

Dynamic Game Object Component System for Mutable Behavior Characters

José Ricardo S. Junior Erick B. Passos Esteban W. Clua Bruno C. Moreira Pedro C. Mourão

Instituto de Computação
Universidade Federal Fluminense

Abstract

Most games today use some form of Game Object Component System to compose game entities. With this approach, components represent anything such as functionalities or just a collection of attributes, and are attached to game objects in order to properly compose it. In this paper we propose an augmented Game Object Component System with automatic activation/deactivation of components based on runtime evaluation of logical conditions. Using this approach, it is possible to compose entities with mutable behavior based on such dynamically activated components. We propose this architecture as an alternative to Finite State Machines to model NPC behavior. This dynamic mechanism decouples the implementation of the behavior itself from its activation and deactivation, providing for easy reuse of such components in different game types by only modifying the activation rules.

Keywords: Dynamic component system, mutable behavior characters, engine architecture, finite state machine.

Authors' contact:

{josericardo.jr,erickpassos}@gmail.com
pedrothiago@hotmail.com
brucostam2004@yahoo.com.br
esteban@ic.uff.br

1. Introduction

In game development, several techniques can be used to implement the behavior of game objects that are not controlled by the player. One example of these techniques are Finite State Machines (FSM), where a finite number of states acts as a memory of what happened in the past to make possible future decisions according to distinct events [WAGNER et al. 2006].

Using this approach, the game objects' behavior is specific for each state and transition rules have to be manually coded so that state switches happen throughout the simulation, providing for the mutable behavior.

However, to design a believable NPC behavior, many states have to be created. In this case, the management and maintenance of such architecture is a complex task, which is only advisable in cases where all the states are previously known [BAILLIE 2004].

In this paper we propose the extension of a Game Object Component System [BILLAS 2002; FOLMER 2007; STOY 2006] to develop Finite State Machines. In this new architecture, behavior components associated with a game object can be activated and deactivated through the dynamic verification of logical conditions. These logical conditions are composed with any number of game object attributes that are evaluated at runtime. Some novel features provided by this system regarding the design of game entities behavior are:

- Automatic dynamic mechanism for the activation of components (behaviors) through the use of logical conditions composed with runtime evaluation of game object attributes;
- Complete decoupling of components activation and implementation, providing for easier maintainability and reuse;
- Possibility to activate multiple/concurrent behaviors at the same time;
- Insertion and removal of new behaviors without increasing the code complexity of the others;

The rest of the paper is organized as follows: section 2 discusses related work, section 3 presents the concepts and design of the game object component system, and section 4 explains the dynamic mechanism for activation and deactivation of behaviors. Section 5 brings a case study of a Finite State Machine designed for a simple RPG NPC. Finally, section 6 concludes the paper and outlines future work.

2. Related Work

As the necessity for more realistic behavior of non player characters (NPC) in games increased, so did the number of states used in finite state machines used for this purpose. To tackle this issue, some architectures can be used to better manage and organize these states. One of them is called Hierarchical Finite State Machine (HFSM) [GIRAULT et al. 1999] where a set of states could be grouped together inside a super-state with some small states inside it. Using this approach, called generalized transitions, redundant transitions could be prevented. However, reusing transitions is not a trivial task and requires a lot of reasoning about the logic of several different contexts.

A technique called Object Oriented Hierarchical State Machine (OOHSM) [MALLOUK and CLUA, 2006]

was developed to better organize finite state machine. In this work, the author proposes the use of an abstract class which every concrete state machine needs to inherit from.

Our work decouples behaviors activation from the representation of the state machine itself, using object-oriented composition for the later and independent activation rules for the first. In the next section we explain our framework architecture and the dynamic activation mechanism for behavior components.

3. Game Objects Architecture

This section presents *GCore* [PASSOS et al. 2008], a game object component game framework implemented in Java, based on JMonkeyEngine [JMonkeyEngine], that has been developed as a platform for our research in game development and game software engineering.

3.1. The Game Object Class

In the proposed architecture, the *GameObject* class has attributes that are common for all objects in a game. Additionally, an object must have a set of components that are attached to it add new functionalities to it.

To enable communication between objects, there is a class responsible to manage all *GameObject* instances created in the game, where each object has its own unique identifier during game execution.

The types of game objects that are available in the game are stored externally in XML files. An example of how such definition could be is shows in code 1.

```
<types>
  <type name="basic">
    <component class="VisualComponent" />
  </type>

  <type name="npc-type" extends="basic">
    <component class="AIComponent" />
  </type>

  <type name="tree-type" extends="basic">
    <component class="VisualComponent">
      <model value="tree.3ds" />
    </component>
  </type>
</types>
```

Code 1: XML game type definition

In this example, a "BASIC-TYPE" is defined, specifying that all its instances will have a *VisualComponent* attached to them. This elementary type is referenced twice to illustrate two possible extension mechanisms: a) create new types by adding more components to the supertype, as shown in the example by the "NPC-TYPE", and b) specializing a type by modifying attributes of existing components.

3.2. The Base Component Class

In this architecture, the *BaseComponent* abstract class is the prototype of the behaviors where the logic of entities is defined. To make new components/behaviors, this class has to be extended, overriding the update method, according to the necessary functionalities. This class has a very important boolean attribute (enabled) to tell the *GCore* framework if each component instance is currently active. One should disable a component to avoid its update in the moment of the container game object's update. The *Template Method* design pattern [GAMMA et al. 1995] was used to implement this dynamic form of update.

Since some components depend on others to be executed properly, one should have a way to guarantee that this dependency will be satisfied in the game objects at runtime. To effectively tackle this issue, a technique called Dependency Injection [PASSOS et al. 2008] is used in our framework.

4. Dynamic Component Activation

Several benefits are obtained with rule-based dynamically activated game objects components. For instance, consider a game in which there is a game object type represents a student. Also, suppose this student could dynamically gain more abilities during the execution of the game. One example of such mutable behavior could be: if this student gets approved in all disciplines he becomes a doctor and gains the ability of doing medicine. Generally speaking, each new ability is represented by a specific component/behavior that should be activated to make the correspondent ability available, thus enabling the game object to practice medicine. Similarly, the deactivation of a component makes the container game object loose the correspondent ability. As long as the different behaviors are implemented independently, they can be very flexible and reusable in several games.

However, the dynamic management of behavior activations in such componentized FSM is a complex task. Doing this management manually is very difficult and error-prone, sometimes leading to unexpected results. A mechanism to do this task automatically can avoid many problems related to game prototyping and code maintenance. Our system aims to provide such mechanism, decoupling the implementation of behaviors from their activation, which is performed according rules optionally attached to each component at the game object types XML specifications.

Each component attached to a game object can have its own activation rule. This rule is a composition of logical expressions that is constantly evaluated at runtime. In case this rule evaluates to true to a specific component during game execution, this component is enabled, only for this particular instance of *GameObject*, and his update code is now executed at each game loop step. This dynamic evaluation is performed inside the *BaseComponent:Evaluate* (Template Method), which guarantees that the

overridden (user-defined) *Update* will be called only after the rule evaluates to true. This *Evaluate* method cannot be overridden and it is the one the *GameObject:Update* method actually calls.

The optional automatic activation condition for a component is defined as an expression composed of a binary logical operator and dynamically evaluated operands. Operands of an expression can be:

- Static values;
- A component attribute;
- Other expression.

Some of the available logical operators are:

- And (&);
- Or (|);
- Equals (=);
- Different (!=);
- Lower (<);
- Greater (>);

As will be shown in the next sections, expressions can be as complex as needed, and to enable their dynamic evaluation, component attributes values need to be obtained at runtime which is achieved by the use of Reflection. For instance, to make a component activation dynamic, based on attributes, an expression as the one shown in Code 2 will be needed. The `<activation>` tag is the container for the activation expression.

```
<type name="student-type">
  <component class="StudyInfo"/>
  <component class="DoctorBehavior">
    <activation>
      (StudyInfo.completed = true)
    </activation>
  </component>
</type>
```

Code 02: Attribute-based activation rule

In this example, the “student-type” game object type definition is composed of two components: *StudyInfo* and *DoctorBehavior*. The *StudyInfo* component does not have the `<activation>` tag, thus being always enabled. The `<activation>` tag included in the *DoctorBehavior* component holds the expression “(study-state.completed = true)”, which is parsed at load-time to a dynamic evaluator for the *Equals* logic operator

From this explanation, it becomes clear that the *DoctorBehavior* of any “student-type” instance will be activated at runtime when the value of its *StudyState* component’s attribute named *completed* evaluates to the Boolean value *true*.

It is also possible to reference attributes from components of other game objects, as well as common available ones in the scene graph structure of the GCore framework. To specify another game object’s component attribute, one has to include this game

object identifier in the operand specification. In this case, both operands can be dynamically evaluated attributes, as can be seen in Code 3.

```
<type name="a-type" extends="basic">
  <component class="Behavior"/>
</type>

<type name="b-type" extends="a-type">
  <component class="Behavior">
    <activation>
      ( (compl.attributeA != 1)
        &
          (compl.attribute2 = 473) )
    </activation>
  </component>
</type>
```

Code 03: Decoupled activation

In this example, game objects of type “a-type” will always have their Behavior components activated, while instances of “b-type” will depend on the runtime evaluation of the logical expression to do so. The code also shows that expressions have to be expressed in their complete parentized form, as illustrated by the condition composed with “&”, “!=” and “=” operators.

With the architecture shown in this section, complex NPC behaviors can be broken in several discrete components that will be automatically managed using dynamic activation and deactivation.

5. Case Study: NPC Simulation

To show how this approach can be used in a real game scenario, an exclusive prototype was developed. This prototype presents a land infested with zombies, which like to “kill” each other. The zombies that are put in the scene interact with each other through dynamic activated mutable behaviors. To achieve the desired result, a simple *ZombieState* component was created to hold common referenced attributes. This component is responsible to gather and cache all information that is needed by the dynamic behaviors such as the zombie’s health, nearest zombie (enemy) and the distance to it.

For the zombies to interact with each other, some behaviors were developed and added to them. The available behaviors and the desired activation rules for them are listed in Table 1.

To use such behavior activations with a Finite State Machine, one has to discover what are the possible states represented by them. One way of doing this is to construct a truth-table for all the conditions expressed at the activation rules, remembering to evaluate only the ones that are independent. Also notice that with the above definitions, more than one behavior could be active at the same time. Table 2 shows how even such simple example increases the complexity of an equivalent FSM, as number of states is directly dependent on how many independent activation rules exist.

Based on the complete activation rules given in Table 1, it is possible to realize that some condition

combinations lead to the activation of the same behaviors, so the actual Finite State Machine that needed to be specified would have only 5 states.

Table 1: Behavior and their activation rules

Behaviors	Activation Rule
Wander	state.enemyDistance >= 100
Pursue	state.enemyDistance >= 20 & state.enemyDistance < 100 & state.health >= 40
Shot	state.health >= 40
Evade	state.health < 40 & state.enemyDistance <100

Table 2: Truth-table of possible states

State\Condition	Distance < 100	Distance >= 20	Health >= 40
State 1	T	T	T
State 2	T	T	F
State 3	T	F	T
State 4	T	F	F
State 5	F	T	T
State 6	F	T	F
State 7	F	F	T
State 8	F	F	F

Even in this simple case, the programmer would have to implement, besides the behaviors, the FSM manager component and the transition evaluations and consequent behavior activations. Our automatic mechanism largely simplifies this task, and as more complex the FSM becomes, the more difficult the manual implementation is, eventually leading to maintainability issues.

6. Conclusion

Data driven game object composition is a proved approach to put down risks in game development. Such flexible architecture has been used in several successful engines, frameworks and games. However, the design and implementation of game entities that need mutable behavior brings some issues related to maintainability and code reuse. In this paper, we presented a game framework with a dynamic activation mechanism for components that can be used to

implement complex characters. This feature provides for the decoupling of behavior implementation and its activation, which permits behaviors to be seamlessly reused with other entity types, even when its activation rules (state machine) are different.

The *GCore* framework is a constant work in progress and we are now investigating how use different activation mechanisms for the behaviors such as scene triggers and other in-game events. We also plan to extend this to include timed behavior activation, where it will be possible to also specify for how long should last the activation. This will facilitate the design of temporary behaviors, which are useful in some situations in game development. The short-term roadmap also includes a level-editor, enabling game objects behaviors composition and activation rules editing in visual form.

References

- BAILLIE, P., 2004. *Programming Believable Characters in Computer Games*, Massachusetts: Charles River Media, Inc.
- BILLAS, S., 2002. A data-driven game object system. Talk at the Game Developers Conference '02.
- FOLMER, E. 2007. Component based game development – a solution to escalating costs and expanding deadlines? *In Component Based Software Engineering*, Springer, vol. 4608 of Lecture Notes in Computer Science, 66-73.
- GAMMA, E., HELM, R., JOHNSON, R., AND VLISSIDES, J. 1995. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Professional.
- GIRAULT, A., BILUNG, L., LEE, E., 1999. Hierarchical finite state machines with multiple concurrency models. *In: IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 06 August 1999 Berkeley, California: IEEE Press, 742-760.
- JMONKEYENGINE. Jmonkey engine 1.0 online documentation. <http://www.jmonkeyengine.com/>.
- MALLOUK, W., CLUA, E., 2006. An Object-Oriented Approach for Hierarchical State Machines. *In: Proceedings of the SBGames conference in Computing*, 8-10 November 2006 Recife, Brazil.
- PASSOS, E., SILVA, J., CLUA, E., 2008. Fast and Safe Prototyping of Game Objects with Dependency Injection. Tech Report. Universidade Federal Fluminense -- UFF. 2008. <http://www.ic.uff.br/PosGraduacao/RelTecnico/385.pdf>
- STOY, C., 2006. Game object component system. In: *Game Programming Gems 6*. Charles River Media, M. Dickheiser, Ed., 393-403.
- Wagner, F., Schmuki, R., Wagner, T., Wolstenholme, P., 2006. *Modeling software with Finite State Machine: A practical approach*, United States: Taylor & Francis Group.