CAS 741, CES 741 (Development of Scientific Computing Software)

Fall 2017

18 MIS Continued

Dr. Spencer Smith

Faculty of Engineering, McMaster University

November 10, 2017

McMaster University

MIS Continued

- Administrative details
- Questions?
- Exceptions
- Quality criteria
- Modules with external interaction, enviro variables
- GUI modules
- ADTs
- Generic modules
- OO design spec
- Examples

Administrative Details

- GitHub issues for colleagues
  - Assigned 1 colleague (see `Repos.xlsx` in repo)
  - Provide at least 5 issues on their MG
  - Grading as before
  - Due by Tuesday, Nov 14, 11:59 pm
- MIS template in CAS 741 repo

Administrative Details: Deadlines

MIS Present	Week 10	Week of Nov 13
MIS	Week 11	Nov 22
Impl. Present	Week 12	Week of Nov 27
Final Documentation	Week 13	Dec 6

Administrative Details: Presentation Schedule

- MIS Present
  - **Tuesday: Isobel, Keshav, Paul**
  - **Friday: Shusheng, Xiaoye, Devi**
- Impl. Present
  - Tuesday: Alexander S., Steven, Alexandre P.
  - Friday: Jason, Geneva, Yuzhi

MIS Presentations and Documentation

- For each module
  - Module or Template Module or Generic Template Module
  - Syntax, especially access programs
  - State variables
- Do not need a formal spec for everything
- Goal is communication with a developer or maintainer
- Clarifying comments in the MIS are helpful
- Use notation from SRS wherever possible

Questions?

- Questions about MIS presentations?
- Questions about MIS documentation?
- Other questions?

Exception Signalling

- 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

SWHS Example

Look at SWHS repo

Modules with External Interaction

- In general, some modules may interact with the environment or other modules
- Environment might include the keyboard, the screen, the file system, motors, sensors, etc.
- Sometimes the interaction is informally specified using prose (natural language)
- Can introduce an environment variable
  - Name, type
  - Interpretation
- Environment variables include the screen, the state of a motor (on, direction of rotation, power level, etc.), the position of a robot

External Interaction Continued

- Some external interactions are hidden
  - Present in the implementation, but not in the MIS
  - An example might be OS memory allocation calls
- External interaction described in the MIS
  - Naming access programs of the other modules
  - Specifying how the other module's state variables are changed
  - The MIS should identify what external modules are used

MIS for GUI Modules

- Could introduce an environment variable
- window: sequence [RES_H][RES_V] of pixelT
  - Where window[r][c] is the pixel located at row r and column c, with numbering zero-relative and beginning at the upper left corner
  - Would still need to define pixelT
- Could formally specify the environment variable transitions
- More often it is reasonable to specify the transition in prose
- In some cases the proposed GUI might be shown by rough sketches

Display Point Masses Module Syntax

Exported Access Programs

Routine name	In	Out	Exc
DisplayPointMassesApplet		DisplayPointMassesApplet	
paint			

Display Point Masses Module Semantics

Environment Variables

win : 2D sequence of pixels displayed within a web-browser

DisplayPointMassesApplet():

- transition: The state of the abstract object ListPointMasses is modified as follows:
  ListPointMasses.init()
  ListPointMasses.add(0, PointMassT(20, 20, 10))
  ListPointMasses.add(1, PointMassT(120, 200, 20))
  ...

paint():

- transition *win* := Modify window so that the point masses in ListPointMasses are plotted as circles. The centre of each circles should be the corresponding x and y coordinates and the radius should be the mass of the point mass.

Specification of ADTs

- Similar template to abstract objects
- "Template Module" as opposed to "Module"
- "Exported Types" that are abstract use a ?
  - pointT = ?
  - pointMassT = ?
- Access routines know which abstract object called them
- Use "self" to refer to the current abstract object
- Use a dot "." to reference methods of an abstract object
  - p.xcoord()
  - $self.pt$.dist(p.point())
- Similar notation to Java
- The syntax of the interface in C is different

Syntax Line ADT Module

Template Module

lineADT

Uses

pointADT

Exported Types

lineT = ?

Syntax Line ADT Module Continued

Routine name	**In**	**Out**	**Exceptions**
new lineT	pointT, pointT	lineT	
start		pointT	
end		pointT	
length		real	
midpoint		pointT	
rotate	real		

Semantics Line ADT Module

State Variables

s: pointT
e: pointT

State Invariant

None

Assumptions

None

Access Routine Semantics Line ADT Module

new lineT (p_1, p_2):

- transition: $s, e := p_1, p_2$
- output: $out := self$
- exception: none

start:

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

end:

- output: $out := e$
- exception: none

Access Routine Semantics Continued

length:

- output: $out := s.\mathrm{dist}(e)$
- exception: none

midpoint:

- output: $out :=$

  $$\mathrm{new\ pointT}(\mathrm{avg}(s.\mathrm{xcoord}, e.\mathrm{xcoord}), \mathrm{avg}(s.\mathrm{ycoord}, e.\mathrm{ycoord}))$$

- exception: none

rotate (φ):
φ is in radians

- transition: $s.\mathrm{rotate}(\varphi), e.\mathrm{rotate}(\varphi)$
- exception: none

Line ADT Local Functions

Local Functions

avg: real $\times$ real $\rightarrow$ real
avg$(x_1, x_2) \equiv \frac{x_1+x_2}{2}$

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

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

Assumptions

Semantics

State Variables

s: sequence of T

State Invariant

$|s| \leq$ 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()
$s' = s$

Object Oriented Design

- One kind of module, ADT, called class
- A class exports operations (procedures) to manipulate instance objects (often called methods)
- Instance objects accessible via references
- Can have multiple instances of the class (class can be thought of as roughly corresponding to the notion of a type)

Inheritance

- Another relation between modules (in addition to USES and IS_COMPONENT_OF)
- ADTs may be organized in a hierarchy
- Class B may specialize class A
  - B inherits from A
  - Conversely, A generalizes B
- A is a superclass of B
- B is a subclass of A

Template Module Employee

Routine name	In	Out	Except
Employee	string, string, moneyT	Employee	
first_Name		string	
last_Name		string	
where		siteT	
salary		moneyT	
fire			
assign	siteT		

Inheritance Examples

Template Module Administrative_Staff **inherits** Employee

Routine name	**In**	**Out**	**Exception**
do_this	folderT		

Template Module Technical_Staff **inherits** Employee

Routine name	**In**	**Out**	**Exception**
get_skill		skillT	
def_skill	skillT		

Inheritance Continued

- A way of building software incrementally
- Useful for long lived applications because new features can be added without breaking the old applications
- A subclass defines a subtype
- A subtype is substitutable for the parent type
- Polymorphism - a variable referring to type A can refer to an object of type B if B is a subclass of A
- Dynamic binding - the method invoked through a reference depends on the type of the object associated with the reference at runtime
- All instances of the sub-class are instances of the super-class, so the type of the sub-class is a subtype
- All instances of Administrative_Staff and Technical_Staff are instances of Employee

Dynamic Binding

- Many languages, like C, use static type checking
- OO languages use dynamic type checking as the default
- There is a difference between a type and a class once we know this
  - Types are known at compile time
  - The class of an object may be known only at run time

Point ADT Module

Template Module

PointT

Uses

N/A

Syntax

Exported Types

PointT = ?

Point ADT Module Continued

Exported Access Programs

Routine name	In	Out	Exceptions
new PointT	real, real	PointT	
xcoord		real	
ycoord		real	
dist	PointT	real	

Semantics

State Variables

xc: real
yc: real

Point Mass ADT Module

Template Module

PointMassT **inherits** PointT

Uses

PointT

Syntax

Exported Types

PointMassT = ?

Point Mass ADT Module Continued

Exported Access Programs

Routine name	In	Out	Exceptions
new PointMassT	real, real, real	PointMassT	NegMassExcep
mval		real	
force	PointMassT	real	
fx	PointMassT	real	

Semantics

State Variables

ms: real

Point Mass ADT Module Semantics

new PointMassT(x, y, m):

- transition: $xc, yc, ms := x, y, m$
- output: $out := self$
- exception: $exc := (m < 0 \Rightarrow \text{NegativeMassException})$

force(p):

- output:

$$out := \mathrm{UNIVERAL\_G}\frac{self.ms \times p.ms}{self.\mathrm{dist}(p)^2}$$

- exception: none

Examples

- Example Point Line and Circle
- Example Robot Path
- Example Vector Space
- Example Othello Program
- Example Maze Formal Specification (Dr. v. Mohrenschildt)
- Mustafa ElSheikh Mesh Generator [1]
- Wen Yu Mesh Generator [2]
- Sven Barendt Filtered Backprojection
- Sanchez sDFT

References I

Jacques Carette, Mustafa ElSheikh, and W. Spencer Smith.
A generative geometric kernel.
In *ACM SIGPLAN 2011 Workshop on Partial Evaluation and Program Manipulation (PEPM'11)*, pages 53–62, January 2011.

W. Spencer Smith and Wen Yu.
A document driven methodology for improving the quality of a parallel mesh generation toolbox.
Advances in Engineering Software, 40(11):1155–1167, November 2009.