A PATTERN CATALOG FOR COMPUTER ROLE PLAYING GAMES

C. Onuczko, M. Cutumisu, D. Szafron, J. Schaeffer, M. McNaughton, T. Roy,
K. Waugh, M. Carbonaro* and J. Siegel
Department of Computing Science, *Department of Educational Psychology
University of Alberta, Edmonton, AB, Canada T6G 2E8
{onuczko, meric, duane, jonathan, mcnaught, troy, siegel }@cs.ualberta.ca, *mike.carbonaro@ualberta.ca

KEYWORDS
Generative design patterns, scripting languages, code generation, computer games.

ABSTRACT

The current state-of-the-art in computer games is to manually script individual game objects to provide desired interactions for each game adventure. Our research has shown that a small set of parameterized patterns (commonly occurring scenarios) characterize most of the interactions used in game adventures. They can be used to specify and even generate the necessary scripts. A game adventure can be created at a higher level of abstraction so that team communication and coding errors are reduced. The cost of creating a pattern can be amortized over all of the times the pattern is used, within a single adventure, across a series of game adventures and across games of the same genre. We use the computer role-playing game (CRPG) genre as an exemplar and present a pattern catalog that supports most scenarios that arise in this genre. This pattern catalog has been used to generate ALL of the scripts for three classes of objects (placeables, doors and triggers) in BioWare Corp.'s popular Neverwinter Nights CRPG campaign adventure.

MANUAL SCRIPTING

A computer game typically contains a game engine that renders the graphical objects and characters, and manages sound and motion. A programming team writes a game engine that can be re-used across multiple game adventures and enhanced for future games. They also produce a set of computer aided design (CAD) tools that are used by a team of writers, artists, musicians, voice actors and other skilled craftspeople to create content such as backgrounds, models, textures, creatures, props, sounds, and music that are shared across game adventures. Adventure (story) designers also use these tools to create individual adventures. For example, Figure 1 shows BioWare's (http://www.bioware.com/) Aurora Toolset that is used to create game adventures for Neverwinter Nights (NWN) (http://nwn.bioware.com/).

A game engine typically dispatches game events to scripts that support interactions between the player character (PC) and game objects. These interactions vary for each game adventure and programmers must write the scripts that control them. For example, an adventure designer may want the PC to agree to complete a quest before allowing the PC to enter a castle. To ensure that the quest is accepted, a heat source is placed close to the castle door that prevents the PC from getting close enough to the door to use it. The designer provides a non-player character (NPC) with a cloak of fire resistance that will be given to the PC after the PC has agreed to undertake the quest

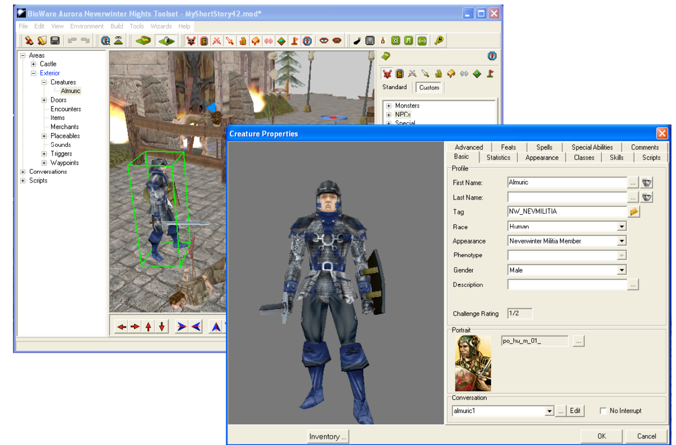

Figure 1: The Aurora Toolset CAD Tool for NWN Adventure Designers

```
void main() {
  object Enterer;
  object Cloak;
  object FireCenter;
  location jumperLocation;
  vector jumperPosition;
  float jumperFacing;
  object jumperArea;
  vector targetPosition;
  vector heading;

  Enterer = GetEnteringObject();
  Cloak = GetObjectByTag("CloakofFireResistance");
  if( ! GetItemInSlot(INVENTORY_SLOT_CLOAK, Enterer)
    == Cloak) {
    ApplyEffectToObject(DURATION_TYPE_INSTANT,
      EffectVisualEffect(VFX_IMP_FLAME_S), Enterer);
    FireCenter = GetObjectByTag("firecenter");
    jumperLocation = GetLocation(Enterer);
    jumperPosition = GetPositionFromLocation
      (jumperLocation);
    jumperArea =
      GetAreaFromLocation(jumperLocation);
    jumperFacing = GetFacingFromLocation
      (jumperLocation);
    targetPosition = GetPositionFromLocation
      (GetLocation(FireCenter));
    heading = Vector(targetPosition.x -
      jumperPosition.x, targetPosition.y -
      jumperPosition.y, targetPosition.z -
      jumperPosition.z);
    heading = VectorNormalize(heading);
    jumperPosition = jumperPosition - 2.5*heading;
    jumperLocation = Location(jumperArea,
      jumperPosition, jumperFacing);
    AssignCommand(Enterer, JumpToLocation(
      jumperLocation));
    FloatingTextStringOnCreature
      ("The heat is too strong.", Enterer, FALSE);
  }
}
```

Figure 2: An NWScript Script for a Barrier

The adventure designer may ask a programmer to implement this scenario as scripts attached to game objects. One of these scripts prevents the PC from getting to the door unless the PC is wearing the cloak. The other script gives the cloak

© EUROSIS-ETI

to the PC, after the PC agrees to undertake the quest. Figure 2 shows the script that prevents the PC from getting near the door, written in NWScript, the NWN scripting language. This script is long and complex, containing local variables, literal tags (representing objects created by the designer using the Aurora Toolset), conditional expressions and many function calls to the NWScript library. A professional game designer without extensive programming experience could not write this script.

Computer games typically have thousands of game objects that must be scripted. There are four serious disadvantages to scripting these objects manually: 1) poor script tracking, 2) simplistic scripts that take too long to write, 3) scripting errors and 4) team communication problems.

Manually written scripts are *hard to track*. The large number of game objects makes it difficult to organize and track objects during adventure development. Organizing and tracking scripts is even more difficult since most scripts involve interactions between several objects. A change to an object or a script often results in unexpected negative consequences.

Most scripts provide only *simplistic game behaviors*. Since a large number of objects require scripts, all but the most important objects must have very simple scripts. Unless an object is on the critical path of the main plot line, it usually has a single trivial script. This makes the game repetitive and predictable, and therefore boring. More interesting interactions are desirable, but are not cost effective to write because of the large time investment needed. Even scripting simple behaviors consumes *too much programmer time*, which could be better spent on developing better game engines and additional tools for the game designers.

Many common *scripting errors* are difficult to detect without manually playing through all of the game scenarios and trying all of the different combinations of user choices. For example, scripts are often created using cut-and-paste techniques and it is not uncommon for the programmer to cut-and-paste scripts without making all the changes needed for the new context. There are so many game objects and scripts that it has become standard practice to use object numbers or script numbers as part of their names. An off-by-one error in a name often results in a legal script that performs incorrectly.

Many designers are unable to write scripts themselves and must rely on programmers. Delegating the scripting to a programmer can lead to inconsistencies between the game designer's intent and the programmer's scripts due to *communication errors*.

At least professional adventure designers have access to programmers who can write scripts for them. Recently, there has been a trend to create computer games where amateur designers can create adventures and post them online for others to play. For example, NWN has an active adventure designer community. Thousands of players contribute adventure modules of their own creation. Contributed modules can be freely downloaded from the Neverwinter Nights Vault web site (http://nwvault.ign.com). The most popular of the 4,100 community-created modules at this site has been downloaded over 252,000 times (as of May 2005), and the tenth-most, 88,000 times. Most of these community designers are non-programmers, so they cannot write the scripts themselves. Instead, they try to copy existing scripts from other adventures and paste them into their own adventures. This does not work very well, since the copied scripts usually have many adventure-dependent literal tags and other context dependent code. Therefore, the adventure designers turn to the community for help by posting to one of the NWN scripting forums. There have been about 150,000 scripting-related postings to the BioWare forums (http://nwn.bioware.com/forums/viewforum.html?forum=47) (as of May 2005). Unfortunately, the help they receive from these forums is often not very useful (Carbonaro et al 2005).

FROM SCRIPTING TO PATTERNS

Our approach to solving these problems is to provide an alternative to manual scripting, based on a higher-level abstraction than scripting code. Our first goal was to identify a set of patterns that describe all of the object interactions that commonly occur in CRPGs. A pattern is a familiar commonly occurring scenario or idiom in an adventure of the appropriate genre. An example of a pattern in the CRPG genre is a secret door that appears when a protagonist gets close enough to it and notices it. Our second goal was to build a tool that uses this set of patterns to generate the scripting code automatically.

Our work is inspired by the use of design patterns to describe object collaborations (Gamma et al. 1994) in general purpose programs. A design pattern specifies the solution to a software design problem at a higher level of abstraction than a program that implements the design. For example, recall the scenario from the previous section. It required two scripts. One script was used to prevent the PC from getting close to the door. The other script allowed an NPC to give a protective cloak to the PC after a conversation had taken place. We provide the game designer with a set of re-usable patterns that can be used to specify such scripts at a higher level of abstraction. The adventure designer would still use two scenes, the conversation and the encounter near the door. We provide two general patterns that can be adapted for these scenes and for many other scenes as well.

In this example, the game designer uses two patterns, *Trigger enter/exit – barrier* and *Conversation when/what*. The first pattern prevents a PC from entering or exiting a trigger region and the second pattern controls whether a particular conversation point can be reached and makes something happen if it is reached. Patterns support adventure design at a higher level of abstraction. For example, the first pattern applies whenever the designer wants to prevent the PC from getting into or out of a region. It is not restricted to being near a door or due to a heat source. Patterns reduce and clarify design team communication, since low-level details do not obscure the designer intent. Designers and programmers can quickly grasp the meaning of a *barrier* pattern. Patterns foster re-use across scenarios in the design of a particular adventure, across the design of multiple adventures for a single game and even across games of the same genre.

© EUROSIS-ETI

A traditional software design pattern is *generic*. It provides a set of solutions to a general design problem (Gamma et al. 1994). The pattern must be adapted to a specific context during application design. The designer refines the design solution family to a single solution in the context of the application.

Our CRPG patterns are also generic. Each pattern must be adapted to a specific scenario by the adventure designer. For example, the *Trigger enter/exit – barrier* has several options, including a *trigger*, a *distance* and a *visual effect*. The *trigger* is a polygonal region that the designer paints into the adventure using a computer-aided design tool. When a character in the game steps into or out of the region, the game engine generates an *onEnter* or *onExit* trigger event respectively and the appropriate script attached to the trigger object is executed. The *trigger* option of the barrier defines the region that a creature cannot enter or exit. The *distance* option defines the distance that the creature trying to cross the barrier will bounce back from the barrier. The *visual effect* option defines the visual effect that will occur when the creature tries to cross the barrier. These options are "hard-wired" into the script that appears in Figure 2, but not into the pattern. Using a pattern is much more generic and safer than cutting and pasting and changing these "hard-wired" values. Setting options is only the simplest form of seven kinds of adaptation that can be used to adapt CRPG patterns. These other forms of adaptation differentiate a pattern from a simple function call and are discussed in the next section. The important idea is that each pattern defines a *generic* family of solutions that must be adapted to the particular context of a game adventure, analogously to the way traditional software design patterns must be specialized to the context of an application.

A traditional software design pattern is *descriptive* (Gamma et al. 1994). Each pattern provides a design lexicon, describes a set of solution structures and describes the reasoning behind the solutions. A programmer selects an appropriate descriptive design pattern, adapts it to the application program and manually *translates* it to code. Experienced programmers who have implemented the same design pattern in other contexts can usually perform adaptation and coding more quickly than novice programmers, where unfamiliar or ambiguous natural language pattern documentation can lead to slow progress and coding errors. Our CRPG patterns can be used descriptively so that the adventure designer can communicate adventure designs more concisely and accurately to a programmer who can implement these descriptive patterns by writing scripting code. However, there is another kind of design pattern.

A *generative design pattern* (GDP) (Budinsky et al. 1996) (Florijn et al. 1997) (MacDonald et al. 2002) has all of the characteristics of a descriptive pattern. In addition, it generates application code automatically so that a programmer does not have to manually translate the pattern to code. Novice and experienced designers can use GDPs to speedup code production and reduce coding errors. Recently, we have used GDPs in computer games for the first time (McNaughton et al. 2003). If GDPs are used, then the adventure designer gains one more advantage from using patterns rather than manual scripting. GDPs allow a game designer to generate scripting code automatically without any need for programmer intervention. This eliminates any chance of communication error between the adventure designer and the programmer.

The pattern catalog presented in this paper is supported as a generative pattern catalog in a tool called ScriptEase (http://www.cs.ualberta.ca/~script/scripteasenwn.html) that automatically generates scripting code for NWN. An adventure designer can also use this pattern catalog to create descriptive design patterns for any CRPG. In this case, each pattern serves as a template for the explicit specification of a scenario that can be implemented by a script programmer. A pattern catalog is not a static entity. It is meant to evolve by expanding (and contracting) as appropriate to satisfy the needs of adventure designers. Therefore, ScriptEase also includes a pattern design tool, which allows adventure designers to modify existing patterns and to create new ones.

PATTERN ADAPTATION

To understand pattern adaptation, it is necessary to understand the component parts of a pattern (McNaughton et al. 2004b). Each *pattern* contains one or more event-driven scenarios called situations. Each *situation* (icon *S*) contains the *event* (icon *V*) that activates it and a set of *definitions* (icon *D*), *conditions* (icon *C*) and *actions* (icon *A*). For example, Figure 3 shows the components of the *Trigger enter/exit – barrier* pattern.

The first situation has been opened to show its components, but the other three are closed (for brevity). There are no conditions in this pattern. The first action *(jumps towards with effect*) is an example of an *action encounter*, which is a re-usable action that contains other actions. It has four options, *Jumper* – bound to *Enterer*, *Target* – bound *to The Center*, *Distance* – bound to *Negative Bounce Distance*, and *Impact Effect* – bound to *Touch Effect*.

To use a pattern, a designer creates an instance of the pattern and adapts it for a specific scenario. The simplest form of adaptation is to set the pattern options as described previously. However, setting options provides only limited abstraction, equivalent to setting function parameters and is not sufficient for the kinds of adaptation needed to support CRPG patterns. Other forms of adaptation include adding or removing components. Table 1 lists the various kinds of adaptation in increasing order of complexity.

Table 1: Kinds of Pattern Adaptation in Increasing Order of Complexity

1.	Setting options
2.	Removing situations
3.	Removing actions and definitions
4.	Removing conditions
5.	Adding actions and definitions
6.	Adding conditions
7.	Adding situations

© EUROSIS-ETI

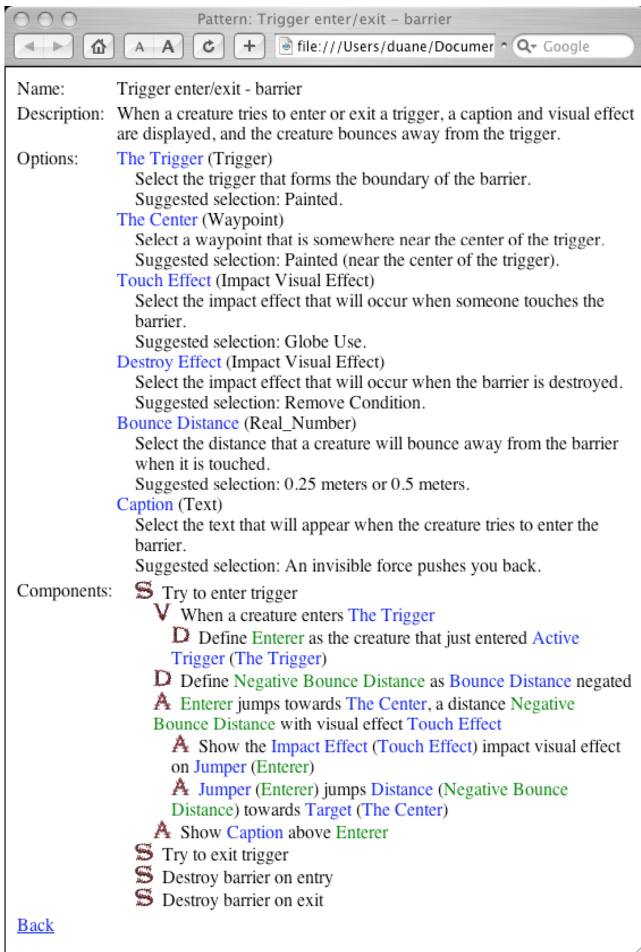

Figure 3: The *Trigger enter/exit - barrier* Pattern

To use this pattern for the scenario described in the previous section, the designer adapts the instance by:

1. setting the options to the appropriate objects and values: *The Trigger* – a region near the door (“Firetrigger”), *The Center* – a waypoint near the center of the trigger (“firecenter”), *Touch Effect* – A flame visual effect (VFX_IMP_FLAME_S), *Destroy Effect* – not used, *Bounce Distance* – 2.5, *Caption* – “The heat is too strong.",

2. removing the unwanted scenarios: *Try to exit trigger*, *Destroy barrier on entry* and *Destroy Barrier on exit*, and

3. adding a definition and a condition so that the barrier will not work on a creature that is wearing the cloak.

After adapting this instance, it looks like the pattern in Figure 4. This instance can serve as a specification for a programmer. Alternately, if the adventure designer is designing for NWN, then ScriptEase can be used to generate the scripting code automatically.

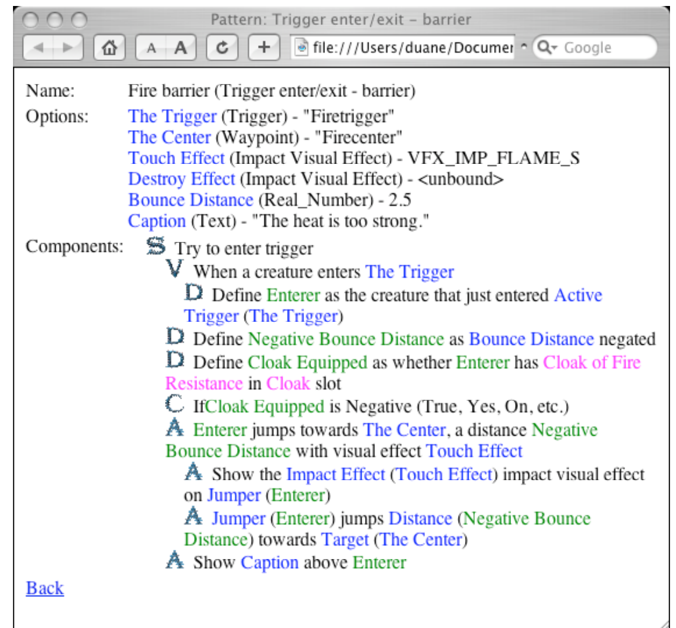

Figure 4: An Adapted Instance of the *Trigger enter/exit - barrier* Pattern

THE PATTERN CATALOG

We have identified four kinds of patterns that are necessary to generate all of the scripts found in CRPGs: *encounter*, *dialogue*, *behavior*, and *plot*. In total our pattern catalog has 60 patterns, consisting of 56 encounter patterns, 1 dialogue pattern and 3 behavior patterns. We are actively engaged in adding more patterns, especially dialogue, behavior and plot patterns. Our pattern catalog is available online at http://www.cs.ualberta.ca/~script/patterncatalog/.

An *encounter pattern* is used to script an interaction between the PC and an inanimate game object. It is useful to divide inanimate objects into groups that can be interacted with in different ways. Three examples of inanimate object groups are: *placeables*, *doors* and *triggers*. A placeable is an inanimate object that can be placed anywhere in the story world. Examples include chests, statues, chairs, tables, levers, and piles of rubble. A placeable is considered a *container* if it can hold items. A *door* can only be placed at the entrance to a structure or between two rooms in a structure. A *trigger* is a region of space that generates an event when a character enters or exits its perimeter. The *Trigger enter/exit – barrier* pattern described earlier is an example of an encounter pattern. The pattern catalog contains 28 placeable, 15 door and 13 trigger encounter patterns, for a total of 56 encounter patterns.

A *dialogue* pattern is used to control conversations. A tree is a common model for conversations in an interactive adventure. At alternate levels in the tree, either the game player selects a conversation node from those available for the PC, or a script selects a conversation node for the NPC. Figure 5 shows an example NWN conversation tree in ScriptEase, for the scenario described previously. Nodes marked [OWNER] (red) are for the NPC and the other nodes (blue) are for the PC.

© EUROSIS-ETI

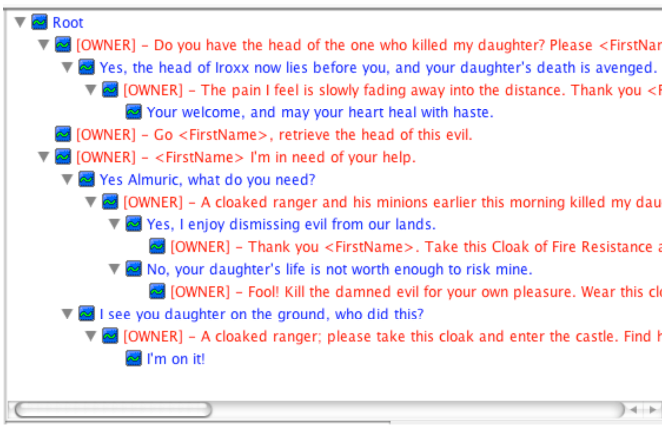

Figure 5: A NWN Conversation Tree in ScriptEase

There are actually two kinds of scripts that can be attached to a conversation node. A *when* script evaluates a *Boolean* that indicates whether the node should appear in the conversation or not. A *what* script provides actions that are taken if the conversation node is reached. Our pattern catalog currently contains a single generic dialogue pattern. The *Conversation when/what* pattern allows the adventure designer to generate *when* and *what* scripts for a conversation node. The sample scenario described earlier can be created using an instance of this pattern. Figure 6 shows the instance of this pattern that is attached to the conversation node "[OWNER] – Thank you <FirstName>. Take this Cloak …". This pattern instance transfers the cloak from the NPC to the PC and fires a visual effect. In general, this pattern has two situations: *When displayed* and *What actions*. The designer has deleted the first situation during adaptation, since this node should always be displayed if its parent node in the tree is displayed.

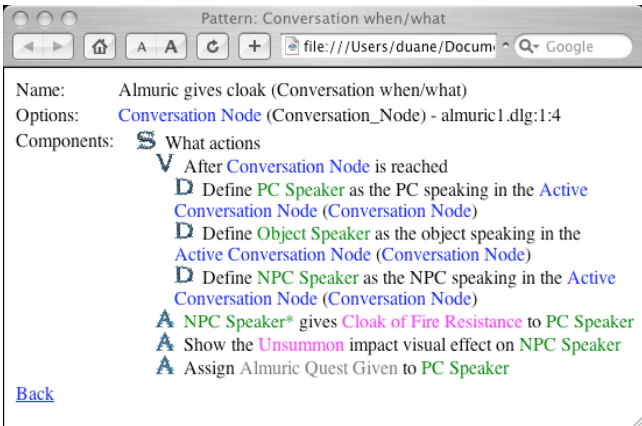

Figure 6 An Adapted Instance of the *Conversation when/what* Dialogue Pattern that uses the *What actions* Situation

Figure 7 shows an example of using this pattern to control whether a conversation node appears in a conversation or not. The adventure designer would like the first [OWNER] node in Figure 5 to appear only if the PC has completed the quest, and the second [OWNER] node to appear only if the PC has accepted the quest, but not yet completed it. Notice from Figure 6 that the PC is given a plot token called *Almuric Quest Given* after agreeing to complete the quest. This plot token can be used in a *Conversation what/when* pattern to hide the second [OWNER] node until the PC has obtained the plot token. Figure 7 shows an adapted instance of the *Conversation what/when* pattern that achieves this objective.

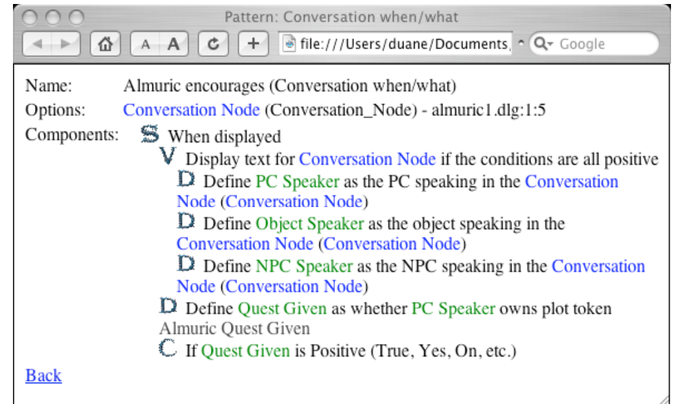

Figure 7: An Adapted Instance of the *Conversation what/when* pattern Dialogue Pattern that uses the *When displayed* Situation

A different instance of this pattern is used to hide the first [OWNER] node until the PC returns with the head of the evil-doer. This pattern is not split into two separate patterns since there are often times when a conversation node is both guarded by a "when" and requires "what" actions. This single dialogue pattern in our pattern catalog can be used to control all conversations on a node-by-node basis. We are currently developing other dialogue patterns that can be used at a higher level of abstraction to model frequently occurring conversation patterns consisting of many nodes.

A story designer can use a *behavior pattern* to specify the actions of an NPC. For example, the adventure designer may want an NPC to stay near an object and to start a dialogue whenever the PC gets close to that object. Our pattern catalog has a pattern called *Creature heartbeat – (PC near object) show dialogue* that supports this behavior. There are three behavior patterns in our catalog at the current time and we are actively adding more behavior patterns.

A plot pattern guides the player character (PC) through the story. For example, in CRPGs it is common to give the player quests. The player advances through the quest in a series of states: *unassigned*, *assigned*, *resolved* and *closed*. A common way to have the player participate in a quest is through a dialogue with a non-player character (NPC), which consists of a series of conversations. The dialogue pattern *Simple verbal quest* specifies which conversation is used for each of the various states of the quest. This pattern depends on other patterns to set a plot token which causes the quest state to change. External patterns are used to provide flexibility since a quest can involve solving a riddle posed by a different NPC, defeating a creature, opening a door, obtaining a specific item, etc. We are currently building a basic set of plot patterns to add to our catalog. Although there are currently no plot patterns in our catalog, many are under development. In the meantime, we have introduced the concept of a *plot token* as illustrated in the previous example.

© EUROSIS-ETI

EVALUATION OF THE PATTERN CATALOG

In a previous paper (McNaughton et al. 2004a), we described how we used encounter patterns to generate all of the scripting code attached to placeable objects in the NWN official campaign story. In that experiment, we replaced 497 calls to 182 different scripts comprising 1925 non-comment lines of hand-written code by pattern-generated code using 431 instances of 23 different encounter patterns and our 1 dialogue pattern.

To ensure that our pattern catalog could be used by non-programmers, we invited a high school English class to use the Aurora Toolset, our pattern catalog and ScriptEase, to write short stories as adventures in NWN. The students succeeded in using our patterns to generate interesting stories (Szafron et al. 2005) that play as NWN adventures.

Besides NWN, we have identified patterns in two other CRPGs, Fable (http://www.fablegame.com) by Lionhead Studios and The Elder Scrolls III: Morrowind (http://www.morrowind.com) by Bethesda Softworks. For example, *Placeable use – toggle door* can be found in Fable where there are four rocks and a door. The player must hit the rocks in the correct order to open the door. Currently, the PC must attack the rocks, but it makes more sense to restructure the puzzle so that the PC is required to touch the rocks rather than hit them. In Morrowind, this pattern is used to open a door when the PC uses a lever. The pattern *Door click – show monologue* can be observed in several areas of Fable that involve the use of riddles. Throughout the game there are several doors called Demon Doors, which require the player to solve a riddle to open them. When the user clicks on the door, the door speaks a monologue giving the user the riddle that must be solved. This pattern is used in Morrowind near the beginning of the game. When the user clicks on a door, the PC is told to look in a nearby barrel for a ring. The pattern *Trigger enter – spawn creature near object* is used in Fable for an ambush. The player at one point in the game is asked to escort a person to a nearby farm. When the person being escorted enters a trigger, an enemy is spawned nearby to attack the person. In Morrowind this pattern is used to spawn a person high above the player that falls to his death, due to the misuse of a jumping potion.

CONCLUSION

In this paper, we have presented a pattern catalog for CRPGs. This catalog contains 60 patterns that can be used by adventure designers to effectively communicate their stories to programmers who must write the scripts to make these adventures come alive. These patterns can also be used to automatically generate scripts for adventure designers working with the NWN system.

ACKNOWLEDGEMENT

This research was supported by grants from the (Canadian) Institute for Robotics and Intelligent Systems (IRIS), the Natural Sciences and Engineering Research Council of Canada (NSERC), Alberta's Informatics Circle of Research Excellence (iCORE), BioWare Corp. and Electronic Arts (Canada) Ltd. We thank former ScriptEase team members James Redford (M.Sc.), Dominique Parker (M.Sc.), Stephanie Gillis (High School Teacher) and Sabrina Kratchmer (WISEST summer student) for their efforts on ScriptEase. We especially thank our many friends at BioWare for their feedback, support and encouragement, with special thanks to Mark Brockington.

REFERENCES

Budinsky, F., Finnie, M., Vlissides, J. and Yu, P. 1996. "Automatic code generation from design patterns". *IBM Systems Journal*, 35, 2, 151-171.

Carbonaro, M., Cutumisu, M., McNaughton, M., Onuczko, C., Roy, T., Schaeffer, J., Szafron, D., Gillis, S., Kratchmer, S. 2005. "Interactive Story Writing in the Classroom: Using Computer Games" In *Proceedings of DiGRA 2005 Conference: Changing Views – Worlds in Play*, (Vancouver, Canada, June), 323-338.

Florijn, G., Meijers, M. and van Winsen, P. 1997. "Tool support for object-oriented patterns". In *Proceedings of the 11th European Conference on Object-Oriented Programming*, Vol. 1241 of Lecture Notes in Computer Science, Springer, 472-495.

Gamma, E., Helm, R., Johnson, R. and Vlissides, J. 1994. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, MA.

MacDonald, S., Szafron, D., Schaeffer, J., Anvik, J., Bromling, S. and Tan, K. 2002. "Generative Design Patterns", In *Proceedings of the 17th IEEE International Conference on Automated Software Engineering*, (Edinburgh, UK, September), 23-34.

McNaughton, M., Cutumisu, M., Szafron, D., Schaeffer, J., Redford, J., and Parker, D. 2004a. ScriptEase: "Generative Design Patterns for Computer Role-Playing Games", In *Proceedings of the 19th Interntational Conference on Automated Software Engineering*, (Linz, Austria, September), 88-99.

McNaughton, M., Redford, J., Schaeffer, J., and Szafron, D. 2003. "Pattern-based AI Scripting using ScriptEase", *In Proceedings of the 16th Canadian Conference of Artificial Intelligence*, (Halifax, Canada, June), 35-49.

McNaughton, M., Schaeffer, J., Szafron, D., Parker, D. and Redford, J. 2004b. "Code Generation for AI Scripting in Computer Role-Playing Games", In *Proceedings of the Challenges in Game AI Workshop at AAAI-04*, (San Jose, USA, July), 129-133.

Szafron, D., Carbonaro, M., Cutumisu, M., Gillis, S., McNaughton, M., Onuczko, C., Roy T. and Schaeffer, J. 2005. "Writing Interactive Stories in the Classroom", *Interactive Multimedia Electronic Journal of Computer-Enhanced Learning (IMEJ)*, Volume 7, Number 1, May.

© EUROSIS-ETI