This is a preprint of an article that will appear in Proceedings of the Sixth Artificial Intelligence and Interactive Digital Entertainment Conference (AIIDE), 6 ms, Palo Alto, USA. October 2010. AAAI Press.

# Quest Patterns for Story-based Computer Games

**Marcus Trenton, Duane Szafron, Josh Friesen**
Department of Computing Science, University of Alberta
Edmonton, AB, CANADA T6G 2H1

**Curtis Onuczko**
BioWare Corp. 200-4445 Calgary Trail
Edmonton, Alberta, CANADA T6H 5R7

marcus.trenton@gmail.com, dszafron@ualberta.ca, joshd.friesen@gmail.com, curtis.onuczko@gmail.com

## Abstract

As game designers shift focus from graphical realism to immersive stories, the number of game-object interactions grows exponentially. Games use manually written scripts to control interactions. ScriptEase provides game designers with generative patterns that generate scripting code to control common interactions. This paper describes a new kind of generative pattern, quest patterns, that generate scripting code to control story plot. We present our quest pattern architecture and study results that show quest patterns are easy-to-use and reduce plot scripting errors.

## Introduction

Scripting is a major bottleneck in computer games. Scripts control object interaction in a similar way that a movie script controls character actions. Game scripts control simple interactions, such as making a ghost appear by a window when the hero enters a room. Scripts select dialog lines for the player character (PC) and non-player characters (NPCs), based on past actions and character attributes. For example, a script is used to ensure that a conversation with the villain cannot occur until the hero possesses a gemstone. Scripts also remember data related to PC actions, such as a list of places visited.

This paper focuses on scripting that controls the plot in story-based computer games. Conceptually, the plot is a decision graph, where each decision point is a game event. These events can vary in abstractness. Concrete events include the death of a specific NPC or the PC's acquisition of an item PC. Abstract events include the PC becoming the leader of an organization or becoming famous. It is natural to aggregate plot events into more cohesive, self-contained units to ease comprehension. Example aggregations in other domains include book chapters, play acts, and television episodes. In games, plot events are commonly combined into quests or missions. An example quest is to retrieve an ancient book from a ruined castle and return it to its rightful owner. In today's open world games, PCs are not restricted to a linear story. Instead they are free to roam about the world and interact with a variety of game objects in "almost" any order. Therefore, the game's plot graph must be comprehensive and flexible enough to accommodate any meaningful PC or NPC action or event in the game. For example, if the PC destroys the ancient book then that quest fails, but the overall plot of the game should continue. The challenge lies in making the progression of events described in the plot graph easy to understand and edit by the game story author.

A scripting language is an interface between the game's engine, which knows when a lever is pulled, and the author of a game story who knows why the lever should be pulled and what should happen next. Since scripts are changed and tested frequently they are usually interpreted (not compiled). Some games use a general scripting language. Vampire: The Masquerade – Bloodlines and Sid Meier's Civilization IV use Python, while Homeworld 2 and World of Warcraft use Lua. Other games use custom scripting languages. Neverwinter Nights (NWN) uses NWScript, and Elder Scrolls IV: Oblivion uses TES Script. Scripting languages are like C or Java with much less functionality and scripts are written manually using a text editor.

With the increased complexity of game production, skill specialization has occurred and many game story authors have no programming (scripting) skills. Either an author dictates story details to a programmer, or a technical designer serves as an intermediary. This can result in miscommunication that produces errors and delays.

Even after a game's release, scripts are still created. Many games, such as NWN and Warcraft 3: Reign of Chaos, support user created content to extend the lifespan of games and generate greater user interest. A game user must write scripts to add meaningful interactive content, and the complexity of scripting foils many users.

One way to solve the scripting problem is to reduce the difficulty of programming so more game authors can program. Environments like Alice and Scratch are designed to simplify programming. Iconic programming systems such as Kodu have also been used to create simple games. Our solution is to provide an environment in which game authors manipulate patterns that generate scripting code. Generative game patterns that describe basic game interactions are usable by the general population without knowledge of computer scripting (Carbonaro et al. 2008). In this paper, we introduce patterns that control plot and show that these patterns are easier to use than manual

Copyright © 2010, Association for the Advancement of Artificial Intelligence (www.aaai.org). All rights reserved.

scripting and that their use generates scripts with fewer plot errors than manually written scripts.

Quest script errors have higher consequences than other script errors. A bug in one quest may not only cause that quest to be unplayable, but it may also affect dependent super-quests. This can cause entire sections of a game to be unplayable. For commercial games, a bug in a single quest may impede the work-flow of several individuals for several hours or a day. These individuals are often able to work around the bug, but it results in them being able to work at a fraction of the capacity that they could before. The continued cost of these individual bugs adds up over the project to be several hundreds of hours of wasted time. The cost-savings in preventing quest bugs can be huge.

## ScriptEase

ScriptEase (ScriptEase 2009) generates scripting code from patterns (Gamma et al. 1994). An author creates instances of patterns from a catalogue and adapts the instances to a game story context. The adapted pattern instances are used to generate scripting code. If the patterns in the catalogue are insufficient for the current story, an author can construct new patterns and add them to the pattern catalogue for use in the current story and re-use in future stories. Although the pattern catalogue is game independent, the current implementation of ScriptEase generates NWScript code for the NWN game. This implementation is sufficient to show the utility of using generative patterns in story-based games

### Neverwinter Nights

NWN (released by Bioware Corp. in 2002) provides a testbed for evaluating the concepts introduced by ScriptEase. NWN is popular story-based game where a player controls a single player character (PC) that interacts with a fantasy world through movement, interaction with game objects, conversation, magic spell casting, and combat. NWN won 86 awards and its popularity was enhanced by releasing the Aurora Toolset, which enabled players to create their own stories (modules). The Aurora Toolset supports visual tools for conversation authoring, NPC creation, environment creation and game object creation. What is lacking in the toolset is a simple way of designing game-object interactions. An author must manually write scripts in the NWNScript language using a text editor and a predefined list of API functions, variables and constants. There are no debugging tools and to preserve game immersion for the player, script errors fail silently rather than presenting an error notification.

### ScriptEase Patterns

ScriptEase allows an author to create story interactions in a top-down manner. The author starts with an abstract intent by selecting a pattern and creating an instance. The author then adapts the pattern in a hierarchical manner, starting from general options and proceeding to specific details. This is the opposite of constructing a code script by creating expressions that are placed into statements that are placed into scripts. With ScriptEase an author always selects from a small number of explicit options and never has to create any kind of construct on a blank page.

Suppose an author wants a *Skeleton* to be spawned when the *Emerald of Protection* is removed from the *Statue*. This is an encounter example, where the PC interacts with a game object. The author first selects an intent, the *When the Placeable loses Specific Item spawn Creature* pattern from the encounter pattern catalogue to create an instance of it, as represented by the *E* line in Figure 1.

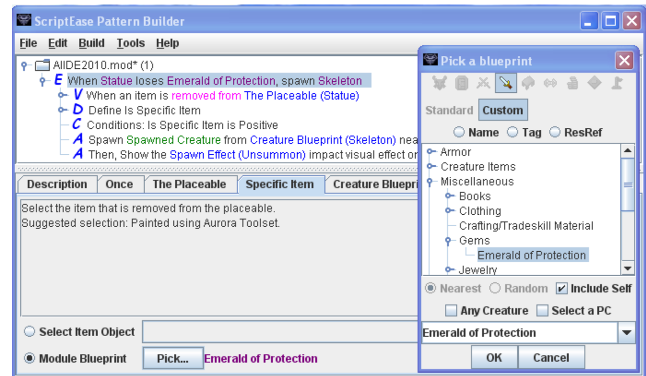


Figure 1 An encounter pattern in ScriptEase

Figure 1 shows the author selecting the *Emerald of Protection* as the *Specific Item* option from a list. The author has already selected the *Statue* as *The Placeable* option and *Skeleton* as the *Creature Blueprint* option. If the author wanted to adapt this encounter further by changing or deleting the visual effect or adding another action, the author could open this encounter, as shown in Figure 1. An encounter can contain definitions (*D*), conditions (*C*) and actions (*A*). To add an action, the author selects it from a list and sets its options. A case study (Carbonaro et al. 2008) showed that generative encounter patterns are a solution to the manual scripting problem for interactions between game creatures and game objects. It has also been shown that behavior patterns can specify which tasks are performed by NPCs and when behaviors can be initiated, interrupted, resumed and learned (Cutumisu et al. 2008). Dialogue patterns (Siegal and Szafron 2009) can be used to control conversation by generating scripts that determine when a line should be spoken. In this paper, we introduce quest patterns to control the plot in story-based games.

### Quest Patterns

A story-based game has a non-linear plot that forms a large decision graph that controls the potential decisions a player can make during the story. The graph is traversed as the story unfolds. We divide the quest graph into units called quests. A *quest* is single mission that the PC can or must complete and is divided into *quest points*. An in-game journal lists active and completed quests and has entries for each quest point to summarize quest progress and remind

the player what the PC should try to achieve next. For example, a *Defeat the Dragon* quest can have three quest points: *Converse*, to learn that the dragon is a menace; *Kill*, when the dragon is actually killed; and *Converse* again to report success. This example is represented by the *Exterminate* quest pattern, the first line in Figure 2.

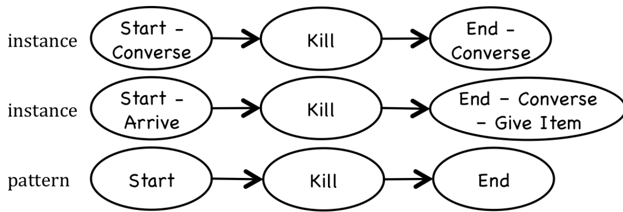


Figure 2 An exterminate pattern using meta quest points with a pair of possible instances

Figure 3 shows a ScriptEase textual quest representation, where *Q* denotes a quest, ▪ (bullet) denotes a quest point and the *Converse* quest point is open to reveal its components. A quest pattern guides a player through a quest by enabling quest points at the appropriate time. The first quest point (*Converse* in Figure 3) is enabled when the game begins, so the PC can converse with the quest giver at any time. An enabled quest point is reached when it either succeeds or fails. Each quest point has at least one encounter pattern that determines when a quest point succeeds (*E+* in Figure 3), and zero or more encounter patterns that determine when a quest point fails (*E-*). The *Converse* quest point in Figure 3 succeeds if the PC reaches a line of dialog and fails if the quest giver dies before the conversation occurs. Success and failure can each have a journal entry. If a quest point succeeds, its successor quest points are enabled. A quest point maintains a list of which quest points can enable it and how many of those quest points must succeed before it is enabled. For linear quests, when a quest point is reached, the lexically next quest point is enabled. When the *End* quest point succeeds, the quest succeeds. If a quest point fails then it enables no other quest points. If the *End* quest point cannot succeed, then the quest fails automatically.

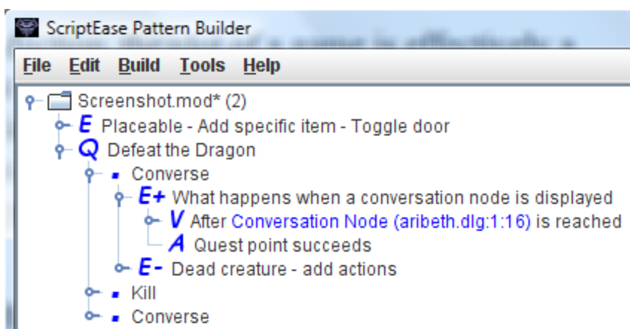


Figure 3 A ScriptEase *Exterminate* quest, with the *Converse* quest point opened to reveal its success and fail encounters

A quest point cannot be reached if it is not enabled. What happens if an encounter occurs before its quest point is enabled? For example, assume the PC kills the dragon before conversing with the quest giver. The *Kill* quest point has not been enabled when the dragon is killed. The generated script records that the quest point encounter occurred, but does not mark the quest point as reached, since it was not enabled. However, in this case, as soon as the PC *Converses* with the quest giver to receive the quest, the *Kill* quest point will be enabled. Since the success encounter has already occurred, the *Kill* quest point will be immediately reached, with success. Therefore, the PC can perform the encounters in the first two quest points of the *Exterminate* quest in the opposite order without breaking the quest. The separation of the concepts enabled and reached allows quests to progress even if the encounters in the quest points occur in an alternative order. ScriptEase automatically generates the complex scripts that determine at run-time whether any quest point is enabled, reached, successful, or failed. The author's task is reduced to specifying the success and failure encounter for each quest point and the enable relationships between quest points.

Quests are abstracted to increase reusability. The *Exterminate* pattern instance (line 1 of Figure 2) is very specific – it starts and ends with a *Converse* quest point. However, similar quests often start and end in a variety of different ways – the author may want the PC to arrive in a village and witness the menacing dragon leaving, and after the dragon is defeated a villager may reward the PC with some jewelry. In this case, the author may want to start the quest with an *Arrive* quest point when the PC enters the village and end the quest with the *Converse – Give Item* quest point shown as the second line in Figure 2.

Although the two quest instances in Figure 2 have only one quest point in common, they are conceptually similar. A *meta quest point* is an abstract quest point that can easily be adapted to one of a small set of intended choices or to any other quest point, if necessary. The *Start* and *End* quest points are actually meta quest points whose intended choices include *Converse* and *Arrive*, and *Converse* and *Converse – Give Item* respectively. The third line of Figure 2 shows the *Exterminate* pattern with meta quest points. Figure 4 shows an instance of the *Exterminate* quest in ScriptEase, where ¤ represents a meta quest point. The *Start* meta quest point has been bound to an *Arrive* quest point and the *End* meta quest point has not yet been bound. There are 17 meta quest points in the pattern library including *Discover* that has 7 common choices including: *Converse*, *Arrive*, *Approach* and *Use Placeable*.

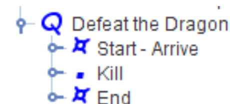


Figure 4 An Exterminate quest instance with meta quest points in ScriptEase

Many quests are non-linear. For example, suppose the dragon problem could also be resolved by persuading the dragon to leave the village. Figure 5 shows this more general quest with two branches: one contains a *Kill* quest

point and the other contains a *Verbal Skill* (e.g. intimidate, bluff, diplomacy, etc.) quest point. The notation indicates that the second *Converse* quest point is enabled when at least “1” of these two quest points succeeds.

The additional choices complicate the logic for determining if the quest fails or succeeds. Although the PC’s lack of skill may cause the *Verbal Skill* quest point to fail, the *Kill* quest point must also fail for the quest itself to fail. ScriptEase generated scripts support this logic by enabling the second *Converse* quest point when either quest point succeeds and by closing the quest if both fail.

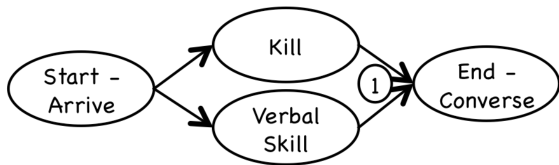


Figure 5 A non-linear quest

The plot may consist of many dependent quests. A *subquest* is a quest pattern that can be used as a single quest point in another quest. When a subquest succeeds/fails it succeeds/fails as a quest point in the superquest that contains it. For example, suppose the dragon cannot be persuaded to leave using a *Verbal Skill* and will only leave voluntarily if it receives a magical gemstone that the villagers stole from it. The acquisition and delivery of the gemstone can be represented by a *Retrieve* quest pattern. The dragon quest can then be represented by the *Do one of many* quest pattern shown in Figure 6, where subquests are represented by rectangles.

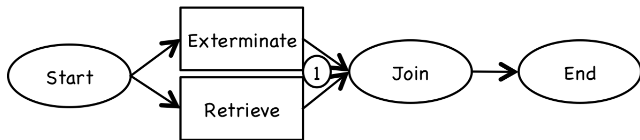


Figure 6 A do one of many quest with subquests

The *Join* quest point succeeds as soon as the required number of enabling quest points succeed. Changing the number of enablers from 1 to the number of entering arcs converts a quest from a “do one of many” to a “do all”. Supporting an arbitrary number of enablers gives even greater utility, such as “do 2 of 4”. The ScriptEase textual view of this pattern, with expanded subquests is shown in Figure 7. The ScriptEase textual tree view does not explicitly display the branches. However, viewing the enablers of the *Join* quest point shows that it is not linear.

Figure 7 Subquests and non-linear quests in ScriptEase

## User Study

A previous user study showed that quest patterns can be used to randomly generate side quests (Onuczko et al. 2008) that were judged as good as manually constructed side quests (Onuczko 2007). Another study demonstrated that high school students are capable of using encounter patterns to insert player-object interactions into NWN stories (Carbonaro et al. 2008). In this paper, we present user study results that measure the ease-of-use and reliability of quest patterns compared to manual scripting of quests. University students scripted NWN quests using both ScriptEase and the native text-based NWScript.

The user study had an incomplete repeated measures design with four phases. Participants were randomly divided into two equal-sized groups: one group used quest patterns first and the other group used NWScript first. In the first phase a demographics survey identified participant age, field of study, and prior use of ScriptEase and NWScript. In the second phase, each group used one tool. In the third phase each group used the other tool. In the fourth phase, we recorded participant preferences.

There were two study objectives: which tool produced scripts more effectively (in a shorter time with fewer errors) and which tool was preferred (easier-to-use). For effectiveness, we measured the number of quests completed, the number of completed quests that functioned correctly, and the amount of time needed to complete quests. Preference was determined by compiling survey responses about individual quests and about overall experience. Each preference question used a 5-point scale. Each quest had a preference question concerning ease-of-use, speed, and ease-of-debugging. There were two overall preference questions: which tool was preferred during the study and which would be preferred for future use.

The participants were screened for familiarity with the concepts needed to complete the study. All but one of the twenty-three participants were computing science or engineering students at the University of Alberta. The participants were required to know at least one C-like programming language, to shorten the time needed to learn NWScript. Participants had an average of 5.3 years of programming experience with a range of 1 to 30 years. Familiarity with computer games was required to reduce the time needed to test scripts in the game.

All participants had played at least one story-based game, and 78% had played seven or more story-based games. In fact, 52% of the participants had played NWN before. A smaller percentage, 21%, had previously used NWScript and 17% had used ScriptEase (without quest support). The participants were familiar enough to know what capabilities to expect from the tools but not familiar enough to know the tested tools thoroughly.

Participants were instructed to script eight quests with each tool. Except for the quests, the story was otherwise complete: all required terrain, objects, and conversations were present. We assumed that eight quests were more than a participant could complete in the 175 minutes allotted for each tool. A tutorial with an example quest was

provided with each tool. The quests were constructed in a specified order. The same order was used for each tool. The quests varied in complexity from killing zombies to implicating a tax collector by planting contraband in his possessions. After each quest was completed with the second tool, participants marked which tool they preferred for that quest. After a time limit was reached for each tool they completed an overall preference questionnaire.

The order of quests was chosen to measure specific characteristics of the tool. The complexity of each successive quest increased, except for the final quest. The first quest was a simple linear quest. The second quest was simple and linear, but the pattern catalogue lacked the corresponding quest pattern to force the participants to adapt a different quest pattern. The third quest was simple but could be naturally completed two different ways. The fourth through six quests were linear but successively increased in length. The seventh quest was linear with two subquests. The final quest was simple and similar to the first quest to try to distinguish between those who ran out of time and those who could not understand how to create the most complex quest.

## User Study Results

Figure 8 shows the number of participants who completed each quest with each tool. Six participants completed the third quest with quest patterns and seven did so with NWScript, There is no statistically significant difference between the average number of quests completed with each tool (1.74 for ScriptEase and 2.17 for NWNScript).

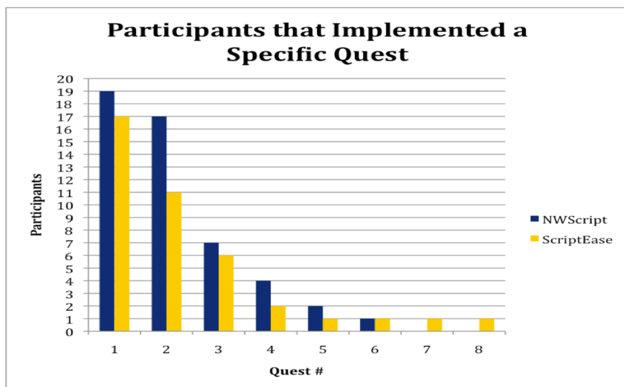


Figure 8 Number of participants who completed a specific quest

Quest patterns produced more reliable quests overall (Figure 9). However, for quest 1, quest patterns were less reliable than NWScript, perhaps due to a steeper learning curve. All participants were experienced programmers, but only 17% had used ScriptEase before. Note, 30 of the 40 quests completed using quest patterns functioned correctly, compared to 26 of the 50 written in NWScript. The percentage of correct quests per participant had a mean of 75.25% for quest patterns and 47.89% for NWScript, which is significantly different. A one-tailed T-test, assuming unequal variance of the samples, produces a p-value of 0.0122, well within statistical significance at the 95th percentile. Scripts generated by quest patterns were 1.57 times more reliable than scripts written in NWScript. For quests 2 and 3 the multipliers were 2.43 and 4.67 respectively. Even with fewer samples the reliability difference between the tools for quests 2 and 3 is significant at the 95th percentile; a one-tailed T-test for quests 2 and 3 produced p-values of 0.0001021 and 0.03482 respectively. Neither tool was statistically more reliable for quest 1. The most frequent bugs found in NWScript quests originated from quest events being completed in an unintended order.

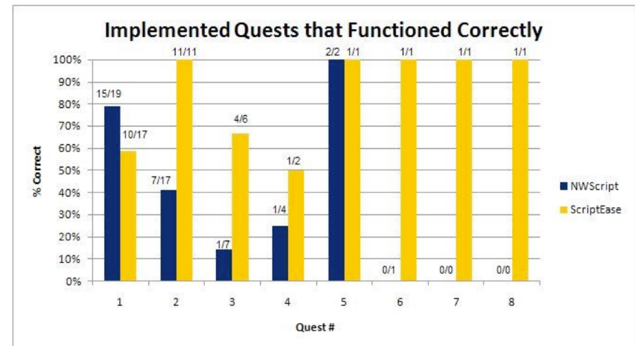


Figure 9 Percentage of completed quests that functioned correctly

Intuitively, Figure 10 shows that after a sizeable learning curve (time spent on the tutorial plus time used to implement the first quest), quest patterns are quicker to use. However, this assertion cannot be supported statistically. The mean time for this learning curve was 129 minutes for quest patterns, compared to 90 minutes for NWScript. Using a one-tailed, unequal variance T-test – this difference is statistically significant at the 95th percentile with a p-value of 0.000394. On a quest-by-quest basis, NWScript was 61% faster for quest 1 with a p-value of 0.000685, quest patterns were 42% faster for quest 2 with a p-value of 0.00384, and no tool was significantly faster for quest 3 (the p-value was 0.379). Further quests were not analyzed since so few participants implemented those quests. Even though NWScript was quicker to learn, it did not result in quicker times for later quests.

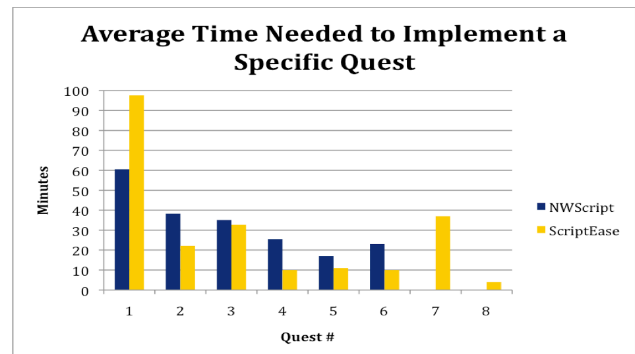


Figure 10 Average minutes needed to complete an individual quest

Z-tests were used for calculating preference, with a hypothesized population mean of neutral preference and using the sample's standard deviation. Two-tailed Z-tests showed that participants were neutral about tool preference (Figure 11) for the first (learning) quest (p-value of 0.959), and overall (p-value of 0.486). However, after the first quest they significantly preferred quest patterns. The one-tailed Z-test on the preferences scores for quests 2 through 6 shows quest patterns were preferred at the 95th percentile with a p-value of $2.042\times10^{-12}$.

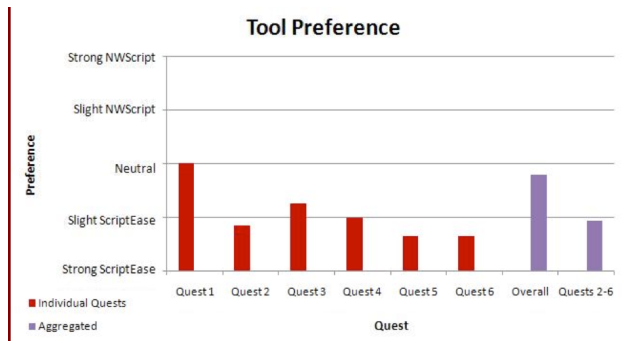


Figure 11 Tool preference overall and by quest

We identified two important factors in the design of the study. First, participants commented that they only felt comfortable with the tools near the end of the allotted time. The study should have been longer to reduce the impact of the learning curve and allow more data to be gathered. Participants preferred quest patterns for the later quests, once they became familiar with the interface. The second factor is that participants were "fooled" into thinking that they had finished a quest with NWScript, even though it did not meet the specifications. Therefore they proceeded to the next quest before actually finishing. This artificially reduced the times for NWScript quests and allowed them to implement more quests with NWScript. If a verifier checked each implemented quest then it would have caught the bugs in the quests, slowing the development of the buggy NWScript quests but increasing the percentage of correct quests. Since the reliability of the respective approaches was part of the test, no such verifier was provided. The fact that quest patterns automatically generate correct scripts to prevent many of these bugs indicates that quest patterns are more robust and in designing real games, no verifier is available.

## Conclusion of User Study

Generative quest patterns were easier to use and more reliable than manual scripting. Quest patterns were significantly preferred after the learning phase (completion of the tutorial and first quest). Significant results favouring quest patterns were found in quest reliability, both overall and for quests 2 and 3 specifically. The data showed that quest patterns were 1.57 times more reliable (when correctness is binary) than using manual scripting. More specifically, when an appropriate pattern was not available for a quest, quest patterns were still 2.43 times more reliable and 42% faster than using manual scripting, supporting the assertion that quest patterns are easily adaptable. Another impressive result is that quest patterns were 4.67 times more reliable for the quest involving multiple branches, showing that automated management of quest progression simplifies the process of creating more complex quests. In contrast, participants implemented more (but not by a statistically significant margin) quests using NWScript, (the T-test yielded a p-value of 0.194); however, they frequently suffered from bugs.

For a commercial video game, more reliable scripts reduce the cost of testing, especially for story-based games with branching plots. Therefore, the superior reliability and preference for quest patterns supports our assertion that they are more effective than manual scripting.

# Conclusion

Generative design patterns provide reliability and efficiency by automatically generating scripts for frequently used domain concepts. A study was conducted to determine if generative design patterns could be used to support quests for story-based games. The study provided some evidence that generative quest patterns are effective.

It showed that quest patterns generated more reliable scripts than manual scripting and that adapting patterns was a viable alternative to manual script writing. After a higher learning curve, participants also preferred to use quest patterns over manual scripting. Overall, quest patterns were more effective than manual scripting.

This study supports the use of quest patterns in story-based games. They will reduce the bottleneck in scripting computer games since they eliminate the requirement of programming knowledge for game designers.

# References

Carbonaro, M., Cutumisu, M., Duff, H., Gillis, S., Onuczko, C., Siegel, J., Schaeffer, J., Schumacher, A., Szafron, D. and Waugh, K. 2008. Interactive story authoring: A viable form of creative expression for the classroom. *Computers and Education 51 (2)*, 687-707.

Cutumisu, M., and Szafron, D. 2009. An Architecture for Game Behavior AI: Behavior Multi-Queues. In *Proceedings of Artificial Intelligence and Interactive Digital Entertainment*, 20-27.

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

Onuczko, C. 2007. Quest Patterns in Computer Role-Playing Games. MSc Thesis.

Onuczko, C., Szafron, D., and Schaeffer, J. 2008. Stop Getting Side-Tracked by Side-Quests. In *AI Game Programming Wisdom 4*, Editor S. Rabin. Charles River Media, 513-528.

ScriptEase. 2009. http://www.cs.ualberta.ca/~script/.

Siegel, J., and Szafron, D. 2009. Dialogue Patterns - A Visual Language For Dynamic Dialogue. *Journal of Visual Languages and Computing 20 (3)*, 196-220.