This is a pre-print of a paper that will appear in *Science of Computer Programming*, *Volume 67, Issue 1*, *1 June 2007*, *Pages 32-58.* The final version can be obtained from: *ScienceDirect.com*

ScriptEase: A Generative/Adaptive Programming Paradigm for Game Scripting

Maria Cutumisu[a], Curtis Onuczko[a], Matthew McNaughton[a], Thomas Roy[a], Jonathan Schaeffer[a], Allan Schumacher[a], Jeff Siegel[a], Duane Szafron[a, *], Kevin Waugh[a], Mike Carbonaro[b], Harvey Duff[b], Stephanie Gillis[a]

[a] *Department of Computing Science, University of Alberta, Edmonton, AB, T6G2E8, Canada*

[b] *Faculty of Education, University of Alberta, Edmonton, AB, T6G2G5, Canada*

Received:

Abstract

The traditional approach to implementing interactions between a player character (PC) and objects in computer games is to write scripts in a procedural scripting language. These scripts are usually so complex that they must be written by a computer programmer rather than by the author of the game story. This interruption in the game story authoring process has two distinct disadvantages: it increases the cost of game production and it introduces a disconnect between the author's intentions and the interactions produced from the programmer's written scripts. We introduce a mechanism to solve these problems. We show that game authors (non-programmers) can generate the necessary scripts for implementing meaningful interactions between the PC and game objects using a three-step process. In the first step, the author uses a generative pattern (concept) to create a high-level description of a commonly occurring game scenario. In the second step, the author uses a standard set of adaptation operations to customize the high-level description to the particular circumstances of the story that is being told. In the third step, the author presses a button that automatically generates scripting code from the adapted pattern. We describe the results of three studies in which a combined total of 56 game story authors used this three-step process to construct *Neverwinter Nights* game stories, using a tool called ScriptEase. We believe that this generative/adaptive process is the key to future game story scripting. More generally, this paper advocates the development of adaptive programming as an alternative to current constructive programming techniques, as well as the application of adaptive programming in many domains.

Keywords: Generative pattern, Computer game, Scripting language, Adaptive programming, Game scripting, Game authoring, Game agent.

[*] Corresponding author. Tel.: 1-780-492-5468; fax: 1-780-492-1071.
E-mail address: duane@cs.ualberta.ca.

1. Introduction

World wide, the annual sales in computer games (hardware and software) is many tens of billions of dollars (US). The games industry is often compared to the movie industry, based both on financial aspects (the industries are similar in size) and public attention. Upcoming game titles are hyped for months in advance of their release and the debut of a major game can garner massive media attention. This public attention comes with increased consumer expectations. The technological and creative sophistication required to produce a big seller in today's discerning market continues to increase. Game development cycles of three to four years are quite common and many games have budgets in excess of $25 million (US).

Unlike the movie industry where actors directly affect the commercial success and are compensated accordingly (A-list actors receive more than $10 million per picture), in the games industry most of the cost is in content creation: writers to design the game story, artists to create the visual effects, and computer scientists to build the technology and tools needed to realize the game. In many ways, it is more difficult and labor-intensive to build a blockbuster game than a blockbuster movie.

Most of the effort in game development increasingly revolves around content creation. A major aspect of content is the game story. This is a multi-faceted component, which includes creating the storyline, sub-plots, characters, and all their interactions. The situation is also complicated by the interactive nature of games. Unlike movies with their linear story lines, in state-of-the-art games, the player can influence what happens in the story (to a greater or lesser extent, depending on the game). Non-linear story lines considerably complicate the content creation process, since the author must anticipate all possible player actions and handle them in a meaningful manner. The sophistication of the game stories (in particular in game genres that depend on a good story, such as role-playing games) and the growing need for non-linearity in the story lines are increasing the cost of game development (not just in the content creation side, but also in other aspects, such as quality assurance and playability testing).

Game companies have identified content as a significant bottleneck in game delivery. Companies construct or buy a set of good tools for creating 3D content, such as artwork, user interfaces, or game story levels. However, none of these tools automates the process of writing the scripts that enable these objects to interact with each other and with the PC. For any complex scripting that cannot be accomplished using available components, the user has to manually write code. This is done in a custom scripting language such as VSL (Virtools Dev [41]) or in languages such as Lua [26] (Anark Gameface [3]) or C/C++ (Gamebryo [18] and 3D GameStudio [1]). Even in Alice [2], where the scripting issue is addressed, there are no higher-level constructs to generate code. As a consequence, if no appropriate methods already exist, all code must be written manually in Python [31] [14].

Creating game-story-related content and translating this into the program code necessary to create the desired behaviour at game-play time has always been a bottleneck in the game-development process. For example, when an author creates some content, it must be specified precisely to the programmer who then writes code to implement this content in the game. This process is fraught with errors: the author might make errors in the specifications given to the programmers and/or the programmers might make errors in implementing the author's specifications. Ideally, the programmer should be eliminated to avoid these errors. One of the dreams of game development is for the creative authors to specify the game content without having to rely on programmers to implement their vision.

The need to simplify the scripting process has long been known in the games industry. This has led to the development of a number of tools aimed at simplifying the process. For

example, scripting languages have become ubiquitous in this context. They offer a "higher level" abstraction over a standard programming language (C++ being the usual default). The hope is that use of this language would simplify the content implementation processes, thereby reducing programming time, errors, and quality assurance efforts. Common scripting languages, such as Python, have been adopted by some companies for AI scripting, such as *Vampire: The Masquerade – Bloodlines* by Troika Games [39], *Battlefield 2* by Electronic Arts [15], *Civilization 4* by Firaxis Games [17], and *Backyard Hockey* by Humongous Entertainment (now Atari) [21]. Other companies use their own in-house language (e.g. NWScript, developed for BioWare Corp.'s *Neverwinter Nights* [5], and subsequently used for other games). Custom languages, of course, are expensive to develop. In contrast, Lua is a freely-available programming language that is used for extending applications, and is often used for scripting. Games that use Lua for Artificial Intelligence (AI) scripting include *MDK2* by BioWare Corp. [5], *FarCry* by Ubisoft [40], *Painkiller* by People Can Fly [30], and *Homeworld 2* by Relic Entertainment [32]. Epic Games [16] built a visual programming language to overcome the content specification gap. Kismet allows the user to draw and annotate flowcharts. Unfortunately, the tool is proprietary [23] and its low level focus makes it difficult to use, especially when a large number of objects is involved. All of these tools simplify the translation of specifications into code, but do not solve the fundamental problem of requiring authors to interface with programmers (with the possible exception of Kismet; public information is scarce). In an ideal world, a tool would allow an author to specify and test games directly, without using programmers as executors of intentions.

ScriptEase is a tool that attempts to remove programmers from the content specification process. The goal is that authors (content creators) should be able to create game stories without knowing how to program and without having to express themselves through programmers. The tool was inspired by the introduction of design patterns as an effective tool for general software design [19]. However, a twist is required. Traditional design patterns are descriptive in nature; the specifications still need to be manually turned into code. Instead, we use generative design patterns – patterns that can automatically be translated into executable code [8] [13]. Other researchers have discussed the problems of using generative design patterns [7], but have not generated code in the context of computer role-playing game (CRPG) story creation. From an author's point of view, specifying game content (interactions between the player character and the game objects) is a simple three-step process:

1. Select an appropriate pattern and create an instance of it. A pattern represents a familiar concept or idiom in a context (e.g., story). ScriptEase provides a rich set of patterns for role-playing games. These patterns can be used for specifying the plot, dialogues, character/object interactions, and character behaviours. For example, when an item is removed from a magic chest, the author might want to have a creature spawned nearby. This can be done using the pattern *Container disturb – spawn creature*. Instantiation of a pattern generates a high-level natural language description.
2. Adapt the description. Each description must be customized to match the intent of the author for the specific story being told. The author adapts the description by selecting appropriate story information using menu selection and dialog boxes. For example, for the *Container disturb – spawn creature* pattern, the author can specify the container (the magic chest), the creature spawned (the monster), and a visual effect displayed when the creature is spawned near the container. The simplicity of the adaptation process is a key to success.

3. Generate scripting code. The author presses a button and the scripting code is generated automatically from the adapted description. Most authors never need to (or want to!) see the scripting code. However, the generated code is made available (easily readable and fully commented) and may be customized further by programmers.

Most of the tools designed to aid content development use components, rather than design patterns, to create content. For example, some tools use 3D components (building blocks) or libraries of pre-made scripts to help users create worlds quickly by combining these components. Employing design patterns versus components is similar to employing frameworks versus libraries. Design patterns and frameworks provide the "most difficult to write" scripting glue that ties the components or library calls together.

This methodology was first introduced in the context of parallel computing [28]. The CO_2P_3S system allows a programmer to convert a sequential program to a parallel one, by selecting parallel patterns such as meshes and pipelines. After specifying some options to adapt the patterns to the application at hand, CO_2P_3S generates a framework of Java code that encapsulates all of the parallel program control. The user then adapts the framework, mostly by sub-classing to add the sequential code. The ScriptEase approach is fundamentally the same except for two differences. First, with ScriptEase, adaptations are done using a very high-level language. The user selects patterns and pattern components using menus, and a natural language description of definitions, conditions and actions is created. This is programming, but not in the traditional sense. With CO_2P_3S, some of the adaptation is done by adapting program code, written in a general purpose programming language – Java. Second, with ScriptEase, all adaptation is done before code generation, using the higher level non-traditional ScriptEase programming language instead of the NWScript target language. With CO_2P_3S, some adaptation is done before code generation that determines which framework is generated from a pattern. Some adaptation is done after code generation by adding subclasses and Java methods to the framework that was generated. The key similarity between these approaches is that in both cases, the generative pattern generates the control flow and complex code and the user makes small simple adaptations. The user adapts the generated artifacts instead of constructing artifacts. It is much easier to add actions or conditions to a generated pattern in ScriptEase than creating the actions and conditions. It is much easier to add small amounts of sequential Java code in CO_2P_3S, rather than writing the complex code that implements the parallel algorithms. Generative design patterns are a hot topic in the software engineering community today [13]. To the best of our knowledge, ours is the only effort aimed at applying design pattern technology to the most difficult content creation problem in the computer games industry – script creation. Design patterns have been used in describing the rules and structure of games, both computer and traditional, but not to generate content [6]. Descriptive (not generative) patterns have also been used to build game engines [24].

ScriptEase is only as useful as its pattern catalog. Although we provide a rich set of patterns to authors, there will always be the need for authors to design their own patterns. ScriptEase includes a tool for designing new patterns. The Atom Editor is the interface between the implementation language (NWScript in this case) and the pattern building blocks. The Pattern Designer allows the user to combine and customize building blocks into new patterns using the same menu-driven techniques that are used to adapt existing patterns. There are two secrets to making the generative pattern approach work. First, there must be a *rich enough set of patterns* that an author can use to express their ideas in a simple, intuitive manner. This requires that ScriptEase include three tool components:

1. a story writing tool that allows authors to select and adapt patterns

2. a pattern designer tool for authors that allows them to create new patterns from a set of simple pattern components;
3. an atom builder tool that allows programmers to create primitive building blocks that can be used to assemble pattern components, for a specific game engine. In this case, the user needs to be a programmer, since the atom builder tool constitutes the interface between the ScriptEase language and the target language. The creation of a new atom is only needed when a game author identifies a capability that cannot be met by existing patterns.

As shown in Fig. 1, ScriptEase has a modular architecture. Our pattern development has been targeted at the role-playing game genre and our tool currently generates code for *Neverwinter Nights*. However, the tool supports the design of patterns for other genres with no changes and could be modified to generate scripting code for other target architectures (game engines). The patterns have to be generative, which means the tool needs to support translation of pattern components into code for the underlying game scripting language. We implement this idea using a collection of atomic elements to implement each pattern component. Each *atom* is responsible for generating scripting code to represent itself. In our case, we support NWScript, BioWare Corp.'s scripting language, but there are no impediments to generating code for other languages/game engines by providing alternate implementations of atoms to generate code for a different scripting language.

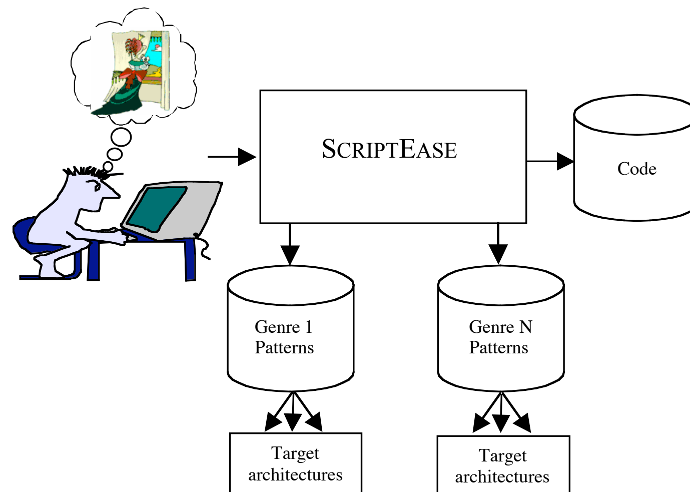

Fig. 1 The ScriptEase architecture

The second requirement is that *the pattern adaptation mechanism must be simple enough for authors (non-programmers) to use*. Requiring authors to perform pattern customization by modifying traditional scripting code (Python, Lua, NWScript, etc.) is not a viable option.

To show that non-programmers can use ScriptEase to create engaging games, we validated our work with three Grade 10 English classes, a summer camp for high school students [34], and a group of first year University students. In each case, the students had none or very limited programming experience. However, they were able to quickly learn the necessary tools (BioWare Corp.'s *Neverwinter Nights* game, BioWare Corp.'s Aurora Toolset, and ScriptEase) and author their own game stories. In this paper we present data from two case studies that each involved one of the Grade 10 high school classes and a third case study that involved the

University students. We show that these authors (with no/limited programming experience) used a broad range of ScriptEase patterns and adaptation operations to produce complex stories. They were able to do this given only a few days to learn the tools and write a story. The analysis gives insight into why these students could succeed without traditional programming skills. The students performed pattern adaptation using operations that could be viewed as a new type of programming called *adaptive programming*, rather than using traditional *constructive programming*.

This paper makes the following contributions:

1. The concept of *adaptive programming* is introduced and differentiated from traditional *constructive programming*. Adaptive program operations are defined for customizing patterns and categorized by conceptual level (the difficulty of applying the adaptive operations).
2. Our case studies show that adaptive programming is simpler than traditional constructive programming in a specific application domain – game story writing. This is shown by having non-programmers author complex game stories (from the programming point of view) with less than a week of total effort – less time than it would take to learn how to program in a traditional language.
3. Our case studies provide initial evidence to support a preliminary rating of the relative conceptual difficulties of using our various adaptive program operations.

Section 2 discusses the generative design patterns in ScriptEase. Constructive programming is contrasted with adaptive programming in Section 3. In Section 4, pattern adaptation is discussed using ScriptEase. Section 5 presents our case study results. Section 6 provides conclusions and future work. We identified four types of patterns: encounter, behaviour, plot, and dialogue. All four kinds of patterns are fundamentally event-based, since that is the nature of the computer role playing games (CRPG) domain.

Programming is undergoing a fundamental change: we envision end-users who can successfully program without writing code. The novelty of this article is its new perspective on the future of programming, generative/adaptive, in which users can solve complicated problems by themselves not by writing code at a higher level of abstraction, but by generating code from concepts (patterns) and adapting the concepts to their specific context. In the short term, the practical application of this work will benefit game story authors who must create and test non-linear stories quickly. However, the consequences of this new approach have the potential to affect problem-solvers in many domains.

2. Generative Patterns

Game story authors are usually not programmers. Their expertise is in creating and detailing storylines, and all the intricacies of (non-)linear plots. The level of detail needed includes creating the story, the (non-)linear lines along which the plot can develop, the scenes, and the scene outcomes. For each scene in the story, the author must define the characters and objects that populate that scene, the interactions between characters and their environment, dialogues, and any side effects that occur. Of course, all this information must be specified precisely, for subsequent translation into a programmed implementation. The strength of a game story author is typically in the creative process; the precision needed to translate a story into a programming specification is both unwelcome and labor intensive. Ideally, one wants to eliminate the intermediary so that the author can directly specify a story using a tool that can automatically translate the specifications into code.

Game story authors are story tellers. Hence, any story specification tool must interface with the story author in a language that they understand (C++ is not appropriate). A natural paradigm is patterns—frequently occurring themes. A tool that supports a rich pattern library can be a natural bridge between the "sentences" of a written story and the "statements" of a programmed implementation.

ScriptEase supports four categories of patterns:

1. Plot. Stories are rich with frequently occurring themes (e.g., "rescue the heir to the throne from the clutches of the villains").
2. Character/object interactions (called encounters in ScriptEase). Game scenes are populated with characters and objects; their interactions are usually well defined. For example, consider a trap door pattern. The typical scenario involves the player character (PC) stepping on a specified place (a trigger), the trap door opens, and the PC falls in.
3. Dialogue. Many types of conversation frequently occur (e.g., asking for directions or ferreting out clues).
4. Behaviour. Scenes are populated by characters that interact with the PC. These characters are often called non-player characters (NPCs). These NPCs need to have their behaviour specified. For example, ascribing the behaviour "guard" or "shopkeeper" to a character immediately implies their expected behaviour.

ScriptEase patterns are applied to NPCs and other game objects. BioWare Corp.'s Aurora Toolset is used to define and populate scenes. The Aurora Toolset is a drag-and-drop CAD tool for creating game worlds. It provides a rich palette of interior and exterior map tiles, objects, creatures, etc. for creating the environments in which the game story will unfold. The tool is intuitive and easy to use. ScriptEase makes the scripting of game objects as easy as the Aurora Toolset makes the creation and placement of game objects. Fig. 2 shows an author placing a container (named *Dresser*) into a room; it will be used in the example of this section and it comes from one of the stories written by a student author in the case study of Section 5.

A story is written by instantiating patterns and adapting the generated descriptions. In the example story, the author wants the PC to open the *Dresser* shown in Fig. 2. When the PC opens it and removes an item from it, the author wants a creature named *Avadel* to be spawned. The author created *Avadel* using the Aurora Toolset (not shown). A similar scenario occurs frequently in role-playing fantasy games (and stories). However, it involves some other container rather than the *Dresser* and some other creature rather than *Avadel*, since these objects are specific to this story. Because this scenario occurs so often, ScriptEase has a pattern for it: *Container disturb – spawn creature*. To use the pattern in a particular game story, the author must adapt the pattern by specifying three options (parameters): the container that the pattern applies to, the creature that gets spawned when an item is removed from the container, and the visual effect (if any) that occurs during the spawning. Fig. 3 shows the generated description of this pattern and one of the options being set by the author.

The author has opened a story file (*MyShortStory*) created using the Aurora Toolset and has created an instance of an *encounter* pattern (identified by the stylized E), *Container disturb – spawn creature*. The author has selected the pattern and four tabs have appeared. If the *Description* tab was selected (it is not), the author would see a summary of the intent of the pattern. The other three tabs are option tabs (*The Container*, *Creature Blueprint* and *Spawn Effect*). The author has adapted this pattern to the story by selecting the *Creature Blueprint* option using a dialog box to select a particular creature named *Avadel*. The author has previously adapted the pattern by setting *The Container* option to the *Dresser* and the *Spawn Effect* to be a *"Pulse, Holy"* visual effect, using similar dialogs.

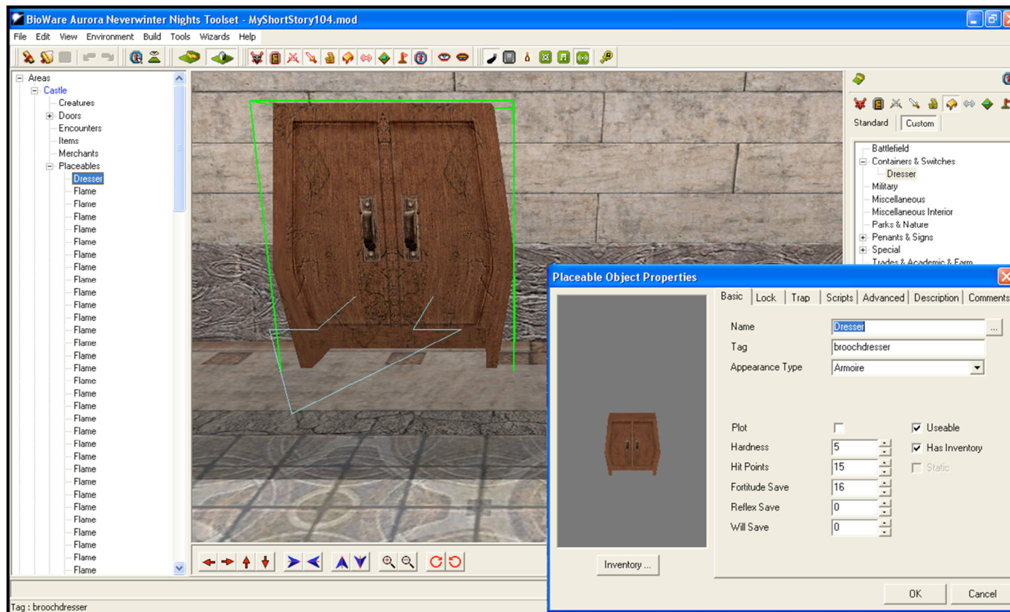

Fig. 2 Creating and placing a container using the Aurora Toolset

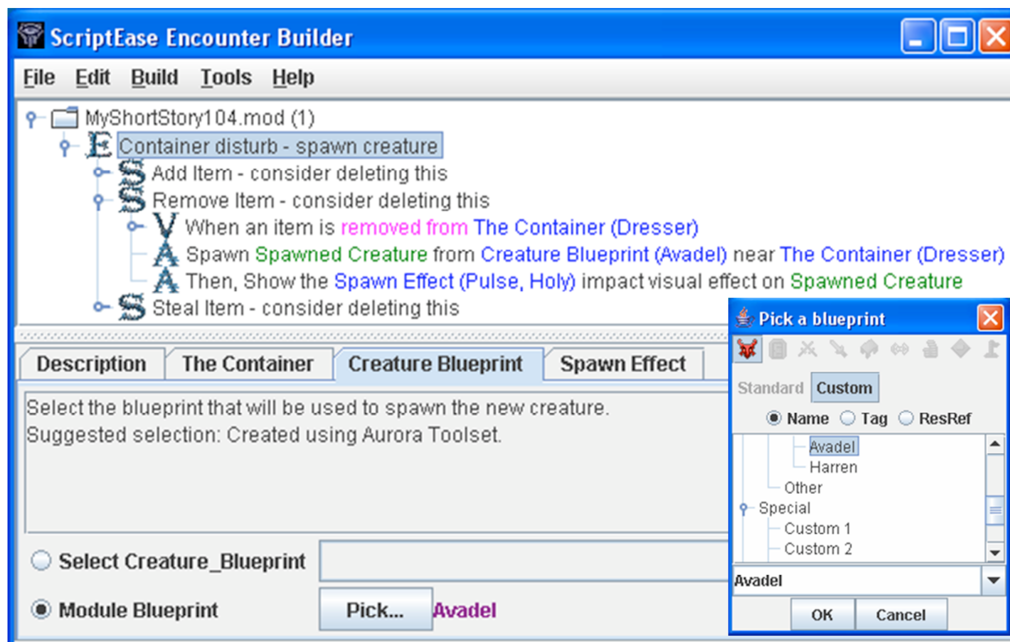

Fig. 3 A generative pattern, its description and a dialog being used to set an option

Every encounter pattern contains one or more *situations* (stylized S). This encounter pattern is shown in Fig. 3 and it has been opened to reveal its three situations. One applies when an item is added to the container, one applies when an item is removed and one applies when an item is stolen. In Fig. 3, the *Remove* situation is expanded to show its components. Every situation contains one *Event* (stylized V) that describes the circumstances under which the situation applies. The *Remove* situation applies when *an item is removed from The Container (Dresser)*. Every situation contains one or more *actions* (stylized A) that will be performed when the situation applies. The *Remove* situation contains two actions, one to spawn a creature and the other to show a visual effect. A situation can also contain two other kinds of components, definitions and conditions, which will be discussed in Section 4. Note the use of colors in the text to clearly indicate the parts of the natural language description that are part of the pattern (black), options (blue), conditions (red), and definitions (green).

As the author adapts the pattern by setting options, the description is updated to reflect the choices. The description in Fig. 3 has been adapted for the author's story as the pattern options were set. If the author selects *Generate Code* from the menu, the NWScript code for the pattern would be automatically generated and inserted into the story file at the appropriate place. Fig. 4 shows a portion of the 58 lines of BioWare Corp.'s NWScript code that was generated for this adapted pattern description. A game author would never need to see the automatically generated commented code. However, a programmer on the team may choose to view the generated scripts and to optionally edit the scripts manually.

Edit

```
void RemoveItemconsiderdeletingthis_1() {
  // The following are all of the variables used in this situation
  object SpawnedCreature_SE3;
  object DisturbedItem_SE2;
  object Dresser_SE0;
  object ContainerDisturber_SE1;
  // This script is attached to the following object's OnDisturbed script slot
  Dresser_SE0 = OBJECT_SELF;
  // When an item is removed from Dresser
  if( ! SE_Ev_ContainerOnDisturbed(Dresser_SE0, INVENTORY_DISTURB_TYPE_REMOVED) ) return;
  // Define Container Disturber as the creature that just removed from Dresser
  ContainerDisturber_SE1 = SE_Df_ContainerDisturber(Dresser_SE0, INVENTORY_DISTURB_TYPE_REMOVED);
  // Define Disturbed Item as the item that was removed from Dresser
  DisturbedItem_SE2 = SE_Df_DisturbedItem(Dresser_SE0, INVENTORY_DISTURB_TYPE_REMOVED);
  // Main code body - checks conditions and executes actions
  // Spawn Spawned Creature from Avadel near Dresser
  SpawnedCreature_SE3 = SE_Ac_SpawnCreatureNearObject("bandit003", Dresser_SE0);
  // Show the Pulse, Holy impact visual effect on Spawned Creature
  SE_Ac_ShowImpactVisualEffectOnCreature(VFX_IMP_PULSE_HOLY, SpawnedCreature_SE3);
}

void StealItemconsiderdeletingthis_2() {
```

Fig. 4 A portion of the code generated for the pattern in Fig. 3

In this story, the author wants the actions to occur only when an item is removed from the *Dresser*. Although adding and stealing from a container are other possibilities supported by the (general) pattern, the author does not want to use them. Therefore, in addition to setting the pattern options, the author can further adapt the pattern description by highlighting the *Add* and *Steal* situations, one at a time, and selecting the *Delete* option from a pop-up menu, before generating the scripting code.

Pattern descriptions can be adapted (customized) by setting options, adding, deleting or replacing pattern components. We have already seen that deleting situations is easy to do. Adding an action to a situation is also simple in ScriptEase. For example, the author could add an action to the existing pattern so that the caption "Run for you life!" is displayed above the spawned creature's head. The author clicks to highlight the *Remove* situation, selects *add an action* from a pop-up menu and navigates a set of hierarchical menus to find the desired action. Fig. 5 shows the results of the author's efforts. Note that the *Add* and *Steal* situations have been deleted. The resulting pattern description would generate 38 lines of NWScript code (not shown). After generating the scripting code, the author can "test-drive" the story by opening it in *Neverwinter Nights* (*NWN*). This allows the author to incrementally change the story and immediately test the new story to verify its correctness.

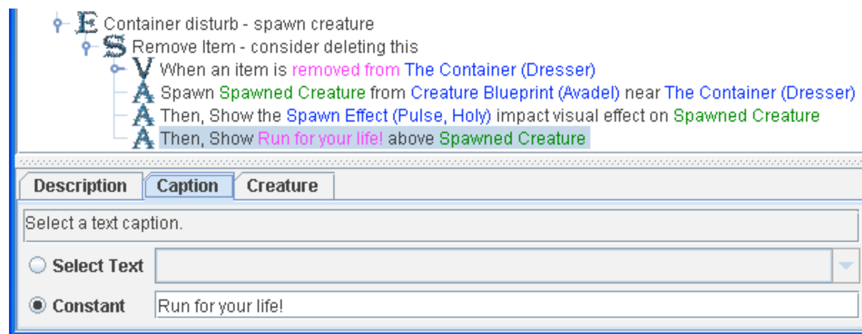

Fig. 5 Adapting a pattern by adding an action

The *Container disturb – spawn creature* example shows how easy it is for an author (non-programmer) to use ScriptEase to create an interactive game story. The ScriptEase three-step approach (instantiate, adapt, generate) provides many benefits for simplifying the creation of interactive stories:

1. All authoring is done using familiar story-element patterns.
2. The author does not need to know anything about programming or scripting languages.
3. The author sees a natural language description of the story.
4. The author adapts the story using a simple menu-driven interface.
5. Many common programming errors are eliminated. The patterns have been tested and debugged by the pattern designer, before the author uses them, not by the author during story writing.
6. Enabling an author to directly generate an interactive story eliminates the programmer as an intermediary and as a potential source of errors.
7. Since there can be thousands of objects that are scripted in a story, each requires a unique label at the script level. ScriptEase manages these transparently using natural language pronouns to refer to objects in context. Hiding these unique labels from the author reduces complexity and allows the author to work at a higher level of abstraction. By reducing the effort required to add scripts, the author has more time to add additional interactive (scripted) NPCs and objects, thereby increasing the richness of the story.
8. Adapted patterns are used to generate scripting code. This code does not have to be viewed by the author. However, it is available for further adaptation by programmers.

ScriptEase does not provide post-facto validity checks to ensure the generation of a valid story. Instead, each of these types of patterns is responsible for a different aspect of a story and ensures that its own correctness responsibilities are met. For example, each encounter pattern represents a correct solution to a game scenario. A *Placeable use/death – toggle door* pattern encapsulates the notion that when the placeable (e.g., a lever) is used (pulled), if the door was unlocked, it is closed and locked. If the door was locked, it is unlocked and opened. A common problem that can occur in this situation is for the player to destroy the lever rather than using it to open the door. In this case, the story is broken, since the door can never be unlocked. However, our pattern ensures that when the lever (placeable) is destroyed, the door is unlocked and opened, so that the PC can continue the adventure. Rather than trying to apply a set of constraints post-facto, after the author makes an error using a free-form constructive system, our approach implicitly imposes constraints by generating code from patterns that are already checked for correctness. In the previous example, the author does not have to independently remember to think about what happens when the lever is destroyed, since the pattern ensures that such a scenario is considered.

3. Why Adaptation Instead of Construction?

There are two reasons why the generative/adaptive approach allows non-programmers (game story authors) to write stories that contain complex scripting code. First, our approach is *adaptive* rather than *constructive*; they do not write new descriptions, they adapt generated ones. Even when a new description is required, the author starts by selecting a basic general concept and adapts it (specializes it). Second, the language we use to represent our *descriptions* is more natural (less formal) than the language used to represent programs. Since our approach can be applied to application domains other than computer games, in this section we describe adaptation-construction and the use of natural language in a broader context.

3.1 Adaptation versus Construction

During the history of computer programming languages and environments, there has been a continual increase in the support for higher levels of abstraction – subroutines, abstract data types, objects, polymorphism, inheritance, etc. Abstractions allow a problem and its solution to be expressed in terms that are closer to the application domain and farther from the computer architecture. Two effects of this progression are increased productivity, as more machine code is generated from individual program statements, and reduced errors, such as subroutine parameter-argument mismatches that cannot happen with modern compilers. This increase in productivity and reduction of errors has enabled software developers to build larger and more complex systems. A Domain Specific Language (DSL) [42] is an extension of the abstraction process in which the programming constructs are not only specified at a higher level of abstraction, but are also specific to the domain of the problems being solved. In fact, our approach uses a DSL (the ScriptEase Programming Language). However, a DSL is not enough. Constructing programs with domain specific building-blocks is still constructive programming and non-programmers find this difficult. Asking game authors to use the ScriptEase programming language to write scripts is equivalent to asking them to create new patterns every time they script a new object in their story. The key is to allow an author to select a pattern and automatically obtain a solution to a problem, expressed in a DSL. The author can then adapt the DSL by modifying a few DSL components and push a button to generate low-level code.

The next major step in the evolution of general software development is to eliminate the programmer from the construction of application programs. Domain authors will not simply write code in a higher-level language. Instead, they will select patterns as solutions and adapt them. Ideally, the programmer should be relegated to building systems software and tools. Automatic program construction [33] has been the holy grail of many programming language researchers and designers. Many attempts have been made, with only limited success. For example, declarative approaches such as logic programming have been attempted under the assumption that it is easy to write declarative statements using first order-logic. Intentional programming [13] tries to directly support programmer intentions, rather than focusing on low-level syntactic language constructs. Program transformation [10] has been used to transform requirements specifications into executable code. These approaches represent innovative ideas, but they have failed to achieve widespread success in mainstream commercial software.

The common thread of these approaches, and many others, is that they are *constructive* – the programmer must create a program the way one constructs a document, by assembling syntactic components. Our approach is fundamentally different. It is adaptive rather than constructive. In the first step of our process, the author selects from the pattern catalog to produce a descriptive artifact – a pattern instance. For a traditional programmer, this would be analogous to selecting a solution technique – for example, deciding to sort some information in ascending order and then display it in a window. It is in the second step of development that the constructive and adaptive approaches differ. Our second step replaces the traditional constructive process of writing a program with what we believe is a much simpler *adaptive* process. Rather than trying to construct a program from a plethora of syntactic components that can be assembled in literally billions of different ways, the author simply adapts the description generated by the pattern using one of a very small set of allowable adaptation operations. For example, instead of writing sort code from scratch, adaptation would involve taking an existing sort description and either setting an option or changing a descriptive line that indicates the information should be sorted into ascending order.

At any point in the constructive programming process, a programmer picks several small syntactic components from a large number of program components using a mental menu, and assembles these components into larger program constructs. For example, to compute and remember a value, a programmer constructs an assignment statement using an assignment variable, an assignment operator and an expression consisting of operators, constants, other variables and function calls. Using a DSL instead of a general purpose programming language increases the level of abstraction that the constructive programmer uses. This does simplify the construction, but does not fundamentally change the fact that the author is still faced with the challenge of creating larger constructs out of smaller one. Wile [42] clearly states that DSLs are not always programming languages: “Moreover, DSLs are not necessarily programming languages: they are languages tailored to express something about the solution to a problem.” His paper nicely divides DSLs into four groups (Full language design, Language extensions, Common Off-The-Shelf approaches, and Interchange representations) and provides a wide range of examples including an interesting Satellite domain application that uses a PowerPoint design editor. Although not all DSLs are programming languages in the traditional sense, the ones constructed to date have been languages that share a constructive approach, rather than an adaptive one.

In the adaptive process, the author selects a generic solution (pattern) to a problem. Since this solution is generic, it will not exactly solve the problem being considered. Therefore, the author selects one of a few adaptation operations from a real menu and then specifies the

details of the adaptation operation using a sub-menu and dialog box. For example, the author starts with a complete solution to a generic problem that may require an additional action to become the desired solution. To add an action, the author selects the add action menu item and then traverses sub-menus to specify what the action should be. For example, Fig. 5 shows an action that displays a caption above the head of a creature. The author then types what the caption should be in a field of a dialog box ("Run for your life!"). Adaptive programming consists of a series of author decisions on how to adapt an existing solution. These decisions are made by selecting items from menus and dialog boxes, possibly combined with the typing of some short strings. Adaptive programming, as we have defined it here, is different than the concept of adaptive programming coined by Lieberherr [25] in the 1990's. He uses the term to refer to applications that are easy to maintain and evolve. We use the term to mean an application that is created by adaptation. Although the ideas are different, our use of the term is a logical progression of his good idea.

Which process is easier – constructive programming or adaptive programming? The analogy of document creation provides some evidence that an adaptive process is easier. Most people would rather adapt a template document or an old version of a document, such as a reference letter, thank-you note or announcement memo, rather than creating a new one. There are two fundamental reasons for this. First, it saves time. Second, the template has already been used one or more times in the past, so common errors have been eliminated.

Traditional constructive programming requires one to learn syntax and semantics. Programming languages are large artifacts with obscure syntax, complex semantics and subtle side-effects. Adaptive programming (as we envision it) has no syntax to learn. As described in the next sub-section, the semantics are expressed in natural language using the vocabulary of the application domain. There might be some subtle side-effects with adaptive programming, but, if so, they are probably the fault of the pattern designer, as opposed to the adaptive methodology. This means that unintentional side effects can be discovered and corrected at the level of pattern design, not at the level of pattern use. Since patterns are designed once and used often, this amortizes the cost of error detection over multiple uses.

3.2 Natural versus formal language

In step one of our process, a pattern instance is generated that contains a description of when the pattern applies and what the pattern does. From a programmer's point of view, the description can be seen as statements in a high-level programming language, but the author just sees them as a description. During step two (adaptation), the designer can manipulate these descriptions analogously to the way programming language statements are manipulated, but in a more constrained way. During step three (script generation), ScriptEase uses each pattern description to generate scripting code in a way similar to how a compiler uses each program statement to generate machine code.

However, there are three fundamental differences between pattern-generated descriptions used by authors and programming language statements constructed manually by programmers that contribute to the success of the generative/adaptive approach. First, generated statements exhibit less variation than manually-constructed statements since, when there is more than one way to express the same concept, generation provides a consistent representation. Hence, the working vocabulary of generated descriptions is smaller than the working vocabulary of manually-written programs that use the same set of concepts. A smaller working vocabulary leads to a faster learning curve and less confusion about which representation to choose.

Second, a description can be expressed in a more natural language if it is generated than if it is constructed. Consider the description from Fig. 3 and the corresponding program statement from Fig. 4, shown together in Fig. 6. The natural language statement is much easier for a non-programmer to understand. However, a system that accepted natural language during constructive programming would require a complex parser and natural language is inherently ambiguous. That is why we have programming languages to support constructive programming. Adaptive programming does not need a formal programming language at the author level, since descriptions are generated from author selections.[1] For example, the constructive approach requires the programmer to create unique labels to reference different objects and functions. With the adaptive approach, unique labels are generated automatically in the scripting code, using prefixes and suffixes. This level of detail is not seen by the author.

Spawn Spawned Creature from Creature Blueprint (Avadel) near The Container (Dresser)

```
SpawnedCreature_SE3=SE_AC_SpawnCreatureNearObject("bandit003", Dresser_SE0);
```

Fig. 6 A natural language description and the equivalent NWScript program statement

Third, an author can only manipulate descriptions in very structured ways, using a limited set of adaptation operations. Performing adaptation operations using context-sensitive menus eliminates errors. With programming language statements, a programmer has a much broader scope for making changes and therefore more chances for errors as well. Errors must then be identified and fixed.

4. Pattern Adaptation in ScriptEase

As mentioned previously, ScriptEase supports four kinds of patterns: encounter, behaviour, dialogue, and plot. The case studies described in this paper focus mainly on encounter patterns, so in this section we provide details on adapting encounter patterns. The other three kinds of patterns are currently under development and they will be added to ScriptEase as first-class patterns in the near future.

Fig. 7 shows a sample encounter pattern, *Container disturb (specific item) – toggle door*, expanded to reveal the internal components of its description. The intent of the pattern is to toggle the status of a door if a specific item in a container is disturbed (added, removed or stolen). Toggling a door means unlocking it and opening if it is currently locked, and closing it and locking it if it is unlocked. An author from the study described in Section 5 (a high school student) was responsible for the adapted pattern shown in Fig. 7. The author created an instance of this pattern and set the options for it using dialog boxes as illustrated in Fig. 3. This particular pattern has three options, which the author set to three specific game objects in the story (*The Container* set to *Bookcasering*, *Specific Item* set to *Queen's Ring*, and *The Door* set to *Statue Door*). Since the author did not adapt this pattern any further, it is not actually necessary to open the pattern to reveal its components. However, we are using this pattern to illustrate all of the possible components of an encounter pattern, so that we can describe the adaptation operations that are available to an author. In the next section, we indicate how many authors performed each adaptation operation and how many times they performed them.

[1] Of course, use of a formal programming language is hidden in the implementation of the adaptive programming process. In step 3 of the adaptive process, patterns are translated into an equivalent code representation. This level is only visible to the programmer.

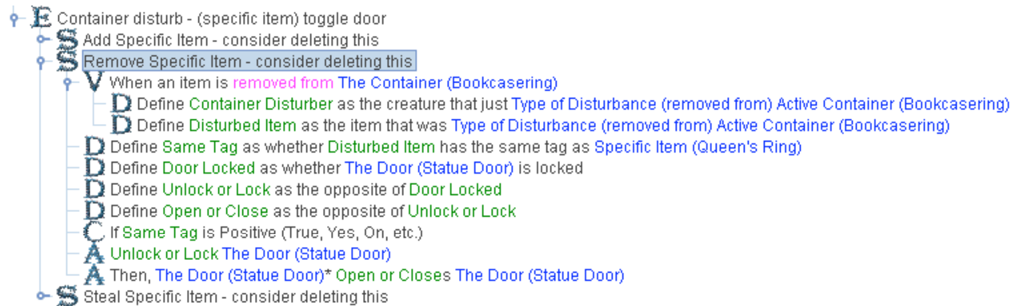

Fig. 7 Components of an encounter pattern description: *Container disturb (specific item) – toggle door*

In Section 2, we indicated that each *encounter* pattern (stylized E) has one or more *situations* (stylized S) and that each situation contains one *event* (stylized V) and zero or more *actions*. We now describe the last two components of situations: definitions and conditions. The same mechanism is used for adding all encounter components – hierarchical menus. Examples of these hierarchical menus will be given later.

Each event contains zero or more implicit *definitions* (stylized D) that refer to objects that play a role in the event. For example, in Fig. 7, the event has been opened to reveal the label *Container Disturber,* referring to whatever creature disturbed the container, and the label *Disturbed Item,* referring to the item that was disturbed during this event. Each situation also has zero or more explicit *Definitions* that can be used to refer to objects and information in the game. The pattern description shown in Fig. 7 contains four explicit definitions. The first definition (*Same Tag*) defines a label that indicates whether the item that was just disturbed (*Disturbed Item*) is the same item as the specific item (*Queen's Ring*) that the author wanted to trigger this situation. This definition will be used in a condition. The second definition (*Door Locked*) indicates whether the door is currently locked or not. This definition is used in the next definition. The third definition (*Unlock or Lock*) indicates what should be done to the door, with respect to locking/unlocking it. This definition is needed in the action that will either unlock or lock the door, to specify the correct version of a lock/unlock. The fourth definition (*Open or Close*) indicates what should be done to the door with respect to opening/closing it. It is needed in the action that will either open or close the door.

Each situation contains zero or more *conditions* (stylized C) that specify other circumstances that are necessary for the situation to apply. For the pattern shown in Fig. 7, the situation should only apply if the disturbed item and the specific item have the *Same Tag*. As indicated in Section 2, each situation can contain zero or more actions (stylized A). In Fig. 7, there are two actions, one to unlock or lock the appropriate door (*Statue Door*) and one to open or close it. The other two situations (*Add* and *Steal*) consider similar code. The author that used the specific pattern description shown in Fig. 7 only wanted the pattern to apply for the *Remove* situation, so the other two situations were deleted.

Although the components of the pattern in Fig. 7 may look to programmers like familiar programming language statements, the components are just descriptions. The author did not construct these descriptions from smaller syntactic components or even write them as natural language sentences. Most descriptions were generated from a pattern that the author selected. Some words in the descriptions were generated after the author selected options using dialogs as shown in Fig. 3. The rest of the descriptions were generated after the author performed adaptation operations by selecting from menus as explained throughout the remainder of this

section. The description shown in Fig. 7 was generated in a few minutes by a Grade 10 English student with no programming experience!

When designing a pattern, there is always a tension between generality and specificity [11]. If a pattern is too general, it will be applicable to many scenarios, but will require many adaptation operations each time it is used. This will increase the work required to use the pattern effectively. If a pattern is too specific, it will not be used often enough to justify its existence. Having a large number of very specific patterns makes the pattern catalog too difficult to use, since finding appropriate patterns becomes problematic.

The topology of a pattern (the number of components of different types) determines its generality. More situations make a pattern more general. For example, the *Container disturb - spawn creature* pattern has three situations: *add an item*, *remove an item*, and *steal an item*. It applies whenever the PC adds, removes, or steals an item from a container. This is more general than a pattern that has only one or two of these situations.

On the other hand, more actions or definitions make a pattern more specialized. For example, the *Container disturb - spawn creature* pattern shown in Fig. 3 has two actions in each of its situations, one action that spawns a creature near the container and another action that shows a visual effect on the spawned creature. A hypothetical pattern, *Container disturb - spawn creature, lock door, create object* that also locks the nearest door and creates an object near the container is more specialized, since it does more actions. Such a pattern would be rarely used.

More conditions also make a pattern more specialized, since each condition reduces the chances that the pattern applies (does something). For example the hypothetical pattern *Container disturb (disturber is female) - spawn creature* pattern would only spawn a creature if the PC that disturbs the container was female. Hence, a second (male) pattern would be needed. The result is twice as many patterns each with half the utility.

Therefore, to strike a balance between general applicability and the need for too many adaptations, the patterns in the ScriptEase pattern catalog tend to have many situations and few actions or conditions. This is an important insight that goes beyond this particular application domain and pattern tool. We have developed metrics for evaluating the effectiveness of general pattern catalogs [11] and used them to evaluate the tradeoffs between specialization and generalization in the ScriptEase pattern catalog.

Since the adaptation process is one of taking a general pattern description and adapting it by specialization to apply in a particular context, the following adaptation operations are common: deleting a situation, adding an action/definition, and adding a condition. On the other hand, the operations of adding a situation, deleting an action/definition, and deleting a condition are rare, since they generalize a pattern rather than specializing it. Generalization rarely happens during adaptation of a general pattern to a particular context.

The number of adaptations required to use a pattern is important. However, the difficulty of applying each desired adaptation must also be taken into account to minimize the work required by an author to create a game story. Based on our experience (and the data shown in this paper) it is easier for an author to delete a component or replace a component than to add one. Deleting a component is a binary decision – one can delete it or not. The location of the component is known. Adding something requires a location to add it and the identification of what to add. If there are many choices, each with options, this can be much more challenging.

To reduce the complexity in the case where a component should always be added and where the specific component cannot be determined when the pattern is created, a placeholder component is used so that the author knows exactly where to place the component. The author

can replace a placeholder with a specified type of component (adaptive), rather than trying to figure out what kind of component to add and exactly where to add it (constructive). For example, it is common for game story authors to want something to happen when the PC removes a specific item from a container. One author may want a creature to be spawned. Another author may want the container to explode. There are many possibilities and they can all be represented by the *Container disturb (specific item)* pattern shown in Fig. 8. In this case the author wants to add an action (at the action placeholder) to spawn a creature. Performing this adaptation is straightforward. The author first selects the action to be added (*Spawn a creature near an object*) from the hierarchical menu shown in Fig. 8 and then sets the options (*Creature Blueprint* to *Slave of Lord Dumont* and *Target* to *Gong of Challenge*) for this new action using dialogs similar to the one shown in Fig. 3. The final step is to delete the placeholder by clicking on the placeholder action to highlight it and selecting *Delete* from the pop-up menu as shown in Fig. 9. The technique for deleting other components (situations, definitions, regular actions, and conditions) is the same.

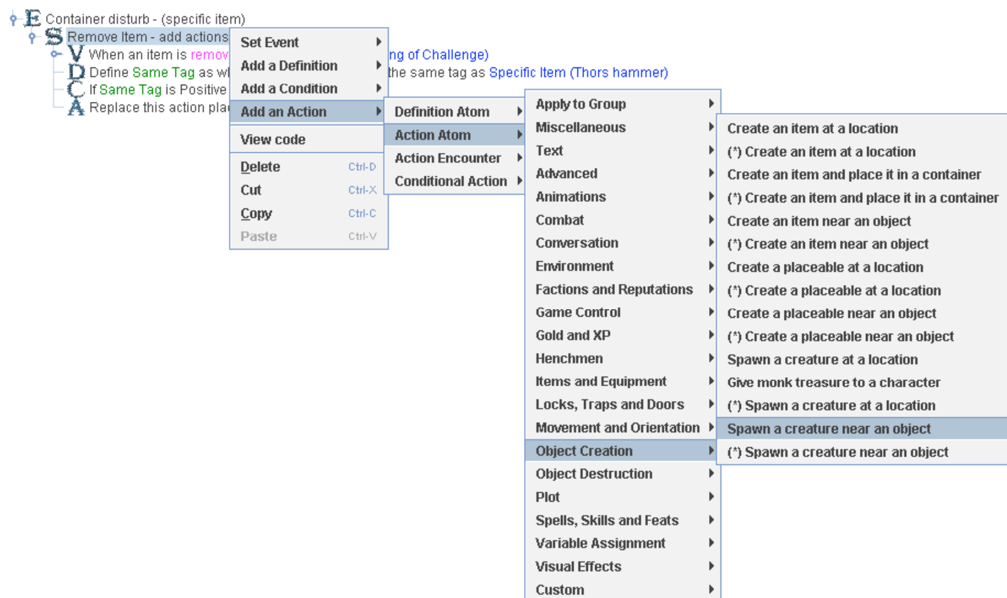

Fig. 8 The hierarchical menus to add an action – *Spawn a creature near an object*

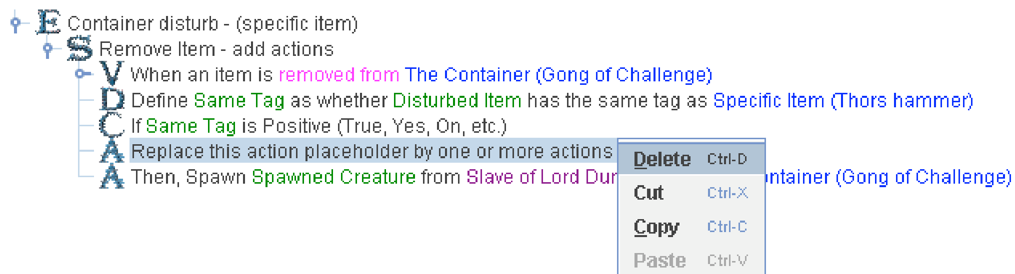

Fig. 9 Deleting the Action placeholder in the Container disturb (specific item) pattern

ScriptEase supports twelve pattern adaptation operations that are grouped into nine categories as shown in Table 1. The number associated with each category is a subjective ranking of the difficulty associated with each kind of adaptation, with a lower number indicating a lower difficulty. The difficulty categories are based on our experience and on the results of the case study described in the next section. For example, adding a definition and adding an action have the same perceived difficulty. Adding a condition is considered to be a more difficult operation, hence its higher difficulty ranking. Note that the rare adaptation operations identified previously (adding a situation, deleting an action/definition, deleting a condition) are marked with an asterisk in Table 1.

Table 1
Adaptation operations for ScriptEase encounter patterns. Operations that are rarely used are marked with an asterisk (*)

Difficulty Category	Adaptations
1	Set options
2	Delete a situation
3	Delete an action*; Delete a definition*
4	Delete a condition*
5	Replace an action placeholder; Replace a definition placeholder
6	Add an action; Add a definition
7	Replace a condition placeholder
8	Add a condition
9	Add a situation*

It should not be surprising that different adaptation operations have different difficulties. Most people would agree that when writing a traditional program, constructing an assignment statement is easier than constructing a selection control structure (if), which is easier than constructing an iteration control structure (loop).

The following sub-sections illustrate the adaptation operations from each of the nine categories. These examples are based on the game stories produced during the case studies described in the next section.

4.1 Set options

After the author selects a pattern and creates an instance, the pattern options are set. Every pattern has at least one option – the object to which the pattern applies. Options are set using a dialog box for each pattern option. Setting options is the simplest conceptual adaptation that designers perform on patterns, since it must be performed for each pattern instance created.

4.2 Delete a situation

Deleting a situation is straightforward. Each situation describes a self-contained circumstance under which the pattern may apply and the actions that should occur under those circumstances. Since each situation is independent of the others and situations are at the top level inside an expanded encounter pattern in the GUI, deleting a situation is a simple concept. Notice that when it is common to delete a particular situation, the situation is marked for the author as “consider deleting this” (see Fig. 7).

4.3 Delete an action or definition (- rare adaptation)*

Deleting an action or a definition requires the author to understand the context of that action or definition. Each action is contained in a situation. A definition may be at the top level of an encounter and apply to all situations in the encounter, or it may be inside a situation and apply only to that situation. This makes the deletion of an action or definition more complex than the deletion of a situation. Nevertheless, it is not difficult. For example, Fig. 3 shows the interior of the *Remove item* situation of the *Container disturb – spawn creature* pattern. There are two actions in this situation, one to spawn a creature and another to show a visual effect. The second action can be deleted if the author does not want to use the visual effect to draw attention to the spawned creature.

Since extra actions are not included in patterns, actions are rarely deleted. In game stories, it is rare for an author to want a creature to be spawned without a visual effect. That is why the pattern designer included a visual effect in this pattern. In our study, there were very few occurrences of an action or definition being deleted.

4.4 Delete a condition (- rare adaptation)*

It is possible to delete a condition from a pattern description. It is rare, since deletion of a condition generalizes a pattern. Here is a hypothetical example that did not occur in the study. Start with the pattern *Container disturb (specific item) – toggle door* shown in Fig. 7 and delete the specific item condition (*Same Tag is positive*) so that the door is toggled whenever any item in the container is disturbed. At first glance, it seems like the basic pattern should not include the specific item condition, making the pattern more general. In practice, few authors want a door to be unlocked when just any item is disturbed. The author usually has a specific item in mind (that is critical to the story), so the pattern available is the one that is actually more useful. In the study, there were only four cases where a condition was removed. In two of these cases (two instances of the same pattern) the condition should not have been deleted. Based on the author's intent, deletion was an error. In the other two cases (again two instances of the same pattern), the author deleted condition placeholders inside a situation, when the situations should have been deleted instead.

4.5 Replace an action or definition placeholder

Replacing an action or definition requires the author to understand the context of that action or definition. Replacement is more complex than deletion, since there are many actions or definitions that can be used as the replacement. Once the author gains experience about what actions and definitions are available, it becomes easier. Fig. 9 shows an action placeholder being replaced by adding an action at the location of the placeholder and then deleting the placeholder.

4.6 Add an action or definition

Adding an action or a definition is slightly more difficult than replacing one, since the author must manually decide the appropriate place to add the action in the situation. Fig. 10 shows a *Container disturb – spawn creature* pattern that has an extra definition and an extra action added to it. The author would like to a have an item (*Blade of the Rashemi*) appear at a particular location. To add this definition, the author first used the Aurora Toolset to create the

item and an object called a waypoint (a flag) with tag (*DumontWaypoint*) to mark the desired story location. Then, in ScriptEase, the definition was added by selecting an appropriate kind of definition (*Define location of object*) from a set of hierarchical menus, similar to the action hierarchical menus shown in Fig. 8. The author then set the label of the definition (*Location*) and set the waypoint object (*DumontWaypoint*) as options in the definition, similar to the way the action options were set in Fig. 3. Finally, a *Create item at location* action was selected from hierarchical menus and its two options were set (*Item Blueprint* to *Blade of the Rashemi* and *Creation Location* to *Location*) as shown in Fig. 10.

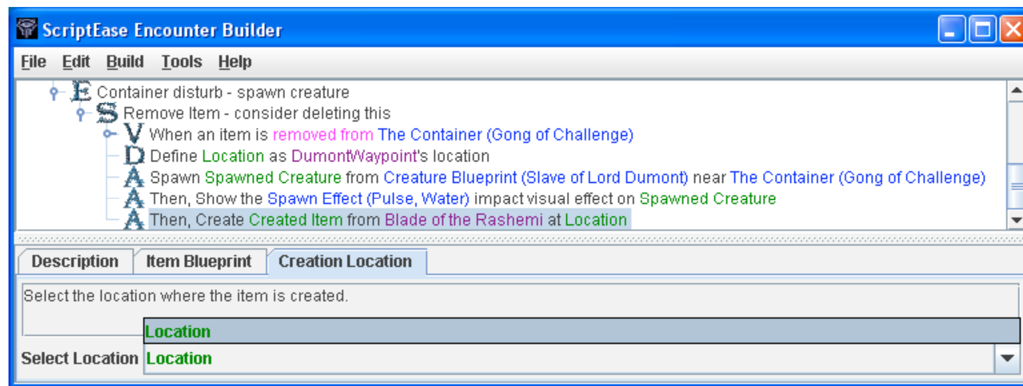

Fig. 10 Adaptation: adding an action or definition

4.7 Replace a condition placeholder

Adapting a condition is more complex than adapting an action or a definition, since each condition uses a definition. Each ScriptEase condition evaluates a Boolean condition and performs its actions if the Boolean condition has one of two specific values. Unlike programming languages, where Booleans have the values false and true or 0 and 1, a Boolean in the ScriptEase description language can have one of any pair of values. Common examples are the pairs False/True, No/Yes, Off/On. New pairs can be defined when a pattern is created. ScriptEase supports only four conditions: *always positive*, *always negative*, *if positive*, and *if negative*. An *always positive* condition always performs its actions and an *always negative* condition never performs its actions. These two conditions are used as placeholders and should be replaced during adaptation. For example, consider Fig. 11 that shows a *Trigger enter/exit – barrier* pattern. An author uses this pattern to prevent a PC from entering or exiting a trigger region (a polygon shaped area drawn on the ground using the Aurora Toolset). By default the pattern contains four situations, *Try to enter trigger*, *Try to exit trigger*, *Destroy barrier on entry* and *Destroy barrier on exit*. In a typical use, an author only wants a region that cannot be entered or a region that cannot be exited, so two of the situations are usually deleted – in this case the exit situations were deleted. The *Destroy barrier on entry* situation contains an action that destroys the barrier and displays a visual effect. A barrier is useless if it is destroyed as soon as the player tries to enter the first time, so the *Destroy barrier on entry* situation is "guarded" by a placeholder condition that is *Always negative*. To achieve the intended story line, the author adapted the pattern by replacing the *Always negative* condition by some appropriate condition that should be satisfied to destroy the barrier. In this case, the author wanted the barrier to be destroyed if the PC was carrying an item whose label was *Tomi's*

contract. Fig. 11 shows the pattern after the author has added a definition (*Has item*) and a condition (*If Has Item is Positive*). To complete the condition replacement, the author deletes the *Always negative* placeholder condition, as shown in Fig. 11.

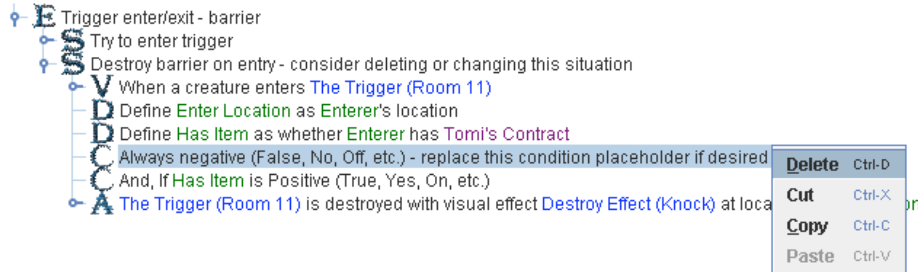

Fig. 11 Adaptation: replace a condition placeholder

4.8 Add a condition

Adding a condition to specialize a pattern is a common adaptation operation. It is more difficult than adding an action or a definition, since adding a condition requires the author to first add a Boolean definition and then use it to set the *definition* option in the condition. Therefore, when a pattern always requires a condition to make sense, a placeholder condition is provided to simplify the adaptation, as shown in the previous subsection. However, some patterns can be used with or without a condition. Fig. 12 shows an example where a condition has been added to the *Container disturb – spawn creature* pattern. The author has added a condition that the disturbed item must be a specific item (*Thors hammer*). It was necessary to add the definition (*Same Tag*) to set the *definition* option of this condition. Note that the author could have achieved the same intention by starting with a *Container disturb (specific item)* pattern and adding the two actions instead. The only way we know that the former was done rather than the latter is that the description on the encounter pattern shows the pattern that was selected (*Container disturb –spawn creature*).

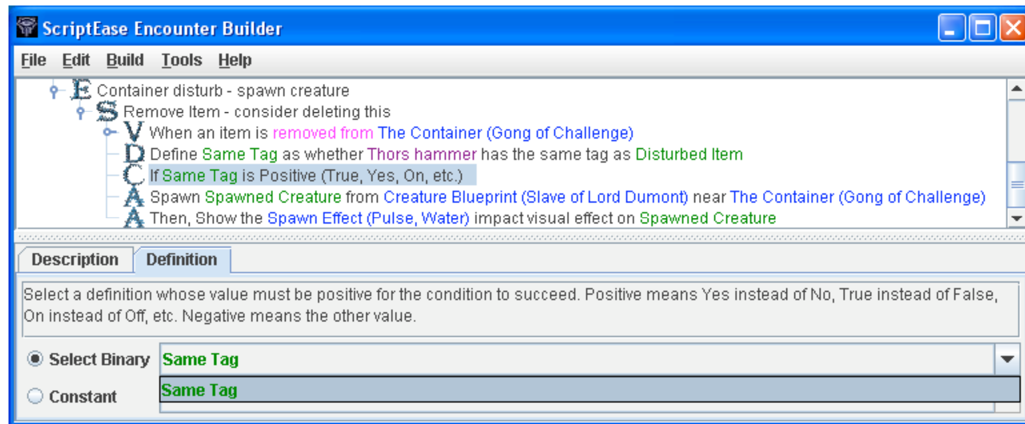

Fig. 12 Adaptation: add a condition

4.9 Add a situation (- rare adaptation)*

Adding a situation is the most complex adaptation operation available in ScriptEase, since a situation contains other components. Fig. 13 shows a *Trigger enter* pattern that contains its original situation (*Trigger enter*), along with a second situation (*Trigger enter*) that has been added by the author. The author has adapted the original situation by adding two definitions, a condition and an action. The intent is that when a creature enters the trigger, if the creature has the *Orb of Xi'Tah Teleportation*, then the creature will be teleported to a different location – the location of *waypoint1a*. Since a *Trigger enter* pattern only has a single situation, the author wanted to add a second situation so that if a creature enters the trigger, but does not possess the *Orb of Xi'Tah Teleportation*, then the creature would be warned by the text, "*You need the Orb of Xi'Tah Teleportation"*.

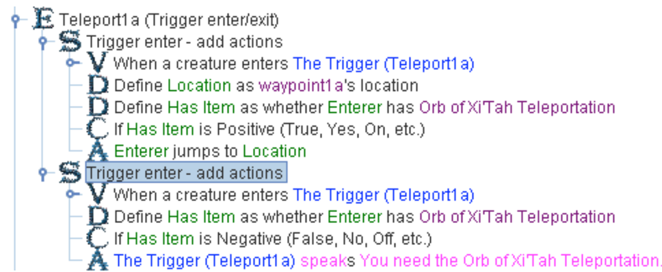

Fig. 13 Adaptation: add a situation

To add a situation, first select the encounter pattern that will contain it and then select a menu item to add a new situation. There is only one kind of situation so that menu is not hierarchical. The next step is to set the event for the situation using the hierarchical menu shown in Fig. 14. Note that a situation will apply only to a particular kind of object (*Area, Creature, Door*, *Trigger*, etc.), so that the event selection depends on the kind of the target object. In this case, the author wanted the event to be a *When a creature enters a trigger* event. After adding this event, the author then adds definitions, actions and conditions as described previously. Alternatively, instead of manually creating the second situation, the author could have simply copied the first situation, pasted it into the pattern and then adapted its contents. In fact, that is what this author actually did during the study.

Fig. 14 Selecting an event for a situation from a hierarchical menu

4.10 Designing new patterns

ScriptEase includes an encounter pattern designer that allows an author (non-programmer) to create new encounter patterns. In this paper, we provide only a brief explanation of pattern design, since our case studies did not include pattern design by the authors. The high school student authors were creating game stories as part of their high school English curriculum. There was not enough time to spend designing new patterns, when the focus was on creating stories. However, a high school summer student (non-programmer) demonstrated the simplicity of pattern design by designing four behaviour patterns to populate our behaviour pattern catalog. In addition, three other high school students (non-programmers) created several new patterns in the process of creating a feature interactive story. We are planning to conduct a full study that includes pattern design in the near future.

Designing a new encounter pattern is similar to adding a situation to an existing pattern. There are only two more small steps. The author first selects a menu item to create a new encounter pattern and gives a name and description to this pattern. The author then uses a menu to create the set of options (parameters) that the pattern will have and assigns a game object type to each option (*Creature*, *Door*, *Container*, etc.). Fig. 15 shows the dialog used to create options (parameters). After this, the author adds situations as described in the previous section. In other words, after an encounter pattern is created, it is adapted using exactly the same adaptation operations that are used to adapt a pattern instance for a story.

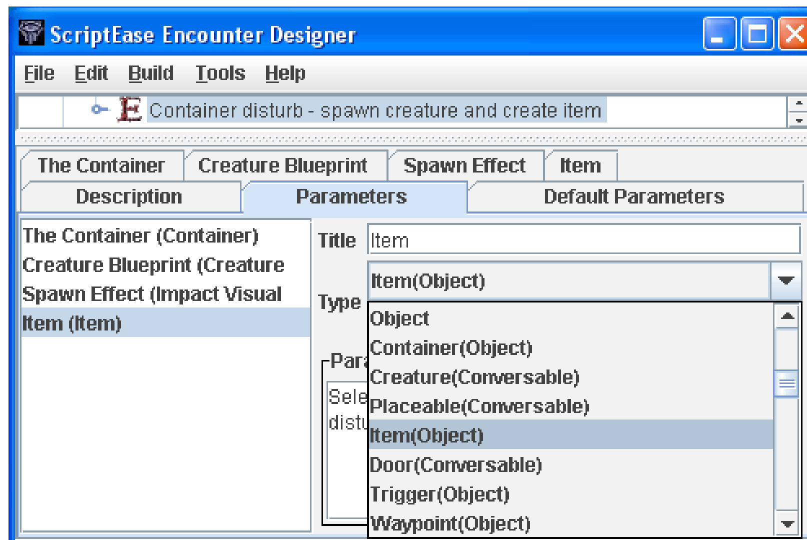

Fig. 15 Creating encounter options (parameters) for a new encounter pattern using the ScriptEase encounter designer

The definitions and actions that can be selected from hierarchical menus during adaptation depend on the particular game (e.g., *Neverwinter Nights*). ScriptEase provides an atom editor that can be used to create these atomic definitions and actions in terms of the API provided by the game engine. However, defining a new atom requires programming skills, since the atoms are defined using the underlying scripting language (NWScript in this case). For example, Fig. 16 shows how the *Spawn creature near object* action is created in the atom editor.

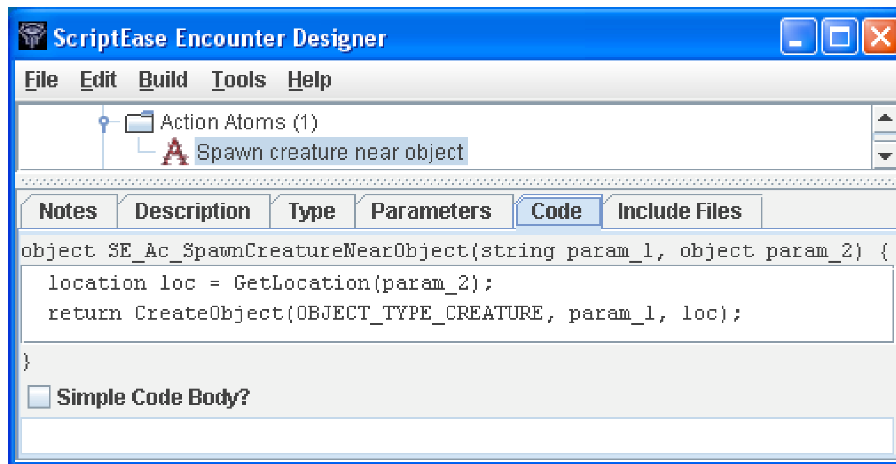

Fig. 16 Creating an atomic action using the ScriptEase atom editor

Since creating a new pattern requires the same adaptation operations as adapting an existing pattern, the author does not have to learn a new skill set. This approach provides a smooth evolution from pattern user to pattern designer. When an author creates a new pattern, the same basic components, such as basic actions (walk to an object or perform arithmetic operations) that have been used in adaptation can be used for pattern creation. In addition to the fixed set of events to which every object responds, the pattern designer may create custom events. This allows unlimited flexibility for any particular situation a story requires. Using patterns does not deter an author from creating original stories. On the contrary, since the existing library of patterns supports rapid story prototyping, the author can use the time saved to experiment with new patterns that can be used in different ways. The productivity gains that resulted from using reusable components in 3D content construction to produce more impressive visual objects (the Gamebryo 3D graphics engine [18] was used to create the worlds of the award-winning CRPG, Oblivion [4]) should be repeatable by using patterns to generate scripts for game object interaction in stories.

5. Case Studies

5.1 Description

To obtain evidence that the ScriptEase generative/adaptive approach to game authoring and interactive fiction writing could be successfully applied by non-programmers, we designed and conducted case studies for three groups of authors. These case studies were designed to meet the following criteria:

1. the authors had to be either non-programmers or inexperienced programmers;
2. the stories produced had to rival those that could be produced by more conventional programming approaches;
3. the time required to learn ScriptEase had to be significantly shorter than the time required to learn the equivalent scripting language, and
4. the time required to produce stories had to be shorter than the time required to produce equivalent stories using conventional programming approaches.

We developed these case studies to determine whether ScriptEase's generative/adaptive paradigm was robust enough to generate stories of acceptable complexity while still being easy to learn. In particular, the learning curve had to be small so that ScriptEase could be tested in a classroom environment. The same case studies were also used to investigate broader educational goals, which are beyond the scope of this paper.

Each case study had the same four steps. In step one, student authors were taught to play *Neverwinter Nights* (*NWN*). A tutorial provided a guided walkthrough of a player character (PC) through the *Prelude* of the *NWN* game. The *Prelude* is the game's training module and is designed to introduce players to the look and feel of the *NWN* world and to teach them how to use the interface. This tutorial was designed to make sure that the students were well aware of those aspects of the interface and the game world that would be of special importance when they shifted roles from game players to game story authors. Students typically took one and a half to two hours to complete this first tutorial on "playing the game".

In step two, a second tutorial about constructing a story world was used to introduce the students to *NWN*'s Aurora Toolset. They were taught how to use the Aurora Toolset to populate a game setting with the props and the computer-controlled NPCs that the PC would interact with when the story was "played". A game module comprises all areas, NPCs, props and their scripted interactions in a game story. The students were provided with three game modules produced by a high school teacher. The first module, *CastleEmpty*, was used in conjunction with the tutorial. This module contained two areas, the interior (*Castle*) and the exterior (*Exterior*) of a medieval castle, but did not contain any props or creatures. The castle was composed of several rooms. The students were guided through the process of placing props (such as furniture, books, weapons, etc.) and NPCs (such as guards and creatures) in these areas. The second module, *CastleFull*, was an augmented *CastleEmpty* with the props and creatures added and scripted as a consequence of completing the tutorial steps. Together with this tutorial, students were provided with maps for the *CastleEmpty* module, whose annotated rooms facilitated the tutorial instructions. The third game module, *MyShortStory*, was used as the basis for their interactive stories.

In step three, the students learned how to use ScriptEase to actually "write" the story associated with the setting, by completing the ScriptEase part of the second tutorial. In *NWN*, stories are a network of possible interactions between the human-controlled PC and the computer controlled NPCs and props. Accordingly, this tutorial guided the students through a typical set of *NWN* interactions. This was done through creating appropriate pattern instances and adapting them. As they worked through the tutorial, the students learned to select the pattern that created a general version of the desired interaction and then to adapt that pattern to obtain the specific interaction desired. As this tutorial was considerably more complex than the "playing the game" tutorial, the students typically took four hours to work through the tutorial (Aurora Toolset and ScriptEase).

In step four of the case study, student authors were told to create their own stories in *NWN*, using the Aurora Toolset and ScriptEase.

The tutorials were specifically designed for the case studies. A high school teacher and a high school student developed and tested the tutorials. Since the success of the case studies was predicated on the presence of clear and effective tutorial materials, the tutorial development process went through several cycles of writing, testing and revision. For example, a high school English class used an earlier version of the tutorials as a test run, without any data being collected, to test the feasibility of the tutorials and techniques. This tutorial material (split into three tutorials) and the *CastleEmpty* module are available on-line [34].

After developing and testing the tutorials, the first case study was conducted with a class of 23 grade 10 English students (referred to as group HB in this paper) enrolled in the International Baccalaureate program. Students in this program are top academic students [22]. The second case study was with a group of six first year University students (group UA) who had just completed either their first or second University computer programming course. The third case study was with a class of 27 grade 10 English students (group HA) enrolled in a regular academic program.

For the high school students (groups HA and HB), the case studies were packaged as part of an actual school assignment. The grade 10 English curriculum requires students to develop their understanding of short stories through one or more writing projects. The interactive writing assignment that made up the core of the case study was inserted as a part of the teachers' regular project work in the short story unit. After writing a traditional (pen-and-paper) short story in class, the students were introduced to the concept of interactive stories and given the assignment of using ScriptEase to write an interactive short story, through participation in the case study. In each case, the high school student authors took part in a two-day workshop conducted at the University of Alberta. The workshop consisted of the tutorials described earlier (total time six hours), along with some limited time (two hours) to start their interactive short story. Students worked under the supervision of their teacher and some of the ScriptEase researchers who were familiar with both the tools and tutorials so they could answer student questions. During the day, there were several extended breaks where the students were exposed to several research projects that were of interest to this age group. At the end of the workshop, the students returned to their high school classrooms to complete their stories.

The students' assignment was to create an interactive story in a given setting. The high school teacher provided the students with a module, *MyShortStory*, which contained only the castle areas, the interior and the exterior of a medieval castle, without any props or scripting. The students were asked to "write" their interactive story using this module by adding their own props and characters and generating appropriate scripts using ScriptEase. The students were provided with blank maps of the castle and they were required to annotate the maps according to their story. These annotated maps were used by the teacher during grading to make the story easier to navigate. Students were given three class periods of computer lab access to write their stories. However, they did have additional access to the labs if they wished to work on their stories outside of class, but no assistance was provided. This allocation of three periods in the lab corresponded to the amount of time that an English teacher assigned to an equivalent traditional writing project. At the end of the allotted time, the stories were submitted to and graded as an English assignment by their teacher, using the same rubric as was used to grade their traditional stories. In all, the high school authors spent an average of four hours working on their stories in the high schools, plus two hours at the University, for a total of six hours.

The University student case study was conducted over a three-day period at the University of Alberta. Student authors used exactly the same tutorial material, completing it in five to six hours, followed by thirteen to fourteen hours of story writing time.

As shown in Section 5.2, in all three studies, the students were able to master the use of the Aurora Toolset and the ScriptEase environment to the point where they were able to produce meaningful stories. They were able to identify a network of interactions that they wished to incorporate into their story, select the appropriate ScriptEase generative pattern

needed to produce each interaction, and use a standard set of adaptation operations to customize the pattern to reflect the particular circumstances of their story.

5.2 Results

Fig. 17 shows a histogram of how many pattern instances were created by the authors in each of the three groups, HA with 27 authors (left bar), HB with 23 authors (middle bar) and UA with 6 authors (right bar). The x-axis shows ranges of the number of pattern instances that were created and the y-axis shows the percentage of authors from each study that created the number of pattern instances in that range. For example, 26% (7/27) of the HA authors created 1 or 2 pattern instances and 7% (2/27) of HA authors created 11 or 12 pattern instances. For the UA authors, the numbers on the x-axis represent 10's of patterns created, so that the information could be put into a single graph. For example, 67% (4/6) of the UA authors created 1-29 (not 1-2) pattern instances.

The graph shows that there is a wide variation in the number of pattern instances used by the authors and only a single author (from HA) out of the 56 involved with the three case studies did not create any pattern instances. This author created a story using the Aurora Toolset, but did not script any objects in the game using ScriptEase. The averages of the number of pattern instances created per author for the three study groups were 5.1 (HA), 5.6 (HB) and 40.2 (UA). The UA authors were more mature, having completed three more years of formal education, along with one or two programming courses. The UA authors also had more time to work on their stories. Either or both of these considerations could account for the fact that these authors used about eight times more pattern instances. Although the two high school groups were both grade 10 classes, the profile of the two classes was different. The HA class was a regular English class and the HB class was an International Baccalaureate English class. This distinction likely contributed to the difference in pattern instances created.

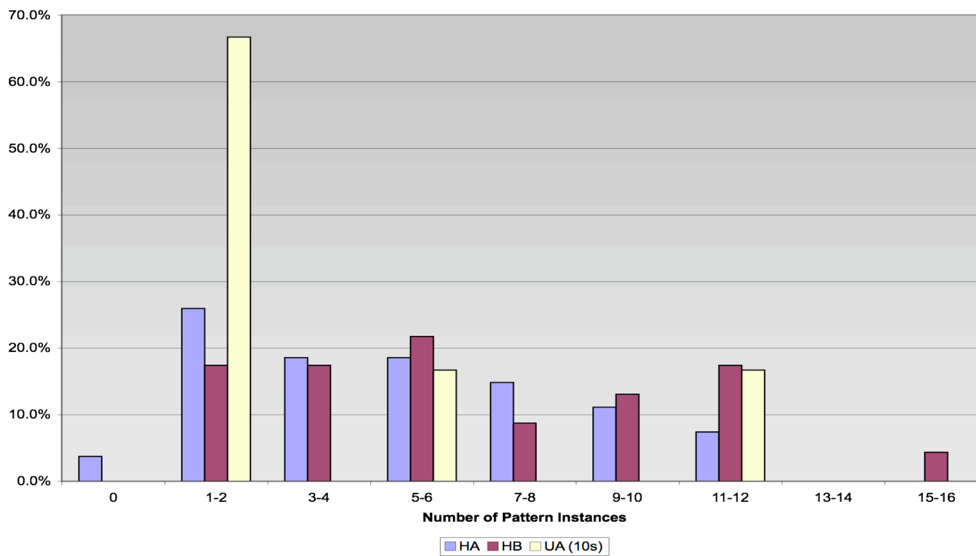

Fig. 17 Number of pattern instances used by students from each school

Fig. 18 shows a histogram of the total number of adaptations that were performed by the authors in each of the three groups. However, setting pattern options (Section 4.1) was not included in the counts, since every pattern that is created must have at least one option set. The x-axis shows ranges of adaptations and the y-axis shows the percentage of authors from each study that used the number of adaptations in that range. For example, 33% (9/27) of HA authors used from 1 to 5 adaptations and 4% (1/27) HA authors used 21 to 25 adaptations. Again, the UA numbers should be multiplied by 10. The graph shows a wide variation in the number of adaptations used by authors in all groups. It also shows that only six authors (five from HA and one from HB) out of the 56 authors involved in the study did not perform any adaptations. Of these six, five used only un-adapted pattern instances (except for setting options) and the sixth was the single author in the study who did not create any pattern instances (so no adaptations were possible). The averages of the number of adaptations used per author for the three study groups were 7.1 (HA), 14.7 (HB) and 156.2 (UA). The average number of adaptations per pattern instance for each of these three groups is 1.3 (HA), 2.2 (HB) and 3.9 (UA). It could be argued that the use of more pattern instances is simply due to more time to write a story. However, an increased number of adaptations per pattern instance indicates more complex interactions during story scenes. This is more likely attributable to higher cognitive skills of the UA authors and to a lesser extent the higher cognitive skills of the International Baccalaureate English class (HA) relative to the regular high school class (HB).

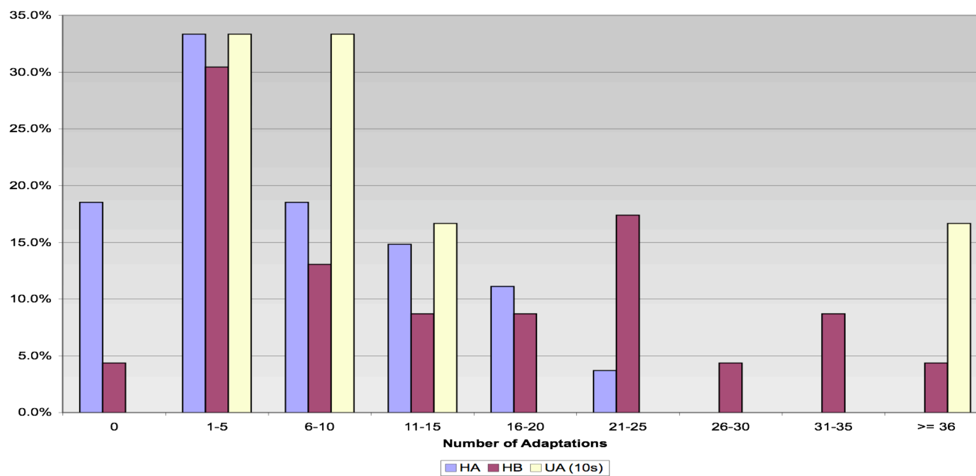

Fig. 18 Number of adaptations used by students from each school

Fig. 19 shows the percentages of authors in each of the three groups that used adaptations for each difficulty level from Table 1 (excluding the *set options* adaptation). The adaptations are arranged from easiest to most difficult (left to right), except that the three rare adaptations are placed on the far right. For example, 73% of the students from HA used the *delete a situation* adaptation, 85% used the *replace action/definition placeholder* adaptation and only 12% used the *replace a condition placeholder* adaptation. The number of students who used a particular kind of adaptation depends on several factors. For example, the adaptations that are rare because they generalize a pattern instead of specializing it have been placed on the right of the graph. Other factors that affect the number of authors who use a particular kind of

adaptation are the following: the patterns selected to create the desired kind of story, the particular way that an author wants to tell that story, and the instructions that the authors received when learning how to use ScriptEase. The number of students who used a particular kind of adaptation does not only reflect the difficulty of using a particular kind of adaptation. However, there appears to be a relationship between categories mastered and category difficulty for the two high school groups. The bars tend to go down from left to right for both high school groups indicating that fewer students mastered the adaptation operations on the right (higher difficulty categories). The slightly higher bars for *add a condition* compared to *replace a condition placeholder* might reflect the fact that there are a limited number of patterns with condition placeholders, so there is more opportunity to add than to replace.

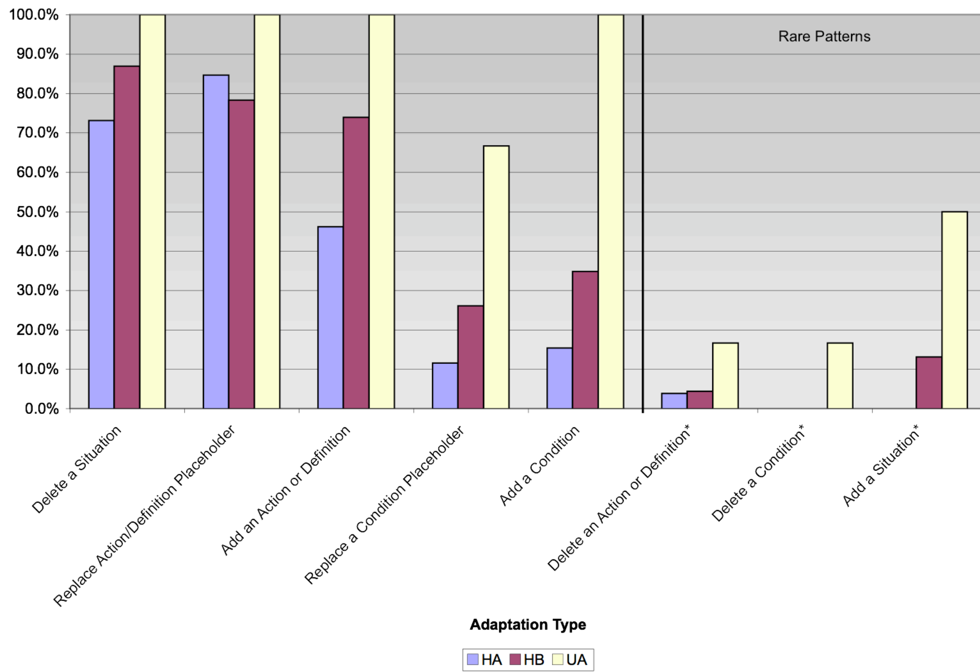

Fig. 19 Percentage of authors using each kind of adaptation operation

We claim that, since the UA authors have higher cognitive skills and experience from more formal education and one or two computer programming courses, the greater difficulty of some pattern adaptation operations was not an impediment – they mastered all of the adaptation operations. Except for the rare adaptations and the *replace a condition placeholder* adaptation, all UA authors used all adaptation operations. None of the stories created by the UA authors who did not perform any *replace a condition placeholder* operation contained any condition placeholders. Therefore, the reason they did not use this operation category was lack of opportunity rather than lack of cognitive skill.

Notice that a greater percentage of HB authors performed each kind of adaptation than the HA authors. We think this result is related to higher cognitive ability. To obtain some evidence for this assertion, we measured IQ scores and creative thinking scores for the authors from groups HA and HB. The average IQ score was 101 for HA authors and 107 for HB authors.

The average creative thinking score was 101 for HA authors and 114 for HB authors. This provides some evidence for our assertion.

The IQ scores were computed using the Shipley Institute of Living Scale (SILS) [44]. SILS is a quick accurate measure of general intellectual functioning of adults and adolescents, ages fourteen and older [43]. The scale consists of two subtests of vocabulary and abstract concept formation. The administration time is ten minutes for each subtest. The total IQ score is reported as an estimated WAIS-R IQ score. The multiple correlation of the SILS estimated total IQ score and WAIS-R IQ score is .87 [44]. The estimated total IQ scores are reported on a scale with a mean of a 100 and standard deviation of 15 [43].

The creative thinking score was measured using the Torrance Tests of Creative Thinking (TTCT) [35] (figural form A test). It evaluates divergent thinking, productive thinking, inventive thinking and imagination, all factors that are commonly thought to be involved with creative achievements [36]. The figural form A test is appropriate for evaluating students in the fourth grade through graduate school in a group setting [38] [9]. It requires responses that are mainly drawn or pictorial in nature, and takes thirty minutes to compete. Artistic quality is not required to receive credit. Standard scores are reported on a scale with a mean of 100 and standard deviation of 20 respectively [37] [35].

Fig. 20 shows the number of each category of adaptation operations that were performed by the authors, aggregated by group. Except for the number of *add action* adaptations performed by UA authors, the decreasing number of adaptations performed by each group across the adaptation categories provides more support for our assertion about the increasing difficulty levels of the adaptation categories. Once again, lower numbers for *replace a condition placeholder* than *add a condition* could be due to opportunity rather than complexity.

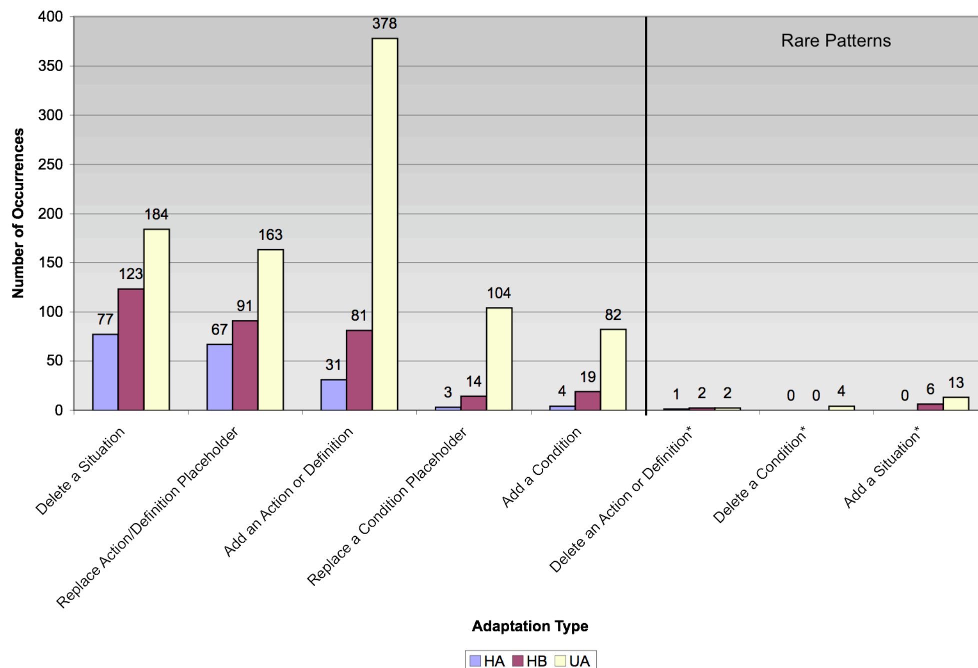

Fig. 20 Number of adaptations from each category that were made by each author group

In this section, we showed empirical evidence that most non-programmers can successfully use the generative/adaptive approach to generate scripting code for computer game stories. Students were not graded on their use of patterns or adaptations, so they used them voluntarily when it made sense to improve their story. All of the students in the high school study produced an interactive story that "worked". This means that it was a playable *NWN* story. In addition, 73% of the students scored a passing grade (at least 50%) on their interactive story. This shows that ScriptEase allows non-programmers to create interactive stories with only six hours of instruction and story writing time. Recall that the same rubric was used to evaluate the interactive stories as the traditional ones. This rubric was based on characters, setting, plot, conflict, theme and style, not on numbers of patterns or adaptations. Of course if a student used the same pattern frequently with the same adaptations, the story would be boring and it would be penalized based on its resulting literary quality, not based on actual pattern use. However, using the same pattern/adaptation combination was not a problem in practice, since the wide range of patterns and adaptations available in ScriptEase encouraged students to write stories that were not repetitive.

The time spent on learning the approach was six hours for each of the two high school groups. Since they are non-programmers, it is reasonable to believe that this time is less than the time they would spend learning the NWScript language if they wanted to do the scripting manually. One could argue that the University students could learn NWScript in the five to six hours they used to learn the generative/adaptive tools, but it is unlikely. Finally, the two high school groups spent about four hours writing their stories. It is highly unlikely they could have manually written the NWScript code that was generated for their stories in this time. Similarly, even if the University students could have learned the NWScript language in five to six hours, it is improbable they could have manually written the scripting code necessary for their stories in the thirteen and a half to fourteen and a half hours they spent writing their story. The number of non-comment lines of ScriptEase-generated code for these six stories ranged from 1,211 to 5,249 lines. Therefore, the case-study satisfied all four of the criteria outlined in Section 5.1.

6. Related Work

We described the related work with respect to scripting languages in Section 1 and other attempts at automatic programming in Section 3. The work most closely related to generative patterns with customization was done in the context of parallel programming environments. Our CO_2P_3S tool (Correct Object-Oriented Pattern-based Parallel Programming System) supported a small number of generative patterns that spanned a large number of application domains [27]. In contrast, ScriptEase has a large number of patterns to span a single application domain. Both approaches are effective, but are fundamentally different with regards to customization.

In CO_2P_3S, the patterns were used to generate Java code that encapsulated the parallel structure of an application (e.g., communication, synchronization, deadlock avoidance). Some customization is done before generation, by setting options to values suitable for the application. The generated code forms an object-oriented framework and further customization consists of creating subclasses of the framework classes and writing hook (call-back) methods in these classes. Although creating these sub-classes can be considered adaptation at the software architecture level (adapting a fixed architecture by providing sub-classes), each class had to be constructed rather than adapted. This means that the "programming" was fundamentally constructive rather than adaptive. In addition, the user would have to write a

large amount of application-specific sequential code (but no parallel code) as well. The goal of the system was to allow experienced sequential programmers to write parallel programs without having to write any parallel code. It removed the need to write parallel code, but not the need to write code. For most parallel applications, the most difficult code to write and debug – the parallel semantics – was generated by the patterns, reducing programming and debugging time [20].

In contrast, ScriptEase is designed for non-programmers. After generation, the customization phase consists of true adaptation rather than construction. The emphasis is on modifying existing descriptions rather than creating them. It is the application-specific nature of the game-genre-specific pattern library in ScriptEase that eliminates the need for constructive programming. In effect, the CO_2P_3S pattern library has application breadth and the ScriptEase pattern library has application depth. The former requires constructive customization and the latter supports adaptive customization. We feel that adaptive customization of pattern-generated descriptions is the "solution" to the automatic programming problem, but only for specific application domains!

7. Conclusion

The thirst for increased and improved quality of content in games is insatiable, driving up the time and cost needed to produce a major game title. The current approaches used by the game industry (manual scripting) do not scale, and game company resources are being stretched to their limit. There is an urgent need for new tools to simplify the game content creation process, including specifying, implementing, and testing game content. Given that the game content is determined by game authors, not by game programmers, the tools used must change to reflect the intended user community. Of necessity, this demands that the specification technique must distance itself from the traditional constructive programming view. Authors are comfortable with natural language. Although a true natural language interface is not possible with today's technology (ideally one would write a story in English, push a button, and create an instant game), one can do more to reduce the conceptual gap. This paper advocates adaptive programming as a major step in that direction. By building on the well-established ideas behind design patterns, ScriptEase allows authors to deal with a level of abstraction that is closer to the language level they normally use for communication. We have demonstrated the validity of our approach by showing that grade 10 English students can build game stories using ScriptEase. Adaptive programming allows non-programmers to build complex programs!

High-school English students participated in two of our case studies to demonstrate that non-programmers can build interesting stories with little experience. Industry game authors are usually computer-savvy, university-educated, skilled story writers, despite potentially limited programming skills. Since high school English students, without programming skills, productively used ScriptEase after only a short learning period, game industry authors can be expected to be even more productive. The programmer remains an important part of the process. As existing pattern libraries evolve and new pattern libraries are created, there may be an ongoing need for new atoms to support them. However, as the pattern libraries mature, the interactions between authors and programmer will be reduced significantly. Game story authors will be able to directly create content without using the programmer as an intermediary. This will eliminate the disconnect between an author's intentions and the game scripts needed to implement them.

There are many directions for future work, but here we highlight two that have generated the most discussion. The first area of research is in patterns. Encounter patterns are relatively straightforward to define and implement in ScriptEase. Considerable research is needed to bring plot, behaviour, and dialogue patterns to a similar state of maturity. For example, behaviour patterns require an underlying concurrency model, since characters have to communicate, synchronize, and avoid deadlock (note the CO_2P_3S connection!). Further, they have to behave in non-repetitive ways; predictability turns game players off. All of this, of course, must be "behind the scenes" from the game author's point of view. Manual sequential scripting by authors is already too hard. Contemplating a process in which authors do manual parallel scripting is unrealistic. Although we have made some progress on generative patterns for behaviours, dialogues and plots, more work needs to be done [12].

The second area of research is in interfaces for authors. ScriptEase uses a textual representation that reads as verbose English. Several authors who have used ScriptEase have commented that a more visually descriptive interface would be better. Kismet, for example, has a visual interface (connect the components in a flow chart). Our experience suggests that a connect-the-components approach is limited and does not scale well to large applications. Using generative patterns that generate visual components whose positions and orientations can be adapted is much more appealing. More work is needed to identify the interface components that simplify the specification and representation of patterns on the screen and the adaptive process that will be used to customize them.

Acknowledgement

This research was supported by the Natural Sciences and Engineering Research Council of Canada (NSERC), the Institute for Robotics and Intelligent Systems (IRIS), and Alberta's Informatics Circle of Research Excellence (iCORE). We want to thank BioWare Corp. for the financial, technical, and administrative support. We would like to thank former ScriptEase team members, James Redford and Dominique Parker who are now "working on the frontline" at BioWare Corp. and Electronic Arts. We would also like to thank the student authors who participated in our study and the teachers at their schools who made these studies possible.

References

[1] 3D GameStudio, Realtime 3D Authoring System, Conitec Datasystems, Inc., 2006, http://www.3dgamestudio.com.
[2] Alice v2.0, Learn to Program Interactive 3D Graphics, Carnegie Mellon, 2006, http://www.alice.org.
[3] Anark Gameface, Anark Corporation, 2006, http://www.anark.com/entertainment/gameface-features.html.
[4] Bethesda Softworks, 2006, http://www.elderscrolls.com/games/oblivion_overview.htm.
[5] BioWare Corp., 2006, http://www.bioware.com.
[6] S. Björk, J. Holopainen, Patterns in Game Design, Charles River Media, 2004.
[7] J. Bosch, Design Patterns as Language Constructs, Journal of Object-Oriented Programming 11(2) (1998) 18-32.
[8] F. Budinsky, M. A. Finnie, J. M. Vlissides, P. S. Yu, Automatic Code Generation from Design Patterns, IBM Systems Journal 35 (2) (1996) 151-171.

[9] C. I. Chase, Review of the Torrance Tests of Creative Thinking, in: J. V. Mitchell Jr. (Ed.), The Ninth Mental Measurements Yearbook, Lincoln: Buros Institute of Mental Measurements, University of Nebraska, 1985, pp. 1631-1632.
[10] J.R. Cordy, T.R. Dean, A.J. Malton, K.A. Schneider, Source Transformation in Software Engineering Using the TXL Transformation System, Special Issue on Source Code Analysis and Manipulation, Journal of Information and Software Technology 44 (13) (2002) 827-837.
[11] M. Cutumisu, C. Onuczko, D. Szafron, J. Schaeffer, M. McNaughton, T. Roy, J. Siegel, M. Carbonaro, Evaluating Pattern Catalogs - The Computer Games Experience, in: 28th International Conference of Software Engineering (ICSE 2006), China, 2006, pp. 132-141.
[12] M. Cutumisu, D. Szafron, J. Schaeffer, M. McNaughton, T. Roy, C. Onuczko, M. Carbonaro, Generating Ambient Behaviors in Computer Role-Playing Games, IEEE Journal of Intelligent Systems (IEEE IS) 21(5) (2006) 19-27.
[13] K. Czarnecki, U. Eisenecker, Generative Programming Methods, Tools, and Applications (Chapter 11 Intentional Programming), Addison-Wesley, Reading, MA, 503-567, 2000.
[14] B. Dawson, Game Scripting in Python, Game Developers Conference (GDC), USA, 2002.
[15] Electronic Arts, 2006, http://www.ea.com.
[16] Epic Games, 2006, http://www.epicgames.com.
[17] Firaxis Games, 2006, http://www.firaxis.com.
[18] Gamebryo Element, 3D Graphics Engine and Tools, Emergent Game Technologies, 2006, http://www.emergent.net/index.php/homepage/products-and-services/gamebryo.
[19] E. Gamma, R. Helm, R. Johnson, J. Vlissides, Design Patterns: Elements of Reusable Object-Oriented Software, Reading, MA, Addison-Wesley, 1994.
[20] Z. Guo, J. Schaeffer, D. Szafron, P. Earl, Using Generative Design Patterns to Develop Network Server Applications, in: 10th International Workshop on High-Level Parallel Programming Models and Supportive Environments (HIPS'2005) at IPDPS, 2005, pp. 178. On CD-ROM.
[21] Humongous Entertainment (Atari), 2006, http://www.atari.com/atarikids.
[22] International Baccalaureate Program, International Baccalaureate Organization, http://www.ibo.org/diploma.
[23] Kismet, Visual Scripting System, Unreal Technology, 2006, http://www.unrealtechnology.com/html/technology/ue30.shtml.
[24] B. Kreimeier, Game Design Patterns, Wordware Publishing Inc., 2004.
[25] K. J. Lieberherr, Adaptive Object-Oriented Software: The Demeter Method with Propagation Patterns, PWS Publishing Company, Boston, 1996.
[26] Lua, The Programming Language, 2006, http://www.lua.org.
[27] S. MacDonald, J. Anvik, S. Bromling, J. Schaeffer, D. Szafron, K. Tan, From Patterns to Frameworks to Parallel Programs, Parallel Computing 28(12) (2002) 1663-1683.
[28] S. MacDonald, D. Szafron, J. Schaeffer, J. Anvik, S. Bromling, K. Tan, Generative Design Patterns, in: 17th IEEE International Conference on Automated Software Engineering (ASE 2002), UK, 2002, pp. 23-34.
[29] M. McNaughton, M. Cutumisu, D. Szafron, J. Schaeffer, J. Redford, D. Parker. ScriptEase: Generative Design Patterns for Computer Role-Playing Games, in: 19th IEEE International Conference on Automated Software Engineering (ASE 2004), Austria, 2004, pp. 88-99.
[30] People Can Fly, 2006, http://www.peoplecanfly.com.
[31] Python, The Programming Language, 2006, http://www.python.org.

[32] Relic Entertainment, 2006, http://www.relic.com.
[33] C. Rich, R. Waters, Automatic Programming: Myths and Prospects, IEEE Computer 21 (8) (1988) 40-51.
[34] D. Szafron, M. Carbonaro, M. Cutumisu, S. Gillis, M. McNaughton, C. Onuczko, T. Roy, J. Schaeffer, Writing Interactive Stories in the Classroom, Interactive Multimedia Electronic Journal of Computer-Enhanced Learning (IMEJ) 7(1) (2005), 13 pages.
[35] E. P. Torrance, Torrance Tests of Creative Thinking – To The Test User, Bensenville, IL: Scholastic Testing Service, 1990.
[36] E. P. Torrance, Torrance Tests of Creative Thinking – Streamlined Scoring Guide Figural A and B. Bensenville, IL: Scholastic Testing Service, 1992.
[37] E. P. Torrance, The Torrance Tests of Creative Thinking Norms – Technical Manual Figural (Streamlined) Forms A & B. Bensenville, IL: Scholastic Testing Service, 1998.
[38] E. P. Torrance, Torrance Tests of Creative Thinking – Directions Manual Figural Forms A and B, Bensenville, IL: Scholastic Testing Service, 2003.
[39] Troika Games, 2006, http://www.troikagames.com.
[40] Ubisoft, 2006, http://www.ubi.com.
[41] Virtools™ 4, Comprehensive Life Platform for Creating Highly Interactive 3D Applications, Virtools, 2006, http://www.virtools.com.
[42] D. Wile, Supporting the DSL Spectrum, Journal on Computing and Information Technology (CIT) 9(4) (2001) 263-287.
[43] R. A. Zachary, Shipley Institute of Living Scale – Revised Manual, Los Angeles: Western Psychological Services, 2003.
[44] R. A. Zachary, M. J. Paulson, and R. L. Gorsuch, Estimating WAIS IQ from the Shipley Institute of Living Scale Using Continuously Adjusted Age Norms, Journal of Clinical Psychology 41(6) (1985) 820-831.