DEPARTMENT OF COMPUTER SCIENCE

ASSESSMENT DESCRIPTION 2013/14 (EXAM TESTS AND COURSEWORK)

MODULE DETAILS:

Module Number:	08981	Semester:	1
Module Title:	Component Based Architecture		
Lecturer:	Dr DJ Grey		

COURSEWORK DETAILS:

Assessment Number:	2	of	2		
Title of Assessment:	Simple Virtual Machine				
Format:	Program	Demonstration	Report		
Method of Working:	Individual				
Workload Guidance:	Typically, you should expect to spend between	50	and	100	hours on this assessment
Length of Submission:	This assessment should be **no** more than: *(**over length submissions will be penalised as per University policy – see below**)*	1000 **words** *(excluding diagrams, appendices, references, code)*			

PUBLICATION:

Date of issue:	Monday 4 November 2013

SUBMISSION:

ONE copy of this assessment should be handed in via:	E-Bridge		If Other (state method)	
Time and date for submission:	**Time**	9:30am	**Date**	Thurs 12 December 2013
If **multiple hand–ins** please provide details*:*				
Will submission be scanned via TurnItIn?	No	If this requires a separate TurnItIn submission, please provide instructions:		
*Late submissions **will be** penalised as per university policy*				

The assessment must be submitted **no later** than the time and date shown above, unless an extension has been authorised on a *Request for an Extension for an Assessment* form which is available from the Departmental Office (RB-308) or http://intra.net.dcs.hull.ac.uk/student/exam/Advice%20regarding%20resits%20in%20modules%20passed%20by%20compe/Forms/AllItems.aspx.

If Turnitin is taking a long time to produce its analysis you must still submit your work, albeit initially without the Turnitin analysis. A delay at Turnitin cannot be used to excuse a late submission.

MARKING:

Marking will be by:	Student Name

COURSEWORK COVERSHEET:

BEFORE submission, you **must** ensure you complete the **correct** departmental ACW cover sheet (if required) and attach it to your work. The coversheets are available from: http://intra.net.dcs.hull.ac.uk/student/ACW%20Cover%20Sheets/Forms/AllItems.aspx	NO coversheet required

ASSESSMENT:

The assessment is marked out of:	100	and is worth	50	% of the module marks
N.B If multiple hand-ins please indicate the marks and % apportioned to each stage above (i.e. Stage 1 – 50, Stage 2 – 50). It is these marks that will be presented to the exam board.				

ASSESSMENT STRATEGY AND LEARNING OUTCOMES:

The overall assessment strategy is designed to evaluate the student's achievement of the module learning outcomes, and is subdivided as follows:

LO	Learning Outcome	Method of Assessment *{e.g. report, demo}*
1	*Identify the key components of a virtual machine used as a runtime environment for a contemporary managed programming language and describe with comprehension their function in detail.*	Report, Demo
2	*Show evidence of a systematic and comprehensive understanding of the role, design and implementation of intermediate languages.*	Demo
3	*Demonstrate research, selection and assessment of component-based software architectures and implementation techniques. Adapt approaches including some at the forefront of the discipline and identify possibilities for originality or creativity*	Demo

Assessment Criteria	Contributes to Learning Outcome	Mark
Task 1 – Program Compilation *Dynamic type instantiation* *Case-insensitive search* *Identification of types implementing IInstruction*	3	 5 5 5
Task 2 – Program Execution	1, 2, 3	

Execution of SML Program *Execution mechanism*		5 5
Task 3 – Adding Core SML Instruction *Incr instruction* *Decr instruction*	3	 5 5
Task 4 – Unit Tests	3	10
Task 5 – Extending the SML Instruction Set *Identification of assemblies* *Loading of assemblies* *Error handling* *Minimisation of memory footprint*	3	 5 5 5 10
Task 6 – Conditionals *Branch instruction implementation* *Label handling*	1, 2, 3	 5 10
Task 7 – Looping *SML program demonstrating loops*	3	 5
Task 8 – Report	1, 2, 3	10

FEEDBACK

Feedback will be given via:	Verbal (via demonstration)	Feedback will be given via:	N/A
Exemption (staff to explain why)			
Feedback will be provided no later than 4 'semester weeks' after the submission date.			

This assessment is set in the context of the learning outcomes for the module and does not by itself constitute a definitive specification of the assessment. If you are in any doubt as to the relationship between what you have been asked to do and the module content you should take this matter up with the member of staff who set the assessment as soon as possible.

You are advised to read the **NOTES** regarding late penalties, over-length assignments, unfair means and quality assurance in your student handbook, also available on the department's student intranet at:

- http://intra.net.dcs.hull.ac.uk/student/ug/Handbooks/Forms/AllItems.aspx (for undergraduate students)
- http://intra.net.dcs.hull.ac.uk/student/pgt/Student%20Handbook/Forms/AllItems.aspx (for postgraduate taught students).

In particular, please be aware that:

- Your work will be awarded zero if submitted more than 7 days after the published deadline.
- The overlength penalty applies to your written report (which includes bullet points, and lists of text you have disguided as a table. It does not include contents page, graphs, data tables and appendices). Your mark will be awarded zero if you exceed the word count by more than 10%.

Please be reminded that you are responsible for reading the University Code of Practice on the use of Unfair means (http://student.hull.ac.uk/handbook/academic/unfair.html) and must understand that unfair means is defined as any conduct by a candidate which may gain an illegitimate advantage or benefit for him/herself or another which may create a disadvantage or loss for another. You must therefore be certain that the work you are submitting contains no

section copied in whole or in part from any other source unless where explicitly acknowledged by means of proper citation. In addition, **please note** that if one student gives their solution to another student who submits it as their own work, **BOTH** students are breaking the unfair means regulations, and will be investigated.

In case of any subsequent dispute, query, or appeal regarding your coursework, you are reminded that it is your responsibility, not the Department's, to produce the assignment in question.

The Simple Virtual Machine

INTRODUCTION

In this assessed coursework exercise you are required to work with and extend a partial implementation of a simplistic virtual machine and its machine language. The goal of the exercise is to make you more familiar with virtual machine concepts and aspects of managed environments, such as metadata and reflection, which support component-based systems.

The exercise consists of seven tasks. You do not need to complete all of these tasks to pass the assignment but you should attempt all of the tasks to maximize the marks that you score.

THE SIMPLE VIRTUAL MACHINE

The Simple Virtual Machine (SVM) is, unsurprisingly, a simplistic implementation of a stack-based virtual machine. The SVM compiles and executes programs written in its Simple Machine Language (SML) format. SML is described further in the following section.

SML programs are assumed to be stored in files with a `.sml` extension and are executed using a command line of the following form

```
svm <program filename>.sml
```

SML programs are expressed in a textual form. The virtual machine loads the textual representation of the SML program, transforms it into an executable form and then executes it.

Internally the virtual machine consists of several important components

- *Stack* – this holds all the data required by the running program. SML instructions take their operands from the stack and leave return values upon the stack
- *Program* – the program is represented as a list of instructions in executable form. The program is executed by stepping through the list and triggering the execution of each instruction in turn
- *Program Counter* – an index into the Instruction List which indicates the index of the instruction that is currently executing. When an instruction completes, the program counter is updated to indicate the next instruction that should be executed.

THE SIMPLE MACHINE LANGUAGE

The software specification of the Simple Virtual Machine required that it had a core instruction set which was easily extensible by third-party developers. Simple Machine Language (SML) is the core instruction set of the Simple Virtual Machine.

As the SVM is a stack-based virtual machine, all of the instructions in the SML operate against the stack. Any type of data can be placed on the stack but the instructions are typed; that is, they expect to work with a particular type of data (e.g. integer) and check that the values they retrieve from the stack are of the correct type before the instruction is executed.

Most instructions do not take any operands but simply retrieve the data they need to operate on from the stack. Some instructions do require operands and a valid SML instruction may have up to two operands. However, instructions in the core SML instruction set have zero or one operand only.

SML consists of the following limited set of instructions:

Instruction	*Action*
`loadint integer`	loads the supplied integer onto the stack
`loadstring string`	loads the supplied string onto the stack
`add`	pops the top two numbers off the stack, adds them and pushes the result on the stack
`subtract`	pops the top two numbers off the stack, subtracting the first from the second, and pushes the result back on the stack
`writestring`	pops the top value from the stack and writes a string representation of it to the console

Table 1: The core SML instruction set

The skeleton Visual Studio solution supplied with this coursework exercise contains a simple sample SML program in a file called `sample.sml`. This program is listed below.

```
loadstring "Calculating (2 + 3) - 1"
writestring
loadint 2
loadint 3
add
loadint 1
subtract
loadstring "The result is"
writestring
writestring
```

When this program is executed it produces the following output

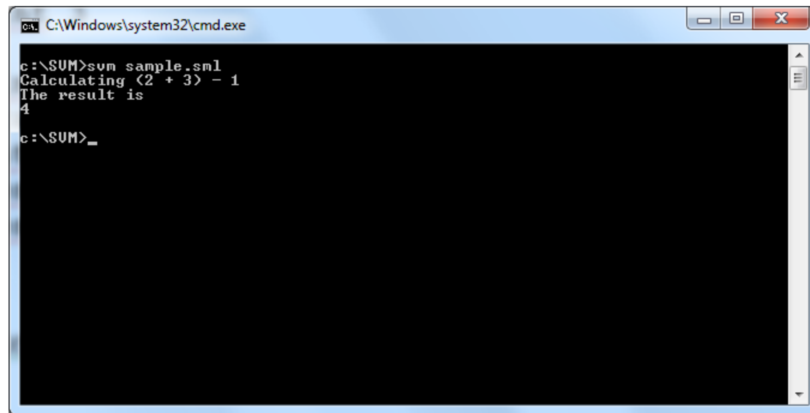

Figure 1 : Output from sample.sml

A valid SML program is a text file, with a `.sml` extension, which contains SML instructions and their operands. Each instruction must exist on a separate line within the source file.

SML INSTRUCTION IMPLEMENTATION

Within the virtual machine, SML instructions are implemented as C# classes which implement the **`IInstruction`** interface. Any class in any namespace which implements **`IInstruction`** is considered a valid SML instruction. SVM also defines an **`IInstructionWithOperand`** interface that inherits from **`IInstruction`**. Any C# class which implements **`IInstructionWithOperand`** is considered to be a valid SML instruction that takes one or more operands.

Instruction classes have to provide an **`Execute()`** method. The code within this method performs the operation defined by the instruction. In carrying out this operation it may be necessary for the instruction to access the stack or other properties of the virtual machine. All instruction classes must therefore provide a **`VirtualMachine`** property; when this instruction is executed a reference to the virtual machine executing the instruction is assigned to its **`VirtualMachine`** property, and the instruction can use this property to access the **`Stack`** of the virtual machine.

To simplify the implementation of instruction types, SVM provides an **`Instruction`** abstract class and an **`InstructionWithOperand`** abstract class which implement the **`IInstruction`** and **`IInstructionWithOperand`** interfaces respectively. Instruction classes can be created by inheriting from the appropriate **`Instruction`** or **`InstructionWithOperand`** classes as appropriate and overriding the **`Execute()`** method

to give the desired behaviour. All of the other properties, etc are implemented by the **`Instruction`** and **`InstructionWithOperand`** classes.

THE SKELETON VISUAL STUDIO SOLUTION

The majority of the SVM has been implemented and is provided for you to work with in a skeleton Visual Studio solution which can be downloaded from the *Assessments (Tests & ACW's)* section of the module SharePoint site.

In this assessment there are six programming tasks that you need to complete and these require you to work with different aspects of the SVM source code. The areas of the source code that you need to work with are clearly marked with C# `#region/#endregion` statements (e.g. `#region Task 1 – To be implemented by the student`). You should add or modify code within these regions as necessary to accomplish the task. Code outside of these regions should not be modified or deleted, although you may add code to existing files or additional code files as you see fit. For example, you may choose to add an additional property to the **`SvmVirtualMachine`** class but you should not remove any existing code outside the marked regions even if it is unused by your solution.

TASK 1- PROGRAM COMPILATION

The implementation supplied in the skeleton Visual Studio solution contains all the code necessary to process the command line and to load up and parse a `.sml` file. The **`SvmVirtualMachine.Compile()`** method is responsible for reading the .sml file and uses the **`SvmVirtualMachine.ParseInstruction()`** method to parse an individual SML instruction. The **`SvmVirtualMachine.ParseInstruction()`** method in turn makes use of the **`CompileInstruction()`** methods of the **`JITCompiler`** class to transform a textual SML instruction into an executable equivalent. The methods of the **`JITCompiler`** class have not been implemented. In this task you need to add code to these methods to convert a textual SML instruction into an executable equivalent.

A textual instruction is converted into an executable equivalent by Reflection. Using reflection you should examine each of the loaded assemblies and the types within those assemblies, searching for a type that implements **`IInstruction`** and which has the same name as the SML opcode. SML opcodes are not case sensitive, so in matching opcodes to C# type names you should ignore case. If no matching type can be found an **`SvmCompilationException`** should be thrown to indicate that an invalid SML instruction has been found in the SML source.

When a C# type matching an SML opcode is found, the search should terminate and an instance of the C# type should be created. This type instance should be returned as the return value of the **`CompileInstruction()`** method. If the SML instruction has operands then the **`Operands`** property of the newly created type instance should be used to assign the operands for that instruction.

Task 2 – Program Execution

The **`SvmVirtualMachine`** class defines a **`program`** field which is of type **`List<IInstruction>`**. This simply contains a list of **`IInstruction`** instances which implement the SML instructions read from the SML source file, in the order that they were read from the file. The SML program can be executed by moving through this list of instructions in the appropriate order and calling **`Execute()`** on each **`IInstruction`** instance.

The virtual machine triggers execution of the SML program by calling the **`SvmVirtualMachine.Execute()`** method. This method is not implemented in the skeleton solution. For this task you need to add the necessary code to the **`SvmVirtualMachine.Execute()`** method to move through the list of instruction in the program field and call **`Execute()`** on them.

Running the SVM project in Visual Studio will cause the SVM virtual machine to start and to compile and execute the `sample.sml` source file. You can verify your implementation by checking that the SML program executes without error and the output that is produced is the same as that shown in **Figure 1**.

Task 3 – Adding Core SML Instructions

The core SML instruction set defined in **Table 1** is somewhat limited in its functionality. In this task you are required to create the C# classes necessary to implement the following two SML instructions.

Instruction	*Action*
`incr`	Increments the integer value stored on top of the stack, leaving the result on the stack. An **`SvmRuntimeException`** should be generated if the value on top of the stack is not an integer.
`decr`	Decrements the integer value stored on top of the stack, leaving the result on the stack. An **`SvmRuntimeException`** should be generated if the value on top of the stack is not an integer.

The classes that implement these instructions should be added to the `SimpleMachineLanguage` folder of the `SVM` Visual Studio project. Test your modification by creating a simple `.sml` program which loads an integer value on the stack, increments and/or decrements it and prints out the result.

TASK 4 – UNIT TESTING THE ADDITIONAL SML INSTRUCTIONS

When developing software it is good practice to write *unit tests* for each class and method implemented to ensure that each specific unit of functionality operates in the expected manner and is fit for use. Each test case should be independent from the others and modules should be tested in isolation, which requires the use of test harnesses and, in many cases, mock objects.

Create an interface called **`IVirtualMachine`** in the VirtualMachine folder of the SVM project. This interface should define the core public behaviours of a virtual machine. You should also refactor the **`SvmVirtualMachine`** class such that this it implements the **`IVirtualMachine`** interface and replace references to **`SvmVirtualMachine`** in the **`IInstruction`** and **`Instruction`** types with references to **`IVirtualMachine`**.

Add a unit test project to your solution and add to this project a sufficient set of unit tests for the classes that implement the SML **`incr`** and **`decr`** operations. Your unit tests should not require the instantiation of the **`SvmVirtualMachine`** class or the loading, parsing and execution of an SML program.

Hint: An SML instruction implementation has a **`VirtualMachine`** property which is set when the instruction is executed. This **`IVirtualMachine`** instance referenced by this property allows the instruction to access the stack and program counter of a virtual machine. If you cannot instantiate an actual instance of **`SvmVirtualMachine`** and store this in the **`VirtualMachine`** property of the instruction under test you will need to provide an alternative implementation of **`IVirtualMachine`**.

TASK 5 – EXTENDING THE SML INSTRUCTION SET

The software specification for the SVM required that the instruction set of the virtual machine be extensible in an unrestricted manner. To achieve this, the virtual machine should look not only in the set of loaded assemblies for C# types implementing SML instructions, but also in any DLL found in the directory in which the SVM executable is located (this directory can be identified by calling **`System.Environment.CurrentDirectory`**).

There are two parts to this task. Firstly, you should modify the code of the **`JITCompiler.CompileInstruction()`** methods that you implemented for Task 1 to search for C# types implementing SML instructions in any DLL found in the directory in which the SVM executable is located. Your code should account for the fact that not all DLL's contain .NET assemblies. For maximum marks you should only load into memory those assemblies containing SML instructions that are used by the SML program being compiled.

The SVM stack can store any type of data. For the second part of this task you should add implementations of the following two instructions to the `SML Extensions` project in the Visual Studio solution. The Visual Studio solution has been configured to automatically copy this assembly built by this project into the directory containing the SVM executable when the solution is built. Test your modification by creating a simple SML program that loads an image on to the stack and displays it on the screen.

Instruction	*Action*
`loadimage imagepath`	Loads the image found at the file path specified by the **`imagepath`** string operand on to the stack. An **`SvmRuntimeException`** should be generated if the file does not exist or is not a valid image.
`Displayimage`	Displays, on screen, the image loaded on top of the stack. An **`SvmRuntimeException`** should be generated if the value on top of the stack is not an image.

You are also required to implement appropriate unit tests for the **`loadimage`** instruction in your test project.

TASK 6 – CONDITIONALS

The SML instruction set does not support conditional (i.e. branch) instructions. The aim of this task is to add support for the following conditional instructions

Instruction	*Action*
`equint value branch_location`	Jumps to the SML instruction labelled by the **`branch_location`** if the **`value`** operand is equal to the integer value on top of the stack. An **`SvmRuntimeException`** should be generated if the value on top of the stack is not an integer.

bgrint value branch_location	Jumps to the SML instruction labelled by the **branch_location** if the **value** operand is greater than the integer value on top of the stack. An **SvmRuntimeException** should be generated if the value on top of the stack is not an integer.
bltint value branch_location	Jumps to the SML instruction labelled by the **branch_location** if the **value** operand is less than the integer value on top of the stack. An **SvmRuntimeException** should be generated if the value on top of the stack is not an integer.

In order to make these conditional instructions work you will need to modify the virtual machine implementation to support labeling of instructions. For example, an SML program that uses a branch instruction may look like

```
%addOne% incr
bltint 5 addOne
```

where **%addOne%** is a label applied to the **incr** instruction (**Note**: that labels must start and end with % delimiters, but the % is not part of the label name). If the value on the stack is less than 5 when the **bltint** instruction is executed, program execution should jump to the instruction associated with the **addOne** label (i.e. the **incr** instruction). Labels must always precede the opcode in the source file and labels must be terminated by a colon (:).

You will need to introduce a mechanism to identify the instruction associated with a label when the program is executing and to force this instruction to become the next one executed. This may require you to introduce additional data structures and change your implementation of the **SvmVirtualMachine.Execute()** method.

HINT: You may find the **lineNumber** variable maintained by the **SvmVirtualMachine.Compile()** useful.

You are also required to implement appropriate unit tests for the **equint**, **bgrint**, and **bltint** instructions in your test project.

TASK 7 – LOOPING

This task does not require you to write any C# code. Instead, you should demonstrate the branching capabilities of your modified virtual machine by writing and executing an SML program which implements behaviour equivalent to the following C# code.

```
int count = 0;
while (count < 5)
{
    count++;
}
Console.WriteLine("Count =");
Console.WriteLine(count);
```

TASK 8 – REPORT

You are required to write a brief report on your implementation and modification of the Simple Virtual Machine which

- briefly describes your algorithm for identifying a C# type which matches an SML opcode and discusses how Reflection has been used
- briefly describes your algorithm for the **`SvmVirtualMachine.Execute()`** method implemented for Task 2
- outlines your mechanism for searching in additional DLL's for SML instruction implementations and only loading assemblies containing instructions that are used into memory
- discusses why it make sense to refactor the **`SvmVirtualMachine`** class and introduce an **`IVirtualMachine`** interface
- describes how you provided an **`IVirtualMachine`** instance to the **`incr`** and **`decr`** instructions during unit testing
- includes a screenshot of the Visual Studio Test Explorer or Test Results window showing all of the unit tests in your solution and the red/green pass/fail status for each test
- outlines your mechanism for maintaining labels for instructions, finding an instruction with a specific label and making this the next instruction to be executed

- lists your SML source code for the solution to Task 7, along with a screen capture showing the output produced by this program when it is executed

- identifies ways in which your virtual machine implantation could be optimized, in terms of improving compilation and execution speed and minimizing the amount of memory used

SUBMISSION INFORMATION

This coursework is to be submitted via e-Bridge. To submit your coursework, you should ZIP up your Visual Studio solution and an electronic copy of your report into a single ZIP file. This ZIP file should then be submitted via e-Bridge.