The Abstract Machine

A Pattern for Designing Abstract Machines

Julio García-Martín Miguel Sutil-Martín
Universidad Politécnica de Madrid[1].

Abstract

Because of the increasing gap between modern high-level programming languages and existing hardware, it has often become necessary to introduce intermediate languages and to build abstract machines on top of the primitive hardware. This paper describes the ABSTRACT-MACHINE, a structural pattern that captures the essential features addressing the definition of abstract machines. The pattern describes both the static and dynamic features of abstract machines as separate components, as well as it considers the instruction set and the semantics for these instructions as other first-order components of the pattern.

Intent

Define a common template for the design of Abstract Machines. The pattern captures the essential features underlying abstract machines (i.e., data area, program, instruction set, etc...), encapsulating them in separated loose-coupled components into a structural pattern. Furthermore, the proposal provides the collaboration structures how components of an abstract machine interact.

Also known as

Virtual Machine, Abstract State Machine.

Motivation

Nowadays, one of the buzziest of the buzzwords in computer science is Java. Though to be an object-oriented language for the development of distributed and GUI applications, Java is almost every day adding new quite useful programming features and facilities. As a fact, in the success of Java has been particular significant of Java the use of the virtual machine's technology[2]. As well known, the Java Virtual Machine (JVM) is an abstract software-based machine that can operate over different microprocessor machines (i.e., hardware independent). Designers of the JVM must comply with the specification of the JVM and make the necessary bridge from the JVM virtual scene into concrete operating systems and microprocessors. This behind the scenes bridge allows the software developers to *"Write Once, Run AnyWhere"* [1] because the JVM must behave the same regardless of the underlying microprocessor according standard specifications of the JVM.

[1] LSIIS Department, Facultad de informática, Campus de Montegancedo, Boadilla del Monte, 28660 Madrid, Spain. Email: juliog@fi.upm.es

[2] In the past, long before the boom of Java, many works on declarative programming had already pointed out the success of abstract machines to compile languages as an alternative to the commonly used implementation techniques. Thus, examples like Prolog [7] or SML [6] mean good examples of high-performance programming languages implemented by abstract machines. Nowadays, it is very common the usage of the expression "virtual machine".

The increasing gap between modern high-level programming languages and existing hardware, it has often become necessary to introduce intermediate languages and to build abstract machines on top of the primitive hardware. However, in many cases the gap is so large that it is either hard to see how the source language relates to the intermediate language or, alternatively, how de intermediate language relates to the hardware. Unfortunately, in many cases the abstract machine is merely presented as consisting of some registers, areas of memory and a set of machine instructions with very little or no correspondence to the source language.

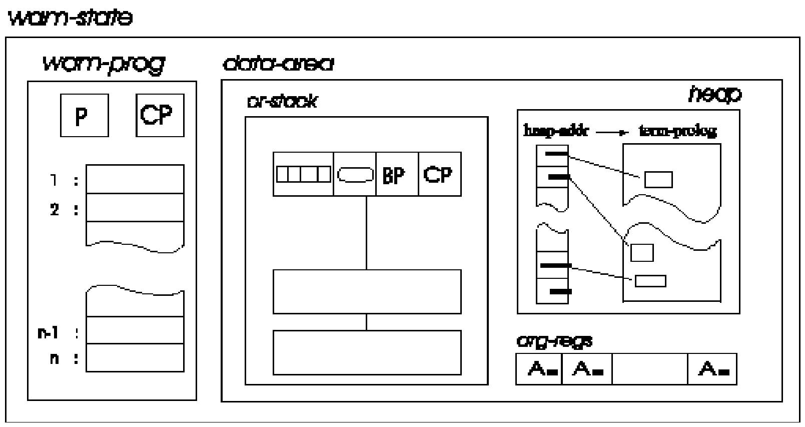
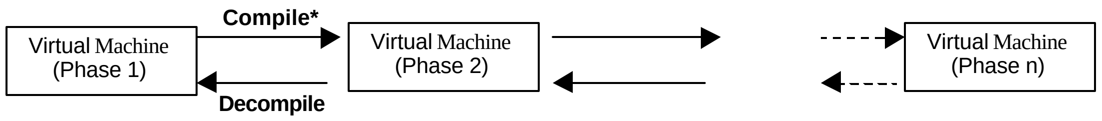


Figure 1. *An example of abstract of machine: The Abstract-WAM [3]*

The objective of this work is two fold: a) firstly, to devise some constituents of a methodology for the design of abstract machines; afterwards, b) to use the methodology to state a pattern-based framework to design programming language compilers. The usage of such a implementation technology (i.e., abstract machines) allows the task of compiling can be planned as a stepwise-refinement process, so, the piecemeal acquisition of high-performance properties is posed in terms of relationships between intermediate abstract machines. The coherence between one machine and the next is preserved thought the process. Each compilation step makes explicit some new features and changes that are added to the global compilation process (see figure 2).


Compile*
Virtual Machine (Phase 1)
Virtual Machine (Phase 2)
Virtual Machine (Phase n)
Decompile

** It might be "specify" also*

Figure 2. *A Compilation process by stepwise refinement of virtual machines*

The process contributes to get a more abstract and systematic way of constructing compilers. Furthermore, it improves the understanding of the process (compilation) and simplifies the task of refining and reusing previous designs. It is likely that a (prototype) compiler can be extracted more or less automatically as a side effect of the design of the abstract machine.

Applicability

Use the ABSTRACT-MACHINE pattern in any of the following situations:

- *When we want to specify an abstract machine.* The pattern provides a common skeleton to write high-level specifications, allowing the programmer puts more attention in specifying the machine's components instead of worrying for their combination (which can be provided by the pattern for free). The specifications can be formal or informal descriptions.
- *When we want to compile a language by using the abstract machine technology.* Abstracts machines provide a well-suited framework to describe compiling processes by stepwise-refinement of intermediate languages (i.e., development '*by prototyping*"). In such a process, the relationships between intermediate languages (i.e., abstract machines) define the whole compilation process.
- *When we want to specify compilers obtained from applying abstract machines.* It is as a result of combining the two previous situations.
- *When we want to test different semantics for the same instruction*. It is possible to define different semantics for the same instruction, as well, to get different implementations for the same semantics.
- *When we want to test different instruction sets for the same abstract machine*. It is possible to define different instruction sets for the same abstract machine definition. Moreover, the compilation process as a stepwise refinement implies that each intermediate abstract machine defines a more refined instruction set than the one managed by the previous machine.
- *When we want to emulate the execution of our programs.* The program emulation forces to incorporate visualization and debugging facilities into the programs. It is quite easy to consider the introduction of these view-components and other characteristics as participants of the Abstract Machine pattern. In particular, the execution code related to visualization or debugging can be defined by enhancing the semantics of machine's instructions.

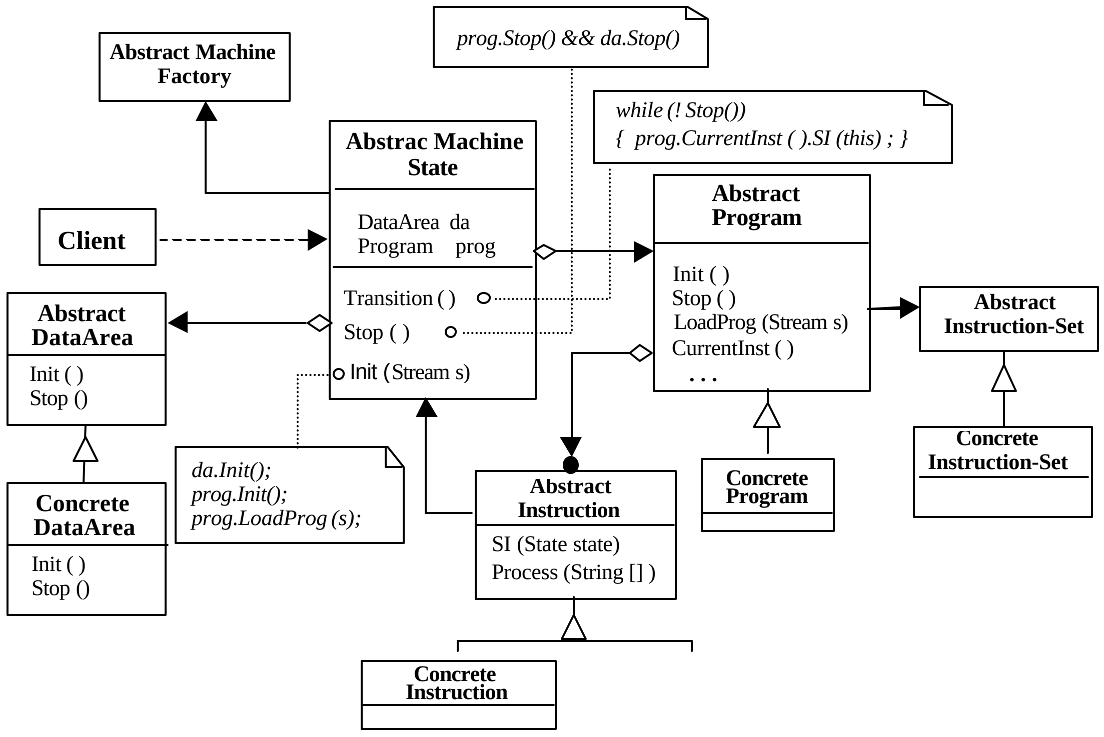


Figure 3. *ABSTRACT-MACHINE pattern (structure)*

Structure

The structure of the ABSTRACT-MACHINE pattern is shown on Figure 3.

Participants

Abstractly, an abstract machine is defined as the union of two parts: (i) the *static* part, consisting of the components related to the state, (ii) the *dynamic* part, determining the element associated with the behavior of the abstract machine. The ABSTRACT-MACHINE pattern organizes and locates the participants of the two parts as components of its structure.

The *state* of an abstract machine consists of the following components:

1. ***Abstract-Machine Factory***: It declares an interface for operations that create the *Abstract DataArea* and the *Abstract Program*.
2. ***Abstract DataArea***: It declares an interface for a type of data area object to configure the machine. It declares two abstract operations:
   - The *Init* operation, to determine the initial data area configuration, and
   - The *Stop* operation, that determines if the machine has got to the end of its execution. If this ending condition does not depend of the data area, then the Stop operation returns *true*.
3. ***Concrete DataArea***: It defines a concrete data area object. The concrete data area may be a simple object or a complex object structure (an object container). This component has to provide implementations for the *Abstract DataArea* interface.
4. ***Abstract Program***: It declares a common interface for assembler programs. Its definition includes has to deal with a collection of instructions (*Abstract Instructions)* and an instruction set *(Abstract Instruction Set)*. Besides, it declares four abstract operations:
   - The *Init* operation, to determine the initial configuration for assembler program,
   - The *Stop* operation, to determine if the machine has achieved its final stage. If this termination condition does not depend of any program configuration, then the Stop operation returns *true*.
   - The *LoadProg* operation is responsible of constructing the assembler instructions of the machine's program. The operation reads a textual representation from an input stream and translates each instruction into a *Concrete Instruction* object.
   - The *CurrentInst* operation returns the instruction to execute by the abstract machine.
5. ***Concrete Program***: It defines a concrete program object. It is defined as the aggregation of a concrete instruction set and some program counters. It has to implement the operations *Init, Stop* and *CurrentInst* defined on the Abstract Program.
6. ***Abstract Instruction Set***: It declares a common interface to represent a set of abstract instructions.
7. ***Concrete Instruction Set***: It defines a set object with Concrete Instruction objects. A concrete instruction set is related to a concrete data area and a concrete program. It has to implement the operations of the Abstract Instruction Set.
8. ***Abstract Instruction***: It declares a common interface to represent an abstract machine's instruction.
9. ***Concrete Instruction***: It defines a concrete instruction object. A concrete instruction is directly related to a concrete data area and a concrete program. It has to implement the operations of the Abstract Instruction.

On its side, the *behavior* of an abstract machine relies on some operations that operate over the static components. These operations are a part of the definition of the machine's state and they are responsible of describing which are the different states the abstract machine achieves during the execution. These operations are described below:

1. ***Abstract-Machine State***: It coordinates the interactions between *DataArea and Program* as a result of executing an assembler Concrete Instruction. So, it has a similar role of a MEDIATOR pattern [4]. The Abstract-Machine State may achieve three different stages (*initial, executing* and *final)* that are related to the following operations:
   - The *Init* operation, to determine the initial state of the machine, at the beginning of the program execution.
   - The *Stop* operation, to determine if the execution has achieved the machine's final state.
   - The *Transition* operation, to perform the execution of the machine's programs. It is defined as a while-loop control structure; at every round of the loop the instruction pointed by the program counter (the CurrentInst) is executed. Transition starts at the initial machine's configuration and continuous until the final stage is achieved.
2. ***Abstract & Concrete Instruction***:
   - The *SI* operation (semantics function) determines how the machine's configuration evolves due to the execution of the instruction. Each instruction defines its own SI semantics function and establishes how the machine's state changes (modifying the data area, the program or both). In order to do these changes, the instructions must be able to access to the components on the current state. It is done through a copy-reference of the machine's state that is passed to the instruction as a parameter.

Collaborations

Three scenarios describe the three different states an abstract machine may achieve during its execution: *initialization*, *transition* and *ending.* Figures 4, 5 and 6 sketches these scenarios.

Scenario 1: *Initialization*

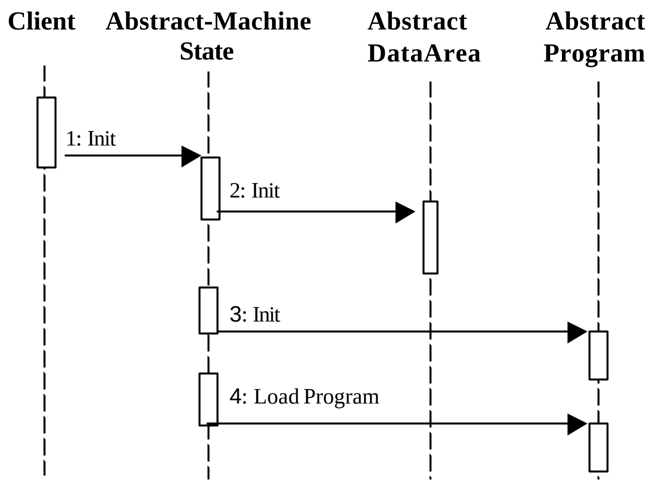


Figure 4 . The initialization stage (collaborations)

At this step, the abstract machine is initialized. As a result, both the DataArea and Program and are initialized. The initialization of the Program is carried out by the Init operation, and involves set its components to the initial values (i.e., program counters, array of instructions, etc…). Next, it is executed the LoadProgram operation, which is in charge of loading the assembler program (i.e., to translate the text representation of assembler instructions from the inputStream into instruction objects of the program). On the other hand, the Init operation initializes the components in the DataArea (i.e., the data registers or control registers, etc).

Scenario 2: *Transition*

As said before, the Transition operation performs the machine execution. So, Transition requests from the program the instruction to execute, which is pointed by the program counter (by the CurrentInst operation). The execution of the machine consists of executing instructions until the ending condition is achieved. Each machine's instruction is responsible of defining its own semantics. So, each instruction provides the SI operation, which execution modifies the machine's state (i.e., the program and/or the data area). Therefore, depending on the instruction will be modified the data area, the program or both. The order the modifications are carried out only depends on the instruction semantics.

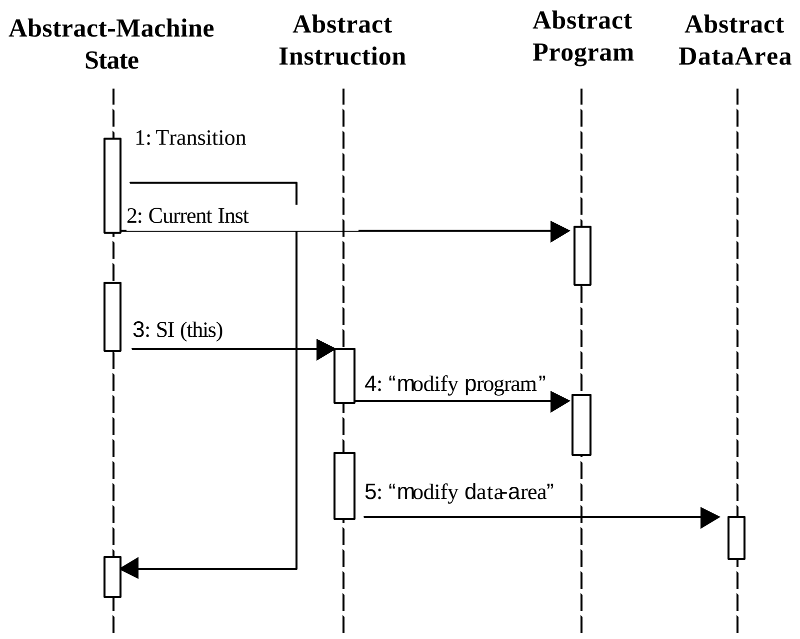


Figure 5 . The transition stage (collaborations)

Scenario 3: Ending

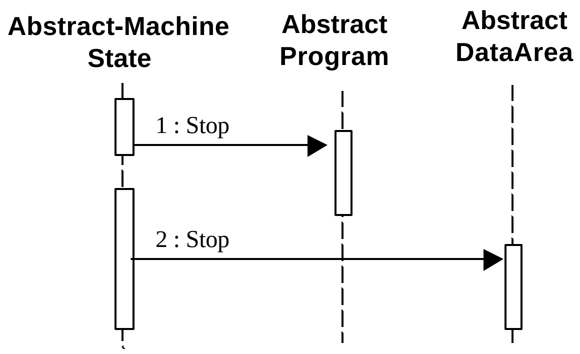


Figure 6 . The final stage (collaborations)

The stage for the end of machine's execution is achieved when any of its two components (or both) achieves the ending condition (e.g., it is executed a concrete stop instruction, there is a data area overflow, etc…). The state evaluates the corresponding Stop operations on the DataArea and the Program, and combines their result.

Consequences

The ABSTRACT-MACHINE pattern presents the following advantages:

- *Provide a framework to develop abstract machines*. The pattern defines a high-level and flexible design to develop abstract machines.
- *Provide a framework to develop compilers by stepwise refinement of abstract machines*. The steps of refinement in the process of compilation can be stated in terms of the refinement in the abstract machine's structure or its components.
- *De-couple the relationships among the abstract machine participants*. The Abstract Machine Factory helps to isolate the Abstract Machine State from concrete data area and program implementations. The Abstract Machine State manipulates instances through their abstract interfaces. On its side, the Abstract Instruction Set behaves as an abstract factory for the instruction sets managed by the programs.
- *Provide a framework to test different instruction sets.* The pattern makes exchanging instruction sets easy, allowing the data area and the program to maintain independent of any concrete instruction set. To change the concrete instruction set re-configures the abstract machine.
- *Provide a framework to test different instruction semantics or implementations for the same machine instruction.* Because the semantics are encapsulated by the instructions (the SI operation), we can redefine them without it affects to the data area and/or the program.
- *Offer a higher degree on the re-use of abstract machines.* In most of the cases, an abstract machine, the whole or a part, can be highly re-used in new developments.
- *Promote a methodology for a most systematic development of abstract machines.* By separating the participants of abstract machines in components (i.e., the data area, the program, the instruction set and the instruction semantics) we introduce some design constraints that force the user to follow a systematic approach.

The ABSTRACT-MACHINE presents the following disadvantages:

- *Yield to obtain inefficient implementations*.
- *Provoke a communication overhead between Abstract-Machine State and Abstract Instruction*.

Implementation

Consider the following implementation issues:

1. *Creating the concrete Data Area and Program.* The *Abstract-Machine Factory* only declares an interface for creating Data Area and Program objects. Follow similar hints that those given for the ABSTRACT-FACTORY pattern [4].
2. *Concrete Data Area might be a complex object composition.* Follow the same implementation issues as in for the COMPOSITE pattern [4]
3. *Defining the Abstract Data Area, Abstract Program, Abstract Instruction and Abstract Instruction Set as interfaces.* For example, following the Java conventions:

```
public interface AbstractInstruction
{
    public void SI (AbstractMachineState state);
    public String Name ();
    public int NumArguments ();
    public String ToString ();
    public void Process (String args [])
}

public interface AbstractInstructionSet
{  ... }

public interface AbstractProgram
{
    public AbstractInstruction CurrentInst ();
    public void Init ();
    public boolean Stop ();
    public void LoadProgram (Stream assemblerCode);
}

public interface AbstractDataArea
{
    public void Init ();
    public boolean Stop ();
}
```

The Abstract Instruction interface provides methods to determine the information about instructions (i.e., *Name*, *NumArguments*), the method *SI* declares the instruction semantics, and finally, the method *Process* compiles/translates textual-representation of the instruction arguments into information managed by the instruction. The SI method receives, as an argument, a reference to the Abstract Machine State.

The Abstract Instruction Set defines the instruction set may be managed by the program. This component may be a directory name, an enumerated set of instruction names, a package, a module, etc … The operations defined in the Abstract Instruction Set interface only concerns the representation chosen.

The Abstract Program defines the common interface for an assembler program. The CurrentInst operation is used to access the instruction to be executed. Besides, the LoadProgram operation is in charge of building the Concrete Instruction objects to be added into the program.

4. *AbstractDataArea,AbstractProgram and AbstractInstruction as template parameters.* In a C++ like syntax, as follows:

```
class AbstractInstruction
{ .. }

template
class AbstractProgram <class AbstractInstruction>
{ .. }

class AbstractDataArea
    { .. }

template
class AbstractMachineState < class AbstractDataArea,
class AbstractProgram <AbstractInstruction>>
{ .. }
                                                  }
```

5. *Omitting the Abstract-Machine State class.* It is similar to the case of the MEDIATOR pattern [4].
6. *Primitive operations.* Some operations defined in Abstract DataArea, Abstract Program and Abstract Instruction are primitive. Then, they must be overridden. For example, they could be declared as pure virtual (in C++ conventions) or as part of an interface (Java conventions). The operations *Init*, *Transition* and *Stop* in the Abstract-Machine State must be never overridden.

Sample Code

The following sample code shows Java implementations for some parts of the ABSTRACT-MACHINE pattern.

1. Firstly, we define the interfaces corresponding to Abstract Data Area, Abstract Program and Abstract Instruction (see the Implementation section)

2. An abstract class (in C++ or Java *jargon*) means an alternative implementation for the Abstract Program. In this case, the abstract class could provide some extra functionality (e.g., a program counter P and the set of instructions).

```
import AbstractInstruction;
import AbstractInstructionSet;

public abstract class AbstractProgram
{
    public int P;
    private Vector program; // A container of AbstractInstruction
    private AbstractInstructionSet instSet;

    AbstractProgram (AbstractInstructionSet instSet)
    {
           this.instSet = instSet;
    }
    public void Init ()
    {
           P = <<init program address>>;
           Program = new Vector ();
    }
    public void LoadProgram (Stream assemblerCode)
    {
           << load instructions from the assembler code >>
    }
    public AbstractInstruction CurrentInst ()
    {
           return program.Get (P);
    }
    public void Next ()
    {
           P++;
    }
    abstract public boolean Stop ();
}
```

3. The implementation of the Abstract Machine Factory:

```
import AbstractDataArea;
import AbstractProgram;

public interface AbstractMachineFactory
{
    public AbstractDataArea  CreateDataArea ();
    public AbstractProgram   CreateProgram ();
}
```

4. The implementation of the Abstract Machine State.

```
import AbstractDataArea;
import AbstractProgram;
import AbstractInstruction

public class AbstractMachineState
{
    private AbstractDataArea da;
    private AbstractProgram prog;
    public AbstractMachineState (AbstractMachineFactory factory)
    {
           prog = factory.CreateProgram();
           da = factory.CreateDataArea();
    }

    public AbstractDataArea DataArea ()
    {
           return da;
    }
    public AbstractProgram Program ()
    {
           return prog;
    }
    public void Init (Stream s)
    {
           da.Init ();
           prog.Init ();
           prog.LoadProgram (s);
    }
    public boolean Stop ()
    {
           return prog.Stop () && da.Stop();
    }
    public void Transition ()
    {
           while (!Stop ())
           {
              prog.CurrentInst().SI (this);
           }
    }
}
```

5. To create a concrete Abstract Machine State (e.g., the WAM machine described in the motivation section) you may follow the next sequence of instructions:

```
    import AbstractStateMachine;
    import WAM_Factory;
    import WAM_InstructionSet;

    class WAM_Client
    {

           static public void main (String args []) throws IOException
           {
              FileInputStream targetCode = new FileInputStream (arg [0]);

              WAM_Factory factory = new WAM_Factory ();
              AbstractMachineState machine = new AbstractMachineState (factory);

              machine.Init (targetCode);
              machine.Transition ();
           }
    }
```

6. The WAM State data area is defined as an aggregation of some data components (i.e., OrStack, Heap and Registers), as it is shown in the figure 1.

```
    import OrStack;
    import Heap;
    import Registers

    public class WAM_DataArea implements AbstractDataArea
    {
           public OrStack orStack;
           public Heap heap;
           public Registers regs;
```

```
            public WAM_DataArea ()
            {
               orStack = new OrStack ();
               heap = new Heap ();
               regs = new Registers ();

            }
            public void Init ()
            {
               orStack.Init ();
               heap.Init ();
               regs.Init ();
            }

            public boolean Stop ()
            {
               return true;
            }
    }
```

7. On its side, the WAM Program is defined as an instance of Abstract Program for a concrete WAM Instruction Set. Moreover, the WAM Program extends the Abstract Program with a new program counter (CP).

```
public class WAM_Program extends AbstractProgram
{
    public int CP; //

    WAM_Program (AbstractInstructionSet instSet)
    {
            super (instSet);
    }
    public void Init ()
    {
            P = 0;
            CP = 0;
    }
    public int Consult_CP ()
    {
            return CP;
    }
    public boolean Stop ()
    {
            return CurrentInst().Name().equals ("stop");
    }
}
```

8. Finally, we describe some examples of instructions in the WAM instruction set. Let's consider how each of the following instructions implements its semantics (how the WAM machine evolves) by modifying the WAM State (i.e., the WAM DataArea and/or the WAM Program).

   a) The following code describes a Java implementation for the WAM instruction "*allocate n*". The execution of this instruction (its semantics) provokes to allocate n unbound variables in the or-stack of the WAM machine. Afterwards, the instruction moves the program counter to the next instruction (calling to *Next* on the WAM program).

```
public class Allocate implements AbstractInstruction
{
        private int numVars = 0;

        Allocate ()
        {}
        public String Name ()
        {
            return "allocate";
        }
        public int NumArguments ()
        {
            return 1;
        }
        public String ToString ()
        {
            return Name() + " " + numVars;
        }
        public void SI (AbstractMachineState state)
        {
            WAM_DataArea WAM_da = (WAM_DataArea) state.DataArea();
            WAM_Program WAM_prog = (WAM_Program) state.Program();

            // Implementation of allocate semantics
            WAM_da.orStack.Push (numVars);
            WAM_prog.Next();
        }
        public void Process (String args[] )
        {
           // Proces the text representation of the instruction
           // arguments into an int value.
           numVars = Integer.valueOf (args [0]).intValue();
        }
}
```

b) The following code describes a Java implementation for the WAM instruction *"call <addr>"*. Its execution means to copy the current value of CP into the OrStack (in the data area), to assign the counter value of CP into P, and afterwards, assign the counter P to progAddr.

```
public class Call implements AbstractInstruction
{
        private int progAddr;
        ...

        public void SI (AbstractMachineState state)
        {
            WAM_DataArea WAM_da = (WAM_DataArea) state.DataArea();
            WAM_Program WAM_prog = (WAM_Program) state.Program();

            // Implementation of call semantics
            WAM_da.orStack.Set_CP (WAM_prog.CP);
            WAM_prog.CP = P;
            WAM_prog.P = progAddr;

        }
        ...
}
```

c) Consider now, a new version of the call instruction able to support visualization facilities. We derive a new ViewCall class from the previous one and redefine the SI operation as follows:

```
public class ViewCall extends Call
{
        public void SI (AbstractMachineState state)
        {
           super.SI (state);  // Execute the Call semantics
           state.notify ();   // Notify the change and redraw
        }

}
```

Known Uses

The present pattern has been used to formalize and implement an Abstract Compiler for the Prolog language [2]. As a result of this work, it has been obtained a multi-phase compilation process based on the abstract machine technology. Furthermore, we have found the ABSTRACT-MACHINE pattern is well suited to easily include visualization facilities and debugging mechanisms and facilities in a non-intrusive way [3].

Related Patterns

The work presented in [5] explores the definition of a pattern language for building virtual machines. Our proposal, unlike [5], is more focused on providing a higher-level design framework, and not so much on describing the low-level of virtual machines.

The Abstract Machine combines some patterns presented in [4]

- It is the case of the Abstract Machine State, a variation of the MEDIATOR pattern,
- The Abstract Machine Factory and Abstract Instruction Set belong two the set of ABSTRACT FACTORY patterns,
- On the other hand, behind the *SI* operation (in the Abstract Instruction) there is a kind hidden STRATEGY pattern, and
- Finally, the *Init*, *Transition* and *Stop* operations in the Abstract Machine State are clear examples of the TEMPLATE METHOD pattern.

Bibliography

[1] Arnold & Gosling. The Java Programming Language. Addison-Wesley, Reading, Massachusetts, 1996.

[2] García J. & Moreno J.J. Visualization as Debugging : Understanding/Debugging the WAM, Automated and Algoritmic Debugging (AADEBUG'93), Lecture Notes in Computer Science (LNCS 749), Springer-Verlag, 1993.

[3] García J. & Moreno J.J. A Formal Definition of an Abstract Prolog Compiler, AMAST'93. Workshops in Computer Science, Lecture Notes in Artificial Intelligence (LNAI), Springer-Verlag, 1993.

[4] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides. *Design Patterns. Elements of reusable object-oriented software*. Addison-Wesley, Reading, MA, 1995.

[5] Jacobsen, E.E. & Nowack, P. A Pattern Language for Building Virtual Machines. *2th European* Conf. Pattern Languages of Programming, Irse (Germany) , July 1996.

[6] Reade, C, Elements of Functional Programming, Addison-Wesley, 1989.

[7] Warren, D. H.D, An Abstract Prolog Instruction Set. Tec. Note 309, SRI International, Menlo Park, California, October 1983.