Abstractions via Mathematical Models

YORK UNIVERSITÉ UNIVERSITY

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
count	imp.count		
top	imp[imp.count]	imp.first	imp.last
push(g)	imp.force(g, imp.count + 1)	imp.put_front(g)	imp.extend(g)
pop	imp.list.remove_tail (1)	list.start list.remove	imp.finish imp.remove

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

Motivating Problem: LIFO Stack (2.1)


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


Motivating Problem: LIFO Stack (2.2)


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


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
```


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
```

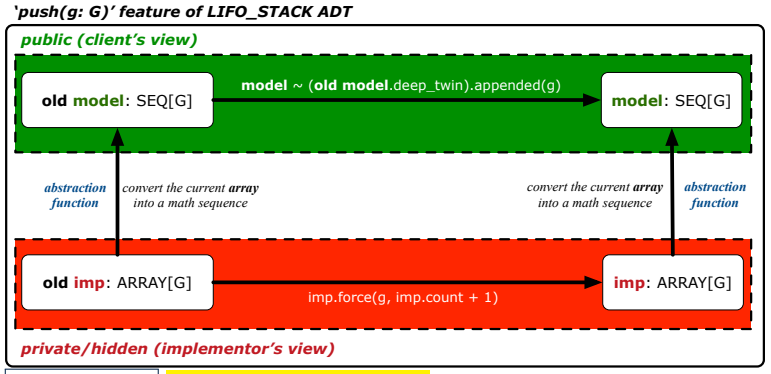
- Side-effect-free ***queries***: Write Complete Contracts

```
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)



- **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)
```

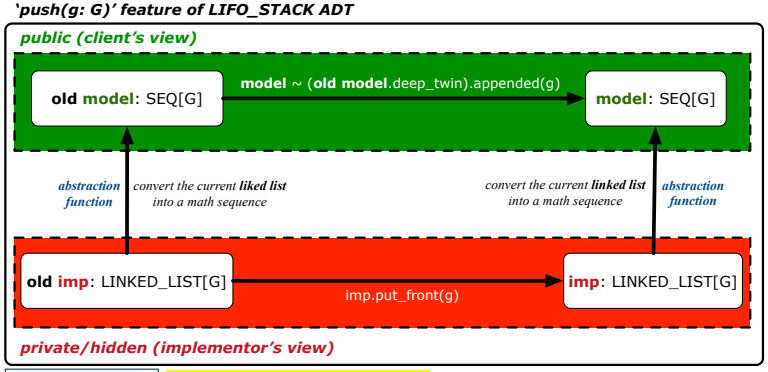
Implementing an Abstraction Function (2)

```
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
```


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
```

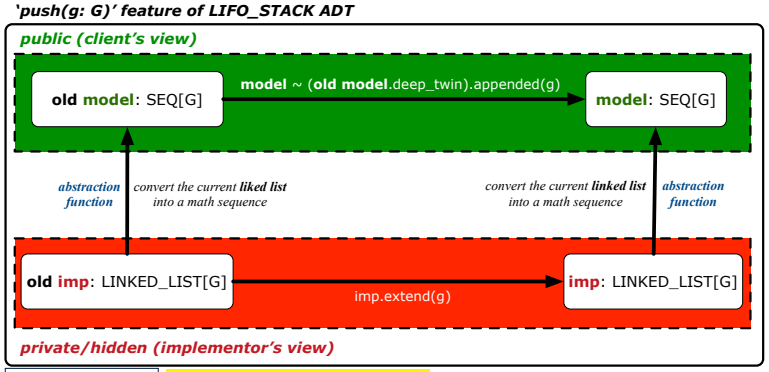

Abstracting ADTs as Math Models (2)



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

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


Abstracting ADTs as Math Models (3)



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

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

Solution: Abstracting ADTs as Math Models

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

Beyond this lecture . . .

- Familiarize yourself with the features of class `SEQ`.

Index (1)

Index (2)