Declarative Approach to Implementing Automata Classes in Imperative Programming Languages

Artyom Astafurov, Anatoly Shalyto (research supervisor),
Fac. of Information Technologies and Programming
St. Petersburg State University of Information Technologies, Mechanics and Optics
Email: Artyom.Astafurov@gmail.com, shalyto@mail.ifmo.ru

Abstract – **This paper describes the declarative approach to implementation of automata objects in imperative object-oriented programming language with static type checking. The main advantage of proposed approach is the ability to use nesting and inheritance of macro-states.**

Keywords – **Automata-based programming, declarative programming, object-oriented programming.**

I. Introduction

There are several approaches to implementing automatons in object-oriented programming languages. Those can be divided on imperative and declarative.

Among imperative approaches the most commonly used is *State* design pattern [1,2]. The main advantage of using this approach is encapsulation of specific behavior into states and implementation of the transition in explicit form in the source code. The primary disadvantage of this pattern is that the implementation of the automaton with a significant number of states will result in a highly complex class hierarchy. This problem can be solved using *Decorator* design pattern [2,3].

The primary disadvantage of imperative approach is explicit delegation of a call from context to the concrete state or delegating the call to the nested automata.

In this paper we propose the approach that acts as a tradeoff between declarative and imperative approaches. We will inherit the ability of declaratively describing the features of automata from declarative approach. The implementation of transition logic will be inherited from imperative approach as it is imperative by its nature.

Even though the described approach to implementing automata classes is declarative it is still applicable for usage in object-oriented languages with static type checking. The approach is illustrated with samples written in C# in this paper.

II. The Approach

When describing the behavior of automaton we'll be using a modified notation of *Statecharts* diagrams [4]. The main difference between *Statecharts* and *SWITCH*-technology [5] is that *SWITCH*-technology clearly defines the term *"automaton"* and *Statecharts* doesn't. The semantics of transition charts used in *SWITCH*-technology is close to the semantics of *Statecharts* but is not the same. In *SWITCH*-technology terms *"automaton"* and *"inheritance of automatons"* are introduced and there are no concepts of *"nested"* and *"orthogonal"* states. The nesting and orthogonalization are achieved using nested automatons and introducing the concept of the *"system of automatons"*. *SWITCH*-technology is good for use in documentation: there are no problems with notation when describing nested automatons as opposed to using nested orthogonal states in *Statecharts*, specifically when it comes to laying out the titles of macro-states containing orthogonal states.

In the given approach the automaton along with all of its nested states and nested automatons is treated as a set of *macro-states* as there are no specific features of the state compared to automaton. Macro-state can contain other macro-states along with their nested states. Also there will be no difference between states and automata when it comes to implementing an automata programming framework. The library designed to support the approach will define the class *State* which represents the state concept and also contains nested macro-states. Every specific State class will also implement the interface (or set of interfaces) that specifies input and output events that can be handled by that state.

Also it is worth mentioning that *UML 2 State Machine* diagram (State Chart diagram in UML) [6] also doesn't define the concept of automaton. It defines only states which can have nested states, thus addressing the problem of nested automatons.

A. Nesting of Automaton Objects

Due to elimination of *Automaton* term and introduction of *Macro-State* instead let's define nesting of automata objects in the following manner: when the state A receives an event, the event will be broadcasted to all of its nested states and then handled by the state A after that. In this paper we'll be using the same concept of broadcasting the event to inner-states and then handling it in outer state.

B. Inheritance of Automaton Objects

Let's discuss the problem of inheritance of automata objects. The approach described in this paper should allow inheriting automatons overriding the states and the transition logic. The inheritance of automata objects is based on overriding of the

states of the base automaton: the resulting automaton should override the behavior of the base automaton at least in one of its states. The resulting object can also contain new states and transitions [7].

III. Declarative Approach When Using Imperative Languages

A. Declarative Approach and Imperative Language

Despite of the imperative nature of many contemporary programming languages we still can leverage a declarative approach when using them. In order to do this we need in encapsulate all non-declarative details in a framework which will be controlled using declarative meta-information, thus allowing using declarative approach. Depending on the language and framework this can be achieved in several ways. For example, in *Java* and *Microsoft .NET* there is reflection framework that allows accessing the components of assemblies and classes in runtime. This approach has the benefits of declarative description of a class behavior and also adds the imperative behavior in the runtime.

B. Using Attributes in Declarative Approach

For more detailed analysis of implementing declarative approach using imperative languages let's use *C#* and *Microsoft .NET* as a sample language/platform. The presence of an attribute clause allows implementing a declarative approach in this language. Attributes in *C#* allow adding meta-information to class members and classes themselves. In runtime this information can be accessed by library that encapsulates non-declarative parts of the program using the reflection mechanism.

Let's discuss the approaches to execution of declarative program in imperative environment.

C. Context Bound Objects

The central concept in the interception mechanism employed by the CLR is the notion of a context associated with a target object. A context is a set of properties or usage rules that define an environment. One or more target objects may share the same context. The rules defined in a context are enforced by the CLR at runtime when the objects are entering or leaving the context. A context is an object of type *System.Runtime.Remoting.Contexts.Context.* Objects that reside in a context and are bound to the context rules are called context-bound objects. Objects that are not context-bound are called agile objects. The context associated with the current thread of execution is obtained using the property *System.Threading.Thread.CurrentContext*. While executing within a method or accessing a field or property of a context-bound object, the CLR will ensure that this thread property always returns a reference to the same context object that the context-bound object was originally bound to thus guaranteeing that the latter executes in the same environment every time. In order to make an object context-bound, its type must be derived from *System.ContextBoundObject* which in turn derives from *System.MarshalByRefObject*. An important distinction between a context-bound and an agile object is that a client can never obtain a direct reference to an instance of a context-bound object.

When the new operator is used to instantiate a context-bound object, the CLR intervenes and executes a series of steps which together constitute the object activation phase. During this phase, the CLR generates two proxies, namely, an instance of *TransparentProxy* and an instance of *RealProxy*. Both these types are defined in the *System.Runtime.Remoting.Proxies* namespace.

The *TransparentProxy* is merely a stand-in for the target object and it is this proxy that a client always gets a reference to whenever it tries to communicate with a context-bound object. The injection of these proxies thus enables the CLR to always intercept any type of access to the context-bound object. In addition to these proxies, the CLR also provides an extensible mechanism for inserting message sinks between the proxies and the target object. A message sink is an object that implements the type *System.Runtime.Remoting.Messaging.IMessageSink*.

After the object activation phase is completed, the objects that appear in the invocation chain between the client and the target object may be visualized. The method invocation by the client is delegated by the *TransparentProxy* to the *RealProxy* which in turn sends it through a chain of *IMessageSink* instances before it reaches the target object. These message sinks provide us an entry point in order to inject the desired aspect at runtime.

This can be used for intercepting the messages and also changing them to modify the behavior of the object without changing its code.

We should also note that *Context Bound Objects* should be used when performance is not critical. The code in this approach is easy to debug so it can be used on prototyping stages when performance is not critical and debugging is essential.

D. Assembly Instrumentation

One key aspect of many modern programming languages is that they are compiled into a portable intermediate form and executed by a language runtime. Typically, the language runtime allows code to be loaded at runtime from a binary source (e.g. from a file or from the network) and executed. Two well known examples are the Java platform, which supports dynamic class loading, and the .NET framework which allows assemblies to be dynamically loaded and executed. One interesting side effect of having dynamic code loading is that before actually loading the code into a virtual machine, it is possible to instrument the code, introducing or removing specific instructions, changing the use of classes, variables and constants. The key idea is that it is possible to alter the code, performing some modifications, before the code is actually executed. These transformations are either performed after compile time, or at load time. This approach can be quite powerful. For instance, it is possible to instrument the code so that proper resource control takes place, change the code so that it is possible to serialize and relocate executing threads in a cluster, perform program consistency checks according to security policies, redirect method calls to proxies, among others.

For making things clearer, let's consider an example. Suppose that you have downloaded an application from the Internet but you cannot really consider it trustworthy. It would be desirable to be able to know all the files that it is reading and writing from disk, so that one could be sure that it is not really stealing confidential information and sending it to a foo. One possible approach is to try to use a disassembler and understand the code structure. But, as it is easy to comprehend, that is not really feasible except for trivial applications. Nevertheless, using code instrumentation, it is simple to replace all the references to classes that perform I/O with corresponding proxies that implement the same interface. Those proxies can log all the calls that are made before allowing the original invocations to take place. This way, the user can examine the log and determine which files have been accessed.

This approach can be used in performance critical solutions as all additional calls are performed inline, without queue management overhead, as it is done in Context Bound Objects approach. The main disadvantage of this approach is that it is difficult to debug instrumented code: the byte-code is modified and doesn't match the debug information generated by compiler.

IV. .NET Implementation

A. *DOME Library Description*

In this paper in order to illustrate the declarative approach in imperative languages we have developed a class library *DOME* (*Declarative Object Machines Extension*) that supports *.NET CLR*-compliant languages, for example *C#* and *Visual Basic .NET*. Fig. 1 illustrates the core class diagram for this library.

On this figure *State* – is the base class for all states. *State.Container* property is the reference to the instance of *State* class that represents the macro-state containing this state. This property is used to access outer macro-state to support nesting. *State.CurrentState* property is the current state of the macro-state. This property is used for nesting purposes and also can be referenced in the code of automata object. *SetState(Type state)* method changes the current state. This method accepts *Type* (class) as a parameter; this parameter represents the state we should transition to. Please note, that each state is represented by a separate class. *StateAttribute* is the attribute that can be applied to the state class to describe its inner states. Type – *CLR*-type (class) representing a single state. *Name* – the name of the state. *StateAttribute* constructor accepts type as a parameter, this type represents the state, and the overloaded constructor provides the mechanism for overriding states in order to support inheritance of automatons. In order to do this, Type parameter should correspond to the original state we are going to override and Overrides parameter should represent the new state that will override the original state. On case of overriding the state, the runtime engine will check that the original state supplied in Type parameter is present in the automata.

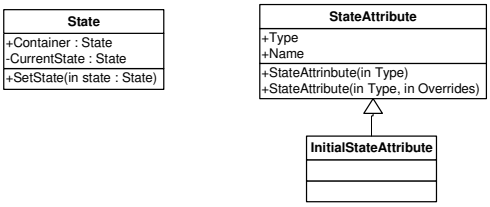

Fig 1. *DOME* Framework class diagram

InitialAttribute has the same semantics as *StateAttribute* and used to define the initial state of the automata or initial state of the inner state.

B. *Implementing Automata Objects*

As the first example, let's take a simple two-state automaton with *ON* and *OFF* states (fig. 2)

Let's define three interfaces: *IOn* – interface for state *ON* containing only one method *E0* which corresponds to the only event the automaton can accept in this state, *IOff* – interface for sate *OFF* containing method *E1* that corresponds to *E1* event, ISwitch – interface describing the entire automaton, implementing *IOn* and *IOff* interfaces.

The interfaces are declared in the following way:

```
public interface IOn
{
 void E0();
}

public interface IOff
{
 void E1();
}

public interface ISwitch : IOn,IOff
{

}
```

After that we can implement the classes that represent actual ON and OFF states:

```
class On : State, IOn
{
 public void E1()
 {
  Container.SetState(typeof(Off));
 }
}
class Off : State, IOff
{
 public void E1()
 {
  Container.SetState(typeof(On));
 }
}
```

Let's discuss the code sample above. The *Container* property references the automaton (or meta-state) this state belongs to. *SetState(Type)* changes the current state of the automaton.

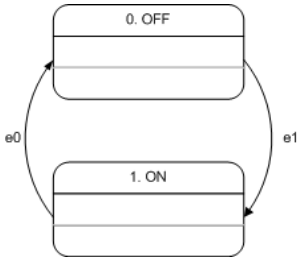

Fig. 2. Simple switch automaton

After that we should implement the automaton itself which, as stated earlier, can be treated as a set of states contained in a macro-state and thus represented by the same *State* class. In order to comply with the events the automaton can receive, it should implement *ISwitch* interface described above.

```
[State(typeof(On)), State(typeof(Off))]
class Switch : State, ISwitch
{
 public void E0() { }
 public void E1() { }
}
```

Let's take a on the attributes of the Switch class. These attributes indicate that the automaton (or macro-state) described in this class contains two states *ON* and *OFF* described by *On* and *Off* classes respectively.
In runtime all the *Switch* object method calls will be intercepted and delegated to the method of the automaton's current state that has the same signature as the automaton's method being called. After that, the original automaton's method will be called. If current state doesn't have a method with the same signature, the call will be delegated back to the automaton which means the current state doesn't support the event received.

C. *Implementing Nested States*

In order to illustrate nesting process let's take a more complex example (fig. 3).
Let's take a state diagram of the Fighter computer game character (*Fighter*) with a detailed "*Jump*" state.
Let's see how nesting states are implemented in the source code:

```
[InitialState(typeof(Jumping.Rising)),
State(typeof(Jumping.Falling)),
State(typeof(Jumping.Hovering)),
State(typeof(Jumping.Finished))]
public class Jumping : State, IFighter
{
 public void Tick()
 {
  if (CurrentState is Finished)
   Container.SetState(
   typeof(Fighter.Main));
 }

 public void ButtonPressed(Keys key)
 {}
 public bool InAir()
 {
  return true;
 }

 public class Rising : State, ITickable
 {
  public void Tick()
  {
   Console.WriteLine("rising");
Container.SetState(typeof(Hovering));
  }
 }

 public class Hovering : State, ITickable
 {
 public virtual void Tick()
 {
  Console.WriteLine("hovering");
  Container.SetState(
  typeof(Falling));
 }
}

public class Falling : State, ITickable
{
 public void Tick()
 {
  Console.WriteLine("falling");
  Container.SetState(
  typeof(Finished));
 }
}

public class Finished : State, ITickable
 {
  public void Tick()
  {
  }
 }
}
```

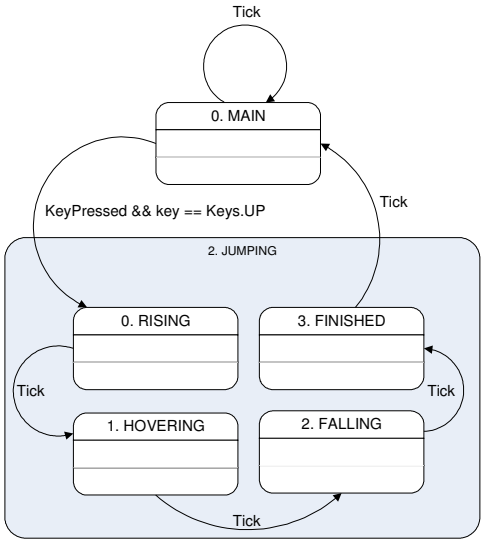

Fig. 3. State chart of the fighter character detailing the JUMPING state

Let's take a look on the nested states of *Jumping* state. In this case when one of the inner states (*Raising*, *Hovering*, *Falling*, *Finished*) is triggered, the event sent to *Jumping* state will be first passed to the current inner state and then handled by the corresponding method of *Jumping* class. Like in the sample above, the call is delegated using message interception mechanism of *Context Bound Object* described in the corresponding section of this paper.

D. Implementing Inheritance

Using the declarative approach it is quite easy and intuitive to describe the inheritance of automatons. In order to do this we just need to implement the class inherited from the automaton base class. Also when state is overridden it needs to be described using *State* attribute. Thanks to terminology, base states can be inherited the same way automatons are: one just needs to override some of its inner states and/or add new.

As an example, let's take the problem of overriding one of the states in *Fighter* automaton from a previous example. Assume we need to override Hovering inner-state of *Jumping* state. In order to do this we need to create a derived class *EasternFighter* inherited from a *Fighter* class. Using attributes we need to describe the overriding of *Fighter.Jumping* state. So let's assume we will be overriding this state with *Fighter.EasternJumping* state which will contain an overridden *Hovering* state.

In *C#* by means of attributes this can be expressed in the following manner:

```
[State(typeof(EasternJumping),
typeof(Fighter.Jumping))]
public class EasternFighter : Fighter
{
}
```

Now we need to create a derived class of the class-state *Jumping* in order to override it's inner state Hovering with a new *EasterHovering* one:

```
[State(typeof(EasternHovering),
typeof(Fighter.Jumping.Hovering))]
public class EasternJumping :
Fighter.Jumping
{
}
```

Eventually we need to implement the state *EasternHovering* itself creating a class derived from *Jumping.Hovering*:

```
public class EasternHovering :
Fighter.Jumping.Hovering
{
 public override void Tick()
 {
  Console.WriteLine("Eastern Hovering");
  Container.SetState(
  typeof(Fighter.Jumping.Falling));
 }
}
```

This way when the new automaton is executed, when the transition to *Jumping* state is made the new *EasternJuping* class will be used which in turn will be using an overloaded *EasternHovering* state which will be logged into console.

V. Conclusions

This paper presents a declarative approach to implementation of automaton objects. During the research process the ways to implement this approach using imperative methods have been defined and supporting *DOME* (*Declarative Object Machines Extension*) framework has been implemented. This framework allows to fully implement the approach described using Microsoft .NET platform. The usage of the library is illustrated with a series of samples of nesting and inheritance. DOME framework has been published on http://is.ifmo.ru.

The approach presented in this paper allows combining declarative and imperative programming paradigms providing a framework to use declarative object-oriented approach in implementing of automatons using imperative languages. Due to the fact that the concept has been described on the level of declarative programming paradigm without referencing to any concrete programming language this approach can be applied in other programming languages like Java and others.

Further research will cover the creation of *DOME*-like library for other languages such as Java and will also solve the issue of overriding virtual and non-virtual states.

VI. References

[1] E. Gamma et al, (1995). *Design Patterns*. Boston: Addison-Wesley.

[2] P. Adamczyk "The Anthology of the Finite State Machine Design Patterns", *The 10th Conference on Pattern Languages of Programs*, 2003.

[3] J. Odrowski, P. Sogaard "Pattern Integration – Variations of State", *Proceedings of PLoP96*, 1996.

[4] D. Harel, "Statecharts: A visual formalism for complex systems", *Sci.Comput. Program. 1987. Vol.8, pp. 231–274.*

[5] Shalyto (1998). *SWITCH-technology. Algorithmization and programming the logical control problems.* St.Petersburg: Nauka. Available: http://is.ifmo.ru/books/switch/1

[6] D. Shopyrin, "Methods for object oriented design and development of the software for reactive systems", Ph.D. thesis, Department of Computer Technology, St.Petersburg State University of Information Technologies Mechanics and optics, St.Petersburg, Russia, 2005. Available: http://is.ifmo.ru/disser/shopyrin_disser.pdf

[7] Booch, G., Rumbaugh, J., & Jacobson, I. (2000). *Complete Uml Training Course, the*. Upper Saddle River: Prentice Hall PTR.

[8] D. Box, C. Sells (2002). *Essential .NET Vol. 1, The Common Language Runtime.* Boston: Addison-Wesley.