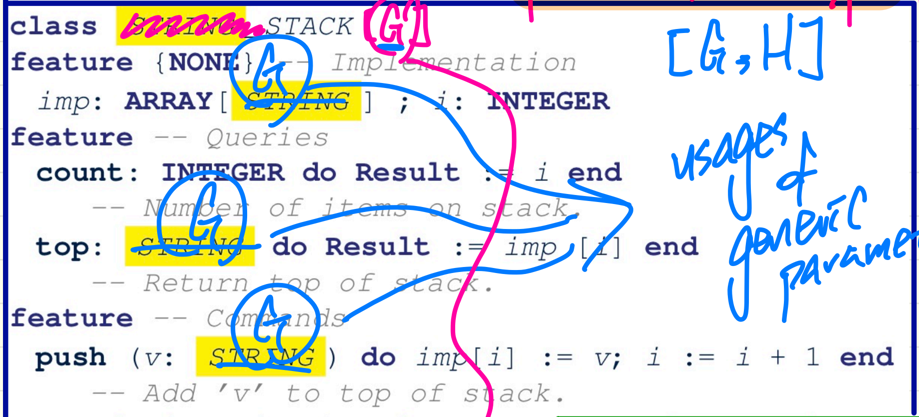
Lecture 3

Part 1

Writing & Using a Generic Class

Stack of Strings vs. Stack of Accounts



```
class ~~STRING_~~STACK [G]
feature {NONE} -- Implementation
  imp: ARRAY [ ~~STRING~~ G ] ; i: INTEGER
feature -- Queries
  count: INTEGER do Result := i end
    -- Number of items on stack.
  top: ~~STRING~~ G do Result := imp [i] end
    -- Return top of stack.
feature -- Commands
  push (v: ~~STRING~~ 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
```

parameter for the type of stack items

[G, H]

usages of generic parameter

```
add (i, j : INTEGER)
do
  R := i + j
end
```

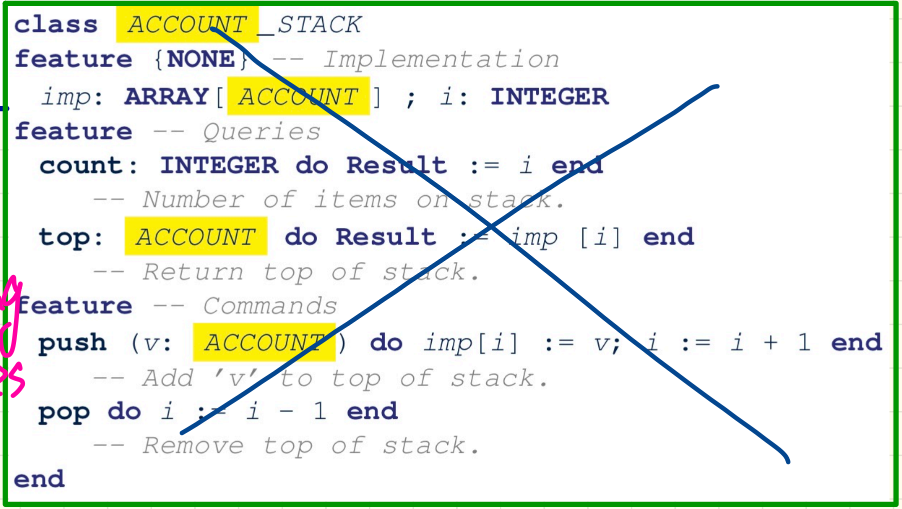
param

type

use of param

supplier of a generic class

should not clash with existing classes



```
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
```

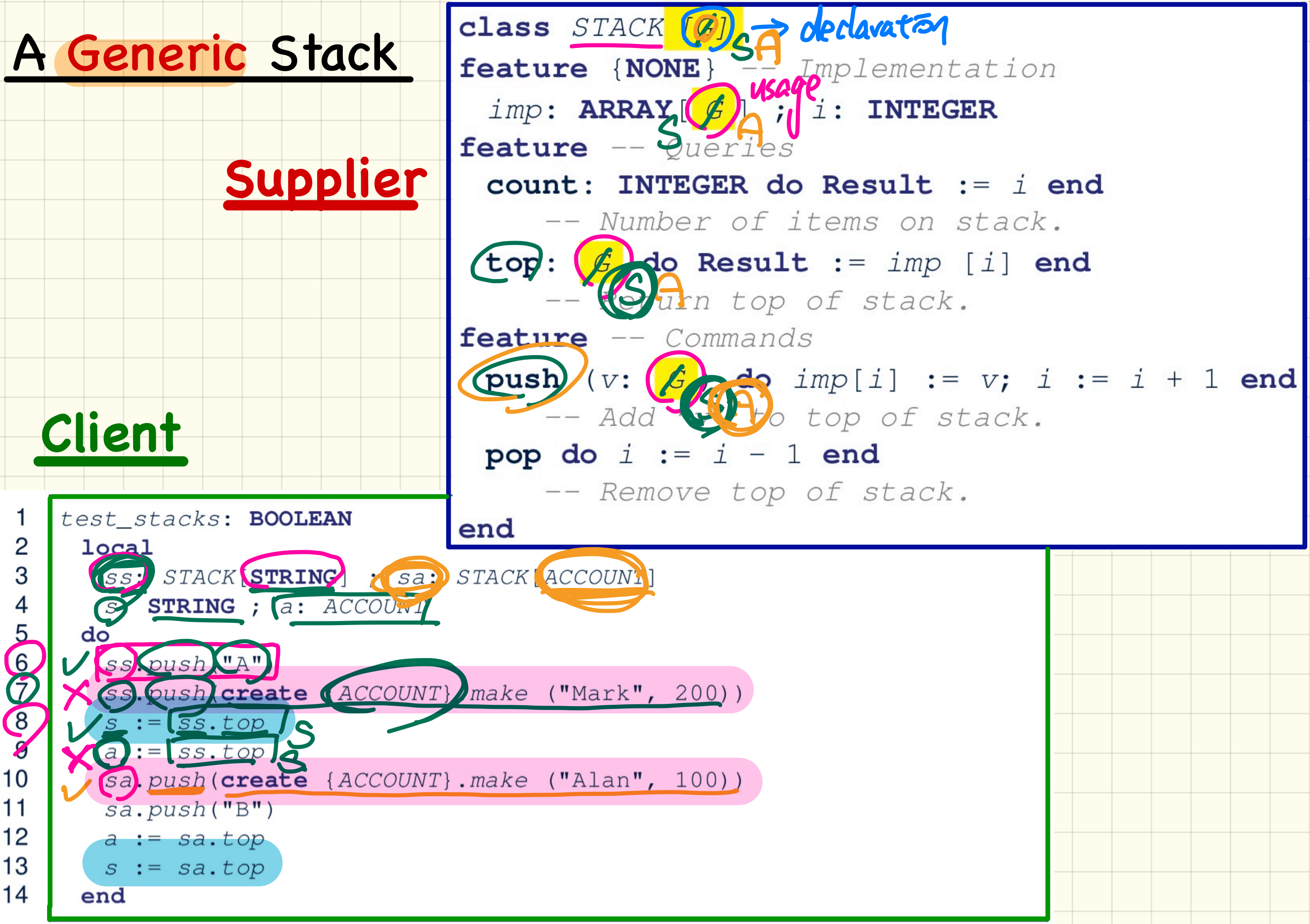
invalid parameter name for type

<u>class</u> STACK [~~STRING~~]

A Generic Stack

Supplier

Client



```
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
```

→ declaration

usage

S A

```
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
```

S

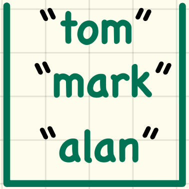
S

Lecture 3

Part 2

Abstractions via Mathematical Models

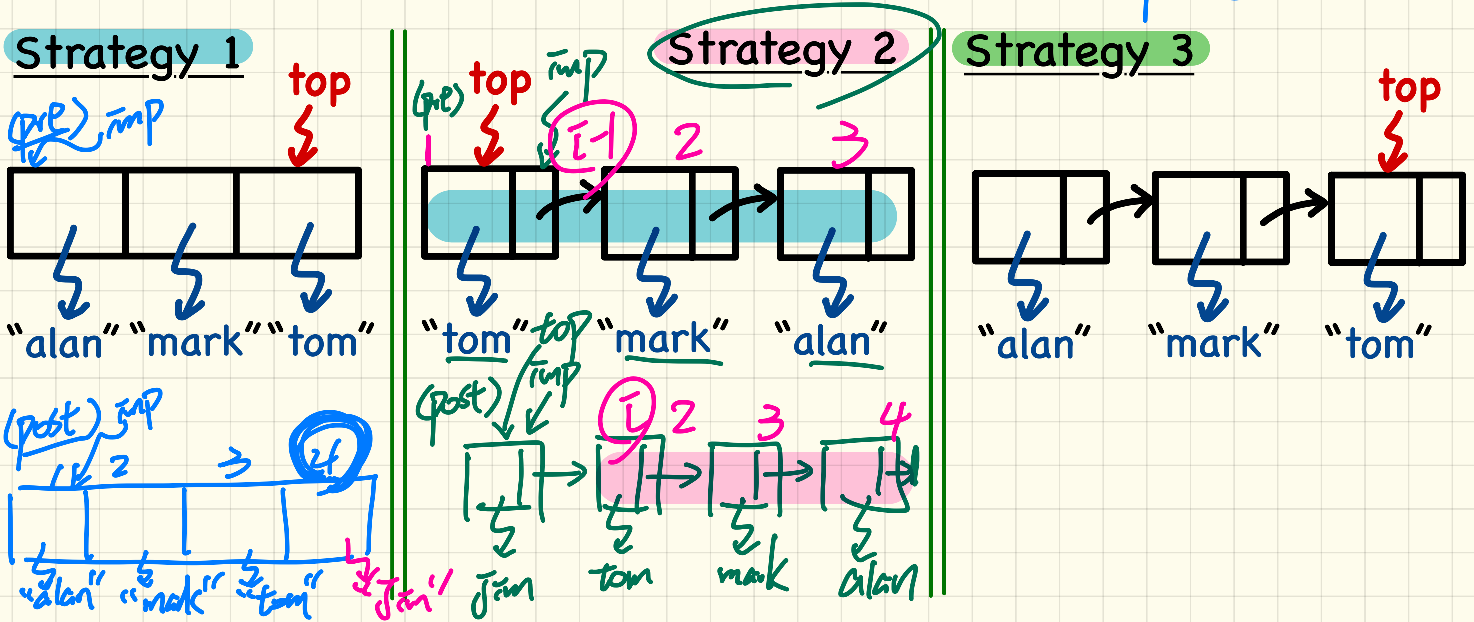
Implementing a LIFO Stack



S.push("Jim")

① $\forall i \mid 1 \le i \le (\text{imp.count} - 1) \bullet \text{old imp.deep\_twin}[i] \sim \text{imp}[i]$

② $\forall i \mid 2 \le i \le \text{imp.count} \bullet \text{old imp.deep\_twin}[i-1] \sim \text{imp}[i]$



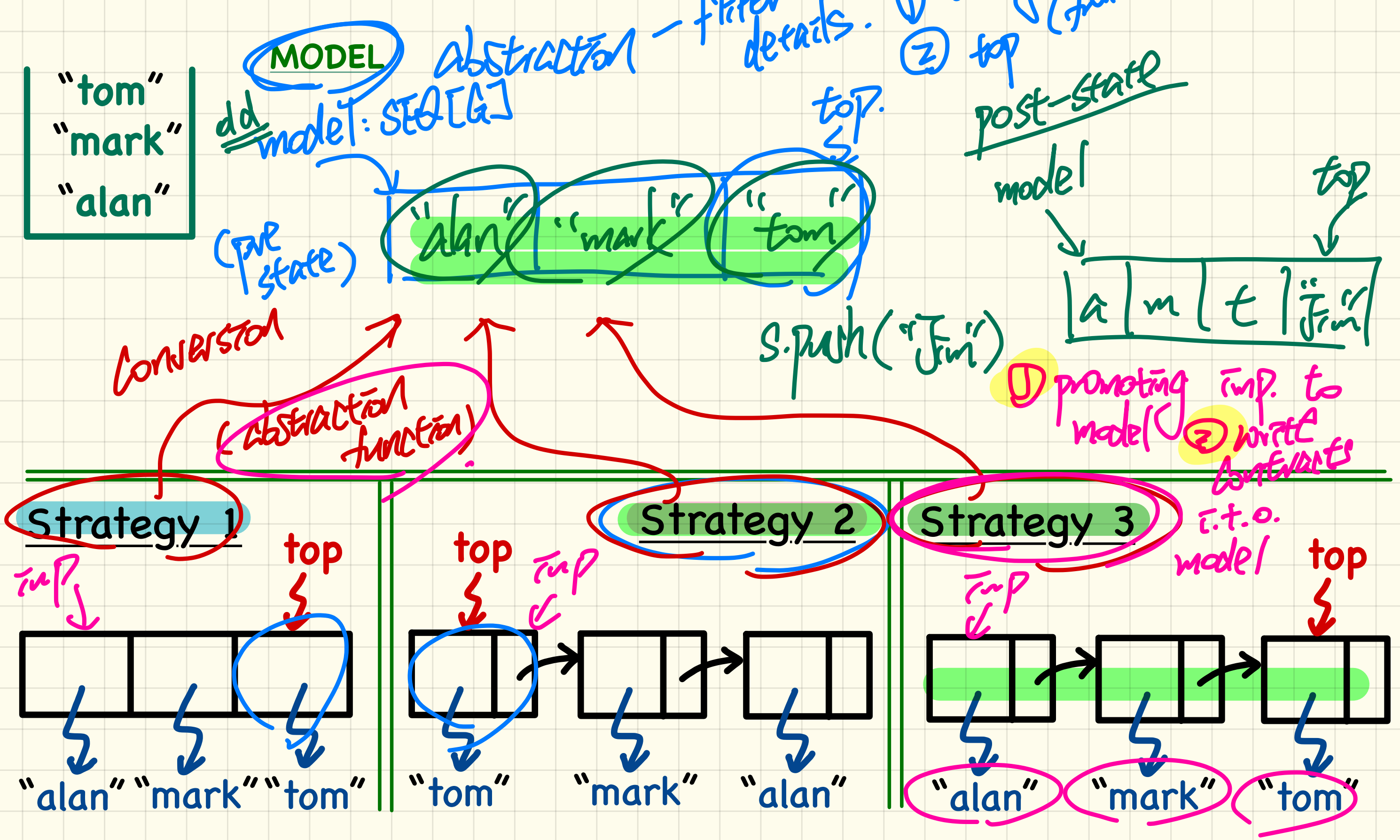
Developing a LIFO Stack

```
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
```

```
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
```

```
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
```

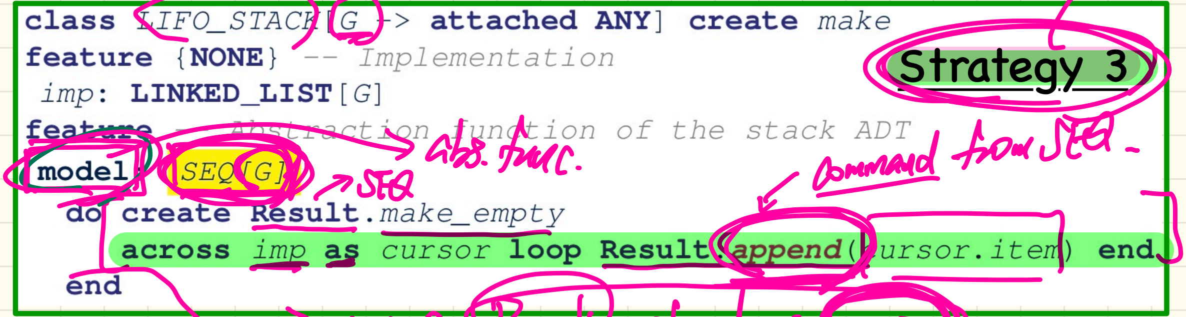
Abstracting a **LIFO** Stack



Using MATHMODELS Library

Implementing an Abstraction Function

strategy 2.



Strategy 3

```
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
```

abs. func.

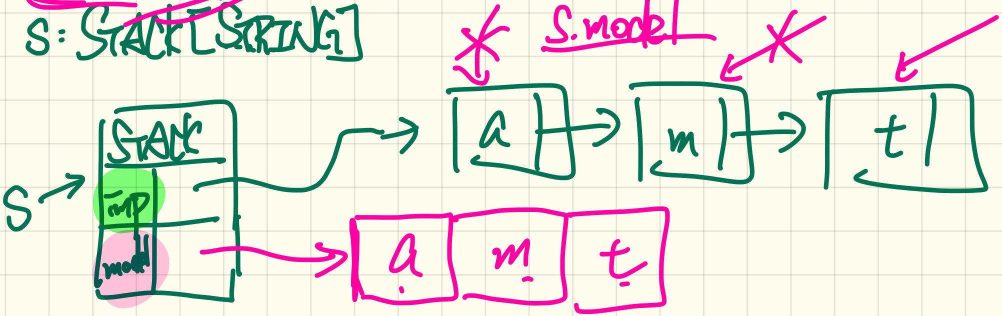
→ SEQ

command from SEQ.

ensure Result related to imp

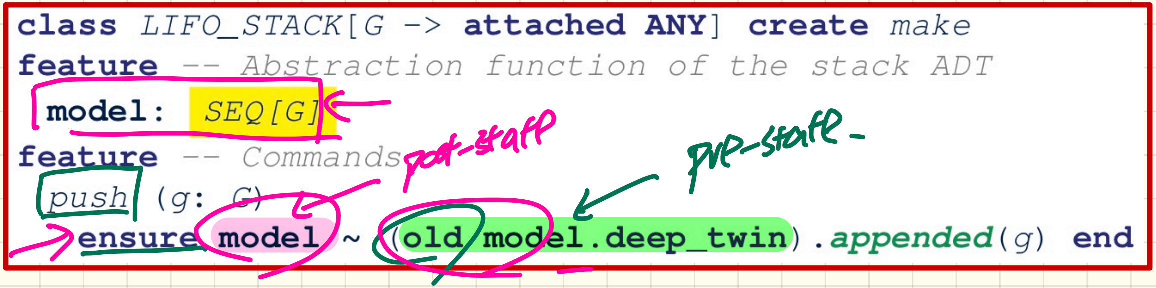
Exercise 1: Write postcondition of **model**.

Exercise 2: What if **Strategy 2** was adopted? Change what?



Using MATHMODELS Library

Writing Contracts using the Abstraction Function



```
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
```

post-state

pre-state -

Question: Can clients tell which strategy is being adopted?

No. ⇒ Information Hiding!

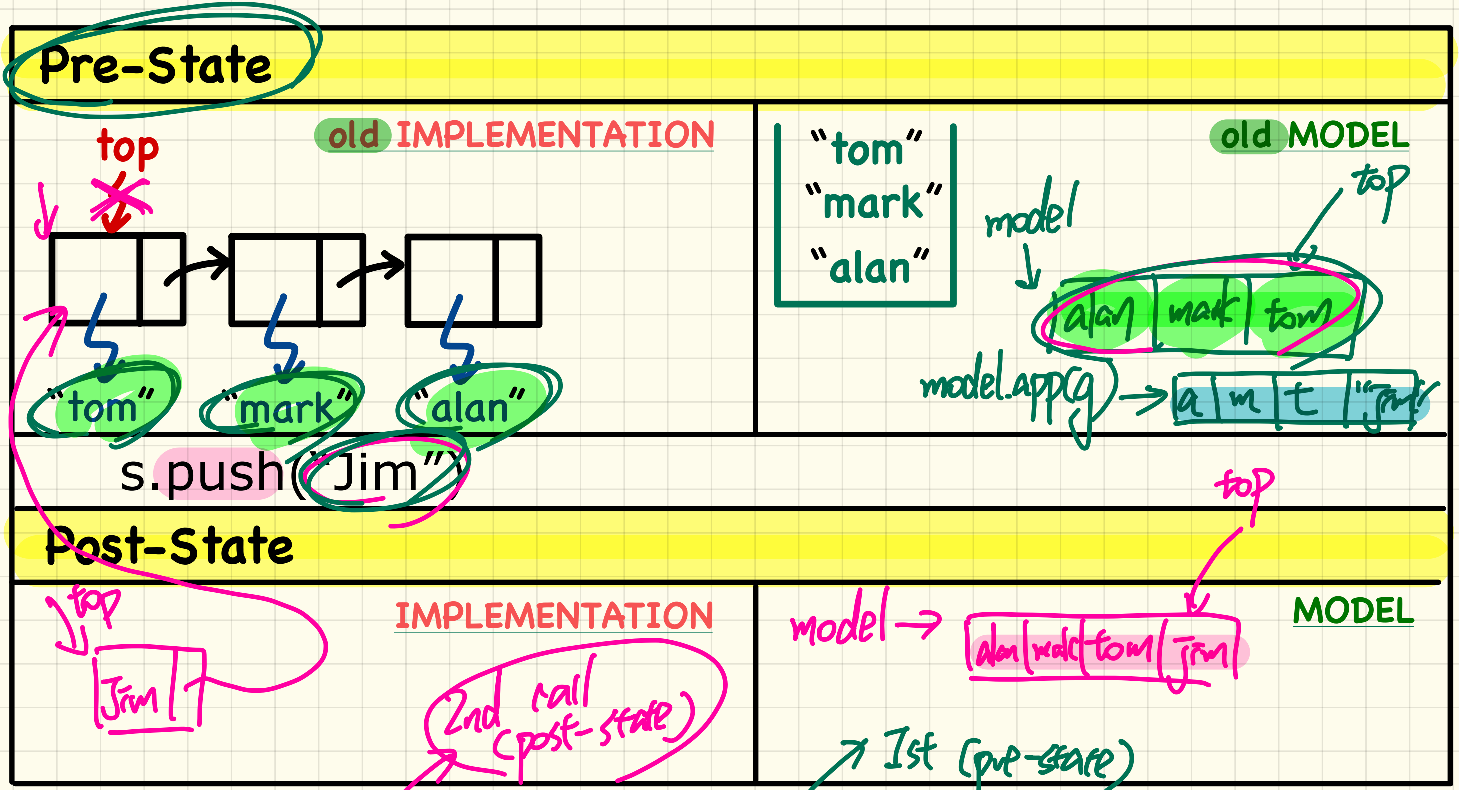
Exercise: What if strategy was changed? Change what?

↳ change the abstraction function

(i.e. imp. and contract of model)

Checking MATHMODELS Contracts at Runtime

Strategy 2



Pre-State

old IMPLEMENTATION

old MODEL

"tom"
"mark"
"alan"

s.push("Jim")

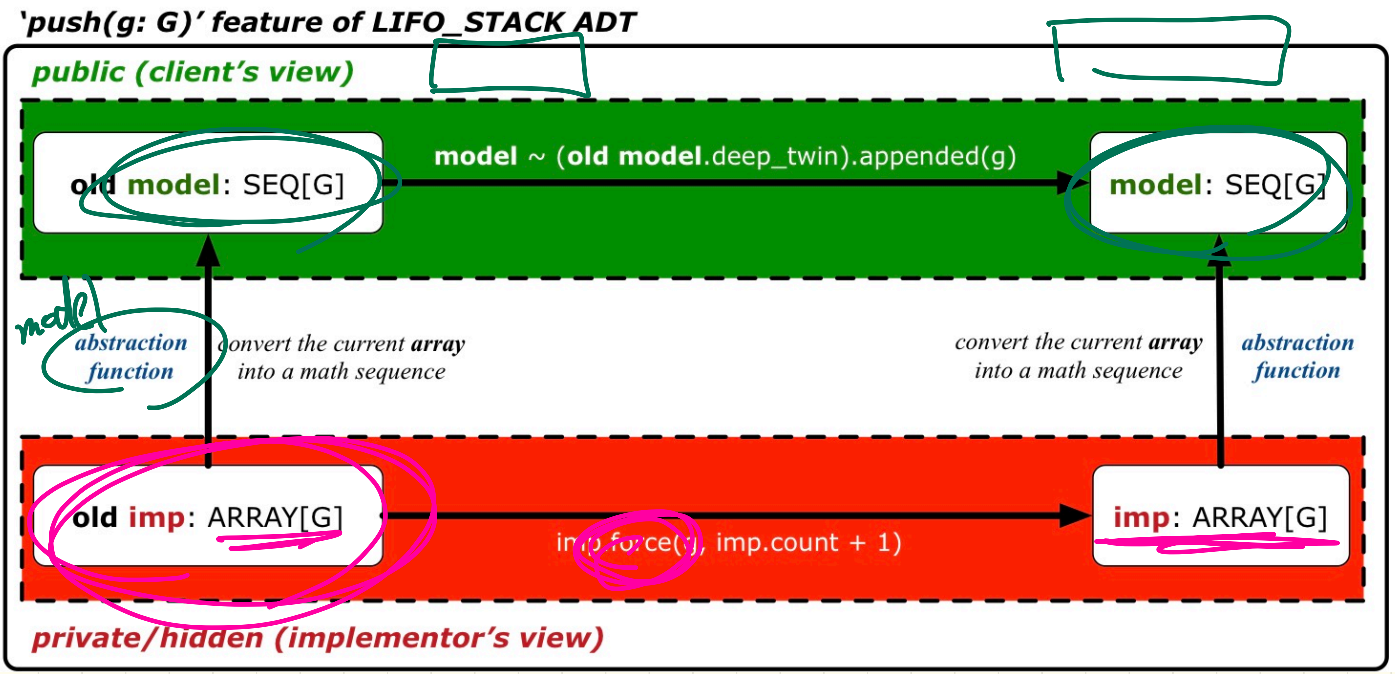
Post-State

IMPLEMENTATION

MODEL

```
push (g: G)
  ensure model ~ (old model.deep_twin).appended (g) end
```

Strategy 1: Mathematical Abstraction



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

public (client's view)

old model: SEQ[G]

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

model: SEQ[G]

model

abstraction function

*convert the current **array** into a math sequence*

*convert the current **array** into a math sequence*

abstraction function

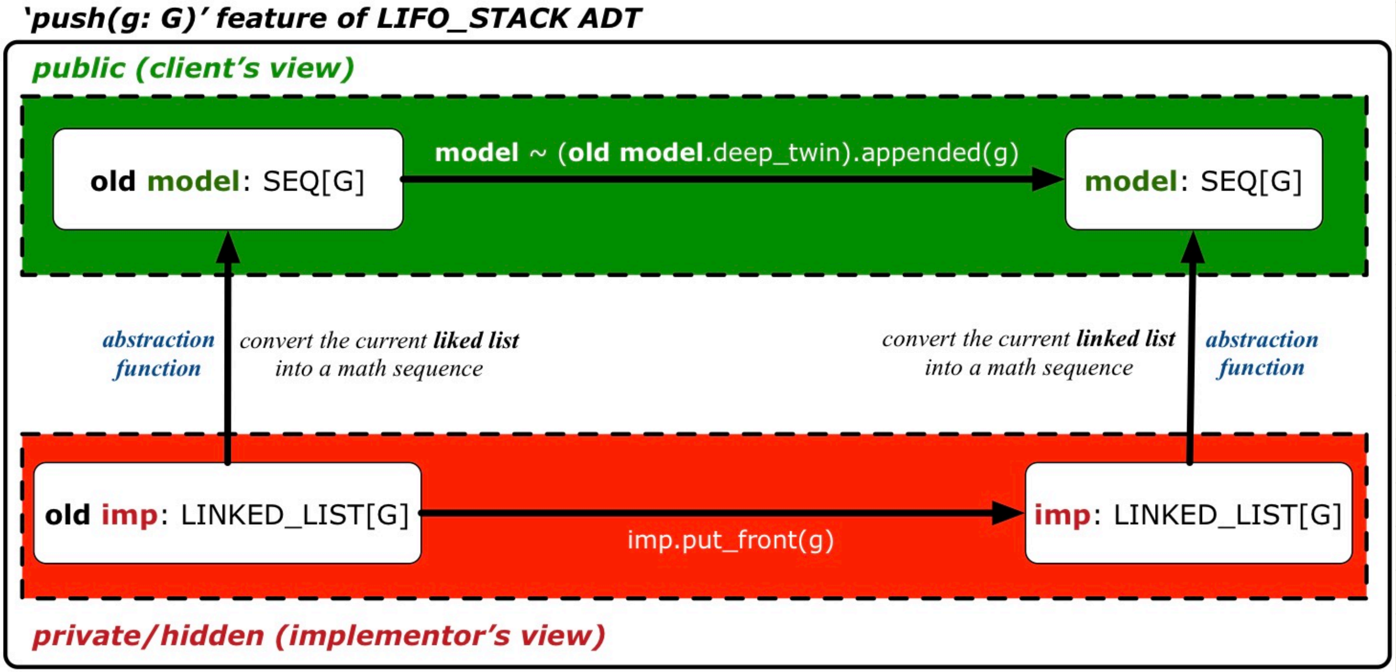
old imp: ARRAY[G]

imp.force(g, imp.count + 1)

imp: ARRAY[G]

private/hidden (implementor's view)

Strategy 2: Mathematical Abstraction



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

public (client's view)

old model: SEQ[G]

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

model: SEQ[G]

abstraction function

*convert the current **liked list** into a math sequence*

*convert the current **linked list** into a math sequence*

abstraction function

old imp: LINKED_LIST[G]

imp.put_front(g)

imp: LINKED_LIST[G]

private/hidden (implementor's view)

Use of MATHMODELS:

Single-Choice Principle

```
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
```

only place spec. rel between Result & imp.

change: s1 -> s2

single place to modify.

```
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
```

cf

imp

post

```
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
```