SE 2AA4, CS 2ME3 (Introduction to Software Development)

Winter 2018

11 Generic MIS (Ghezzi Ch. 4)

Dr. Spencer Smith

Faculty of Engineering, McMaster University

January 28, 2018

11 Generic MIS (Ghezzi Ch. 4)

- Administrative details
- `ssh` with X11 forwarding
- Homework exercise on specification
- Uses for abstract objects
- Implementing objects with other objects as state variables
- Generic modules
- Generic stack abstract object
- Properties exhibited by a stack module
- Generic queue ADT
- Access routine idioms
  - Set idioms
  - Sequence idioms
  - Tuple idioms
- Exceptions
- Quality Criteria

Administrative Details

- Assignment 1
  - Partner files now in your repo
  - Part 2: January 31, 2018
  - If you know how, tag your assignment submissions
- Questions?

X Windows

- If you want to use a gui on mills, you can use X11 forwarding
- On a Mac or with Linux, you use `ssh -X` or `ssh -Y`
- `ssh -X mills.cas.mcmaster.ca` (or `-Y`)
- On a Mac you need XQuartz installed
- If you know the details for Windows, please post to Avenue

Homework Answer: Access Routine Semantics

totalArea():

- output

$$out := ? \quad (i : \mathbb{N} | i \in [0..|s| - 1] : s[i].area())$$

averageRadius():

- output

$$out :=$$

Homework Answer: Access Routine Semantics

totalArea():

- output

$$out := \quad + (i : \mathbb{N} | i \in [0..|s| - 1] : s[i].area())$$

averageRadius():

- output

$$out :=$$

Homework Answer: Access Routine Semantics

totalArea():

- output

$$out := \quad +(i : \mathbb{N} | i \in [0..|s| - 1] : s[i].area())$$

or?

averageRadius():

- output

$$out :=$$

Homework Answer: Access Routine Semantics

totalArea():

- output

$$out := \quad +(i : \mathbb{N} | i \in [0..|s| - 1] : s[i].area())$$

or?

$$out := \quad +(c : CircleT | c \in s : c.area())$$

averageRadius():

- output

$$out := ?$$

Homework Answer: Access Routine Semantics

totalArea():

- output

$$out := +(i : \mathbb{N} | i \in [0..|s| - 1] : s[i].area())$$

or?

$$out := +(c : CircleT | c \in s : c.area())$$

averageRadius():

- output

$$out := +(c : CircleT | c \in s : c.radius())/|s|$$

Uses for Abstract Objects

- Creating a single abstract object corresponds to the singleton design pattern
- Provides "global" variables
- Uses
  - Shared resource
  - Hoffman and Strooper example (assembler)
  - Look up table for global state (read only)
  - Logger (write only)
- Problematic for testing if high coupling and frequent state changes, since many test cases will depend on the state of the abstract object

Objects with Other Objects as State Variables

Potential prob with the naive implement of deque of CircleT?

```
@staticmethod
def pushBack(c):
  Deq.s = Deq.s + [c]
```

Consider

```
p1 = PointT(1.5, 2)
c1 = CircleT(p1, 5)
Deq.init()
Deq.pushBack(c1)
c1.r = 6 #circumventing information hiding
print(c1.rad())
print(Deq.back().rad())
```

Solutions to Potential Problem

- Interface prevents potential abuse
  - Provide no mutators in the interface, just a constructor and selectors
  - Assume information hiding will be respected
- A more robust implementation
  - The state variable stores copies of objects, not references
  - Use copy library in Python

```
import copy
...
  @staticmethod
  def pushBack(c):
    cnew = copy.deepcopy(c)
    Deq.s = Deq.s + [cnew]
```

New Problem for Robust Implementation

```
p1 = PointT(1.5, 2)
c1 = CircleT(p1, 5)
Deq.init()
Deq.pushBack(c1)
print(c1 == Deq.back())
```

Why is the behaviour not what we would naively expect?

What can we do about it?

New Problem for Robust Implementation

```
p1 = PointT(1.5, 2)
c1 = CircleT(p1, 5)
Deq.init()
Deq.pushBack(c1)
print(c1 == Deq.back())
```

Why is the behaviour not what we would naively expect?

What can we do about it?

Redefine Equality for CircleT

```
class CircleT:

  def __init__(self, cin, rin):
    self.c = cin
    self.r = rin

  def __eq__(self, cin):
    return self.c == cin.cen() and
       self.r == cin.rad()
```

What other classes will need __eq__ redefined?

Examples of A2 from 2017

- Solution assuming information hiding in repo of previous assignment solutions (here)
- Robust solution in the same folder as these lecture slides (here)

Generic Modules

- What if we have a sequence of integers, instead of a sequence of point masses?
- What if we want a stack of integers, or characters, or pointT, or pointMassT?
- Do we need a new specification for each new abstract object?
- No, we can have a single abstract specification implementing a family of abstract objects that are distinguished only by a few variabilities
- Rather than duplicate nearly identical modules, we parameterize one generic module with respect to type(s)
- Advantages
  - Eliminate chance of inconsistencies between modules
  - Localize effects of possible modifications
  - Reuse

Generic Stack Module Syntax

Generic Module

Stack(T)

Exported Constants

MAX_SIZE = 100

Exported Access Programs

Routine name	In	Out	Exceptions
...	...	...	...

Stack Module Syntax

Exported Access Programs

Routine name	In	Out	Exceptions
s_init			
s_push	T		FULL
s_pop			EMPTY
s_top		T	EMPTY
s_depth		integer	

Semantics

State Variables

State Invariant

Assumptions

Semantics

State Variables

s: sequence of T

State Invariant

Assumptions

Semantics

State Variables

s: sequence of T

State Invariant

$|s| \leq \text{MAX\_SIZE}$

Assumptions

Semantics

State Variables

s: sequence of T

State Invariant

$|s| \leq \text{MAX\_SIZE}$

Assumptions

s_init() is called before any other access routine

Access Routine Semantics

s_init():

- transition:
- exception:

s_push(x):

- transition:
- exception:

s_pop():

- transition:
- exception:

Access Routine Semantics

s_init():

- transition: $s := <>$
- exception:

s_push(x):

- transition:
- exception:

s_pop():

- transition:
- exception:

Access Routine Semantics

s_init():

- transition: $s := <>$
- exception: none

s_push(x):

- transition:
- exception:

s_pop():

- transition:
- exception:

Access Routine Semantics

s_init():

- transition: $s := <>$
- exception: none

s_push(x):

- transition: $s := s || < x >$
- exception:

s_pop():

- transition:
- exception:

Access Routine Semantics

s_init():

- transition: $s := <>$
- exception: none

s_push(x):

- transition: $s := s || < x >$
- exception: $exc := (|s| = \text{MAX\_SIZE} \Rightarrow \text{FULL})$

s_pop():

- transition:
- exception:

Access Routine Semantics

s_init():

- transition: $s := <>$
- exception: none

s_push(x):

- transition: $s := s || < x >$
- exception: $exc := (|s| = \text{MAX\_SIZE} \Rightarrow \text{FULL})$

s_pop():

- transition: $s := s[0..|s| - 2]$
- exception:

Access Routine Semantics

s_init():

- transition: $s := <>$
- exception: none

s_push(x):

- transition: $s := s || < x >$
- exception: $exc := (|s| = \text{MAX\_SIZE} \Rightarrow \text{FULL})$

s_pop():

- transition: $s := s[0..|s| - 2]$
- exception: $exc := (|s| = 0 \Rightarrow \text{EMPTY})$

Access Routine Semantics Continued

s_top():

- output:
- exception:

s_depth():

- output:
- exception:

Access Routine Semantics Continued

s_top():

- output: $out := s[|s| - 1]$
- exception:

s_depth():

- output:
- exception:

Access Routine Semantics Continued

s_top():

- output: $out := s[|s| - 1]$
- exception: $exc := (|s| = 0 \Rightarrow \text{EMPTY})$

s_depth():

- output:
- exception:

Access Routine Semantics Continued

s_top():

- output: $out := s[|s| - 1]$
- exception: $exc := (|s| = 0 \Rightarrow \text{EMPTY})$

s_depth():

- output: $out := |s|$
- exception:

Access Routine Semantics Continued

s_top():

- output: $out := s[|s| - 1]$
- exception: $exc := (|s| = 0 \Rightarrow \text{EMPTY})$

s_depth():

- output: $out := |s|$
- exception: none

Stack Module Properties

$\{true\}$
    s_init()
$\{|s'| = 0\}$

$\{|s| < \text{MAX\_SIZE}\}$
    s_push(x)
$\{|s'| = |s| + 1 \wedge s'[|s'| - 1] = x \wedge s'[0..|s| - 1] = s[0..|s| - 1]\}$

$\{|s| < \text{MAX\_SIZE}\}$
    s_push(x)
    s_pop()
$\{s' = s\}$

Generic Queue Module ADT Syntax

Generic Template Module

QueueADT(T)

Exported Types

QueueT = ?

Exported Constants

MAX_SIZE = 100

Exported Access Programs

Routine name	In	Out	Exceptions
...	...	...	...

QueueADT Module Syntax

Exported Access Programs

Routine name	In	Out	Exceptions
new QueueT		QueueT	
add	T		queue_full
pop			queue_empty
front		T	queue_empty
isempty		boolean	

Semantics

State Variables

State Invariant

Assumptions

Semantics

State Variables

s: sequence of T

State Invariant

Assumptions

Semantics

State Variables

s: sequence of T

State Invariant

$|s| \leq \text{MAX\_SIZE}$

Assumptions

Access Routine Semantics

new QueueT():

- transition:
- output:
- exception:

add(x):

- transition:
- exception:

pop():

- transition:
- exception:

Access Routine Semantics

new QueueT():

- transition: $s := <>$
- output: $out := self$
- exception: none

add(x):

- transition:
- exception:

pop():

- transition:
- exception:

Access Routine Semantics

new QueueT():

- transition: $s := <>$
- output: $out := self$
- exception: none

add(x):

- transition: $s := s || < x >$
- exception: $exc := (|s| = \text{MAX\_SIZE} \Rightarrow \text{queue\_full})$

pop():

- transition:
- exception:

Access Routine Semantics

new QueueT():

- transition: $s := <>$
- output: $out := self$
- exception: none

add(x):

- transition: $s := s || < x >$
- exception: $exc := (|s| = \text{MAX\_SIZE} \Rightarrow \text{queue\_full})$

pop():

- transition: $s := s[1..|s| - 1]$
- exception: $exc := (|s| = 0 \Rightarrow \text{queue\_empty})$

Access Routine Semantics Continued

front():

- output:
- exception:

isempty():

- output:
- exception:

Access Routine Semantics Continued

front():

- output: $out := s[0]$
- exception: $exc := (|s| = 0 \Rightarrow \text{queue\_empty})$

isempty():

- output:
- exception:

Access Routine Semantics Continued

front():

- output: $out := s[0]$
- exception: $exc := (|s| = 0 \Rightarrow \text{queue\_empty})$

isempty():

- output: $out := |s| = 0$
- exception: none

Queue Module Properties

$$\{true\}$$
$$\quad \text{q\_init}()$$
$$\{|s'| = 0 \wedge is\_init\}$$

$$\{|s| < \text{MAX\_SIZE}\}$$
$$\quad \text{add}(x)$$
$$\{|s'| = |s| + 1 \wedge s'[0] = x \wedge s'[1..|s'| - 1] = s[0..|s| - 1]\}$$

Set Idiom

Routine name	In	Out	Exceptions
set_add	T		Member, Full
set_del	T		NotMember
set_member	T	boolean	
set_size		integer	

Sequence Idiom

Routine name	In	Out	Exceptions
seq_init			
seq_add	integer, T		PosOutOfRange, Full
seq_del	integer		PosOutOfRange
seq_setval	integer, T		PosOutOfRange
seq_getval	integer	T	PosOutOfRange
seq_size		integer	
seq_start			
seq_next		T	AtEnd
seq_end		boolean	
seq_append	T		Full

Tuple Idiom Version 1

Routine name	**In**	**Out**	**Exceptions**
tp_init			
tp_set_f_1	T_1		
tp_get_f_1		T_1	
...	...	...	...
tp_set_f_N	T_N		
tp_get_f_N		T_N	

Tuple Idiom Version 2

Routine name	**In**	**Out**	**Exceptions**
tp_init			
tp_set	T_1, T_2, ..., T_N		
tp_get		T	

Example Subclass Exception in Python

```
class Full(Exception):
  def __init__(self, value):
    self.value = value
  def __str__(self):
    return str(self.value)
```

Example of raising the exception

```
    if size == Seq.MAX_SIZE:
      raise Full("Maximum size exceeded")
```

Exception Signaling

- Useful to think about exceptions in the design process
- Will need to decide how exception signalling will be done
  - A special return value, a special status parameter, a global variable
  - Invoking an exception procedure
  - Using built-in language constructs
- Caused by errors made by programmers, not by users
- Write code so that it avoid exceptions
- Exceptions will be particularly useful during testing

Assumptions versus Exceptions

- The assumptions section lists assumptions the module developer is permitted to make about the programmer's behaviour
- Assumptions are expressed in prose
- Use assumptions to simplify the MIS and to reduce the complexity of the final implementation
- Interface design should provide the programmer with a means to check so that they can avoid exceptions
- When an exceptions occurs no state transitions should take place, any output is *don't care*

Quality Criteria

- Consistent
  - Name conventions
  - Ordering of parameters in argument lists
  - Exception handling, etc.
- Essential - omit unnecessary features
- General - cannot always predict how the module will be used
- As implementation independent as possible
- Minimal - avoid access routines with two potentially independent services
- High cohesion - components are closely related
- Low coupling - not strongly dependent on other modules
- Opaque - information hiding