

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

Fall 2020

## MIS Continued

Dr. Spencer Smith

Faculty of Engineering, McMaster University

November 20, 2020



# MIS Continued

- Start recording
- Administrative details
- Questions?
- Nonfunctional requirements
- Review: Records, Libraries, ADTs, Abstract Objects, Generic ADTs
- Example - Student data
- Exceptions
- Quality criteria
- Modules with external interaction, enviro variables
- GUI modules
- ADTs
- Generic modules
- OO design spec
- Examples

# Administrative Details

- When developing your code, remember that your goal is for someone else to be able to compile and run it
- Upcoming classes
  - ▶ L16 - MIS Continued
  - ▶ L17 - POC + MG Presentations
  - ▶ L18 - MIS Presentations
- Mathematical review ([3] and separate slides)
- Potential software for drawing figures
  - ▶ [draw.io](#)
  - ▶ [Tkiz](#)

# Administrative Details: Report Deadlines

**MG + MIS (Traditional)** Nov 19

**Drasil Code and Report (Drasil)** Nov 19

**Final Documentation** Dec 9

- The written deliverables will be graded based on the repo contents as of 11:59 pm of the due date
- If you need an extension for a written deliverable, please ask
- You should inform your primary and secondary reviewers of the extension
- Two days after each major deliverable, your GitHub issues will be due

# Admin Details: Presentation Schedule

- Proof of Concept Demonstrations (15 min)
  - ▶ **Thurs, Nov 12: Salah, John**
- MG Present (10 minutes)
  - ▶ **Thurs, Nov 12: John, Tiago, Leila, Xuanming, Andrea**
- MIS Present
  - ▶ Mon, Nov 16: Shayan, Parsa, Gaby, Sid, Xingzhi
- Drasil Project Present (20 min each)
  - ▶ Thurs, Nov 26: Andrea, Naveen, Ting-Yu

# Presentation Schedule Continued

- Test or Impl. Present (15 min each)
  - ▶ Mon, Nov 30: John, Salah, Liz, Xingzhi, Leila
  - ▶ Thurs, Dec 3: Shayan, Naveen, Sid, Gaby, Seyed
  - ▶ Mon, Dec 7: Ting-Yu, Xuanming, Mohamed, Andrea, Tiago
- 4 presentations each
- If you will miss a presentation, please trade with someone else

# Questions?

- Questions on administrative details?
- Questions about Module Guide?
- Questions about upcoming presentation?
- Questions about MIS?
- Other questions?

# Nonfunctional Requirements

- Aim to be unambiguous
- Say the quality you want to achieve, not how you are going to achieve it
- Point to the Verification and Validation plan
- Added to the blank SRS template



## Examples of Modules: Record [2]

- Consists of only data
- Has state but no behaviour
- Example
  - ▶ Specification Parameters Module in SWHS

# Examples of Modules: Library [2]

- Collection of related procedures (library)
- Has behaviour but no state
- Procedural abstractions
- Example
  - ▶ Library of trigonometric functions
  - ▶ ODE Solver Module in SWHS
  - ▶ Sequence Services Module

# Examples of Modules: Abstract Object [2]

- Consists of data (**fields**) and procedures (**methods**)
- Consists of a collection of **constructors**, **selectors**, and **mutators**
- Has state and behaviour
- There is only ONE
- Singleton design pattern
- Example
  - ▶ Input Parameters Module for SWHS
  - ▶ Logger

# Examples of Modules: Abstract Data Type [2]

- What you are used to for OO programming
- Consists of a collection of abstract objects and a collection of procedures that can be applied to them
- Defines the set of possible values for the type and the associated procedures that manipulate instances of the type
- Encapsulates the details of the implementation of the type
- Multiple instances of the object
- Keyword **Template** in MIS
- Example
  - ▶ [Curve ADT Module](#)

# Examples of Modules: Generic [2]

- A single abstract description for a family of abstract objects or ADTs
- Parameterized by type
- Eliminates the need for writing similar specifications for modules that only differ in their type information
- A generic module facilitates specification of a stack of integers, stack of strings, stack of stacks etc.
- Example
  - ▶ Generic Sequence ADT Module

# Chemistry Example - Highlight Mathematics

- Problem Description
- Source Code

# 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 [3, p. 83]

- 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

# 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

<b>Routine name</b>	<b>In</b>	<b>Out</b>	<b>Exc</b>
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

<b>Routine name</b>	<b>In</b>	<b>Out</b>	<b>Exceptions</b>
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:  $out := s.dist(e)$
- exception: none

midpoint:

- output:  $out :=$

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

- exception: none

rotate ( $\varphi$ ):

$\varphi$  is in radians

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

# Line ADT Local Functions

## Local Functions

avg:  $\text{real} \times \text{real} \rightarrow \text{real}$

$$\text{avg}(x_1, x_2) \equiv \frac{x_1 + x_2}{2}$$

# 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

<b>Routine name</b>	<b>In</b>	<b>Out</b>	<b>Exceptions</b>
s_init			
s_push	T		FULL
s_pop			EMPTY
s_top		T	EMPTY
s_depth		integer	



# 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:  $s := \langle \rangle$
- exception: `none`

`s_push(x)`:

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

`s_depth()`:

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

# Stack Module Properties

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

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

$\{|s| < MAX\_SIZE\}$   
     $s\_push(x)$   
     $s\_pop()$   
 $s' = s$

# 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

<b>Routine name</b>	<b>In</b>	<b>Out</b>	<b>Except</b>
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

<b>Routine name</b>	<b>In</b>	<b>Out</b>	<b>Exception</b>
do_this	folderT		

**Template Module** Technical\_Staff **inherits** Employee

<b>Routine name</b>	<b>In</b>	<b>Out</b>	<b>Exception</b>
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

<b>Routine name</b>	<b>In</b>	<b>Out</b>	<b>Exceptions</b>
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:

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

- exception: none

# Judging Appropriate Abstraction

- If an MIS is too abstract, it won't capture enough information for someone to do the implementation
- In some cases an MIS is not abstract enough
  - ▶ This can happen when someone is reverse engineering their spec from existing code
  - ▶ Can happen with an operational specification, as opposed to a descriptive specification
- Judge the abstraction level by
  - ▶ If a change in how your code works requires a change in your specification, look for a better abstraction
  - ▶ If writing and maintaining the spec is exceedingly frustrating, the spe could be too concrete



# Examples

- Solar Water Heating System
- 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 [4]
- 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.



Carlo Ghezzi, Mehdi Jazayeri, and Dino Mandrioli.

*Fundamentals of Software Engineering.*

Prentice Hall, Upper Saddle River, NJ, USA, 2nd edition, 2003.

# References II



Daniel M. Hoffman and Paul A. Strooper.

*Software Design, Automated Testing, and Maintenance: A Practical Approach.*

International Thomson Computer Press, New York, NY, USA, 1995.



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.