e.g., A client able to get reminders on a *SMART_PHONE*, when battery level is 12%, will fail to do so on an *IPHONE_11_PRO*.

Inheritance and Contracts (2.3)

```
class SMART_PHONE
  get_reminders: LIST[EVENT]
  require
     $\alpha$ : battery_level  $\geq$  0.1 -- 10%
  ensure
     $\beta$ :  $\forall e$ :Result |  $e$  happens today
end
```

```
class IPHONE_11_PRO
inherit SMART_PHONE redefine get_reminders end
get_reminders: LIST[EVENT]
require else
   $\gamma$ : battery_level  $\geq$  0.15 -- 15%
ensure then
   $\delta$ :  $\forall e$ :Result |  $e$  happens today or tomorrow
end
```

Contracts in descendant class `IPHONE_11_PRO` are *not suitable*.
(e happens ty. or tw.) \Rightarrow (e happens ty.) not tautology.
e.g., A client receiving today's reminders from `SMART_PHONE` are shocked by tomorrow-only reminders from `IPHONE_11_PRO`.

Inheritance and Contracts (2.4)

```
class SMART_PHONE
  get_reminders: LIST[EVENT]
  require
     $\alpha$ : battery_level  $\geq$  0.1 -- 10%
  ensure
     $\beta$ :  $\forall e$ :Result |  $e$  happens today
end
```

```
class IPHONE_11_PRO
inherit SMART_PHONE redefine get_reminders end
get_reminders: LIST[EVENT]
require else
   $\gamma$ : battery_level  $\geq$  0.05 -- 5%
ensure then
   $\delta$ :  $\forall e$ :Result |  $e$  happens today between 9am and 5pm
end
```

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

- **Require the same or less**

$$\alpha \Rightarrow \gamma$$

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

Inheritance and Contracts (2.5)

```
class SMART_PHONE
  get_reminders: LIST[EVENT]
  require
     $\alpha$ : battery_level  $\geq$  0.1 -- 10%
  ensure
     $\beta$ :  $\forall e$ :Result |  $e$  happens today
end
```

```
class IPHONE_11_PRO
  inherit SMART_PHONE redefine get_reminders end
  get_reminders: LIST[EVENT]
  require else
     $\gamma$ : battery_level  $\geq$  0.05 -- 5%
  ensure then
     $\delta$ :  $\forall e$ :Result |  $e$  happens today between 9am and 5pm
end
```

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

- **Ensure the same or more**

$$\delta \Rightarrow \beta$$

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

Contract Redeclaration Rule (1)

- In the context of some feature in a descendant class:
 - Use `require else` to redeclare its precondition.
 - Use `ensure then` to redeclare its postcondition.
- The resulting *runtime assertions checks* are:
 - `original_pre or else new_pre`
 - ⇒ Clients **able to satisfy** *original_pre* will not be shocked.
 - ∴ $true \vee new_pre \equiv true$
 - A **precondition violation** will **not** occur as long as clients are able to satisfy what is required from the ancestor classes.
 - `original_post and then new_post`
 - ⇒ **Failing to gain** *original_post* will be reported as an issue.
 - ∴ $false \wedge new_post \equiv false$
 - A **postcondition violation** occurs (as expected) if clients do not receive at least those benefits promised from the ancestor classes.

Contract Redeclaration Rule (2.1)

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

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

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

$$\therefore \text{true} \vee \text{new_pre} \equiv \text{true}$$

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

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

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

$$\therefore \text{true} \wedge \text{new_post} \equiv \text{new_post}$$

Contract Redeclaration Rule (2.2)

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

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

- Unspecified *new_pre* is as if declaring `require else false`
 $\therefore original_pre \vee \text{false} \equiv original_pre$

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

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

- Unspecified *new_post* is as if declaring `ensure then true`
 $\therefore original_post \wedge \text{true} \equiv original_post$

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?
 $(vertices.count = 5) \wedge (vertices.count = 4) \equiv \text{false}$

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
end
```

(Static) Design Time :

- $original_pre \Rightarrow new_pre$ should be proved as a tautology
- $new_post \Rightarrow original_post$ should be proved as a tautology

(Dynamic) Runtime :

- $original_pre \vee new_pre$ is checked
- $original_post \wedge new_post$ is checked

Index (1)

Aspects of Inheritance

Learning Objectives

Background of Logic (1)

Background of Logic (2)

Inheritance and Contracts (1)

Inheritance and Contracts (2.1)

Inheritance and Contracts (2.2)

Inheritance and Contracts (2.3)

Inheritance and Contracts (2.4)

Inheritance and Contracts (2.5)

Contract Redeclaration Rule (1)

Index (2)

Contract Redeclaration Rule (2.1)

Contract Redeclaration Rule (2.2)

Invariant Accumulation

Inheritance and Contracts (3)

The Composite Design Pattern



EECS3311 A & E: Software Design
Fall 2020

CHEN-WEI WANG

Learning Objectives

1. Motivating Problem: *Recursive* Systems
2. Two Design Attempts
3. Multiple Inheritance
4. Third Design Attempt: *Composite Design Pattern*
5. Implementing and Testing the Composite Design Pattern

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.

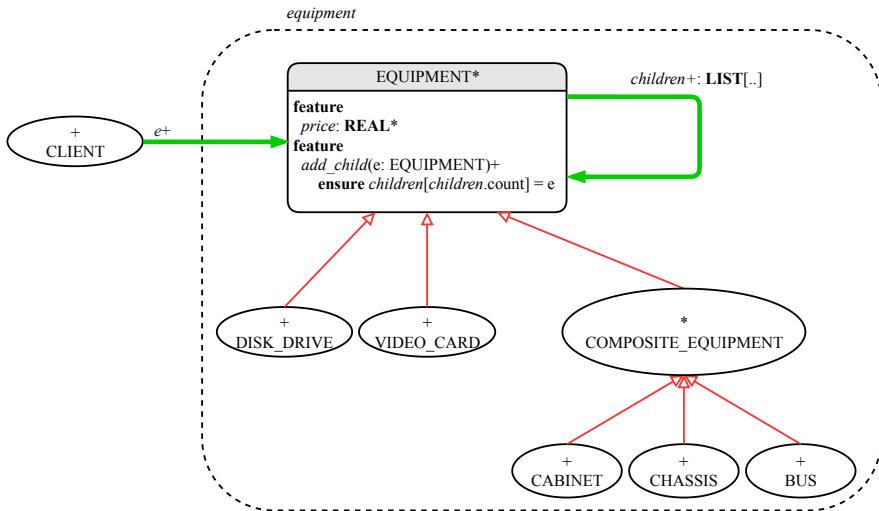
Motivating Problem (2)

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



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

Design Attempt 1: Architecture



Design Attempt 1: Flaw?

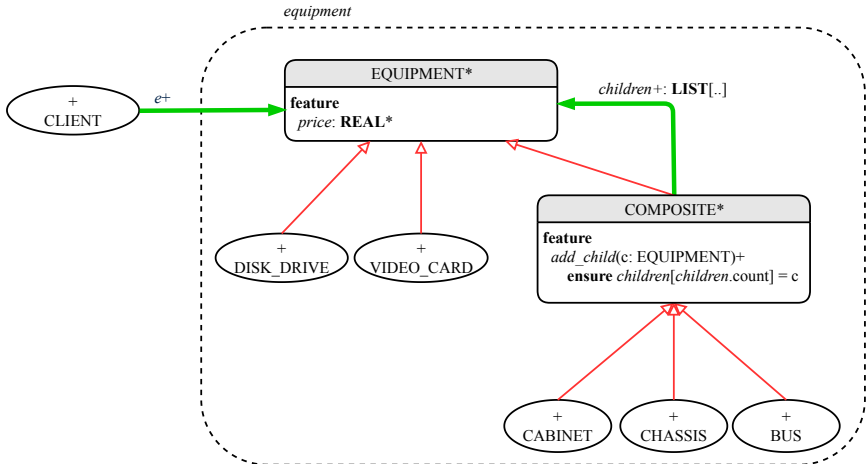
Q: Any flaw of this first design?

A: Two “composite” features defined at the `EQUIPMENT` level:

- `children: LIST[EQUIPMENT]`
- `add(child: EQUIPMENT)`

⇒ Inherited to all *base* equipments (e.g., `HARD_DRIVE`) that do not apply to such features.

Design Attempt 2: Architecture



Design Attempt 2: Flaw?

Q: Any flaw of this second design?

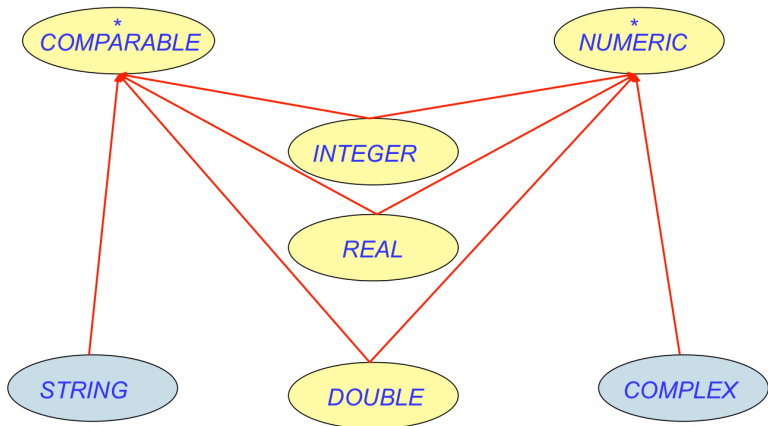
A: Two “composite” features defined at the `COMPOSITE` level:

- `children: LIST[EQUIPMENT]`
- `add(child: EQUIPMENT)`

⇒ Multiple instantiations of the composite architecture (e.g., equipments, furnitures) require duplicates of the `COMPOSITE` class.

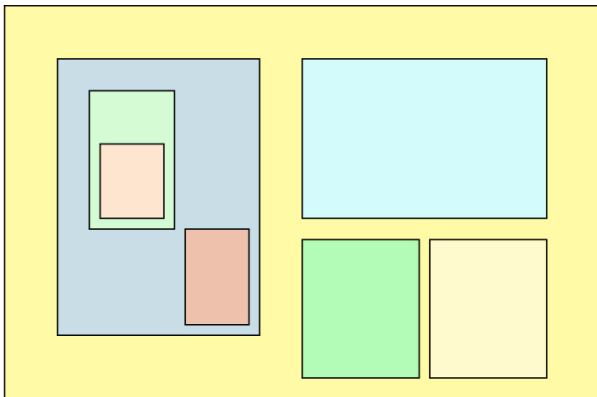
Multiple Inheritance: Combining Abstractions (1)

A class may have two more parent classes.



MI: Combining Abstractions (2.1)

Q: How do you design class(es) for nested windows?



Hints: height, width, xpos, ypos, change width, change height, move, parent window, descendant windows, add child window

MI: Combining Abstractions (2.2)

A: Separating *Graphical* features and *Hierarchical* features

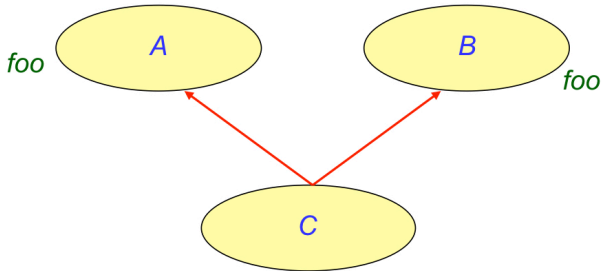
```
class RECTANGLE
  feature -- Queries
    width, height: REAL
    xpos, ypos: REAL
  feature -- Commands
    make (w, h: REAL)
    change_width
    change_height
    move
end
```

```
class TREE[G]
  feature -- Queries
    descendants: ITERABLE[G]
  feature -- Commands
    add (c: G)
      -- Add a child 'c'.
end
```

```
class WINDOW
  inherit
    RECTANGLE
    TREE[WINDOW]
end
```

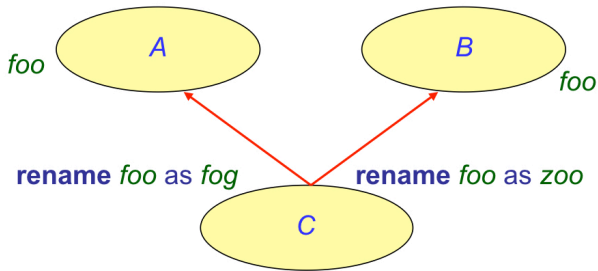
```
test_window: BOOLEAN
  local w1, w2, w3, w4: WINDOW
  do
    create w1.make(8, 6) ; create w2.make(4, 3)
    create w3.make(1, 1) ; create w4.make(1, 1)
    w2.add(w4) ; w1.add(w2) ; w1.add(w3)
    Result := w1.descendants.count = 2
  end
```

MI: Name Clashes



In class C, feature `foo` inherited from ancestor class A clashes with feature `foo` inherited from ancestor class B.

MI: Resolving Name Clashes



```
class C
  inherit
  A rename foo as fog end
  B rename foo as zoo end
  ...
```

		o.foo	o.fog	o.zoo
o:	A	✓	✗	✗
o:	B	✓	✗	✗
o:	C	✗	✓	✓

Implementing the Composite Pattern (1)

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

```
class
  CARD
inherit
  EQUIPMENT
feature {NONE}
  unit_price: REAL
feature
  make (n: STRING; p: REAL)
    do name := n ; unit_price := p end
  price
    do Result := unit_price end
end
```


Implementing the Composite Pattern (2.1)



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

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

Exercise: Make the COMPOSITE class *iterable*.

Implementing the Composite Pattern (2.2)



```
deferred class
  COMPOSITE_EQUIPMENT
inherit
  EQUIPMENT
  COMPOSITE [EQUIPMENT]
feature
  make (n: STRING)
    -- Child classes will declare this command as a constructor.
    do name := n ; create children.make end
  price : REAL -- price is a query
    -- Sum the net prices of all sub-equipments
  do
    across
      children is c
    loop
      Result := Result + c.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
```

Summary: 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.

Index (1)

Learning Objectives

Motivating Problem (1)

Motivating Problem (2)

Design Attempt 1: Architecture

Design Attempt 1: Flaw?

Design Attempt 2: Architecture

Design Attempt 2: Flaw?

Multiple Inheritance:

Combining Abstractions (1)

MI: Combining Abstractions (2.1)

MI: Combining Abstractions (2.2)

Index (2)

MI: Name Clashes

MI: Resolving Name Clashes

The Composite Pattern: Architecture

Implementing the Composite Pattern (1)

Implementing the Composite Pattern (2.1)

Implementing the Composite Pattern (2.2)

Testing the Composite Pattern

Summary: The Composite Pattern

The Visitor Design Pattern



EECS3311 A & E: Software Design
Fall 2020

CHEN-WEI WANG

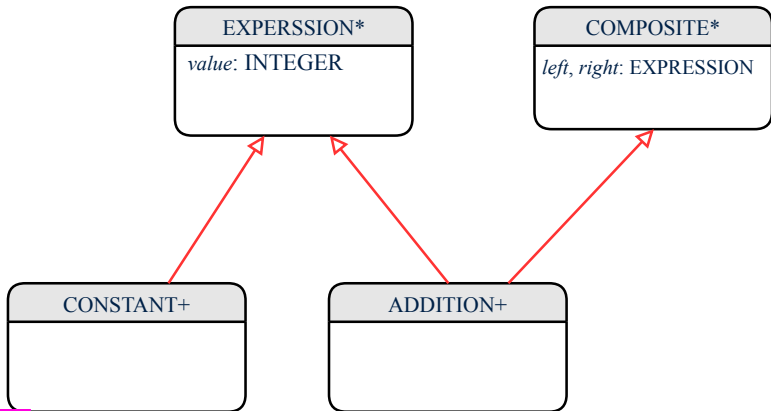
Learning Objectives



1. Motivating Problem: *Processing* Recursive Systems
2. First Design Attempt: Cohesion & Single-Choice Principle?
3. Open-Closed Principle
4. Second Design Attempt: *Visitor Design Pattern*
5. Implementing and Testing the Visitor Design Pattern

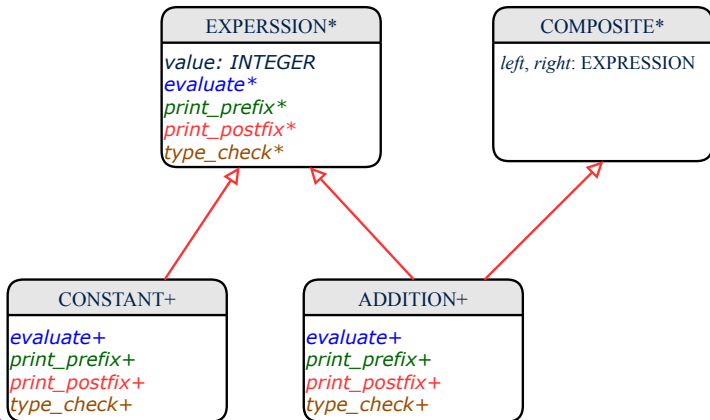
Motivating Problem (1)

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



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.

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/modify the *open* (unstable) part of system.
- May not add/modify the *closed* (stable) part of system.

e.g., In designing the application of an expression language:

- **ALTERNATIVE 1:**
Syntactic constructs of the language may be *open*, whereas operations on the language may be *closed*.
- **ALTERNATIVE 2:**
Syntactic constructs of the language may be *closed*, whereas operations on the language may be *open*.

Visitor Pattern

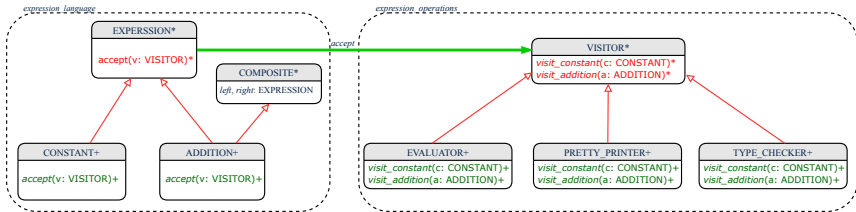
- **Separation of concerns** :
 - Set of language constructs
 - Set of operations

⇒ Classes from these two sets are **decoupled** and organized into two separate clusters.
- **Open-Closed Principle (OCP)** : [**ALTERNATIVE 2**]
 - **Closed**, staple part of system: set of language constructs
 - **Open**, unstable part of system: set of operations

⇒ **OCP** helps us determine if Visitor Pattern is **applicable** .

⇒ If it was decided that language constructs are **open** and operations are **closed**, then do **not** use Visitor Pattern.

Visitor Pattern: Architecture



Visitor Pattern Implementation: Structures

Cluster *expression_language*

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

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

```
class ADDITION
inherit EXPRESSION COMPOSITE
...
  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*.

```
deferred 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 inherit VISITOR
  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** \Rightarrow Call accept in **ADDITION**

v.visit_ **addition** (add)

2. **DT** of v is **EVALUATOR** \Rightarrow 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** change often.
 - ⇒ Do not use visitor if the **structure** changes often.

Beyond this Lecture...



- Learn about implementing the Composite and Visitor Patterns, from scratch, in this tutorial series:

<https://www.youtube.com/playlist?list=PL5dxAmCmjv4z5eXGW-ZBqsS2WZTvyBHY2>

- The Visitor Pattern can be used to facilitate the development of a language compiler:

<https://www.youtube.com/playlist?list=PL5dxAmCmjv4FGYtGzcvBeoS-BobRTJLc>

Index (1)

Learning Objectives

Motivating Problem (1)

Motivating Problem (2)

Problems of Extended Composite Pattern

Open/Closed Principle

Visitor Pattern

Visitor Pattern: Architecture

Visitor Pattern Implementation: Structures

Visitor Pattern Implementation: Operations

Testing the Visitor Pattern

To Use or Not to Use the Visitor Pattern

Index (2)



Beyond this Lecture...

Program Correctness

OOSC2 Chapter 11



EECS3311 A & E: Software Design
Fall 2020

CHEN-WEI WANG

Learning Objectives

1. Motivating Examples: *Program Correctness*
2. *Hoare Triple*
3. *Weakest Precondition* (*wp*)
4. Rules of *wp Calculus*
5. Contract of Loops (**invariant** vs. **variant**)
6. **Correctness Proofs** of Loops

Assertions: Weak vs. Strong

- Describe each assertion as **a set of satisfying value**.
 - $x > 3$ has satisfying values $\{ x \mid x > 3 \} = \{ 4, 5, 6, 7, \dots \}$
 - $x > 4$ has satisfying values $\{ x \mid x > 4 \} = \{ 5, 6, 7, \dots \}$
- 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

Assertions: Preconditions

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

P_2 **requires less** than P_1 if

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

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

More concisely:

$$P_1 \Rightarrow P_2$$

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

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

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

Assertions: Postconditions

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

Q_2 **ensures more** than Q_1 if
 Q_2 is **stricter** on (thus **allowing less**) outputs than Q_1 does.

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

More concisely:

$$Q_2 \Rightarrow Q_1$$

e.g., For query $q(i: \text{INTEGER}) : \text{BOOLEAN}$,

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

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

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

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

\therefore 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

∴ 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*.
 ⇒ 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*: $\{Q\} S \{R\}$
 - e.g., $\{i > 3\} i := i + 9 \{i > 13\}$
 - e.g., $\{i > 5\} i := i + 9 \{i > 13\}$
 - If $\{Q\} S \{R\}$ **can** be proved **TRUE**, then the **S** is correct.
 e.g., $\{i > 5\} i := i + 9 \{i > 13\}$ can be proved TRUE.
 - If $\{Q\} S \{R\}$ **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]

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*.
 - If $Q \Rightarrow wp(S, R)$, then any execution started in a state satisfying Q will terminate in a state satisfying R.
 - If $Q \not\Rightarrow wp(S, R)$, then some execution started in a state satisfying Q will terminate in a state violating R.
- S can be:
 - Assignments ($x := y$)
 - Alternations (**if ... then ... else ... end**)
 - Sequential compositions ($S_1 ; S_2$)
 - Loops (**from ... until ... loop ... end**)
- We will learn 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 x_0 to denote its *pre-state (old)* value.
- We write 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 may choose to write “ b ” rather than “ b_0 ” in preconditions
 \therefore All variables are pre-state values in preconditions
 - We don't write “ b_0 ” in program
 \therefore there might be *multiple intermediate values* of a variable due to sequential composition

wp Rule: Assignments (1)

$$wp(x := e, R) = R[x := e]$$

$R[x := e]$ means to substitute all *free occurrences* of variable x in postcondition R by expression e .

wp Rule: Assignments (2)

Recall:

$$\{Q\} S \{R\} \equiv Q \Rightarrow wp(S, R)$$

How do we prove $\{Q\} x := e \{R\}$?

$$\{Q\} x := e \{R\} \iff Q \Rightarrow \underbrace{R[x := e]}_{wp(x := e, R)}$$

wp Rule: Assignments (3) Exercise

What is the weakest precondition for a program $x := x + 1$ to establish the postcondition $x > x_0$?

$$\{??\} x := x + 1 \{x > x_0\}$$

For the above Hoare triple to be **TRUE**, it must be that $?? \Rightarrow wp(x := x + 1, x > x_0)$.

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

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 $x = 23$?

$$\{??\} x := x + 1 \{x = 23\}$$

For the above Hoare triple to be **TRUE**, it must be that $?? \Rightarrow wp(x := x + 1, x = 23)$.

$$\begin{aligned} & wp(x := x + 1, x = 23) \\ = & \{Rule\ of\ wp:\ Assignments\} \\ & x = 23[x := x_0 + 1] \\ = & \{Replacing\ x\ by\ x_0 + 1\} \\ & x_0 + 1 = 23 \\ = & \{arithmetic\} \\ & x_0 = 22 \end{aligned}$$

Any precondition weaker than $x = 22$ is not OK.

wp Rule: Assignments (4) Revisit

Given $\{??\}n := n + 9\{n > 13\}$:

- $n > 4$ 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 $n > 13$.

wp Rule: Alternations (1)

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

The *wp* of an alternation is such that **all branches** are able to establish the postcondition ***R***.

wp Rule: Alternations (2)

Recall: $\{Q\} S \{R\} \equiv Q \Rightarrow wp(S, R)$

How do we prove that $\{Q\}$ if B then S_1 else S_2 end $\{R\}$?

```

{Q}
if B then
  {Q ∧ B} S1 {R}
else
  {Q ∧ ¬B} S2 {R}
end
{R}
  
```

$$\begin{aligned}
 & \{Q\} \text{ if } B \text{ then } S_1 \text{ else } S_2 \text{ end } \{R\} \\
 & \iff \left(\begin{array}{c} \{Q \wedge B\} S_1 \{R\} \\ \wedge \\ \{Q \wedge \neg B\} S_2 \{R\} \end{array} \right) \iff \left(\begin{array}{c} (Q \wedge B) \Rightarrow wp(S_1, R) \\ \wedge \\ (Q \wedge \neg B) \Rightarrow wp(S_2, R) \end{array} \right)
 \end{aligned}$$

wp Rule: Alternations (3) Exercise

Is this program correct?

```

{x > 0 ∧ y > 0}
if x > y then
  bigger := x ; smaller := y
else
  bigger := y ; smaller := x
end
{bigger ≥ smaller}
  
```

$$\left(\begin{array}{l} \{(x > 0 \wedge y > 0) \wedge (x > y)\} \\ \text{bigger := x ; smaller := y} \\ \{bigger \geq smaller\} \end{array} \right) \\
 \wedge \\
 \left(\begin{array}{l} \{(x > 0 \wedge y > 0) \wedge \neg(x > y)\} \\ \text{bigger := y ; smaller := x} \\ \{bigger \geq smaller\} \end{array} \right)$$

wp Rule: Sequential Composition (1)



$$wp(S_1 ; S_2, R) = wp(S_1, wp(S_2, R))$$

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)

Recall:

$$\{Q\} S \{R\} \equiv Q \Rightarrow wp(S, R)$$

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 $\{ \text{True} \} \text{tmp} := x; x := y; y := \text{tmp} \{ x > y \}$ correct?
 If and only if $\text{True} \Rightarrow \text{wp}(\text{tmp} := x; x := y; y := \text{tmp}, x > y)$

$$\begin{aligned}
 & \text{wp}(\text{tmp} := x; \boxed{x := y; y := \text{tmp}}, x > y) \\
 = & \{ \text{wp rule for seq. comp.} \} \\
 & \text{wp}(\text{tmp} := x, \text{wp}(x := y; \boxed{y := \text{tmp}}, x > y)) \\
 = & \{ \text{wp rule for seq. comp.} \} \\
 & \text{wp}(\text{tmp} := x, \text{wp}(x := y, \text{wp}(y := \text{tmp}, x > \boxed{y}))) \\
 = & \{ \text{wp rule for assignment} \} \\
 & \text{wp}(\text{tmp} := x, \text{wp}(x := y, \boxed{x} > \text{tmp})) \\
 = & \{ \text{wp rule for assignment} \} \\
 & \text{wp}(\text{tmp} := x, y > \boxed{\text{tmp}}) \\
 = & \{ \text{wp rule for assignment} \} \\
 & y > x
 \end{aligned}$$

$\therefore \text{True} \Rightarrow y > x$ does not hold in general.

\therefore The above program is not correct.

- 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

<p style="text-align: center;">BS1</p> <pre> 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) </pre>	<p style="text-align: center;">BS2</p> <pre> 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 </pre>
<p style="text-align: center;">BS3</p> <pre> 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 </pre>	<p style="text-align: center;">BS4</p> <pre> 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 </pre>

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
  Sinit
until
  B
loop
  Sbody
end
{R}
  
```

```

{Q}
Sinit
while ( $\neg B$ ) {
  Sbody
}
{R}
  
```

- In case of C/Java, $\neg B$ denotes the **stay condition**.
- In case of Eiffel, B denotes the **exit condition**.

There is native, syntactic support for checking/proving the **total correctness** of loops.

Contracts for Loops: Syntax

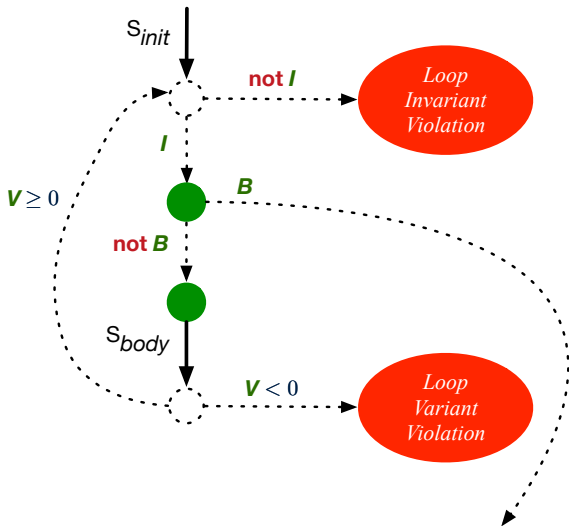
```
from
   $S_{init}$ 
invariant
  invariant_tag:  $I$  -- Boolean expression for partial correctness
until
   $B$ 
loop
   $S_{body}$ 
variant
  variant_tag:  $V$  -- Integer expression for termination
end
```


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 **non-negative** at the end of each iteration.
 - 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: Runtime Checks (1)



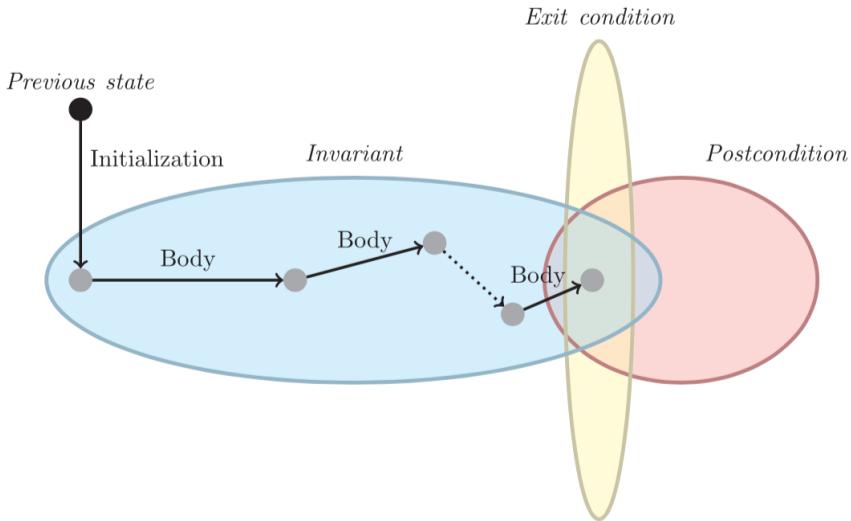
Contracts for Loops: Runtime Checks (2)

```
1 test
2   local
3     i: INTEGER
4   do
5     from
6       i := 1
7     invariant
8       1 <= i and i <= 6
9     until
10      i > 5
11    loop
12      io.put_string ("iteration " + i.out + "%N")
13      i := i + 1
14    variant
15      6 - i
16    end
17 end
```

L8: Change to $1 \leq i$ and $i \leq 5$ for a **Loop Invariant Violation**.

L15: Change to $5 - i$ for a **Loop Variant Violation**.

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: --  $\forall j \mid a.lower \leq j \leq i \bullet Result \geq a[j]$ 
      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: --  $\forall j \mid a.lower \leq j \leq a.upper \bullet Result \geq a[j]$ 
    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:** $\forall j \mid a.lower \leq j \leq i \bullet Result \geq a[j]$
- **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:

$$\forall j \mid a.lower \leq j \leq 3 \bullet 20 \geq a[j]$$

evaluates to **false** $\because 20 \not\geq a[3] = 40$

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: --  $\forall j \mid a.lower \leq j < i \bullet Result \geq a[j]$ 
      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: --  $\forall j \mid a.lower \leq j \leq a.upper \bullet Result \geq a[j]$ 
    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:** $\forall j \mid a.lower \leq j < i$ • $Result \geq a[j]$
- **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 4th iteration

$\because a.upper - i = 4 - 5$ evaluates to **non-zero**.

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: --  $\forall j \mid a.lower \leq j < i \bullet Result \geq a[j]$ 
      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: --  $\forall j \mid a.lower \leq j \leq a.upper \bullet Result \geq a[j]$ 
    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:** $a.upper - i + 1$
- **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: 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:  $\forall k \mid k \in \text{Result} \bullet \text{model.item}(k) \sim v$ 
      no_missing_keys:  $\forall k \mid k \in \text{model.domain} \bullet \text{model.item}(k) \sim v \Rightarrow k \in \text{Result}$ 
    end
end
```

Proving Correctness of Loops (1)

```

{Q}  from
      Sinit
      invariant
      I
      until
      B
      loop
      Sbody
      variant
      V
      end    {R}
  
```

- A loop is **partially correct** if:
 - Given precondition Q , the initialization step S_{init} establishes LI .
 - At the end of S_{body} , if not yet to exit, LI is maintained.
 - If ready to exit and LI maintained, postcondition R is established.
- A loop **terminates** if:
 - Given LI , and not yet to exit, S_{body} maintains LV as non-negative.
 - Given LI , and not yet to exit, S_{body} decrements LV .

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 .

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

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

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

- If ready to exit and LI maintained, postcondition R is established.

$$I \wedge B \Rightarrow R$$

- A loop **terminates** if:

- Given LI , and not yet to exit, S_{body} maintains LV V as non-negative.

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

- Given LI , 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:  $\forall j \mid a.lower \leq j < i \bullet Result \geq a[j]$ 
    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:  $\forall j \mid a.lower \leq j \leq a.upper \bullet Result \geq a[j]$ 
  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:

$$\begin{aligned}
 & (\forall j \mid a.lower \leq j < i \bullet Result \geq a[j]) \wedge i > a.upper \\
 & \Rightarrow \forall j \mid a.lower \leq j \leq a.upper \bullet Result \geq a[j]
 \end{aligned}$$

Proving Correctness of Loops: Exercise (1.3)

Prove that each of the following **Hoare Triples** is TRUE.

4. Loop Variant Stays Non-Negative Before Exit:

```

{ (∀j | a.lower ≤ j < i • Result ≥ a[j]) ∧ ¬(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:

```

{ (∀j | a.lower ≤ j < i • Result ≥ a[j]) ∧ ¬(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)

$$\{Q\} S \{R\} \Rightarrow \{Q \wedge P\} S \{R\}$$

In order to prove $\{Q \wedge P\} S \{R\}$, it is sufficient to prove a version with a **weaker** precondition: $\{Q\} S \{R\}$.

Proof:

- Assume: $\{Q\} S \{R\}$

It's equivalent to assuming: $\boxed{Q} \Rightarrow wp(S, R)$

(A1)

- To prove: $\{Q \wedge P\} S \{R\}$

- It's equivalent to proving: $Q \wedge P \Rightarrow wp(S, R)$
- Assume: $Q \wedge P$, which implies \boxed{Q}
- According to **(A1)**, we have $wp(S, R)$. ■

Proof Tips (2)

When calculating $wp(S, R)$, if either program S or postcondition R involves array indexing, then R should be augmented accordingly.

e.g., Before calculating $wp(S, a[i] > 0)$, augment it as

$$wp(S, a.lower \leq i \leq a.upper \wedge a[i] > 0)$$

e.g., Before calculating $wp(x := a[i], R)$, augment it as

$$wp(x := a[i], a.lower \leq i \leq a.upper \wedge R)$$

Beyond this lecture



Exercise on proving the *total correctness* of a program:

https://www.eecs.yorku.ca/~jackie/teaching/lectures/2020/F/EECS3311/exercises/EECS3311_F20_Exercise_WP_sol.pdf

Index (1)

Learning Objectives

Assertions: Weak vs. Strong

Assertions: Preconditions

Assertions: Postconditions

Motivating Examples (1)

Motivating Examples (2)

Software Correctness

Hoare Logic

Hoare Logic and Software Correctness

Proof of Hoare Triple using wp

Denoting New and Old Values

Index (2)

***wp* Rule: Assignments (1)**

***wp* Rule: Assignments (2)**

***wp* Rule: Assignments (3) Exercise**

***wp* Rule: Assignments (4) Exercise**

***wp* Rule: Assignments (5) Revisit**

***wp* Rule: Alternations (1)**

***wp* Rule: Alternations (2)**

***wp* Rule: Alternations (3) Exercise**

***wp* Rule: Sequential Composition (1)**

***wp* Rule: Sequential Composition (2)**

***wp* Rule: Sequential Composition (3) Exercise**

Index (3)

Loops

Loops: Binary Search

Correctness of Loops

Contracts for Loops: Syntax

Contracts for Loops

Contracts for Loops: Runtime Checks (1)

Contracts for Loops: Runtime Checks (2)

Contracts for Loops: Visualization

Contracts for Loops: Example 1.1

Contracts for Loops: Example 1.2

Contracts for Loops: Example 2.1

Index (4)

Contracts for Loops: Example 2.2

Contracts for Loops: Example 3.1

Contracts for Loops: Example 3.2

Contracts for Loops: Exercise

Proving Correctness of Loops (1)

Proving Correctness of Loops (2)

Proving Correctness of Loops: Exercise (1.1)

Proving Correctness of Loops: Exercise (1.2)

Proving Correctness of Loops: Exercise (1.3)

Proof Tips (1)

Proof Tips (2)

Index (5)



Beyond this lecture