Published in Proceedings of "Software Engineering Conference (Russia) – 2006" (SEC (R) 2006). Russia, Moscow, 2006, pp. 20–23.

A GRAPHICAL NOTION OF STATE-BASED CLASSES' INHERITANCE

Danil Shopyrin
Transas Technologies
Saint-Petersburg State University of Information Technologies, Mechanics and Optics
email: danil.shopyrin@gmail.com

Anatoly Shalyto
Saint-Petersburg State University of Information Technologies, Mechanics and Optics
email: shalyto@mail.ifmo.ru

Abstract

The paper proposes a novel graphical notion of state-based classes' inheritance. The proposed graphical notion allows generalization, decomposition, structurization and incremental extension of state-based classes' behaviour using inheritance.

***Keywords**: Finite state machines, object-oriented design, inheritance.*

1. Introduction

The most widely known finite state machines based graphical notion is the Statecharts visual formalism [1]. A lot of the modern finite state machines based graphical notions are founded on it. Statecharts formalism uses the extended model of traditional finite state machines [2]. The traditional model is enriched by hierarchy that is introduced using nested states (corresponds to XOR logical operation) and parallelism that is introduced using orthogonal states (corresponds to AND logical operation).

One of the main shortcomings of the Statecharts and subsequent notions is the fact that behaviour diagrams are too much cumbersome to be used in real size projects. This shortcoming is (partially) eliminated in the *SWITCH*-technology [3]. Behaviour diagrams that are used in the *SWITCH*-technology are much more compact because they are used together with communication schemas.

Finite state machines are also frequently used in object-oriented systems where they become to be state-based classes. The behaviour of a state-based class depends on its explicitly dedicated current control state and is implemented using finite state machines. The *State Design Pattern* is the most widely known approach to design and implement state-based classes [4]. There are a lot of *State Design Pattern* extensions and modifications browsed in [5].

1.1. Inheritance of state-based classes

Inheritance is one of the main paradigms of object-oriented programming. Inheritance allows derived class to receive properties and characteristics of the base class, normally as a result of some special relationship between the base and the derived [6]. Only new properties of derived class are declared. Properties of the base class get to the derived class incrementally (i.e. automatically).

Inheritance can also be used in state-based classes design and implementation [7–9]. But the particular problem of the state-based classes' inheritance visualization isn't widely discussed in the literature. The new graphical notion of state-based classes' inheritance is presented in this paper. The proposed graphical notion allows generalization, decomposition, structurization and incremental extension of state-based classes' behaviour using inheritance. Note that some questions of inheritance semantics are omitted. The main goal of the paper is to show how the state-based classes' inheritance can be graphically viewed.

2. Formal definition of state-based classes

Formally, a state-based class A can be defined by triple $<I, S, J>$, where:

- I – is a set of state-based class's interface methods;
- S – is a set of state-based class's control states;
- J – is a set of transitions between control states.

There is a function $beg(S) \in S$ defined on the set of control states that returns the initial state. For each control state $s \in S$ following functions are defined:

- $dex(s)$ – is an action that is done at exit from the state s;
- $den(s)$ – is an action that is done at enter to the state s;
- $dact(s)$ – is an activity that is done at the state s.

A transition $j \in J$ can be defined by quintuple $<from, to, ev, cond, do>$, where:

- $from(j) \in S$ – is an origin state of the transition;
- $to(j) \in S$ – is a target state of the transition;
- $ev(j) \in I$ – is a causal call of state-based class interface method;
- $cond(j) \in \{true, false\}$ – is a condition that must be *true* to allow the transition;
- $do(j)$ – is an action that must be performed when the transition is happened.

The transition $j_0 \in J$ is happened if and only if all the following conditions are satisfied:

- current control state of the state-based class is $from(j_0)$;
- method $ev(j_0)$ of the state-based class's interface is called;
- condition $cond(j_0)$ is satisfied.

In this case, the following sequence of doings is done:

- action $dex(from(j_0))$ is performed;
- action $do(j_0)$ is performed;
- current control state is set to $to(j_0)$;
- action $den(to(j_0))$ is performed.

2.1. Formal definition of state-based classes inheritance

Let's consider inheritance of state-based classes. All states and transitions of the base class implicitly get to a derived class. Furthermore, derived class can extend and modify the behaviour of the base class. Modification of base class behaviour is founded on states overriding. Some states from the base class can be marked as overridden. Transitions from the overridden state to other states can be somehow modified. For example, the target state of the transition can be changed. Derived class can also extend logic of the base class by adding new states and transitions.

If the state-based class D is a descendant of the state-based class B, then following conditions are satisfied:

- interface I_b of the base class B is the subset of the interface I_d of the derived class D, $I_b \subseteq I_d$;
- states set S_b of the base class B is the subset of the states set S_d of the derived class D, $S_b \subseteq S_d$;
- initial states of base and derived classes are equal, $beg(S_b) \equiv beg(S_d)$;
- for every transition $j_b \in J_b$ of the base class B there is a transition $j_d \in J_d$ of the derived class D, such as $from(j_d) \equiv from(j_b)$, $ev(j_d) \equiv ev(j_b)$ and $cond(j_d) \equiv cond(j_b)$.

If transition j_b of the base class B is overridden by transition j_d of the derived class D, then following conditions are satisfied:

- origin states of transitions are the same, $from(j_d) \equiv from(j_b)$;
- casual calls are the same, $ev(j_d) \equiv ev(j_b)$;
- conditions are the same, $cond(j_d) \equiv cond(j_b)$;
- target states or actions are different, $to(j_d) \neq to(j_b)$ or $do(j_d) \neq do(j_b)$.

2.2. Formal definition of state groups

Structuring of state-based logic is done using state groups [1]. State groups unite states in which the state-based class has some similar behaviour. State groups can be nested in each other.

State groups can have group transitions also called *beams*. *Beams* are similar to transitions but initial state isn't specified for them. A beam $b \in B$ can be defined by quadruple $<to, ev, cond, do>$. There is a function $beams(s) \subseteq B$ defined for each state $s \in S$. The set $beams(s)$ is conform to the set of transitions originated in the state s.

State group $g \in G$ can be defined by triple $<gbeams, msub, gsub>$, where:

- $gbeams(g) \subseteq B$ is the subset of beams corresponded to the state group g;
- $msub(g) \subseteq S$ is the subset of states included in the state group g;
- $gsub(g) \subset G$ is the subset of state groups nested in the state group g.

For each state group $g \in G$ following statements are true:

- $\forall s \in msub(g), gbeams(g) \subseteq beams(s)$ – for each state s that is included in state group g, the beams set $gbeams(g)$ is the subset of the $beams(s)$;
- $\forall g_0 \in gsub(g), gbeams(g) \subseteq gbeams(g_0)$ – for each state group g_0 that is a subgroup of the state group g, the beams set $gbeams(g)$ is the subset of the $gbeams(g_0)$;
- $\forall g_0 \in gsub(g), \forall s \in msub(g_0), s \in msub(g)$ – if state s is included in state group g_0 and g_0 is a subgroup of state group g, then state s is also included in state group g;
- $\forall g_0, g_1 \in G,$ if $g_0 \in gsub(g1)$ and $g \in gsub(g_0)$, then $g \in gsub(g1)$ – if state group g_0 is a subgroup of the state group g_1 and state group g is a subgroup of the state group g_0, then state group g is also a subgroup of state group g_1.

3. Graphical notion of inheritance

The proposed graphical notion is an extended version of *SWITCH*-technology behaviour diagrams. Main elements of the proposed notation are shown on the fig. 1.

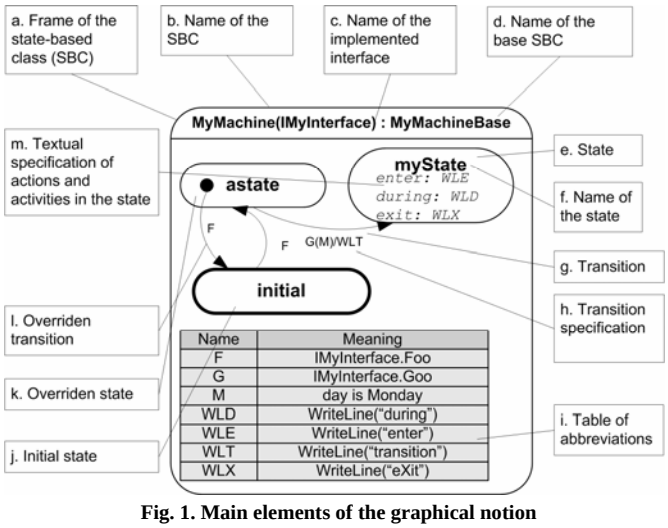

Fig. 1. Main elements of the graphical notion

The matter of the proposed graphical inheritance notation is as follows. Base class is declared in the title of the derived class (see fig. 2 for details). Inheritance of state-based classes is founded on overriding of states of the base class. Overridden states of the base class are marked by bold point. Some transitions from overridden state to other states can be

overridden in derived class. Overridden transitions are originated in the mentioned bold point. Derived class can also contain new states and transitions that aren't presented in the base class. There is a similar overriding syntax for state groups and beams.

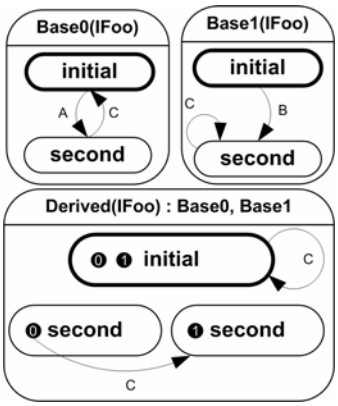

Fig. 2. Graphical notion of multiple inheritance

4. An illustrative example

As an illustrative example of the proposed graphical notation usage let's consider classes' family that provides access to a file:

- `ReadFile` that provides access only for reading;
- `WriteFile` that provides access only for writing;
- `ReadWriteFile` that provides access for reading, writing and reading/writing.

Mentioned classes have state-based nature with states `closed`, `opened` etc. Behaviour diagrams of these classes are shown on the fig. 3.

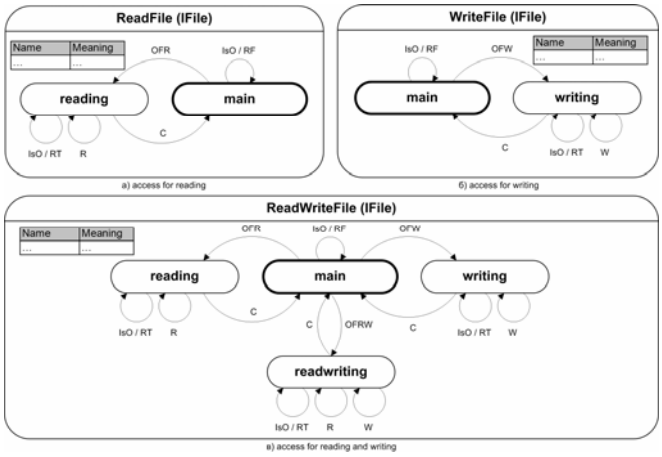

Fig. 3. Logic of file access classes (without inheritance)

The behaviour of these classes can be generalized and structured using inheritance. The root element of the appearing hierarchy is the abstract class that generalizes access to a file. Behaviour diagrams of file access classes that are built using inheritance are shown on the fig. 4.

Further, let's consider the behaviour diagram of the state-based class `AppendFile` that is a descendant of the state-based class `ReadWriteFile` (fig. 5). The state-based class `AppendFile` adds new control state `appending` that allows to append data to a file. Note that `AppendFile` class is built incrementally, i.e. without any changes in the base class. The behaviour diagram from the fig. 5 is equal to the diagram on the fig. 6 that is built without inheritance. Note how duplication can be dramatically reduced by usage of inheritance.

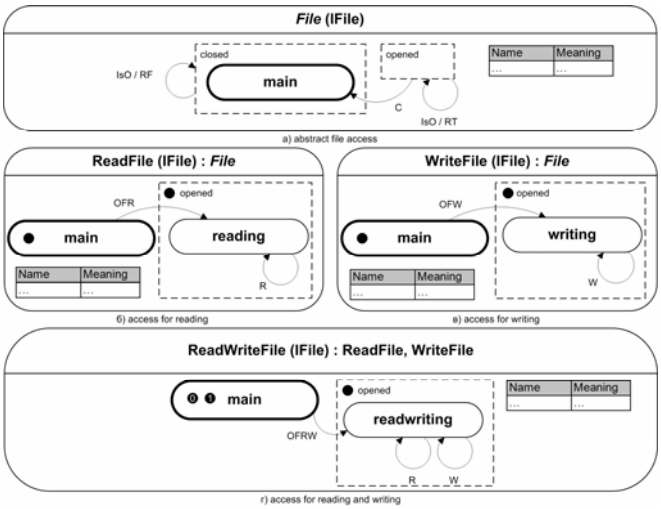

Fig. 4. Logic of file access classes (with inheritance)

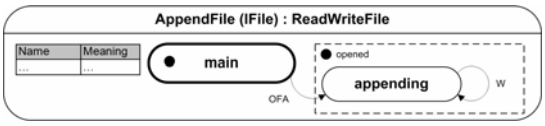

Fig. 5. Logic of `AppendFile` class (with inheritance)

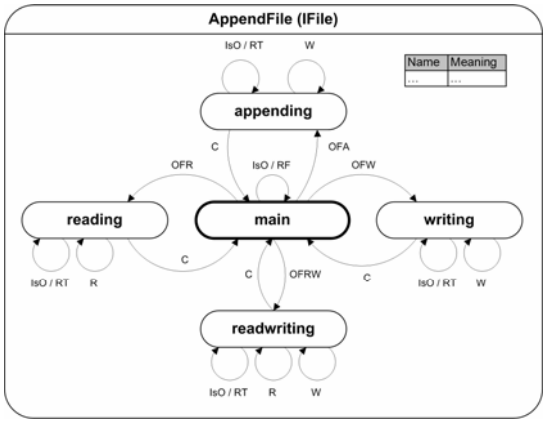

Fig. 6. Logic of `AppendFile` class (without inheritance)

Some quantitative adjectives are given in the table below. The count of used states, state groups and transitions is computed for state diagrams with and without usage of inheritance. Note that count of transitions is significantly reduced when state-based classes' inheritance is used (table).

Table. Inheritance effectiveness

Count	Without inheritance	With inheritance
States	13	5
Overridden states	–	4
State groups	–	2
Overridden state groups	–	4
Transitions	42	12
Total	55	27

Conclusion

The proposed graphical notation is quite simple but powerful. It allows to present inheritance of state-based classes in incremental, intuitive, easy to understand manner. In many cases it allows to significantly reduce the duplication.

One of the shortcomings of the proposed graphical notation is that additional primitives such as overridden states and transitions are introduced. Another shortcoming is that state-based classes' hierarchies appear not so often.

It is significant that two methods of state-based classes' implementation that are isomorphous to the described graphical notation are proposed:

- on the base of virtual methods [10];
- on the base of virtual inner classes [11].

Both of proposed implementation methods conforms to the main principles of object-oriented programming and can be used within different modern object-oriented languages. It increases the practical value of the proposed graphical notion.

References

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

[2] *Automata Studies* / Shannon C.E., McCarthy J. Princeton University Press, 1956. – 296 pp.

[3] *Shalyto A. SWITCH*-technology. Algorithmization and programming of logic control problems. SPb.: Science, – 1998. – 628 pp. (in Russian). http://is.ifmo.ru/books/switch/1

[4] *Gamma E., Helm R., Johnson R., Vlissides J.* Design Patterns – Elements of Reusable Object-Oriented Software. Addison–Wesley. 1995. – 395 pp.

[5] *Adamczyk P.* The Anthology of the Finite State Machine Design Patterns / Proceedings of the Pattern Languages of Programming conference (PLoP), 2003.

[6] *Danforth S., Tomlinson C.* Type theories and object-oriented programming //ACM Comput. Surv. Vol. 20, № 1. 1988, pp. 29–72.

[7] *Sane A., Campbell R.* Object-Oriented State Machines: Subclassing, Composition, Delegation, and Genericity /Proceedings of OOPSLA '95. 1995, pp. 17–32.

[8] *Lee J., Xue N., Kuei T.* A note on state modeling through inheritance // SIGSOFT Softw. Eng. Notes. Vol. 23. 1998. № 1, pp. 104–110.

[9] *Harel D., Kupferman O.* On Object Systems and Behavioral Inheritance // IEEE Trans. Softw. Eng. Vol. 28. 2002. № 9, pp. 889–903.

[10] *Shopyrin, D.* Object-oriented implementation of finite state machines on the base of virtual methods //Information-control systems. 2005. Vol. 3, pp. 36–40.

[11] *Shopyrin, D.* Method of state machines design and implementation on the base of virtual inner classes //Information technologies of modeling and control. 2005, Vol. 1(19), pp. 87–97. (In Russian). http://is.ifmo.ru/works/ruvstate/