

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



EECS3311 A: Software Design
Fall 2019

CHEN-WEI WANG

Why a Design Diagram?

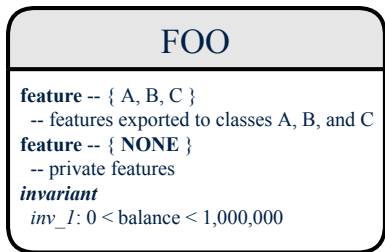
- **SOURCE CODE** is **not** an appropriate form for communication.
- Use a **DESIGN DIAGRAM** showing **selective** sets of important:
 - clusters (i.e., packages)
 - [deferred vs. effective]
 - [generic vs. non-generic]
 - architectural relations
 - [client-supplier vs. inheritance]
 - features (queries 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 ...**, **<=**).

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]`

`HASH_TABLE[STRING, INTEGER]`

`HASH_TABLE[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

`DATABASE[STRING]`

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

`DATABASE[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
feature -- Queries
  search (g: G): BOOLEAN
    -- Perform a linear search on the database.
  deferred 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]
feature -- Queries
  search (g: G): BOOLEAN
    -- Perform a binary search on the database.
  deferred 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'.

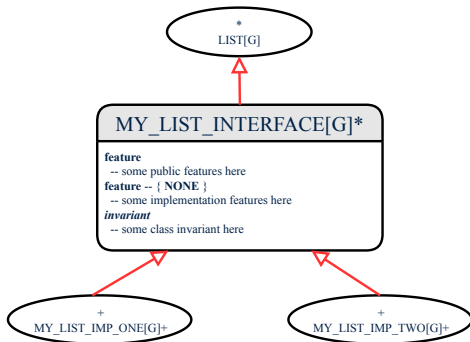
feature -- Queries
count+: INTEGER
  -- Number of items stored in database

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)

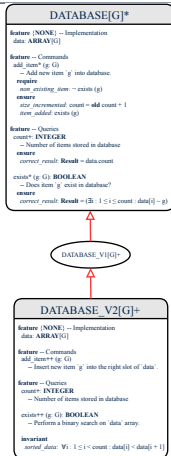
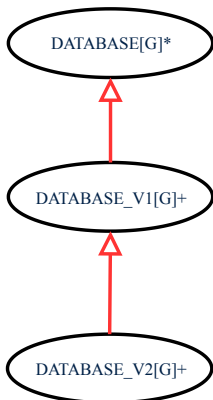
- An **inheritance hierarchy** is formed using **red arrows**.
 - Arrow's **origin** indicates the **child/descendant** class.
 - Arrow's **destination** indicates the **parent/ancestor** class.
- You may choose to present each class in an inheritance hierarchy in either the detailed form or the compact 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 label there should be a *label* indicating the **supplier name** (i.e., variable name).
 - In the case where supplier is an attribute, 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

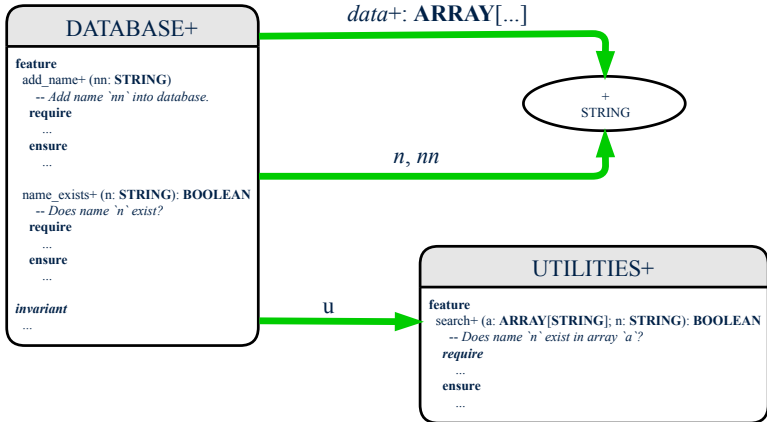
  name_exists (n: STRING): BOOLEAN
    -- Does name 'n' exist in database?
    require ...
    local
      u: UTILITIES
    do ... ensure ... end
invariant
  ...
end
```

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

- Attribute `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)

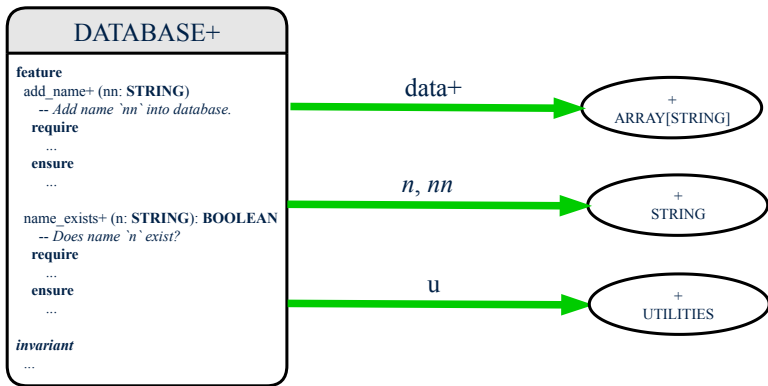
If `STRING` is to be emphasized, label is `data: ARRAY[...]`, where `...` denotes the supplier class `STRING` being pointed to.



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
```

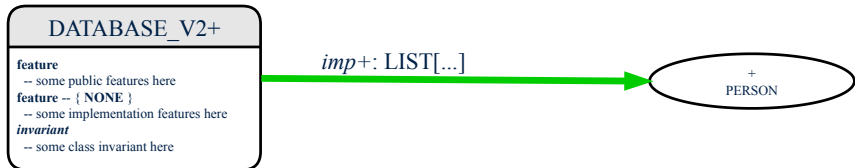
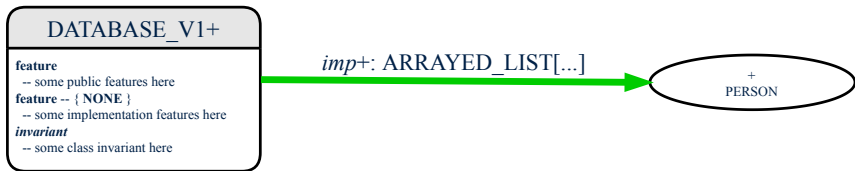
Question: Which design is better?

[DESIGN TWO]

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

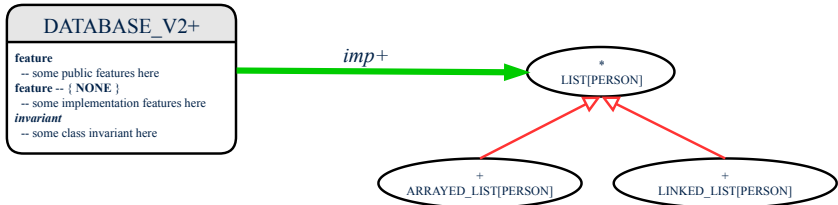
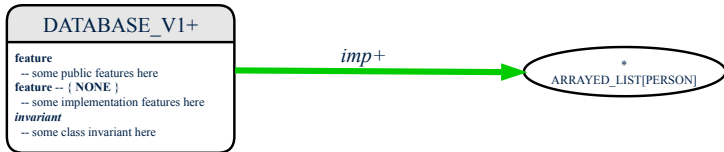
Class Relations: Client-Supplier (3.2.1)

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



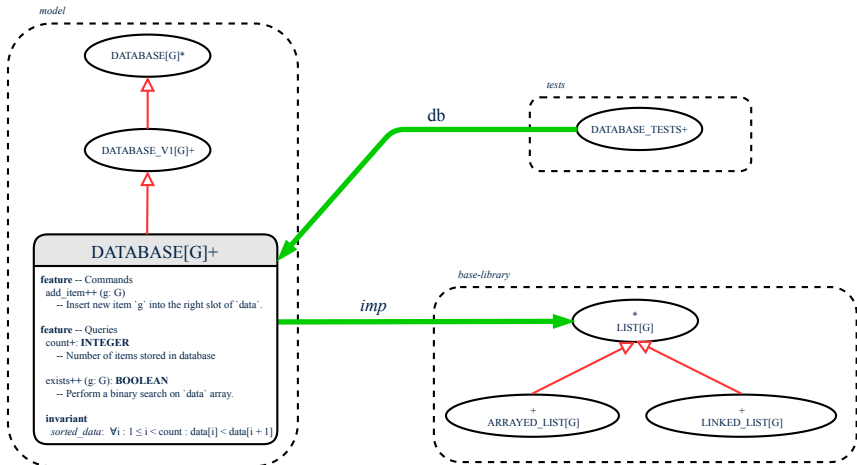
Class Relations: Client-Supplier (3.2.2)

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.



Index (1)

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)

Features: Deferred, Effective, Redefined (2)

Features: Deferred, Effective, Redefined (3)

Classes: Deferred vs. Effective (2.1)

Classes: Deferred vs. Effective (2.2)

Index (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)

Class Relations: Client-Supplier (3.2.1)

Class Relations: Client-Supplier (3.2.2)

Clusters: Grouping Classes