This is a pre-print of a paper that will appear in the Proceedings of 28th International Conference on Software Engineering, held May 2006 and it will be published by ACM.

Evaluating Pattern Catalogs - The Computer Games Experience

M. Cutumisu[1], C. Onuczko[1], D. Szafron[1], J. Schaeffer[1], M. McNaughton[1], T. Roy[1], J. Siegel[1], M. Carbonaro[2]

[1]Department of Computing Science, University of Alberta
[2]Department of Educational Psychology, University of Alberta

{meric, onuczko, duane, jonathan, mcnaught, troy, siegel}@cs.ualberta.ca

mike.carbonaro@ualberta.ca

ABSTRACT

Patterns and pattern catalogs (pattern languages) have been proposed as a mechanism for re-use. Traditionally, patterns have been used to foster design re-use, and generative design patterns have been used to achieve both design and code re-use. In theory, a pattern catalog could be created and used to provide re-usable patterns within a project and across a group of related projects. This idea raises a natural question. How can we measure the effectiveness of a pattern catalog or compare the effectiveness of different pattern catalogs? In this paper, we define four metrics that can be used to measure the effectiveness of pattern catalogs. We illustrate these metrics by applying them to a case study that uses a pattern catalog of generative design patterns to generate scripting code for computer games. The metrics are general enough to assess any pattern catalog, independent of application domain or whether the patterns are generative or descriptive.

Categories and Subject Descriptors

D.2.2 [**Software Engineering**]: Design Tools and Techniques – *Computer-aided software engineering (CASE), Software libraries;* D.2.6 [**Software Engineering**]: Programming Environments – *Integrated environments, Interactive environments;* D.2.8 [**Software Engineering**]: Metrics – *Performance measures, Process metrics.*

General Terms

Performance, Design, Reliability, Experimentation, Languages.

Keywords

Pattern catalog, pattern language, generative design pattern, scripting language, code generation, computer game, metric.

1. INTRODUCTION

Our work is inspired by design patterns used to describe object collaborations in graphical user interfaces and other software systems [7]. A *design pattern* specifies the solution to a general software design problem at a higher level of abstraction than the program code that implements the design. Traditional software design patterns are *descriptive*. Each pattern provides a design lexicon, a set of solution structures and the reasoning behind the solutions. Since each pattern provides a family of solutions, it must be *adapted* to a specific context during program construction and then manually translated to code. Experienced programmers who have implemented the same design pattern in other contexts can usually perform adaptation and coding more quickly than novices, where unfamiliar or ambiguous natural language pattern documentation can lead to slow progress and coding errors. Generative design patterns (GDPs) have been used [4][5][12] to speedup code production and reduce coding errors for novice and experienced programmers alike. Recently, we were the first group to use GDPs to generate scripting code for computer games [14][15]. The author selects a series of patterns, adapts them to the story being written, and our GDP tool automatically generates scripting code for the adapted patterns. A generative design pattern generates code that implements the pattern. The actual code generated depends on the target programming language and API. The same GDP can generate code for different languages and APIs and it is often adapted before code generation. Therefore, the GDP is a design-time artifact (design pattern) rather than a code-time artifact (code pattern).

The current state-of-the-art in computer games is to manually script individual game objects to provide desired interactions for each game story. For example, BioWare Corp.'s [2] popular Neverwinter Nights (NWN) [16] campaign story contains 54,300 game objects of which 29,510 are scripted, including 8,992 objects with custom scripts, while the others share a set of pre-defined scripts. The scripts consist of 141,267 lines of code in 7,857 script files. Our research has shown that a small set of parameterized patterns (familiar commonly occurring scenarios) can characterize most of the object interactions used in game stories. These patterns can be used to specify and generate the necessary scripts [13]. We have identified four kinds of patterns that are necessary to generate all of the scripts found in computer role-playing game (CRPG) stories: *plot*, *behavior*, *dialog* and *encounter*. In this context, a pattern is a commonly occurring scenario or idiom in a story of the appropriate genre. *Plot patterns* guide the player character (PC) through the story. *Behavior patterns* are used to specify the actions of non-player characters (NPC). *Dialogue patterns* control conversations between characters. *Encounter patterns* are used to script interactions between the PC and inanimate objects (props) in the game.

For example, a story may require that a shield be placed into a chest to unlock the door to a passage. The story may also require

ICSE'06, May 20–28, 2006, Shanghai, China.

another script that locks a crypt room door and spawns a mummy whenever a character removes an "eye gem" from a statue. In fact, we can abstract both of these scenarios into a single encounter pattern, which we call *Container disturb (specific item) toggle door*. The author can select two different instances of this encounter pattern and adapt them to create the desired scenarios. This pattern applies when a PC disturbs a container by adding or removing a specific item. When this occurs, a nearby door is toggled (unlocked and opened if it is locked, or closed and locked if it is unlocked). This pattern has three options, the *container*, the *specific item*, and the *door*. To adapt the pattern to the first scenario requires only that the options be set: the *container* as the chest, the *specific item* as the shield, and the *door* as the passage door. To adapt the pattern to the second scenario requires two steps. First the options must be set: the *container* as the statue, the *specific item* as an "eye gem", and the *door* as the crypt room door. Second, this pattern must be further adapted by adding an additional action to the pattern to spawn a mummy. Adaptation is described more fully in Section 3. There are several other kinds of pattern adaptations besides setting options and adding actions. All adaptation occurs before scripting code is generated. Recognizing that patterns must be adapted is essential to building effective metrics for evaluating pattern catalogs (pattern languages).

Our GDP approach has two major advantages for scripting computer game stories. First, a story can be created at a higher level of abstraction. The same patterns can be used to generate scripting code for different game engines on the same platform or for variants of the same game engine on multiple platforms (i.e., game console or computer). Second, the cost of creating a pattern can be amortized over all of the times the pattern is used. This re-use occurs at two levels of scope: 1) patterns created while writing the beginning part of a story can be re-used many times throughout the story, and 2) patterns created for a story can be re-used in other stories. Consequently, both advantages significantly reduce the effort required to script, test, and find errors in a story.

Our goal was to eliminate the need for manual scripting by putting GDPs in the hands of authors. We provided authors with a pattern catalog and a tool called ScriptEase [20] that generates scripts for each pattern. Since most of the uses of general software design patterns involve relatively few patterns that occur relatively few times, it was not clear whether we could obtain the coverage necessary to generate all of the scripting code in a computer game. In addition, a pattern catalog is not a static entity, it is meant to evolve by expanding (and contracting) to satisfy the needs of authors. We realized that if our pattern catalog must evolve, we would need a mechanism to measure its current effectiveness and its potential effectiveness after patterns are added or removed. More generally, we needed a way to compare several different pattern catalogs to decide which one was most effective. To our knowledge, there has not been a documented attempt to evaluate the quality of pattern catalogs, although there has been research aimed at measuring the quality of software produced by design patterns [8][9][22]. In this paper, we present our solution – using metrics to evaluate pattern catalogs. We propose four metrics. We illustrate how to use these metrics by applying them to a pattern catalog that generates scripting code for computer role-playing games. However, the metrics we propose can be applied to any pattern catalog for any application domain, regardless of whether the catalog contains descriptive or generative patterns.

In Section 2, we provide an introduction to BioWare Corp.'s popular Neverwinter Nights game system that our pattern catalog was built to support. In Section 3, we provide high-level descriptions of some of the patterns, using two other CRPGs in addition to NWN. We also discuss pattern adaptation, since it is an essential factor in measuring the effectiveness of pattern catalogs. In Section 4, we define our four metrics for evaluating pattern catalogs and provide some rationale about why these metrics are desirable. In Section 5, we illustrate how these metrics are used, by applying them to a pattern catalog for NWN stories.

Rather than creating several artificial pattern catalogs for a series of small applications, we decided to test our idea of using metrics to measure pattern catalog effectiveness on a large pattern catalog for a sizable application. This decision has the disadvantage of providing only weak evidence for the generality of our idea – the case for generality rests with the intuition of the reader. However, the advantage of our decision is a demonstration that pattern catalog metrics can be used effectively to guide the development of a real pattern catalog for a real commercial application.

2. NEVERWINTER NIGHTS

Neverwinter Nights [16] is a critically-acclaimed award-winning (86 awards) multi-player CRPG published by BioWare Corp. NWN consists of a general-purpose game engine that renders graphics for game objects and characters, plays sounds, interprets user input, and runs scripts in response to game events. Authors create modules containing chapters of their story and the game engine interprets the modules. The game engine can play game modules in any setting imaginable. The storywriter has only to provide the appropriate backgrounds, props, and NPCs. The NWN game has an official campaign story and two expansion-pack stories set in a fantasy world. The campaign contains more than 13 separate modules that can be played alone or online with friends. NWN has an active player community with thousands of players who contribute modules of their own creation that can be freely downloaded from the NWN Vault web site [18]. The most popular of the 4,100 modules at this site has been downloaded over 259,000 times as of February 2006, and the tenth-most, 95,000 times. The NWN game includes the Aurora Toolset, the same CAD tool for building story modules which BioWare authors themselves used to build the official campaign modules. The toolset allows an author to create the physical landscape of a module and populate it with game objects. The author can also write scripts in BioWare's NWScript language and attach them to game objects, to specify how they will react to game events. The NWN game engine operates on eleven types of game objects: *modules*, *areas*, *creatures*, *doors*, *placeables*, *triggers*, *monster encounters*, *merchants*, *items*, *sounds*, and *waypoints*. The first eight object types may have scripts attached to them. Our pattern catalog contains encounter patterns that generate scripts for placeables, triggers and doors. A *placeable* is an inanimate object that can be placed anywhere in the story world. Examples include chests, statues, chairs, tables, levers, and piles of rubble. A placeable is considered a *container* if it can hold items. A *door* can only be placed at the entrance to a structure or between two rooms in a structure. A *trigger* is a region of space that generates an event when a character enters or exits its perimeter. The *trigger* object type supports the following scriptable events: *OnClick*, *OnEnter*, *OnExit*, *OnHeartbeat* and *OnUserDefined*. Our behavior patterns generate scripts that are attached to creatures. Scripts attached to one object may refer to other objects by a string identifier ("tag") set by the writer.

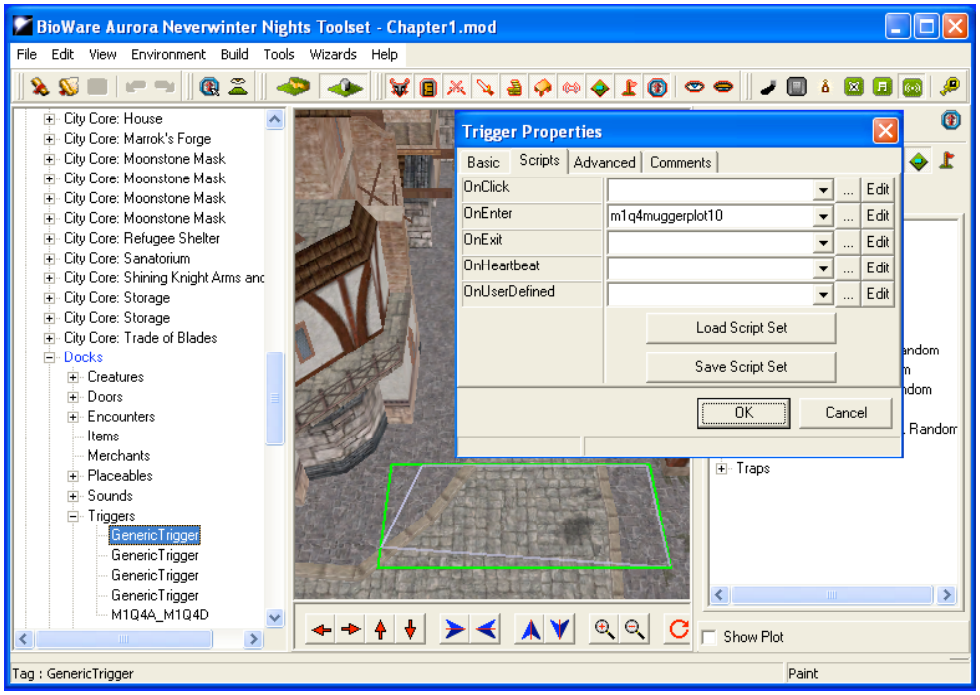

Figure 1. The main window of the Aurora Toolset.

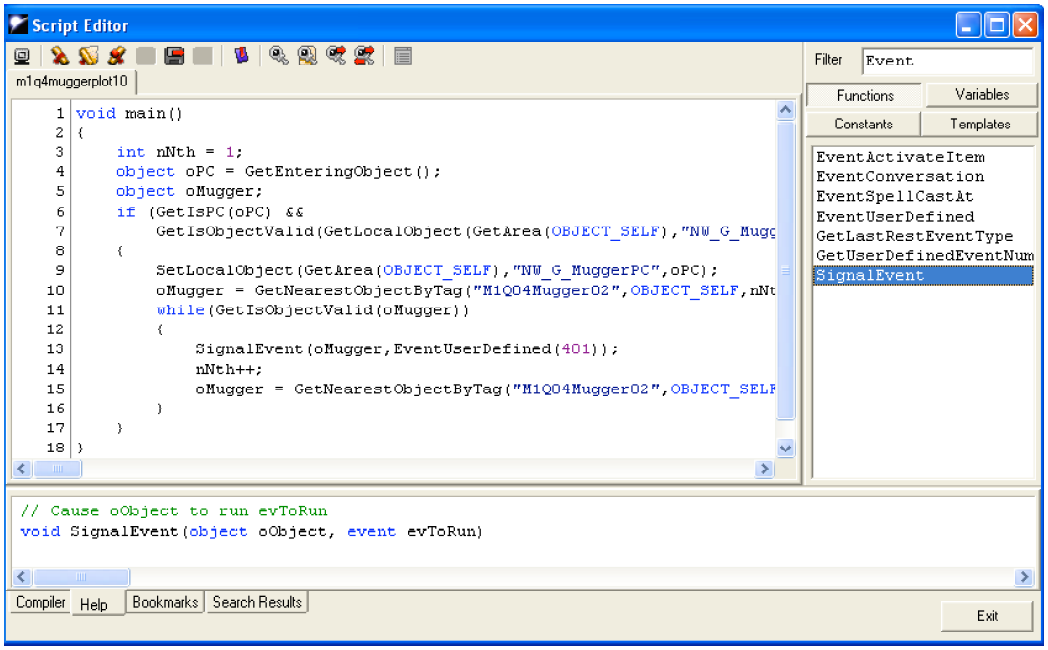

Figure 2. The Aurora Toolset Script Editor.

Figure 1 shows part of an area in the Chapter One module of the campaign story as it appears in the Aurora Toolset. A *trigger* object is selected in the main pane and its perimeter is highlighted. The *Trigger Properties* dialog box for the selected trigger object is opened and the *Scripts* tab selected to show that there is a script named *m1q4muggerplot10* that will execute if the PC enters the trigger area (event *OnEnter*). Figure 2 shows the Script Editor that can be used to enter scripts manually and the complete NWScript script that is attached to the *OnEnter* event of the trigger object selected in Figure 1. This script iterates over all objects in the area with tag *M1Q04Mugger02* and signals an *EventUserDefined* to each of these muggers so that they will attack the PC. NWScript is a C-like language, which requires the user to understand variables, objects, loops, conditionals, function calls and a (large) API library. The library functions that include "Event" and the prototype (signature) of the highlighted *SignalEvent* function are

shown in Figure 2. The script for the *EventUserDefined (401)* call that causes an individual mugger to attack the PC is not shown.

Scripting is very difficult for the many authors who lack programming experience. BioWare's scripting forums [17] are full of questions from eager authors who are trying to learn both the scripting language and general programming concepts. Frequent queries ask for help identifying which function calls are needed to achieve certain effects in the game. Another set of popular questions ask for help identifying which event slot of an object a script should be attached to, so that it will run at the desired time. General programming questions like "what is a local variable?" are also seen from some novice authors. Publicly available tools such as ScriptEase [20] and Lilac Soul Script Generator [10] have been created to reduce or eliminate the need for manual scripting.

3. USING PATTERNS

The pattern catalog described in this paper contains encounter patterns for triggers, placeables and doors. An example of a trigger pattern is *Trigger enter – creatures attack*. This pattern can be used to automatically generate a script with the same functionality as the manually-written script shown in Figure 2. The ScriptEase version of this pattern is shown in Figure 3. An *E* icon represents the encounter pattern. To use this pattern, a story designer selects the pattern name from a menu to create an instance of the pattern and then adapts the pattern instance by setting the two options shown as tabs in the bottom pane: *The Trigger* and *Creature From Faction*.

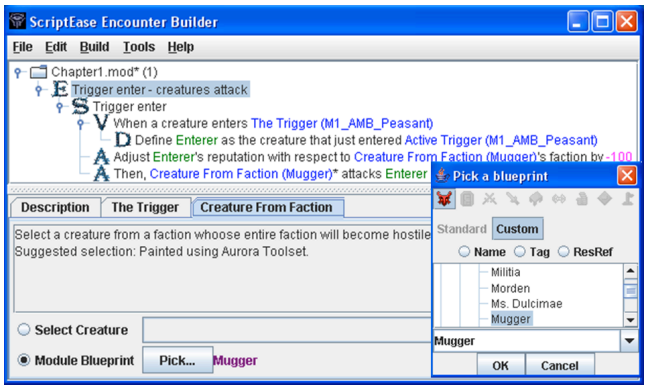

Figure 3. A simple instance of the *Trigger enter – creatures attack* encounter pattern.

The dialog in Figure 3 sets the *Mugger* creature as the second option. A similar dialog sets the first option to the *Generic Trigger* shown in Figure 1. The pattern in Figure 3 shows its component parts. Each encounter pattern contains one or more event-driven scenarios called *situations* (*S* icon). Each situation contains the *event* (*V*) that activates it and a set of *definitions* (*D*), *conditions* (*C*) and *actions* (*A*). Figure 3 shows no conditions, but Figure 4 does. Although ScriptEase supports generative patterns, this encounter pattern would have the same components if it were used in a descriptive pattern catalog. However, in that case, the components would serve as a specification of what a programmer should implement rather than generating the appropriate scripting code automatically, as is done in ScriptEase. This pattern can be used to replace 6 scripts in the NWN campaign story.

An example of a placeable encounter pattern is *Placeable use – toggle door*. When a creature uses the placeable, the status of a door is toggled from opened and unlocked to closed and locked, or vice versa. In the NWN campaign story, this pattern is often applied to a lever to allow it to control a door. This pattern can be used to generate 8 scripts for the NWN campaign story.

An example of a simple door encounter pattern is *Door open failed – show monologue*. This pattern can be used to automatically generate a script that displays a line of text when the PC tries to open a door that is locked. This simple pattern can be used to replace 5 scripts in the NWN campaign story.

3.1 Encounter Patterns in Other CRPGs

All CRPGs have similar patterns to those described in this Section. In this paper, we use two other CRPGs to illustrate the generality of the encounter patterns contained in our pattern catalog. The Elder Scrolls III: Morrowind by Bethesda Softworks [1] and Fable by Lionhead Studios [11] both contain scripts that could be generated by encounter patterns, including the example patterns described in this Section.

In Morrowind, there are many levers that open doors so the *Placeable use – toggle door* pattern could be used extensively to generate these scripts. Often we find situations where an author will re-use an existing script to save work, rather than writing a more appropriate script from scratch. A pattern catalog with many patterns solves this problem. For example, in Fable there is a situation where the PC encounters four rocks and a door. The scripts are set up so that the PC must attack the rocks in the correct order to open the door. However, the pattern *Placeable use – toggle door* could be used to generate improved scripts to control this situation in a more appropriate way, by having the PC simply use (touch) the rocks rather than attacking them.

The pattern *Door open failed – show monologue* could be used in several situations in Fable where the PC encounters doors called Demon Doors. These are doors that require the player to solve some sort of riddle in order for them to open. When a PC tries open a locked door, the door speaks a riddle to the PC. This pattern could also be used in Morrowind. For example, at the beginning of Morrowind, the player is given many hints. When a PC tries to enter a door, the door reminds the PC to obtain a ring from a nearby barrel.

Either of the patterns, *Trigger enter – spawn creature near object* or *Trigger enter – creatures attack*, could be used to generate scripting code for ambushes in Fable. At one point in the Fable story, the PC is asked to escort a person to a nearby farm. When the person being escorted enters a trigger, an enemy is spawned nearby to attack the person. This pattern can be used for many purposes other than ambushes. For example, in Morrowind, there is a situation where a manually-written script spawns a creature high above the PC and the creature falls. This situation is used to illustrate what happens when a jumping potion is misused. A script that provides the intended semantics could be generated from the *Trigger enter – spawn creature near object* pattern.

3.2 Pattern Adaptation

Whenever a pattern is used, it must be adapted to meet the context of the story being designed. The simplest form of adaptation is setting the options of the pattern. For example, in Figure 3, there are two options: *The Trigger* and *Creature From Faction*. Other forms of adaptation include adding or removing components. As an example, there is a scene in Chapter 2 of the NWN campaign story where a creature attacks the PC if the PC gets too close and has not yet answered a riddle correctly. Instead of a script for this

scene, an author could generate the script from an instance of the *Trigger enter – creatures attack* pattern. This would require the author to add a definition that determines whether the player has the plot token (indicating that the riddle has been answered correctly) and to add a condition that tests the plot token. The adapted pattern is shown in Figure 4. The author can add this definition and condition by selecting them from menus and setting options. The author can also add the *Show caption above object* action highlighted in Figure 4, and set its options, such as the *Caption* option shown.

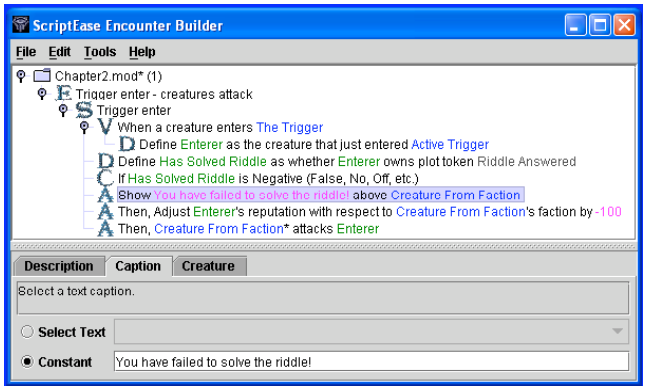

Figure 4. A more adapted instance of the *Trigger enter - creatures attack* pattern.

Table 1. Cognitive levels of pattern adaptation.

Cognitive Level	Adaptation	Number of Adaptations
1	Set pattern options	154
2	Delete a situation	123
3	Delete an action/definition	2 (*)
4	Delete a condition	0 (*)
5	Replace an action/definition placeholder	91
6	Add an action/definition	81
7	Replace condition placeholder	14
8	Add a condition	19
9	Add a situation	6 (*)

We identified and added ScriptEase support for the nine pattern adaptation levels of Table 1, in increasing order of cognitive difficulty. These load levels constitute a ranking, but not a scale. We showed that each adaptation technique is simple enough for non-programmers in a case study (Section 5.4) where students from a high school English course used these adaptation techniques to write an interactive short story with ScriptEase [20]. The third column in Table 1 shows the number of adaptations that were done of each kind during the case study. Asterisks denote adaptations that are rarely used, independent of their difficulty. The reasons for rareness of adaptation use are beyond the scope of this paper and will be discussed in a future paper. Determining a relative scale of difficulty for each kind of adaptation will require another study.

4. METRICS FOR PATTERN CATALOGS

We define four metrics for pattern catalogs, with respect to a particular application, *usage*, *coverage*, *utility*, and *precision*. A good pattern catalog (catalog) is one that has high usage, coverage utility and precision across a broad range of applications. The usage and coverage metrics are based on counting patterns in the catalog and the application, while the utility and precision are based on instances of patterns and adaptations of instances. Formal definitions are given in Figure 5.

$$PCat \overset{def}{=} \{patterns\ in\ pattern\ catalog\}$$

$$IApp \overset{def}{=} \{adapted\ instances\ of\ patterns\ used\ in\ the\ application\}$$

$$i_p \overset{def}{=} an\ unadapted\ instance\ of\ pattern\ p$$

$$\overline{i_p} \overset{def}{=} an\ adapted\ instance\ of\ pattern\ p \Leftrightarrow$$

$$\exists\ adaptations\ a_1 \cdots a_n \ni \overline{i_p} = a_n \cdots a_1 i_p$$

$$PApp \overset{def}{=} \{patterns\ used\ in\ the\ application\} \overset{def}{=} \left\{ p \ni \overline{i_p} \in IApp \right\}$$

$$IAppCat \overset{def}{=} \left\{ \overline{i_p} \in IApp \ni p \in PCat \right\}$$

$$usage(catalog, application) \overset{def}{=} \frac{|PCat \cap PApp|}{|PCat|}$$

$$coverage(catalog, application) \overset{def}{=} \frac{|PCat \cap PApp|}{|PApp|}$$

$$utility(catalog, application) \overset{def}{=} \frac{|IAppCat|}{|PCat \cap PApp|}$$

$$precision(catalog, application) \overset{def}{=} \frac{|PCat \cap PApp|}{\sum_{\overline{i_p} \in IAppCat} n \ni \overline{i_p} = a_n \cdots a_1 i_p}$$

Figure 5. Formal definitions of the four effectiveness metrics.

The *usage* is the ratio of patterns used in the application that come from the catalog to the total number of patterns in the catalog. For example if there are 24 patterns in the catalog and only 8 of these are used in an application, the usage is 8/24 = 0.33. The usage is always in the range 0.0 to 1.0 and a higher usage is better. Usage is important since the cognitive load of using a catalog increases if there are many patterns in the catalog that are not used in an application. The existence of many non-applicable patterns in the catalog may make it hard to find the few applicable patterns.

The *coverage* is the ratio of catalog patterns used in the application to the total number of patterns used in the application. For example, if the application requires 10 different patterns and only 8 are found in the catalog, the coverage would be 8/10 = 0.8. The coverage is always in the range 0.0 to 1.0 and higher coverage is better. Coverage is important since using a catalog without all the necessary patterns means that the application designer will either have to design new patterns or write the necessary application code by hand. The time/cost required to create a new pattern (or code by hand) is usually much greater than the time/cost to instantiate and adapt an existing pattern.

The *utility* is the ratio of pattern instances in the application whose patterns are in the catalog to the total number of patterns used in the application that come from the catalog. Alternately, the utility is the average number of times each catalog pattern that appears in

the application is actually used. For example, if 40 instances of patterns are used in the application that come from 8 different patterns that are in the catalog, the utility would be 5. This indicates that on average, for each pattern in the application taken from the catalog, there are 5 instances used in the application. The utility is important since there is a cognitive load for learning to use a particular pattern from the catalog, which is amortized over the number of times a pattern is used in an application. Higher utility is better. Using 5 instances of 8 different patterns is easier for the application designer than using 1 instance each of 40 different patterns, since it takes time to learn about the patterns. Utility ranges from 0.0 to any positive value, with higher utility better. The utility of a pattern catalog is not affected by patterns used in the application that do not come from the catalog.

The *precision* is the ratio of the total number of patterns used in the application that come from the catalog to the number of adaptations required for these pattern instances. Alternately, it is the reciprocal of the average number of adaptations that must be performed on the pattern instances in the application that come from the catalog. If an application requires a total of 100 adaptations distributed over 40 pattern instances from the catalog that are used in the application, then the precision would be 40/100 = 0.4. Since at least one adaptation is needed per pattern (to set the options), precision can be any number between 0.0 and 1.0, with higher precision being better.

In the case of CRPG authoring, the possible adaptations are given in Table 1. If an author sets the pattern options, adds two actions and deletes a condition, the number of adaptations is 4. Each pattern needs at least one adaptation since setting the options counts as one adaptation. For other kinds of generative design patterns, adaptation steps could include setting function or method parameters, adding code to methods or functions, adding methods to classes or call-back functions to the application, and adding sub-classes. For descriptive design patterns, adaptation involves making design decisions, such as using safe or transparent sub-classes in a *Composite* design pattern [7], as well as the design of interfaces and sub-classes. Some adaptation is always necessary, since patterns are general solutions to design problems and they must be customized for the applications that use them.

Precision is important since any pattern can be turned into any other pattern by removing all components from the old pattern and replacing them by new components. However, the fewer adaptations that must be done when using a pattern catalog, the easier it is to design the application. An alternate definition of precision replaces the number of adaptations with a cost metric that differentiates between the cognitive levels of adaptations or the time required to make different adaptations. For CRPG patterns, a ranking of adaptation costs is given in Table 1. Here is the more general formal definition of precision:

$$precision(catalog, application) \stackrel{def}{=} \frac{\left|PCat \cap PApp\right|}{\sum_{\overline{i_p} \in IAppCat} \sum_{k=1,n} cost(a_k) \ni \overline{i_p} = a_n \cdots a_1 i_p}$$

In this paper, we use a simple cost metric that assigns an equal cost of 1 to each kind of adaptation listed in Table 1. Hence, this formula reduces to the Figure 5 precision formula.

The precision of a pattern catalog does not measure the simplicity/complexity of the code produced from the pattern instances, nor the simplicity/complexity or entertainment value of the story in our CRPG case study. Precision measures the required amount of manual adaptation of the patterns available in a pattern catalog to adapt them to the specific pattern instances that the application programmer (author) wishes to use.

There is a tension between usage and coverage. In general, adding many patterns increases the coverage of a catalog, but reduces the usage. There is also a tension between utility and precision. Generalizing the patterns in a catalog improves utility since more instances of the same pattern can be used. Unfortunately, generalizing patterns also decreases precision, since a more general pattern requires more adaptation to use. For example, in the CRPG domain, assume we have a pattern catalog that contains the two patterns *Container disturb – (specific item) toggle doo*r and *Container disturb – (specific item) spawn creature*. The first pattern has been described previously. The second pattern simply spawns a creature instead of toggling a door, when a specific item is placed in (or removed from) a container. Assume we have a story that contains both patterns. If we replace these two patterns by a more general pattern such as *Container disturb – (specific item)*, then the utility of the pattern catalog for this story increases since the denominator (the number of patterns from the catalog used in the story) is reduced by one and the numerator (the number of pattern instances used in the story) remains the same. However, the precision decreases since the story designer will have to adapt the more general pattern*, Container disturb – (specific item)* by manually adding actions to each pattern instance, to toggle the door and to spawn a creature, respectively.

There are two challenges for a pattern catalog designer.

- Deciding on whether to include a pattern in the catalog or not by trying to give high coverage across a wide range of applications without reducing usage by including too many patterns that are only used in a few applications.
- Deciding on the level of generality or specificity of a pattern by balancing the high utility (re-use) provided by many instances of a general pattern against the low precision of many instances of a general pattern that will require too many adaptations.

These metrics cannot be used to build a pattern catalog independent of the applications that will be written using the catalog. The process is an iterative one. As an application is built, a set of patterns is created and a preliminary decision is made about how general/specific each pattern should be. As the application matures or new applications are created, the metrics can be used to adjust which patterns are added or removed from the catalog and the generality/specificity of each pattern in the catalog. This involves computing the metrics for alternate designs and comparing them to make the best overall decision that balances generality against required numbers of adaptation.

5. CASE STUDIES

We developed a pattern catalog for CRPG authors containing 60 patterns [19], implemented these patterns in ScriptEase and made the catalog available online [20]. Many NWN authors are using ScriptEase and its associated pattern catalog. As of February 2006, there have been more than 12,000 downloads. To show that this pattern catalog and tool are industrial strength, we are using them to replace all of the manually-written scripts that are used in BioWare's Neverwinter Nights official campaign stories. We illustrate our metrics by applying them to this pattern catalog and the stories in the NWN official campaign. We also apply our metrics to a set of 23 short stories that were designed by a class of grade 10 high school English students.

5.1 The Full Pattern Catalog

In a previous paper [13], we described how we used a set of 24 encounter patterns to generate all scripting code attached to placeable objects in the NWN official campaign story. In that experiment, we replaced 497 calls to 182 different scripts comprising 1,925 non-comment lines of hand-written code by pattern-generated code using 431 instances of the 24 patterns. In that paper, we focused on showing how patterns could be used to foster re-use and reduce errors, rather than focusing on the pattern catalog and how it could be evaluated. However, we can use the data gathered in that experiment to help compute the usage, coverage, utility and precision of our expanded pattern catalog.

Our new pattern catalog consists of 60 patterns, including the 23 placeable encounter patterns and 1 dialog pattern used in [13], as well as 5 additional placeable encounter patterns, 15 door encounter patterns, 13 trigger encounter patterns, and 3 behavior patterns (attached to creatures). To compute the usage, coverage, utility and precision of our pattern catalog with respect to the NWN campaign story, we augmented the data gathered from the previous experiment with new data obtained by using ScriptEase. We eliminated all of the hand-written scripts for doors and triggers in the NWN campaign story using the door and trigger encounter patterns from this catalog. The reason for including the three behavior patterns is that they are used to replace scripts that were attached to triggers in the NWN campaign stories. In each of these cases, a behavior pattern rather than a trigger simplified the implementation considerably. Our pattern catalog includes 11 patterns that are not used in the campaign story. These patterns were created to write our own stories before any case studies on the campaign story were done. We did not remove any patterns in our catalog prior our case studies, since they have been useful in other stories and we feel that they will be useful in future stories.

In addition to the broader goal of introducing metrics to evaluate pattern catalogs, the new study involves twice as many patterns and twice as many lines of replaced manual scripting code. Table 2 shows the difference in scope of the two studies.

Table 2. Pattern usage and code replaced in two studies.

Study	Previous Study	New Study
Patterns used	24	49
Pattern instances	431	796
Script calls replaced	497	884
Scripts replaced	182	516
Lines of code replaced	1,925	4,694

In the previous study, we did not compute the number of adaptations made to each pattern instance. However, in this study we needed the adaptation data to apply our precision metric to the pattern catalog. Therefore, in addition to computing adaptations for the new patterns, we computed them for the patterns used in the previous study as well.

In the previous paper [13], we reported the extent of re-use of placeable patterns created for a particular chapter in subsequent chapters. The official campaign story consists of seven chapters: The Prelude, Chapter One, Chapter One Finale, Chapter Two, Luskan and Host Tower, Chapter Three and Chapter Four. These chapters comprise five sub-stories (each with its own sub-plot and all the other components of a short story), where The Prelude, Chapter One and Chapter One Finale form the first story, which we call One*. We concluded that as more stories are written, the high degree of pattern re-use across stories reduces the number of new patterns that would need to be created for later stories. Since we included trigger patterns (13) and door patterns (15) in our pattern catalog metric study, we also recomputed our previous re-use statistics to see if our previous conclusion was still valid across a wider set of patterns. Figure 6 shows that our conclusion is still valid. The bars show which story a pattern was created for and where it was re-used. For example, first we replaced the manually-written scripts in story One* by pattern generated scripts. Story One* uses 24 new patterns created for it and 9 original patterns created before any of the stories were written. Story Two was converted next and it uses 12 original patterns, 11 patterns written for story One* and only three new patterns. Story Luskan has only four new patterns, story Three has five new patterns and story Four has no new patterns. Once a pattern catalog is established, it grows very slowly.

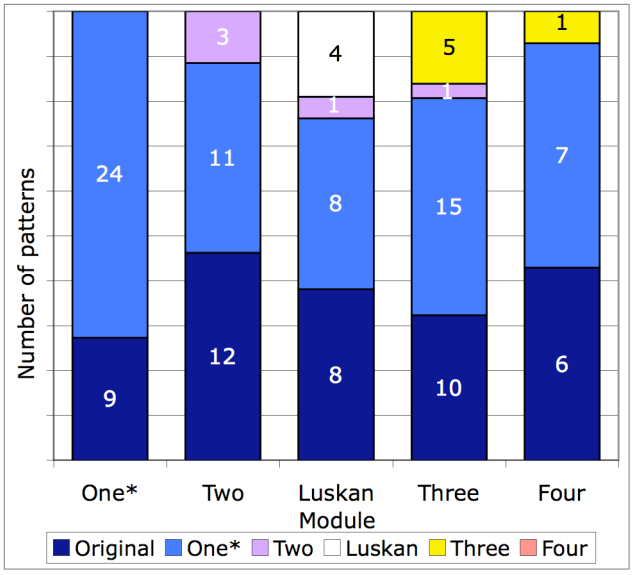

Figure 6. Pattern re-use across a sequence of stories.

5.2 Evaluating the Full Pattern Catalog

We used our four metrics to evaluate our full pattern catalog on six different stories, each of these five short stories considered as a separate story (One*, Two, Luskan, Three and Four) and the complete story (Composite). The results are shown in Table 3. The coverage is 1.00 for all of the stories since the pattern catalog was designed to include all patterns needed for these stories (plus some others). The usage for each story is consistent with the number of patterns used – which is consistent with the relative sizes of the stories. The usage for the composite story is 0.82, which indicates that 82% of the 60 patterns in the pattern catalog (49 patterns) are used in at least one of the sub-stories. The utility of the pattern catalog varies from 4.33 for Luskan to 7.52 for One*. This indicates good pattern re-use in each story, since on average, the patterns were re-used 4 to 8 times. In the composite story, the 49 patterns were re-used an average of 16.24 times. The precision ranges from 0.29 for One* to 0.17 for Luskan. This indicates that a consistent effort was needed to adapt the patterns in the catalog, regardless of the story in which they were used.

Sometimes it is useful to decompose a pattern catalog into sub-catalogs and evaluate some metrics on the sub-catalogs to see which kind of patterns contribute to the differences. Our pattern catalog naturally decomposes into three sub-catalogs, based on the type of object that the pattern is attached to: 1) placeable (including 1 dialog pattern), 2) door and 3) trigger (including 3 behavior patterns). Table 4 shows the precision for the three sub-catalogs and the full catalog for each of the stories. The table shows the effort required to adapt each of these kinds of patterns.

Table 3. Usage (Use), coverage (Cov), utility (Util) and precision (Prec) for the full pattern catalog (60 patterns), where PI is the number of pattern instances used in the story.

Story	PI	Use	Cov	Util	Prec
One*	248	0.55	1.00	7.52	0.29
Two	184	0.43	1.00	7.08	0.22
Luskan	91	0.35	1.00	4.33	0.17
Three	196	0.52	1.00	6.32	0.20
Four	77	0.23	1.00	5.50	0.28
Composite	796	0.82	1.00	16.24	0.23

Table 4. Precision for three sub-catalogs.

Story	Placeables	Doors	Triggers	Full
One*	0.38	0.25	0.24	0.29
Two	0.33	0.13	0.16	0.22
Luskan	0.20	0.12	0.16	0.17
Three	0.21	0.28	0.17	0.20
Four	0.40	0.25	0.17	0.28
Composite	0.28	0.20	0.18	0.23

Table 4 reveals that there are wider variations in the number of adaptations required across two of the sub-catalogs (placeables and doors) than across the full catalog. The low precision in the door sub-catalog for stories Two (0.13) and Luskan (0.12) is due to two different reasons (that are not apparent in the table). In Two, there are 6 doors out of 14 doors whose scripts are unique and moderately complex, each requiring 12 to 16 adaptations. In Luskan, there are 2 doors out of 12 doors whose scripts are unique and very complex, one requiring 35 adaptations and the other requiring 40 adaptations. Table 4 shows that the stories Luskan and Three have placeable sub-catalog precision (0.20 and 0.21) that require almost twice as many adaptations as the other stories. Once again, this phenomenon is due to a small number of complex pattern instances that require a large number of adaptations. For example, the Luskan has four placeable pattern instances out of 51 with 24 – 63 adaptations and Three has five instances out of 124 that require 24 – 86 adaptations.

As indicated in Section 4, a pattern catalog designer must make trade-offs between the generality and specificity of patterns put in a catalog to obtain the best overall scores for the four metrics. The precision scores for our pattern catalog shown in Table 3 allow us to highlight an example of such a trade-off. We consider the set of patterns from the catalog that are based on the *Container disturb* event. The names, number of uses and number of adaptation steps required for the composite campaign story are listed in Table 5.

There are several alternatives to including these three patterns in the catalog. One alternative (Catalog 1) would be to use a single pattern, *Container disturb*, and use an instance of this general pattern in the story wherever an instance of any of these three patterns is currently used. A second alternative (Catalog 2) would be to keep the first two patterns and adapt instances of the second pattern for all instances where the third pattern is currently used. Table 6 shows how the metrics would change (Δ) for each of the alternative pattern catalogs compared to the current full pattern catalog. Coverage does not change.

Table 5. Usage statistics for the *Container disturb* patterns.

Pattern	Uses	Adaptations
Container disturb - spawn creature	1	1
Container disturb (specific item)	13	140
Container disturb - (specific item) toggle door	3	7

Table 6. Changes in metrics for alternative pattern catalogs from the full catalog for the composite story.

Catalog	Δ Usage	Δ Cov	ΔUtil	ΔPrec
1	-0.01	0.0	+0.70	-0.009
2	-0.01	0.0	+0.34	-0.004

Catalog 1 reduces the number of patterns in the story and the catalog by two (three removed and one added). Therefore, pattern catalog usage is reduced by a negligible amount (bad) from 49/60 to 47/58. However, the utility of the catalog is increased (good) since it does not change the number of pattern instances and it reduces the number of patterns by two. This translates to an overall increase of 0.70 in the utility of patterns in Catalog 1. In the original catalog, each pattern was used an average of 16.24 times in the story. In Catalog 1, each pattern is used an average of 16.94 times. This may translate to a decrease in the time a story designer may need to understand the patterns, since one general pattern is being used instead of three specific patterns.

Although increasing the utility by 0.70 is advantageous, the penalty is a decrease in precision (bad). With Catalog 1, there are 147 more adaptations, spread over the 17 pattern instances. This increase in the number of adaptations required makes Catalog 1 inferior to the original catalog. If a more specific pattern is used frequently enough, it should be included in the pattern catalog, along with the more general version. This strategy will avoid the loss in productivity that results when a designer is forced to make manual adaptations when they have already been abstracted into the more specific pattern.

Catalog 2 is a compromise between the original catalog and Catalog 1. It reduces the number of patterns by one, which yields an increase (good) in utility of 0.34. This is half the increase for Catalog 1, but with a precision decrease of only 0.004 instead of 0.009. With Catalog 2, there are only 63 more adaptations, spread over 3 pattern instances. This is a fair trade-off for this story, but we know that the removed pattern is useful for other stories (not discussed here), so we have not removed it from the catalog.

5.3 Evaluating a Reduced Pattern Catalog

To determine how well a more limited pattern catalog would perform, we evaluated a reduced pattern catalog across the same six stories. The reduced catalog consists of only those patterns

needed to replace scripts in One* (33 patterns). Therefore, the usage and coverage of this reduced catalog are both 1.00 for One*. Table 7 contains the values of the metrics for this catalog, where ΔPI is the number of instances of patterns used in the story that were not in the reduced catalog and ΔPat is the number of patterns used in the story that were not in the reduced catalog.

Table 7. Metrics for the reduced pattern catalog (33 patterns).

Story	ΔPI	ΔPat	Use	Cov	Util	Prec
One*	0	0	1.00	1.00	7.52	0.29
Two	-22	-7	0.58	0.73	8.53	0.23
Luskan	-22	-7	0.42	0.67	4.93	0.25
Three	-63	-10	0.64	0.68	6.33	0.25
Four	-44	-3	0.33	0.79	3.00	0.20
Composite	-151	-11	1.00	0.67	19.55	0.25

Overall, in switching from the full catalog to the reduced catalog, the coverage for the composite story decreased by 33% (bad), the usage increased by 18% (good), the utility increased by 20% (good) and the precision increased by 10% (good). This sounds like a good trade-off, but it is not! The coverage reduction means that the application designer must create 11 patterns to compensate for the reduced catalog. In exchange for this effort, the 18% reduced size of the catalog, makes it easier to find the other 33 patterns used in the story. While having fewer patterns in the catalog reduces the effort needed to find the desired patterns, this does not compensate for the extra effort required to create 11 new patterns out of a total of 44 that must be used.

Changes in utility and precision only apply to the patterns that are used from the catalog and do not take into account any new patterns that must be added. If these 11 new patterns are added and their utility and precision are included in the calculations, the utility and precision return to the values reported in Table 3. These two metrics are very useful in deciding whether to include a general or specialized version of a pattern in a catalog, as described earlier in this Section. They should not be used to determine whether a pattern should be included in a catalog or not – coverage and usage should be used to decide that. In general, a large gain in usage must be made to justify a small loss of coverage, due to the effort required to create a missing pattern. Nevertheless, there are some patterns that could be removed from the full catalog that would not decrease the coverage for the composite story at all, and therefore increase the usage. The patterns that could be removed are the 18% of patterns (11 patterns) in the full catalog that are not even used in the composite story. Removing them would have no effect on coverage, while increasing the usage from 0.82 to 1.00. The reason that they are in our full catalog is that we have other stories where they are useful.

The lesson is that a pattern catalog should be evaluated on many different applications (stories in our case). A pattern should be removed from the catalog only if: 1) it is not used often in any of these applications (stories), and 2) generalization or specialization will not allow it to be used more often. On the other hand, no pattern should be added to a catalog unless 1) it has been used several times in one or more stories that have been written using the catalog or 2) generalizing or specializing the pattern can allow it to be used several times.

To reduce application designer effort, high coverage is much more important than high usage, and high precision is much more important than high utility. This strategy can result in higher costs for pattern construction and maintenance. However, these costs can be amortized over many uses of the pattern catalog throughout its lifetime. Our "original" CRPG pattern catalog has been tuned over the past twelve months using the metrics presented in this paper. As new patterns are proposed, we re-evaluate coverage and usage to determine whether the pattern should be added to the catalog. If it should be added, we re-evaluate utility and precision to determine how general/specific the version of the added pattern should be.

5.4 An Extended Catalog and Story-Set

A class of 23 grade 10 high school English students wrote NWN stories using our ScriptEase encounter patterns. Each student spent 6 hours total, learning NWN, the Aurora Toolset and ScriptEase. Then each student spent 6 hours designing a story. Since these stories are short, we have combined all 23 stories together into a single story-set and applied the metrics to the combined story-set. The students used a pattern catalog that consisted of the 60 patterns described early in this Section, augmented by 4 additional patterns for a total of 64 patterns. One new pattern applies when the PC enters an *area* (region of the world that has its own map), one applies when the PC enters the *module* (story) and two apply when the PC acquires an *item*. The story-set consisted of 152 instances of 22 patterns that required 357 adaptations. The usage was 0.33. Most of the stories contained instances of patterns that were used in the tutorial material that they used to learn ScriptEase, so it is not surprising that only 33% of the patterns in the catalog were used. The coverage was 95%, since one student used ScriptEase to create one new pattern (*Creature combat ends*) out of the 22 patterns used. The utility was 7.19 since on average each student used about 7 instances of each pattern that they used. The precision was 0.43, indicating that on average each pattern instance required 2.32 adaptations (setting the options plus an average of 1.32 other adaptations). This precision is significantly larger than the precision (0.23) of the official NWN campaign story. This is a reflection that the student stories were considerably simpler than the professionally produced official campaign story. This study was originally designed to support our conjecture that complex tasks, such as creating a game story, currently the prerogative of professional game designers who can program, can be made accessible to non-programmers by using generative design patterns [6][21]. However, the stories constructed from this study provide another good data point for evaluating our pattern catalog, using our four metrics. These 23 student authors used 154 pattern instances to generate 16,051 lines of non-comment code during the 6 hours spent. That is an average of 698 lines of code per student. Professional programmers rarely write 698 lines of code in 6 hours so the effort required to apply this pattern catalog is much lower than manual implementation. One can argue about how many manual lines of code that a professional programmer can write in a day, but even if we allow for 100 lines per (6 hour) day, this is still 7 times as efficient (by non-programmers).

6. CONCLUSION

We defined four metrics for evaluating the effectiveness of pattern catalogs for designing applications. Although we illustrated these metrics in the context of a public pattern catalog for CRPGs that is

being used to design NWN stories, the metrics are general enough to be applied to any pattern catalog (generative or descriptive) in any application domain. We have learned that to minimize design effort, high coverage should be favored over high usage and high precision should be favored over high utility. This should be true whenever a pattern catalog will be used many times. If this is not the case, it is not clear whether it is worth building a pattern catalog in the first place. We have not touched upon the difficult issue of evaluating the metrics themselves, to select the best metrics from a series of proposals. This is a complex problem that will require user studies and is the subject of future work. At this point it is necessary to gain experience with a reasonable set of metrics (four of which have been proposed here). However, our application is well suited to studying alternate metrics since there is a rich set of patterns that can be used to evaluate them.

Pattern catalog metrics are most important for applications in which a large portion of the code is generated or manually written based on design patterns. However, as the use of design patterns becomes more widespread, this set of applications will grow. Since a design pattern captures considerable application-specific expertise and practice, they codify standard techniques, reduce errors and improve efficiency. These are hallmark characteristics of what is required of modern software. The emergence of domain-specific pattern catalogs (often called pattern languages) [23][3] demands the creation of metrics to evaluate their effectiveness. This is especially true as non-programmers are empowered to generate applications themselves.

Although this paper describes the four metrics in the context of pattern catalogs, the same four metrics could be applied to components and software libraries. The adaptation of patterns is analogous to the configuration of components in component-based software. For example, patterns that are too general/specific are analogous to components that are too general/specific.

7. ACKNOWLEDGMENTS

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

8. REFERENCES

[1] Bethesda Softworks, Morrowind. http://www.morrowind.com.

[2] BioWare Corp. http://www.bioware.com.

[3] Braga, R. T. V., Germano, F. S. R., Masiero, P. C. A Pattern Language for Business Resource Management. In *Proceedings of PLoP* 7 (Monticello, USA, 1999). 1-33.

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

[5] Florijn, G., Meijers, M., and van Winsen, P. Tool Support for Object-oriented Patterns. *ECOOP*, 1241 (1997). 472-495.

[6] Carbonaro, M., Cutumisu, M., McNaughton, M., Onuczko, C., Roy, T., Schaeffer, J., Szafron, D., Gillis, S., and Kratchmer, S. Interactive Story Writing in the Classroom: Using Computer Games. In *Proceedings of DiGRA* (Vancouver, Canada, June 2005). 323-338.

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

[8] ISO/IEC 9126-1. Institute of Electrical and Electronics Engineers, Part 1: Quality model, 2001. http://www.iso.ch.

[9] Khosravi, K. and Guéhéneuc, Y.-G. Open Issues with Software Quality Models. *ECOOP Workshop on Quantitative Approaches to Object-Oriented Software Engineering* (Glasgow, UK, July 2005).

[10] Lilac Soul's NWN Script Generator V2.0. http://nwvault.ign.com/View.php?view=Other.Detail&id=625.

[11] Lionhead Studios, Fable. http://www.fablegame.com.

[12] MacDonald, S., Szafron, D., Schaeffer, J., Anvik, J., Bromling, S., and Tan, K. Generative Design Patterns. In *Proceedings of Automated Software Engineering (ASE '02)* (Edinburgh, UK, September 2002). 23-34.

[13] McNaughton, M., Cutumisu, M., Szafron, D., Schaeffer, J., Redford, J., and Parker, D. ScriptEase: Generative Design Patterns for Computer Role-Playing Games. In *Proceedings of Automated Software Engineering (ASE '04)* (Linz, Austria, September 2004). 88-99.

[14] McNaughton, M., Redford, J., Schaeffer, J., and Szafron, D. Pattern-based AI Scripting using ScriptEase. In *Proceedings of AI 2003* (Halifax, Canada, June 2003). 35-49.

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

[16] Neverwinter Nights. http://nwn.bioware.com.

[17] Neverwinter Nights Scripting Forum. http://nwn.bioware .com/forums/viewforum.html?forum=47.

[18] Neverwinter Nights Vault. http://nwvault.ign.com.

[19] Onuczko, C., Cutumisu, M., Szafron, D., Schaeffer, J., McNaughton, M., Roy, T., Waugh, K., Carbonaro, M., and Siegel, J. A Pattern Catalog For Computer Role Playing Games. In *Proceedings of GameOn North America* (Montreal, Canada, August 2005). 30-38.

[20] ScriptEase for Neverwinter Nights. http://www.cs.ualberta.ca/~script/scripteasenwn.html.

[21] Szafron, D., Carbonaro, M., Cutumisu, M., Gillis, S., McNaughton, M., Onuczko, C., Roy T., and Schaeffer, J. Writing Interactive Stories in the Classroom, *IMEJ* 7, 1 (May 2005).

[22] Tahvildari, L. *Assessing the Impact of Using Design-Patterns-Based Systems*. Master's Thesis, University of Waterloo, Canada, 1999.

[23] Zhao, L., Foster, T. A Pattern Language of Transport Systems (Point and Route). *Pattern Languages of Program Design 3*, Addison-Wesley, 1998, 409-430.