Pattern-based AI Scripting using *ScriptEase*

Matthew McNaughton, James Redford, Jonathan Schaeffer and Duane Szafron

Department of Computing Science, University of Alberta,
Edmonton, Alberta, Canada T6G 2E8
{mcnaught,redford,jonathan,duane}@cs.ualberta.ca

Abstract. Creating realistic artificially-intelligent characters is seen as one of the major challenges of the commercial games industry. Historically, character behavior has been specified using simple finite state machines and, more recently, by AI scripting languages. These languages are relatively "simple", in part because the language has to serve three user communities: game designers, game programmers, and consumers – each with different levels of programming experience. The scripting often becomes unwieldy, given that potentially hundreds (thousands) of characters need to be defined, the characters need non-trivial behaviors, and the characters have to interface with the plot constraints. In this paper, the ScriptEase model for AI scripting is presented. The model is pattern-template based, allowing designers to quickly build complex behaviors without doing explicit programming. This paper describes ScriptEase's behavior patterns and user interface. This is demonstrated by generating code for BioWare's *Neverwinter Nights* game. In addition to behaviors, the model is being extended to include encounter, dialog, and plot patterns.

1 Introduction

The commercial games industry is currently worth $15 billion. In the past, better computer graphics have been the major technological sales feature of games. With faster processors, larger memories, and better graphics cards, this has reached a saturation point. The perceived need for better graphics has been replaced by the demand for a more realistic gaming experience. All the major computer games companies are making big commitments to artificial intelligence (AI). This activity has been accelerated by the recent success of AI-based games like *The Sims* and *Black and White*.

Historically, the artificial intelligence research community has ignored the commercial games industry. However, the AI challenges that this industry faces are daunting. For a number of years, John Laird has been advocating commercial games as a fruitful venue for AI research (AI's "killer application") [10].

Computer games are the ideal application for developing characters that appear to have realistic, artificially-intelligent behavior. There is already a need for it, since human game players are dissatisfied with computer characters. The characters are shallow, too easy to predict, and, all too often, exhibit artificial stupidity. This has led to the success of on-line games where players compete against other humans. The current state of the art in developing artificially intelligent characters can be described as rather primitive. The lack of sophistication is due to the lack of research effort [1] (Laird's group being a notable exception). This is changing, as more researchers recognize the value of the research problems facing the commercial games industry. Artificial intelligence allows us to create simulated environments where the human has the feeling that they are interacting in the real world. While an immediate application of this technology is games, the technology has wider applications (for example, training [8]).

As a first step, it is necessary only to create the illusion of intelligence. The state of the art has each character scripted, usually using a rule-based system or a finite state machine [11]. In both cases, behavior patterns are limited, repetitive, and non-adaptive. In contrast, human-level behavior should not be prescribed, should avoid repetition, and should adapt to changing conditions.

In designing a more ambitious, more robust system for defining behaviors, many issues must be considered:

- Knowledge management. There can be hundreds – even thousands – of characters in a game, each with a (possibly complex) combination of behaviors. All this information has to be organized to simplify maintenance issues.
- Knowledge acquisition. The system must simplify the task of defining characters and their behavior (especially important for game designers).
- Rapid prototyping. Game design is accomplished using an iterative approach. Typically one wants to quickly create the desired functionality, and then incrementally tune it to improve the quality of play.
- Simple model. The AI scripting facility will be used principally by three user communities: game designers (who, typically, have little programming experience), consumers (who want to create their own characters, but have variable programming experience), and game developers (usually programming experts). The programming model has to be simple enough to accommodate non-experts, but rich enough to allow developers to do anything that they want to do.
- Testing. Any definition of AI behavior must be easy to verify for correctness.
- Non-determinism. The system must support "intelligent" (pseudo-random) behavior selections to avoid predictability (while at the same time not hurting the testability of the system).
- Adaptive. The language must support learning – characters must adapt to their circumstances. Few commercial games do more than non-trivial types of learning.
- Rich set of behaviors. Realism demands that any AI behavior specification system must support a large and varied selection of behaviors.
- Complex behaviors. The system must support the creation of complex behaviors, either individual behaviors or a combination of simpler behaviors.
- Extensibility. The basic tool must support the addition of new behaviors and capabilities.

In effect AI scripting for a non-trivial game has all the problems of maintaining a large evolving software repository, while incurring the challenges of AI knowledge acquisition, maintenance, and usage.

Our experience with AI scripting languages comes from working with BioWare products *Baldur's Gate II* and *Neverwinter Nights*. These languages allow the game designer to define characters, and for users to create their own characters. The AI scripting has limited capabilities, and requires a lot of programming expertise to understand what is going on. As one adds more "intelligence" to the system, the scripts become unwieldy and hard to debug.

This paper introduces *ScriptEase*, a tool for defining complex behaviors. The objective is to address all of the above issues in a powerful yet easy-to-use tool. Behaviors are defined using *behavioral patterns* – taking an analogy from software engineering, these are the "design patterns" [7] of artificial intelligence behavior. This work builds on our experience with design patterns for parallel programming [5]. For example, one behavior pattern could be "to guard". The default would have a character stand guard over something and not allow any access to it without a fight. This behavior could be parameterized to, for example, allow the game designer to define how to guard (stand stationary; patrol around) or who to allow to have access to the object. To create a guard involves creating a character, assigning the guard behavior to it, and then customizing the behavior.

Our vision for a scripting language is to have it support a rich set of behavior patterns. And, as with our parallel programming tool CO_2P_3S[5], there is tool support for defining and refining these patterns. It is surprising to see that our research into parallel computing tools can be applied to something as seemingly remote as defining AI characters. The leap is not all that surprising given that the fundamental nature of both applications – defining and using patterns – is the same.

Note that there are multiple audiences for a scripting language, ranging from non-programmers to experts (this can be a serious issue in language design [4]). The former needs access to a simpler more intuitive interface to the language than the latter. Indeed, most users are not programmers, and exposing a textual programming language to them is undesirable. Hence, a visual representation – one that abstracts away textual programming – is important. Again, a CO^2P^3S-like approach seems to work here – parameterized behaviors can be defined graphically.

This paper describes the behavioral patterns in *ScriptEase*, and introduces the reader to our *behavior patterns*. Additional patterns, including those for encounters, dialogs and plot, are only briefly mentioned. We are fortunate to have access to industrial code to work with. *ScriptEase* is used to generate code for BioWare's multi-award-winning role-playing fantasy game *Neverwinter Nights*. Unfortunately, using a real application limits the expressiveness of behavior. The *Neverwinter Nights* scripting language does not support facilities for learning – something we want to see added to the language. Our hope is that our work will influence the future design of AI scripting languages.

Section 2 describes our patterns-based model. Section 3 illustrates our tool *ScriptEase* that implements the model. Section 4 discusses ongoing work, while Section 5 presents the conclusions.

2 Using Design Patterns to Design Computer Games

Consider the situation where a game designer of a fantasy role-playing game wants to include four icons (objects, or in this case, specifically shards), that when gathered together form a single larger icon called a moon-stone. Each shard is guarded, but the game designer wants the guarding done differently in each case:

1. Shard-1 is in a guarded chest. The guard should have a patrol route near the chest. However, if any "enemy" creature gets near the chest, the guard should warn the enemy and then run over to the chest and stand in front of it. If the enemy actually tries to open the chest then the guard should attack the enemy. If the "enemy" moves away from the chest without opening it, the guard should resume the patrol. Note that if a "friendly" creature approaches the chest, the guard will not react. In fact, if a friendly creature removes Shard-1 and takes it away, the guard will continue to guard the chest (not the shard).
2. Shard-2 is in a room with a single door. The guard attacks any enemy that gets close to the door. Again, the guard is guarding the door (not the shard) and the guard will continue to guard the door, even if the shard is removed from the room.
3. Shard-3 is in the possession of a creature. The guard will protect the creature (not the shard) from harm or from stealing. If an enemy comes near, the guard will shout and if the enemy tries to steal from the creature being guarded or attacks the creature, the guard will attack.
4. Shard-4 is protected by a guard who will attack any enemy who tries to possess it. Note that Shard-4 may be moved to any location by a friendly creature and the guard will follow along to attack any enemy that obtains it.

In addition, we want the guards in each of these scenarios to exhibit "natural" behaviors. For example, the designer wants the "chest" guard to have a fixed patrol path around the chest. However, this path should not be exactly identical each time around, since a real guard would have some variation. The second guard should be mostly stationary near the door. However, he should occasionally walk to one or more nearby objects. The third guard should stay close to the individual that is being guarded. The fourth guard should begin by staying near the icon. However, as time goes on without anything happening, this guard should become bored and move farther away. However, if any creature is spotted, the guard should immediately return to the shard and not wander far for a while (until the guard becomes bored again).

It would take considerable effort to program the behaviors of the four guards we have described in most computer game scripting languages. For example, we have manually programmed these four behaviors in the *Neverwinter Nights* scripting language and there are 500 lines of code (over 1,000 if you include white space and comments). However, these four behaviors have some similarities. They all have a common theme that something is being guarded. We should be able to abstract this commonality and use the abstraction to generate the game code for each of these four situations. This observation has already been made in other domains and has led to the construction of *design patterns*. A design pattern is a mechanism for encapsulating the knowledge of experienced designers into a re-usable artifact. By definition, a design pattern is a descriptive device that fosters re-use during the design phase of an activity. Although design patterns have been used in architecture [2], they have also become an important tool in software development [7].

The most common form of a design pattern is a document, such as a chapter in a pattern catalog or a Web page. This form preserves the instructional nature of patterns, as a cache of known solutions to recurring design problems. Patterns in this form are easy to distribute and readily available to designers. Patterns provide a common design lexicon, and communicate not only the structure of the design but also the reasoning behind it. This common form of design pattern is called descriptive. In the context of role-playing fantasy games, humans use high-level patterns to describe characters and behaviors. For example, the notion of "wizard" or "shop-keeper" immediately infer attributes on the character they are ascribed to.

Until very recently, design patterns have only been applied during the design phase of software development. They have not been used to generate code. There are several reasons why design patterns are not used as generative constructs that support code re-use. The most fundamental reason is that design patterns describe a set of solutions to a family of related design problems and it is difficult to generate a single body of code that adequately solves each problem in the family. No adequate mechanism exists for a developer to understand the variations in code that spans the family of solutions and to adapt this code for an application. A second important reason is that it is difficult to construct and edit generative design patterns. This limits the number of design patterns that can be made generative and results in a poor selection of patterns for the end user. Faced with a small selection of rigid generative design patterns, end-users are reluctant to use such a limited approach for real software development.

We have created a new approach to generative design patterns that solves these difficult problems and have embodied our approach in tools called CO_2P_3S (Correct Object-Oriented Pattern-based Parallel Programming System) and $MetaCO_2P_3S$ (and their newer sequential counterparts). The first tool generates code for a wide variety of patterns that exist in the domain of general programming and the specialized domain of parallel programming. The second tool supports the design and implementation of new generative design patterns. Our approach solves the adaptation problem by parameterizing each design pattern with a fixed set of parameters. The programmer provides application domain-specific values for each of these parameters before generating code.

In the context of computer game design, the use of generative design patterns has six positive effects:

1. Pattern re-use. A pattern can be identified, designed and implemented once and then can be instantiated many different times across the same game and different games to amortize its development cost
2. Pattern adaptation. A single pattern can provide a rich texture of different game experiences by varying its parameters.
3. Pattern abstraction. Game designers can discuss and design game components at a higher level of abstraction by discussing the design of new patterns and the adaptation of existing patterns to create new game situations.
4. Pattern code generation. Game designers can generate game code without knowing anything about programming.
5. Pattern prototyping. If a game designer has an idea for a novel new game construct, it can be evaluated more quickly. Instead of having a programmer code the new construct from scratch, an existing pattern can be adapted to generate code that implements a construct that is similar to the new idea, and this code can be modified by a programmer.
6. Pattern correctness. Once a pattern has been tested, the pattern instances that are generated from it will need less quality assurance time during game testing.

In the specialized domain of role-playing computer games, we have identified several kinds of generative design patterns that can be used by game designers with little or no programming experience: behavior, encounter, dialog, and plot patterns. In this paper, we will focus on behavior patterns, although we will also discuss encounter patterns.

Each specific pattern describes a set of roles. A role is a placeholder for a game object. For example, the guard behavior pattern defines two roles: the guard and the guarded. A pattern is instantiated by adapting it for a particular use in the game. For example, we will use four different instantiations of the Guard Pattern to generate the four different scenarios described earlier. To instantiate a pattern, each role is filled by an individual game object, who is said to play the role (in the movie sense, not the programming languages sense). For example, in the first scenario, a particular Orc (a monster) may be cast in the guard role and a particular chest may play the guarded role. In the third scenario, a particular fighter may play the guard and a particular wizard can be cast in the role of guard.

Each game may have a different ontology for classifying the kinds of game objects it has. In this paper, we will use a simple ontology consisting of *actors* (animate game objects that can perform actions) and *props* (inanimate game objects that can be manipulated but cannot perform actions). Props can be further sub-classified as *containers* (that can hold other props) and *simple props* (that cannot hold other props). We use the term *object* to refer to a game object that might be an actor or a prop. Each role is typed. For example, in the Guard Pattern, the guard role must be played by an actor, but the guarded role may be played by any object.

One of the roles of each behavioral pattern is special and is called the principal role of the behavioral pattern. The other roles are called supporting roles. An actor (not a prop) must always play the principal role of a behavior pattern since it prescribes some actions that the actor will take. The actor that is cast in the principal role is called the principal of the behavioral pattern. In fact the goal of a behavior pattern is to prescribe all of the potential actions of the principal. We say that the principal is bound to a behavioral pattern since an actor can only be the principal of one behavioral pattern at any one time. The actions of a behavioral pattern's principal are completely determined by the behavioral pattern it is bound to. A principal stays bound to a behavioral pattern until it is unbound. This can be done if the principal is bound to a different behavioral pattern or is destroyed. Recently, complex schemes for allowing an actor to choose a principal role amongst several behavioral patterns have been proposed in the literature [6]. However, it is not clear that they will be easy to use in cases where the action taken by the character is significant to the plot of the game.

Although an actor may be cast in only one principal role at any given time, it may be cast in an arbitrary number of supporting roles simultaneously. For example, if the principal (guard role) of a Guard Pattern that is guarding something (chest, door, individual, shard, etc.) is itself being guarded by three other creatures, then the principal plays the guarded role in three other guard instantiations.

A pattern role is a special kind of pattern parameter. However, each pattern can have a set of other parameters as well as its roles. Every behavioral pattern has a situation list parameter that describes all of the possible basic situations that comprise the behavioral pattern. Each situation consists of a set of conditions and a set of actions. For example, in scenario 1, one situation is: if an enemy comes near the chest and the guard is currently patrolling, then warn the enemy and move near the chest. A second situation is: if an enemy opens the chest then attack the enemy.

In general, patterns can also have other parameters. Two other common parameter types are labels that refer to specific game objects and composite parameters that refer to other pattern instances. For example, the Guard Pattern has a list of patrols, where each patrol is an instance of another behavior pattern called a Patrol Pattern. At instantiation time, the game designer must assign a value to each pattern parameter. Of course, casting the roles of a pattern to specific objects is a special case of assigning pattern values to those parameters that are role parameters. In the next section, we provide an example of patterns, pattern parameters, and the instantiation of pattern parameters using the Shard-1 chest guard of this section as an example.

When patterns are used as parameters in other patterns there is sometimes a need to require roles from the two different patterns to be cast by the same object. For example, the guard in the Guard Pattern and the patroller in the Patrol Pattern that is attached to it, must be cast as the same actor. In the simplest case, the principal role of two patterns is shared and we say that we are attaching a pattern to another pattern. This is the only situation where an actor may play the principal role in more than one behavior. It is allowed since the attached behaviors are considered as components of the behavior they are attached to.

3 Designing Characters: A *ScriptEase* Walk Through

Consider an example of the Guard Pattern, described as scenario 1 (Shard-1) from Section 2. As a default behavior, the guard patrols the room that the chest is in. When an enemy approaches the chest, the guard yells a warning, runs over to the chest, and stands in front of it. When the enemy moves away from the chest, the guard goes back to patrolling the room. If the enemy ignores the guard's warning and opens the chest, the guard attacks. Figure 1 describes all of the information needed to specify this instance of the Guard Pattern.

We have developed a tool called *ScriptEase* that allows this instance and many other variations of the Guard Pattern to be implemented quickly and easily. All user input is menu driven, with all options for behaviors and scenarios given in natural language. The user never does programming in the conventional senses and, indeed, never knows the existence of an underlying programming language. Figure 2 shows a screen shot of this tool.

Situations can be constructed by selecting conditions and actions from a list. The Situations panel on the left side of Figure 2 contains a list of all the situations that have been defined for this pattern instance. After a new situation has been created, it appears in this list and makes its condition and action lists available to be edited. Once a condition is added, it appears in a list inside the Conditions panel on the right side of Figure 2. Actions appear in the corresponding Actions panel. A condition or action can then be selected to make its parameters available for editing. In Figure 2, the first condition in the Conditions list is highlighted. Its parameters are shown inside of the Near Condition panel underneath the Situations panel.

Patrols are lower level patterns that can be attached to a Guard Pattern. All of the patrols that are attached to a Guard Pattern are listed in the Attached Patrols panel at the bottom of Figure 2. There are two types of Patrol Patterns used in this example. The "Room Patrol" from Figure 2 is an instance of a Way-point Patrol Pattern, which means the patrol is defined by a series of way-points that the guard walks along. The "Chest Post" is an instance of a Post Patrol Pattern, which means the guard just stands at a particular spot. Table 1 gives a description of these two instances. Any number of patrols can be attached to a guard, but only one is active at a time. There is a condition to test which patrol a guard is currently using, and an action to change a guards patrol.

Once all of the patrols and situations have been specified, code can be generated by clicking the "Generate" button at the very bottom of Figure 2. The user may further customize the situations using the *Situation Editor* tool of *ScriptEase*, shown in Figure 3. Finally, *ScriptEase* generates *Neverwinter Nights* scripting language code.

Figure 4 shows a game scenario of a guard in action. Part of the *Neverwinter Nights* scripting code generated by *ScriptEase* for this scenario is shown in Fig-

```
Pattern Type           : Guard Pattern
     Instance Name      : Chest Guard
     Guard Tag          : chest_guard
     Guarded Object Tag : chest
     Friend Identifier  : guard1_friend

Attached Patrol Patterns:
     1. Instance Name   : Room Patrol
     2. Instance Name   : Chest Post

Situation List:
     Name               : Spawn Situation
     Conditions         : The guard is created
     Actions            : Set the guard's patrol to "Room Patrol"

     Name               : Warning Situation
     Conditions         : An creature is within 5 meters of the guarded
                          chest. The guard is currently using patrol
                          "Room Patrol".
     Actions            : The guard yells "Hey! Get away from there."
                          Set the guard's patrol to "Chest Post".

     Name               : Continue Patrol Situation
     Conditions         : No enemy is within 5 meters of the guarded
                          chest. The guard is currently using patrol
                          "Chest Post".
     Actions            : Set the guard's patrol to "Room Patrol"

     Name               : Attack Situation
     Conditions         : An enemy creature opens the guarded chest
     Actions            : The guard says "I warned you!".  The guard
                          attacks the enemy.
```

Fig. 1. The Chest Guard instance of the Guard Pattern

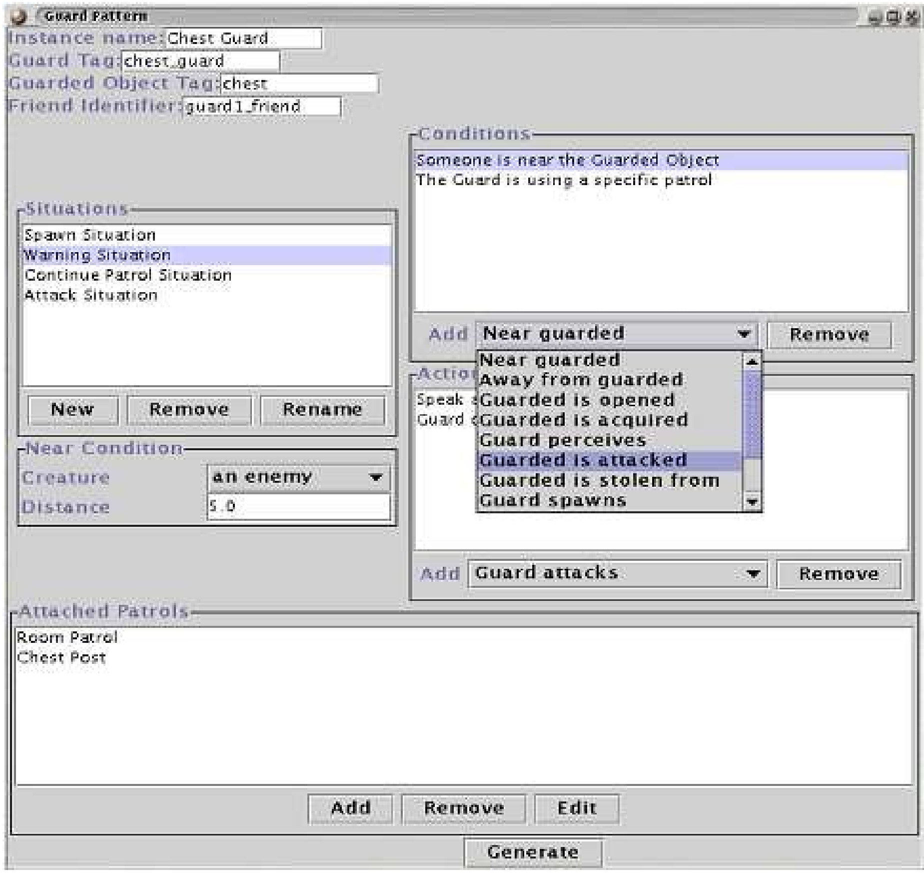

Fig. 2. Editing an instance of the Guard Pattern using *ScriptEase*

ure 5. Notice that the code is self documenting, enabling the user to easily find which portions of code correspond with the situations specified in *ScriptEase*. This is very useful if the user desires to fine tune the code on the lowest level.

We have also identified several *encounter* patterns. Instead of describing the behaviors of a principle actor, an encounter pattern defines a list of situations that describe some notable event in the game. For instance, in Baldur's Gate II, there is an interesting encounter in the Shade Lord's temple. There is a pedestal with an icon, called the Sun Stone, on it. There is also a ring of lights around the pedestal. When a particular type of monster, called a Shadow, enters the ring of lights, it is killed in a spectacular flash of light. If the Sun Stone is removed from the pedestal, the ring of lights disappears and the Shadows can then approach the pedestal without being killed. We have defined an encounter pattern called the Icon-Container Pattern. This pattern has 3 roles: an icon, a container, and an optional perimeter. The icon is a prop that can be placed

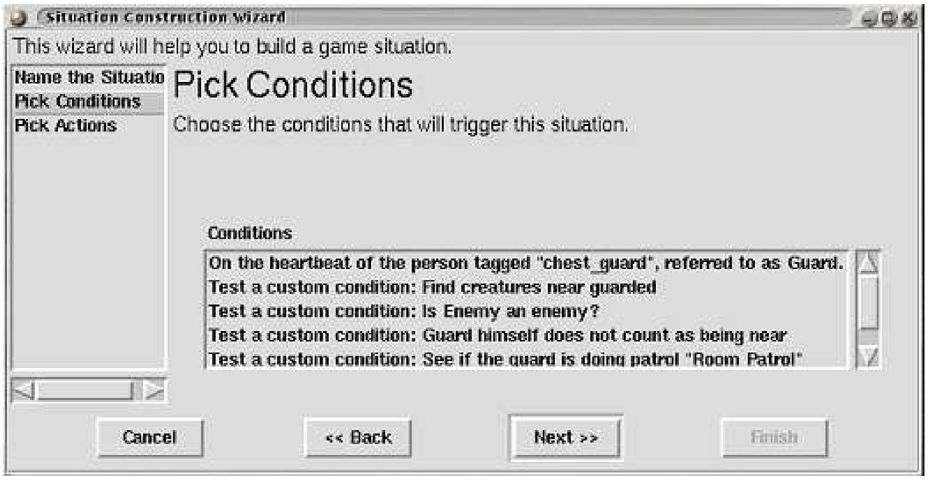

Fig. 3. Situation editing using *ScriptEase*

Fig. 4. *Neverwinter Nights* guarding scenario (using the Icon-Perimeter Pattern)

Pattern Type	Way-point Patrol Pattern	Pattern Type	Post Patrol Pattern
Instance Name	Room Patrol	Instance Name	Chest Post
Way-point Prefix	room 1	Post Tag	chest_post
Num of Way-points	8		
Initial Way-point	1		

Table 1. Two patrol instances attached to the Guard Pattern

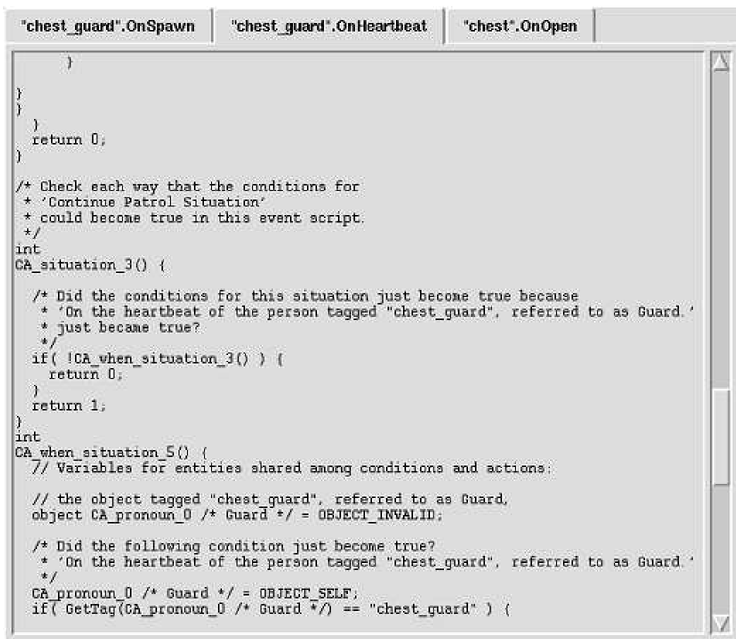

Fig. 5. *ScriptEase* code generation

into the container's inventory. The container is either an actor or a container prop. The perimeter is a polygonal area that an actor can enter and exit. This pattern involves four situations: adding the icon to the container, removing the icon from the container, an actor enters the perimeter while the icon is in the container, and an actor exits the perimeter while the icon is in the container. In each of these situations, the condition listed above is implicitly included in the condition list. The user can add more conditions to the list, and define actions to execute when the conditions are satisfied. Figure 6 shows these four situations instantiated for the Sun Stone Icon example.

We have identified multiple instances of this pattern in the Shade Lord temple alone, demonstrating that this pattern is useful in terms of abstraction, adaptation, and re-use. We have not yet created an interface specific to this particular

```
Pattern Type            : Icon-Container
     Icon               : Sun Stone
     Container          : Pedestal
     Perimeter          : Ring of Lights

Situation List:
     Implicit Condition : The Sun Stone is added to the Pedestal
     Other Conditions   : None
     Actions            : Activate the ring of lights

     Implicit Condition : The Sun Stone is removed the Pedestal
     Other Conditions   : None
     Actions            : Activate the ring of lights

     Implicit Condition : The Sun Stone is on the Pedestal and a creature
                          enters the ring of lights.
     Other Conditions   : The entering creature is a Shadow
     Actions            : Display an impressive visual effect.  Kill the
                          entering Shadow.

     Implicit Condition : The Sun Stone is on the Pedestal and a creature
                          exits the ring of lights.
     Other Conditions   : None
     Actions            : None
```

Fig. 6. An instance of the Icon-Container Pattern

pattern, however, all of the situations defined in Figure 6 have been implemented in the *ScriptEase* Situation Editor.

ScriptEase has been demonstrated to BioWare and we have received positive feedback. The tool is especially appreciated by the game designers, who prefer to work in terms of the story and characters, not at the level of programming. The tool is evolving, as we get more feedback from BioWare. Indeed, the project is expanding at a pace that is difficult to keep up with. The immediate goal is to give *ScriptEase* the functionality so that it can replicate all the capabilities in *Neverwinter Nights*. To do this requires adding a few more patterns (research) and a lot more scenarios (data input). The design of *ScriptEase* makes both issues easy to address.

4 Ongoing Work

Behavior patterns are only the beginning. The industry needs a tool that properly defines a complete game script, in much the same way that a script is used to outline a movie. A movie script must include information on each scene, including the physical arrangement of the scene, the characters that are present, how the characters interact, the dialogs, and the outcomes. A series of scenes has to be stitched together to give a coherent plot. Defining these components in isolation of each other (as is currently done in commercial games) is clearly wrong. Most (but not all) of these components touch on AI issues.

To cover the gamut of issues in game design, a ScriptEase-like tool needs other components, including dialog and plot. We believe that both of these can also be described by patterns. We have ideas for how to integrate these patterns into ScriptEase, and this is the subject of ongoing work.

The preceding description of ScriptEase described scripted behavior, where each character's behavior was predictable, modulo some random number generation. The reality is that creating realistic characters requires more sophisticated behavior. Machine learning is the answer. Unfortunately, compatibility with existing scripting languages (such as that in *Neverwinter Nights*) make this difficult. We expect our work to eventually lead us to the design of a new scripting language, one that supports core AI functionality (such as learning) as basic operators in the language.

Learning is a touchy issue in commercial games. Currently, machine learning plays a limited role in the commercial games industry. "[Learning] takes place as part of the game development cycle, never after the game ships" [9]. The reason for this is that the program developers have no control over how a learning game evolves; the results might be embarrassing. However, things are changing. The success of games like *The Sims* and *Black and White* have demonstrated the power (and commercial appeal) of games that learn.

The issue of testability of a learning algorithm before the product ships is of paramount importance to the games industry [3]. There are no guarantees with any learning algorithm, since the user (game player) can deliberately expose the learner to a contrived set of learning experiences. One way of addressing this is the ScriptEase approach of using patterns. Patterns can be tested individually and verified to be correct. Building on top of verified patterns allows one to create composite patterns that can have guarantees of correctness.

5 Conclusions

This paper discussed one aspect of ScriptEase – behavioral patterns. Our research and development efforts concentrated on this aspect of our vision for the simple reason that it has the highest potential for an impact in the short-term. Given the enormous effort that goes into defining behaviors in a complex game like *Neverwinter Nights*, any tool (such as ScriptEase) that can can reduce this effort translates into enormous costs savings and improved product reliability.

The commercial games industry is pushing AI technology into new and innovative directions. The demand for realism, high-performance, and real-time responses make the problems especially challenging. One could argue that this industry is one of the biggest receptors for AI technology, and yet it has historically been ignored by the AI community. There are wonderful opportunities here for ground-breaking innovative research.

6 Acknowledgments

Financial support was provided by the Institute for Robotics and Intelligent Systems (IRIS), the Natural Sciences and Engineering Research Council of Canada (NSERC), and Alberta's Informatics Circle of Research Excellence (iCORE). This research was inspired by our many friends at BioWare. We thank BioWare for their support and encouragement.

References

1. E. Adams. In defense of academe. *Game Developer*, pages 55–56, November 2002.
2. C. Alexander, S. Ishakawa, and M. Silverstein. *A Pattern Language*. Oxford University Press, New York, 1977.
3. J. Barnes and J. Hutchens. Testing of undefined behavior as a result of learning. In S. Rabin, editor, *AI Game Programming Wisdom*, pages 615–623. Charles River, 2002.
4. M. Brockington and M. Darrah. How not to implement a basic scripting language. In S. Rabin, editor, *AI Game Programming Wisdom*, pages 548–554. Charles River, 2002.
5. S. Bromling, D. Szafron, J. Schaeffer, S. MacDonald, and J. Anvik. Generalising pattern-based parallel programming systems. *Parallel Computing*, 2002. To appear.
6. R. Evans and T. Lamb. Social activities: Implementing wittgenstein, 2002. `http://www.gamasutra.com/features/20020424/evans_01.htm`.
7. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns*. Addison Wesley, 1995.
8. R. Hill, C. Han, and M. van Lent. Applying perceptually driven cognitive mapping to virtual urban environments. *AAAI National Conference*, pages 886–893, 2002.
9. N. Kirby. GDC 2001 AI roundtable moderator's report, 2001. http://www.gameai.com.
10. J. Laird and M. van Lent. Human-level AI's killer application: Interactive computer games. *AAAI National Conference*, pages 1171–1178, 2000.
11. P. Tozour. The evolution of game AI. In S. Rabin, editor, *AI Game Programming Wisdom*, pages 3–15. Charles River, 2002.