Proceedings of the 17th IEEE International Conference
on Automated Software Engineering
September 20-24, 2004, Linz Austria

ScriptEase: Generative Design Patterns for Computer Role-Playing Games

M. McNaughton, M. Cutumisu
D. Szafron and J. Schaeffer
Dept. of Computing Science,
University of Alberta, Edmonton, AB
Canada T6G 2E8
{mcnaught, meric, duane,
jonathan}@cs.ualberta.ca

J. Redford
BioWare Corp.
200, 4445 Calgary Trail,
Edmonton, AB
Canada T6H 5R7
jamesr@bioware.com

D. Parker
Electronic Arts Inc.
4330 Sanderson Way
Burnaby, BC
Canada V5G 4X1
dparker@ea.com

Abstract

Recently, some researchers have argued that generative design patterns (GDPs) can leverage the obvious design re-use that characterizes traditional design patterns into code re-use. This paper provides additional evidence that GDPs are both useful and productive. Specifically, the current state-of-the-art in the domain of computer games is to script individual game objects to provide the desired interactions for each game adventure. We use BioWare Corp.'s popular Neverwinter Nights computer role-playing game to show how GDPs can be used to generate game scripts. This is a particularly good domain for GDPs, since game designers often have little or no programming skills. We demonstrate our approach using a new GDP tool called ScriptEase.

Keywords: generative design patterns, scripting languages, code generation, computer games.

1. Introduction

Traditional design patterns are *descriptive*. Each design pattern describes a set of known solutions to a recurring design problem, by providing a design lexicon, solution structures and the reasoning behind the solutions [12]. Since each pattern is a family of solutions, it must be *adapted* to a specific context during program construction. An adapted design pattern is a detailed specification that can be *translated* to code quickly by experienced programmers who have implemented the same design pattern in other contexts.

For novice programmers, adapting a design pattern into code is usually more difficult and error-prone. The ambiguous natural language pattern documentation can lead to slow progress and coding errors.

1.1 Generative Design Patterns

We claim that *generative design patterns* (*GDPs*) can increase the speed of code production and reduce coding errors, for novice and experienced programmers alike. They can also be used to promote rapid prototyping and provide a mechanism for code reuse at a high level of abstraction. There have been several efforts to produce GDPs. Some of these approaches are ad-hoc and others are more structured [4][6][7][8][15].

We have identified four problems with using GDPs.

1. *generality* - Since each design pattern represents a family of solutions, it is difficult to generate a single code base that can cover each potential solution.
2. *performance* - Since design patterns are formulated at a high level of abstraction, generated code may have poor execution performance.
3. *coverage* - Programmers must be able to amortize the overhead of learning a generative system over the amount of code they are able to generate with it. Therefore, there must be a rich set of GDPs.
4. *evolution* - New problem variants and completely new problems require new solutions, so there must be an easy way to edit GDPs and create new ones.

The *generality* problem of GDPs is caused by the fundamental nature of design patterns as a broad family of solutions to a set of related problems. The generality of non-generative design patterns is an advantage and is responsible for their success. However generality often makes it impossible to generate a single static software architecture that implements the entire family of solutions.

For example, when implementing the *composite design pattern*, there is a choice about where to declare the child management operations (safe or transparent) [9]. In the safe implementation, they are declared in the composite class, and in the transparent implementation, they are declared in the component class. There are trade-offs with either choice, but the important point is that it is impossible to generate a single static software architecture that will support both choices: safety and transparency.

Some of these simple static architecture problems can be solved by cross-cutting support such as aspect-oriented programming (AOP) [11]. However, the choice of safe versus transparent in the composite design pattern is an example where cross-cutting support does not help.

A simple solution to the generality problem is to narrow the scope of GDPs. However, this naïve approach leads to the coverage problem described later. Our solution to the generality problem is tied to design pattern adaptation, where we use the context of the problem to narrow the scope of a design pattern before code is generated. We proposed (and created) a simple three-phase adaptation process consisting of [12]:

1. initial adaptation,
2. code generation of a framework, and
3. final adaptation.

The *initial adaptation* phase adapts the design pattern to a single structural and control flow model, by allowing the user to select *generation options* before code generation. The *code generation* phase generates a software framework and the *final adaptation* phase performs standard specialization operations on the framework to adapt it to the final application. In the case of object-oriented frameworks, final adaptation involves setting method parameters, implementing hook methods and creating subclasses. In non-object-oriented frameworks, final adaptation involves setting function parameters, adding and removing code fragments and creating call-back functions. Our approach solves the problem of trying to adapt a single framework to support different structures and control flows, while maintaining the advantage of using framework specialization for adapting code to a particular application [10].

The *performance* problem of GDPs is due to code that is too generic. Design patterns introduce flexibility to support application evolution, "designing for change", by using indirection techniques [9]. Unfortunately, this approach requires more code to be written and maintained and the indirection can reduce performance. In effect, the indiscriminate use of patterns can result in a slower application [16]. Again, a naïve solution to this problem is to reduce the scope of the GDPs to remove these indirections. However, this can cause a coverage problem.

Partial adaptation before code generation is a good technique for managing the generality problem. Generation options produce more efficient static code by reducing dynamic choice points in the code for control flow, indirection and polymorphism.

The *coverage* problem occurs when there are few opportunities to use GDPs. It may take longer to learn about and use a set of generative patterns than it takes to write and debug the generated code by hand. There are two factors that contribute to the coverage problem.

First, the obvious way to mitigate the generality and performance problems of GDPs is to reduce the scope of each GDP. Our approach for mitigating the first coverage factor is to reject the naïve approach of reducing coverage to solve the generality and performance problems. By solving the generality and performance problems in other ways, we reduce the pressure to lower the coverage of GDPs.

Second, to have broad coverage, each application domain needs domain-specific GDPs. The time for creating a suitable library of patterns must be amortized over the time gained by using the library. Our approach to mitigating this second coverage factor is to create a programming environment for creating GDPs and to provide a mechanism for sharing GDPs between users by creating repositories of shared GDPs. This has the same positive effect as the open source software movement, where the development load is shared among users and the reliability of the software is increased.

The *evolution* problem for GDPs occurs because even if the coverage problem is solved, new problems will require new solutions, which will require either new GDPs or new versions of existing GDPs. Our programming environment for editing and creating new GDPs alleviates this problem as well.

In previous papers, we demonstrated how our three-phase approach could mitigate the four GDP problems in the domain of parallel processing [12][13], where final adaptation used object-oriented framework specialization in Java. However, it could be argued that the domain of parallel processing is very general and the question as to whether our approach works in more vertical specialized domains was unanswered. It was also unclear how well final adaptation would work when the target code was not object-oriented (no inheritance), so that object-oriented framework specialization could not be used.

1.2 GDPs in Computer Role-Playing Games

In this paper, we show how our three-phase approach solves the four GDP problems in a more specialized vertical domain – computer role-playing games (CRPGs). We also describe several different adaptation techniques that can be used when the generated code is not an object-oriented framework. Although this is a narrow domain, it is economically significant. The North American games industry is currently worth more than $10 billion per year. In the past, computer graphics have been the major sales feature of games. With more powerful computers, the perceived need for better graphics has been replaced by the demand for a more entertaining gaming experience and this is often provided through scripting.

CRPGs have many thousands of non-player characters (NPCs) and other game objects to script.

This is a daunting task to do manually, yet sadly, this is the state-of-the-art. The consequences are serious:

1. There are so many objects that it is difficult to organize and track them during game development.
2. Most objects have simplistic behaviors.
3. Testing is difficult, as games are very interactive.
4. Many designers are unable to do the scripting themselves.

Objects must be tracked using many independent ontologies, such as which area they are in, which sub-plots they are associated with, whether they are static or adaptive, and many others. Tracking the objects is hard, but tracking the scripts is even more difficult since most scripts involve the interaction of several objects.

Unless an object is on the critical path of the main plotline of the game, it usually has a single trivial scripted behavior. More behaviors (and more realistic behaviors) are desirable, but are not cost effective to write because of the large time investment needed.

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

Game designers create the game's story line, but often do not do the scripting themselves, since many are not programmers. Therefore, programmers implement the story line by writing scripts. The extra level of indirection in the process (the programmer) increases the chances of creating a product that does not match the designer's intentions. Such miscues are analogous to the ones that occur between customers/requirements analysts and designer/programmers during the development of more general software systems.

We have attempted to solve these four problems by putting GDPs in the hands of game designers. Most of the documented uses of design patterns involve relatively few patterns that occur relatively few times. CRPGs are demanding. There are hundreds of patterns used and they will appear in thousands of places. We performed a case study in which we used GDPs to generate the code for all scripts in BioWare's Neverwinter Nights (NWN) CRPG that are attached to a particular representative kind of object called a *placeable*. We accomplished this using a tool called ScriptEase [14]. In all, 497 calls to 182 different scripts comprising 1925 non-comment lines of hand-written code were replaced by ScriptEase-generated code using 431 instances of 24 different patterns. If we generalize this result to include all of the objects (not only the placeables), many thousands of lines of code could be replaced by a few thousand instances of a few hundred patterns.

ScriptEase solves the four scripting problems listed for the CRPG domain:

1. ScriptEase provides knowledge management support for organizing the thousands of scripts.
2. ScriptEase can create scripted objects with reduced programmer effort. Saved effort can result in lower costs or this effort can be concentrated on writing better stories, resulting in a more satisfying game experience.
3. Since ScriptEase generates the code, many common programming errors are eliminated. Patterns are used only after they have been fully tested, and since ScriptEase generates fully documented code, patterns are easy to understand during debugging. Reduced testing time for individual patterns translates to less cost or more testing time on game play scenarios, which leads to a more reliable product. The pattern instantiation process of ScriptEase also eliminates literal tags that are a source of error during copy-and-paste.
4. ScriptEase allows game designers to generate their own script code. This eliminates the middle-man (programmer) during story-telling and allows the programmers to focus on pattern writing, pattern testing and the interface with the game engine.

In Section 2, we introduce the domain of CRPGs by describing NWN, its game development tool, Aurora, and the non-object-oriented NWScript language. In Section 3, we describe the kinds of patterns found in CRPGs. In Sections 4 and 5, we introduce ScriptEase, our implementation of GDPs for NWN. We show how ScriptEase can be used to adapt GDPs and generate their script code, using several non-object-oriented adaptation mechanisms. We also show how ScriptEase can be used to create and edit GDPs. The scripts described in Sections 4 and 5 were originally hand-written by BioWare. In Section 6, we present the results of a case study where hand-written scripts from the original NWN game were replaced by scripts generated by ScriptEase. We also present some anecdotal data about ScriptEase users.

2. Neverwinter Nights, the Aurora Toolkit and NWScript

NWN is a multi-award winning (86 awards) CRPG from BioWare Corp. The game contains an engine that renders the graphical objects and characters, manages sound and motion, and dispatches game events to

scripts. This game engine is designed to play stories composed of individual modules constructed by game designers. A *module* contains areas (map sections), NPCs and other game objects that can be scripted to respond to game events using the NWScript language.

The game comes with a story that contains seven modules, and recently expansion packs for two more stories have been released. However, NWN is a community-based game. Thousands of people write stories and post them on the web for others to play. For example, Neverwinter Vault (http://nwvault.ign.com/Files/modules/modulesTop3.shtml) hosts more than 3,300 adventures. The most popular community adventure has been downloaded more than 222,000 times and the tenth most popular has been downloaded more than 75,000 times, as of June 2004.

Since most community-based designers are not programmers, they find scripting difficult. They often try to copy-and-paste scripts from existing adventures without understanding the code. There are also forums where they ask other designers who know how to program to write individual scripts for them.

An adventure can be created using BioWare's Aurora Toolset – a CAD tool for designing areas and placing customized versions of pre-designed objects. Aurora can also be used to attach scripts to objects, by selecting one and typing an NWScript for any of the events that the object can respond to. Figure 1 shows part of an area of the basic NWN *Chapter One* module in the Aurora Toolset and the properties dialog box for the selected chest object.

The game objects are divided into eleven categories. There are eight *scriptable object* categories: *modules, areas, creatures, doors, placeables, triggers, encounters* and *merchants*. There are three *non-scriptable object* categories: *items*, *sounds*, and *waypoints*. Non-scriptable objects can be referred to in the scripts attached to scriptable objects. Each scriptable object has a set of *event types* that apply to it and a separate script can be associated with each event type for each scriptable object instance. For example, a *placeable* (chest, pedestal, pile of rubble, etc.) supports the events: *OnClose*, *OnDamaged*, *OnDeath*, *OnHeartbeat*, *OnDisturbed*, *OnPhysicalAttacked*, *OnSpellCastAt*, *OnOpen*, *OnLock*, *OnUnlock*, *OnUsed* and *OnUserDefined*.

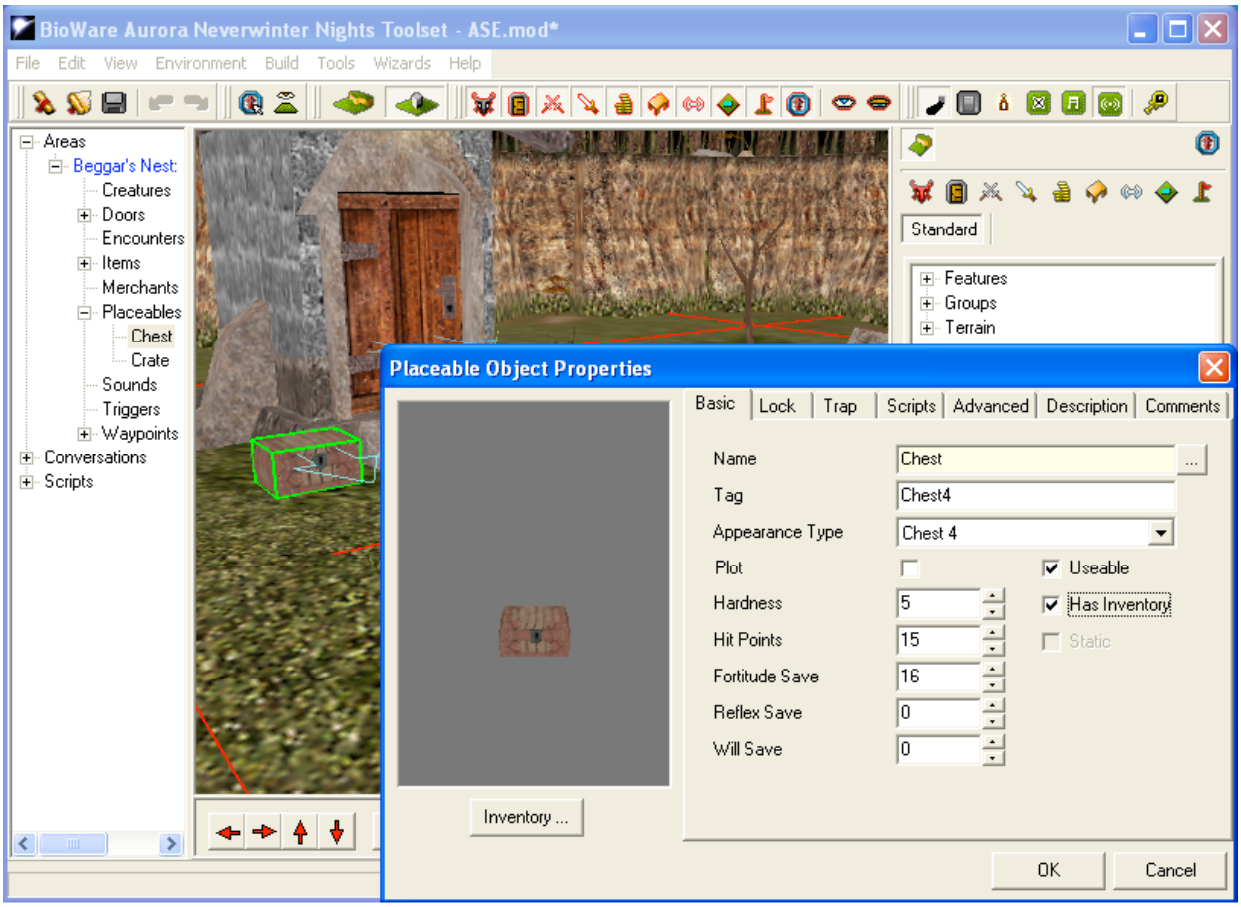

Figure 1. A screenshot of the BioWare Aurora toolset (2004 © Atari, Inc.).

```
void main()
{
 object oItem = GetInventoryDisturbItem();
 int nItemBase = GetBaseItemType(oItem);
 if(GetLocalInt(OBJECT_SELF,"NW_L_M1S1Opened") == FALSE
   && GetTag(oItem) == "M1S1Shield" )
 {
   DestroyObject(oItem);
   object oDoor =
      GetNearestObjectByTag("M1Q5F03_M1Q5J1");
   AssignCommand(oDoor,ActionOpenDoor(oDoor));
   SetLocked(oDoor,FALSE);
   SetLocalInt(OBJECT_SELF,"NW_L_M1S1Opened",TRUE);
   RewardXP("m1q1_Never",50,GetPCSpeaker());
 }
}
```

Figure 2. *OnDisturbed* script for a chest.

Figure 1 shows a chest near a door. The designer created a locked door that cannot be opened unless the player character (PC) performs a specific action on the chest. The PC must place a specific shield object in it. If this is done, the shield is destroyed, the door opens and the PC is given a reward called experience points (XP). Figure 2 shows the hand-written script (written in NWScript – a C-like language) that implements this scenario. The script was attached to the *OnDisturbed* event of the chest, which fires if an item is added to or removed from the chest.

Here is a natural language description of the script in Figure 2. If this chest has not yet served its purpose, and if the disturbed item is a specific shield (`"M1S1Shield"`), then destroy the shield, open the closest door (tagged `"M1Q5F03_M1Q5J1"`), unlock the door, tell the chest to remember that it has served its purpose and award 50 experience points to the PC that disturbed the chest. Note that the `nItemBase` integer variable is bound, but not used.

There are two other locations in the same module where a specific item must be placed into a chest to open a nearby door. The script programmer must have recognized the similarity of all three scenarios and used copy-and-paste to create the other scripts, since the same unused value `nItemBase` appears in all three scripts. All the scripts share the same high level goal and code pattern, although the tag for the special item and the tag for the door to be opened are different.

3. Design Patterns For CRPGs

The scenario described in the previous section occurs frequently enough in NWN modules and in CRPGs in general that we have defined the concept of placing a specific item in a container to open/unlock (or close/lock) a door, as a GDP called *disturb container – (specific item) toggle door*. In fact, this pattern is a specialization of a more general GDP, which we call *disturb container – (specific item)*. In this pattern, some actions are taken whenever a specific item is added or removed from a container. The container does not have to be a chest. For example, in NWN any *placeable* or *creature* can be granted *container* status (so that objects can be added or removed from it) by checking the *Has Inventory* box in the properties dialog of the object, as shown in Figure 1.

We have divided our GDPs into four pattern groups: *encounter*, *behavior*, *dialog* and *plot*. An *encounter pattern* applies to a scenario that is started by an event involving a *module, area*, *door*, *placeable*, *trigger* (a polygon drawn on the ground), *merchant* or *encounter*. A *behavior pattern* is used to generate scripts for *creatures*, a *dialog pattern* is used to generate conversation scripts and a *plot pattern* is used to control the plot of the story being told. In this paper we discuss only encounter patterns and for brevity, we focus on scripts that are attached to *placeable* objects.

The Aurora toolset provides exactly one abstract kind of object called an *Encounter*. An *Aurora Encounter* consists of a *trigger* and one of several pre-defined groups of creatures. To create an Aurora Encounter, the designer paints the *trigger*, selects one of the pre-defined groups of creatures and indicates a number of creatures. During the game, when a PC steps on the *trigger*, the specified number of creatures from the selected group are created near the *trigger*. A ScriptEase *encounter pattern* is a generalization of an Aurora Encounter object. In a ScriptEase encounter pattern, the event is generalized from stepping on a trigger to any *module*, *area*, *door*, *placeable* or *trigger* event, and the action is generalized from creating a number of creatures of a particular kind to any supported actions. In this paper, the term *encounter* always refers to a ScriptEase encounter GDP, unless the term *Aurora encounter object* is used.

4. ScriptEase Generative Design Patterns

ScriptEase is a tool to help CRPG designers script their adventures. The goal of ScriptEase is to solve the four CRPG problems described in Section 1.2, by allowing game designers to generate scripted adventures without writing any code. ScriptEase effectively contains a pattern language [5] for CRPGs.

We present an abbreviated walkthrough of ScriptEase to highlight the complete scripting process. We demonstrate how the designer would generate a script equivalent to the code in Figure 2 for the chest from Figure 1. The designer starts by creating the physical layout of a module (called ASE.mod) using Aurora – without attaching any scripts to any objects. The module file is then opened in ScriptEase and the symbol table is read. There are three steps to script an encounter:

1. Select an encounter pattern and create an instance.
2. Adapt the design pattern for the module.
3. Generate the scripting code.

Figure 3 shows the ScriptEase Encounter Builder being used to create a new instance of the *disturb container – (specific item) toggle door* encounter pattern. The designer adapts the pattern instance for the context using the information in the symbol table. In this case, the designer names the encounter instance (*Shield Chest*) and binds the parameters: *The Container*, *The Item*, *The Door* and *The XP* to appropriate objects in the module.

Figure 4 shows the *Pick a blueprint* window being used to bind the parameter *The Item* to the *Ceremonial Shield* object created earlier using Aurora. All objects in the module are categorized by type. The icons near the top of the window (*Creatures*, *Doors*, *Encounters*, *Items*, *Placeables*, *Sounds*, *Merchants*, *Triggers* and *Waypoints*) are used to select objects of a particular type. Icons for inappropriate types are grayed so they cannot be selected. In this case, since the parameter *The Item* has type *Item*, only the *Item* icon can be selected.

Each encounter contains one or more situations. Each *situation* corresponds to a legal event for the first encounter parameter. The *disturb container – (specific item) toggle door* encounter contains two situations. *Add item* applies when an item is added to the container and *Remove item* applies when an item is removed. In this context, the *Remove item* situation can be deleted since it is not needed. To specify that the *Add item* situation should only apply once (the first time the specific item is placed in the container), the designer selects it and clicks the *Plot* tab in the lower pane of the window as shown in Figure 5. By clicking on the middle radio button, the designer ensures that the actions (door will be toggled) occur only when the *Ceremonial Shield* is put into the chest for the first time. As illustrated by the case study described in Section 6, one-time situations are common in CRPGs, so this form of adaptation is useful. The designer can save the module, generate the scripts and compile them by selecting a single menu command.

Figure 6 shows the NWScript code automatically generated for this adapted encounter pattern instance. The generated code is equivalent to the hand-written code in Figure 2. It appears longer for four reasons. First, ScriptEase code is fully commented. Second, ScriptEase puts variable declarations and assignments on separate lines. Third, ScriptEase uses a separate variable for the receiver object (`Chest_SE0 = OBJECT_SELF`). Fourth, ScriptEase always computes the creature that disturbed the container since this creature is usually referenced in the actions (in this case it is not). This code can be removed by hand to improve performance. We plan to automatically remove such unnecessary computations. The important point is that the code was generated by the game designer spending less than a minute selecting items from menus and using dialogs to pick game objects, instead of hand-coding.

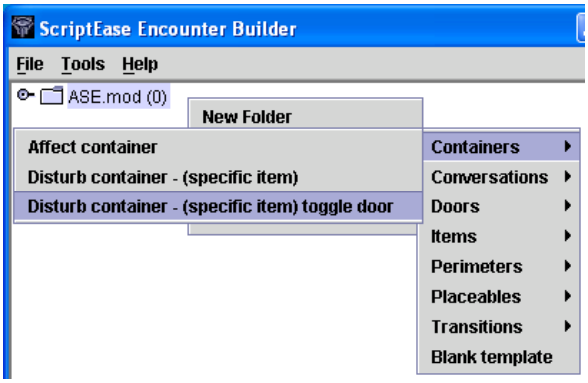

Figure 3. Creating an encounter.

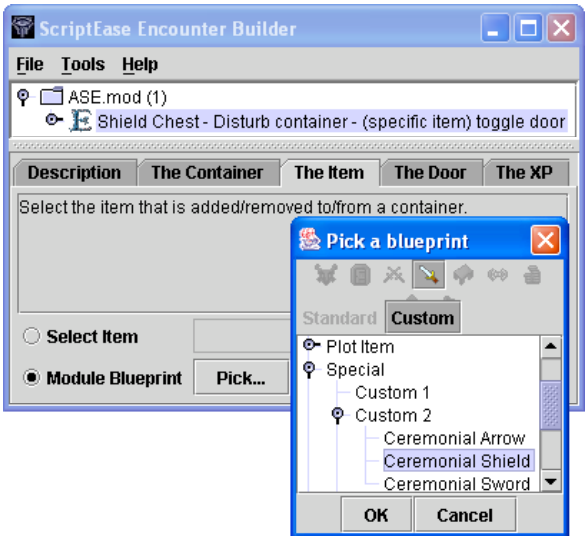

Figure 4. Adapting an encounter.

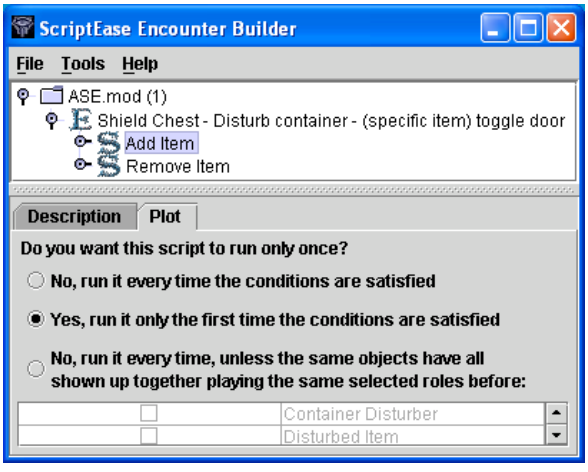

Figure 5. Setting a one-time situation.

```
void AddItem_0() {
/* This situation should only execute once, ever. */
  if( GetLocalInt(GetModule(),"SE_ONCE_19") != 0 )
    return;

  // All of the variables used in this situation
  object Door_SE5;
  object DisturbedItem_SE2;
  object CeremonialShield_SE3;
  int SameTags_SE4;
  object Chest_SE0;
  object ContainerDisturber_SE1;
  // Attached to following object's OnDisturbed slot
  Chest_SE0 = OBJECT_SELF;
  // When an item is added to Chest
  if( ! SE_Ev_ContainerOnDisturbed(Chest_SE0,
    INVENTORY_DISTURB_TYPE_ADDED) ) return;
  // Define Container Disturber as the creature that
  // just added to Chest
  ContainerDisturber_SE1 =
    SE_Df_ContainerDisturber(Chest_SE0,
    INVENTORY_DISTURB_TYPE_ADDED);
  // Define Disturbed Item as the item that was added
  DisturbedItem_SE2 = SE_Df_DisturbedItem(Chest_SE0,
    INVENTORY_DISTURB_TYPE_ADDED);
  // Get the object with tag "M1S1Shield"
  CeremonialShield_SE3 = GetObjectByTag("M1S1Shield");
  // Define Same Tags as whether Ceremonial Shield has
  // the same tag as Disturbed Item
  SameTags_SE4 = SE_Df_SameTag(CeremonialShield_SE3,
    DisturbedItem_SE2);

  // Main code - check conditions & execute actions
  // If Same Tags is Positive (True, Yes, On, etc.)
  if( SE_Co_IsPositive(SameTags_SE4) ) {
    // Make sure this situation doesn't execute again.
    SetLocalInt(GetModule(), "SE_ONCE_19", 1);
    // Get the object with tag "ShieldDoor"
    Door_SE5 = GetObjectByTag("ShieldDoor");
    // Door* opens Door
    SE_Ac_OpenCloseDoorTodo(Door_SE5, Door_SE5, TRUE);
    // Unlock Door
    SE_Ac_LockUnlockDoor(Door_SE5, FALSE);
    // Destroy Ceremonial Shield
    SE_Ac_DestroyObject(CeremonialShield_SE3);
    // Assign 50 XP to Container Disturber
    SE_Ac_AssignXP(ContainerDisturber_SE1, 50);
  }
}
```

Figure 6. Script for an adapted pattern.

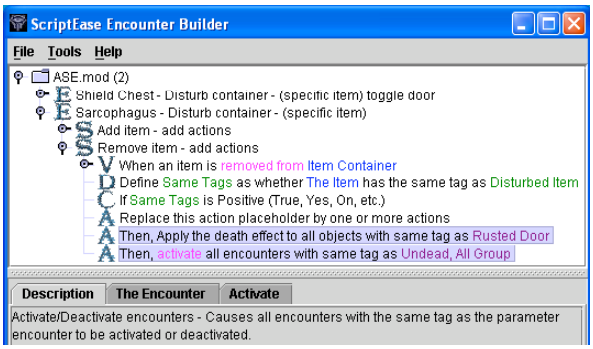

Figure 7. The Encounter Builder.

5. Solving the Generative Problems

ScriptEase addresses the *generality* and *coverage* problems of GDPs by including general patterns, together with a rich set of adaptation mechanisms. Simple parameterization and selection between one-time and many-time situations does not provide enough expressive power for adaptation. ScriptEase supports eight other forms of adaptation: addition/removal of actions, definitions, conditions, and situations.

For example, the *disturb container – (specific item)* pattern applies any time a specific item is added to or removed from a container. The game designer is free to add any series of actions to the pattern instance during adaptation. Specifically, *Chapter One* contains a script that fires when a particular item is removed from a sarcophagus. The script destroys three nearby doors and activates three Aurora encounter objects. Each activated object spawns creatures at the location of one of the destroyed doors. This script can be generated from a *disturb container – (specific item)* pattern by adding two actions, one to destroy the doors and one to activate the encounter objects. ScriptEase uses *actions* to manipulate game objects. ScriptEase currently supports 174 actions. Of these, 164 are *atomic actions*, which are implemented as direct calls to NWScript code. The other 10 are called *action encounters* and each is just a sequence of actions (atomic actions or other action encounters).

Figure 7 shows the adapted pattern in the ScriptEase Encounter Builder. It contains two *situations* (icon *S*), labeled *Add item – add actions* and *Remove item – add actions*. The second situation has also been opened to reveal its components. A *situation* always contains a single *event* (icon *V*) that specifies when the situation is applicable. Each situation also contains 0 or more *definitions* (icon *D*), 0 or more *conditions* (icon *C*) and 0 or more *actions* (icon *A*). When an instance of a *disturb container – (specific item)* encounter is created, its *Remove item* situation contains a single *placeholder* action (labeled *Replace this action placeholder by one or more actions*). Figure 7 shows the two actions in gray that the designer has added to adapt this pattern, one to destroy the doors and one to activate the Aurora encounter objects. The designer can delete the *placeholder action* and can also delete the *Add item* situation since they are not needed in this context.

The *performance* problem for GDPs is an important one in computer games. In fact, in NWN scripts that do not terminate in a fixed number of virtual machine instructions are forcibly aborted, since scripts are only allocated a small portion of the CPU resources. ScriptEase addresses the performance problem in two ways.

First, when a specialization of an action is popular, ScriptEase supports the specialization as a separate more efficient action. For example, the code generated by ScriptEase for the action in Figure 7, that destroys all of the *Rusted Door* objects, is a call to the atomic action *SE_AcKillObjects* shown in Figure 8.

```
void SE_AcKillObjects(Object param_1) {
  int nth = 0;
  string tag = GetTag(param_1);
  effect eDeath = EffectDeath(FALSE, FALSE);
  object anObject = GetObjectByTag(tag, nth);
  while (GetIsObjectValid(anObject) && !
    GetIsPC(anObject))
  {
    SetPlotFlag(anObject, FALSE);
    ApplyEffectToObject(DURATION_TYPE_INSTANT, eDeath,
      anObject);
    nth++;
    anObject = GetObjectByTag(tag, nth);
  }
}

void SE_Ac_DestroyObjectWithAnimation(object parm1) {
  effect death = EffectDeath(TRUE);
  ApplyEffectToObject(DURATION_TYPE_INSTANT, death,
    parm1);
}
```

Figure 8. Action specialization for improved execution performance.

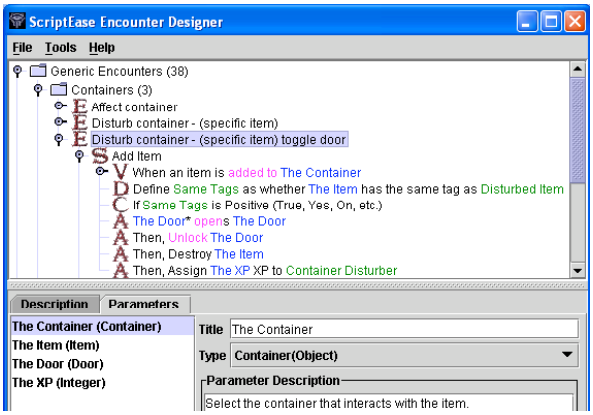

Figure 9. The Encounter Designer.

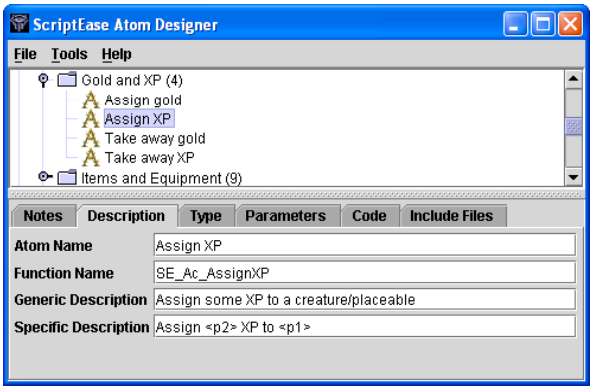

Figure 10. The Atom Designer.

A game designer could use this same atomic action to destroy a single object. However, to improve execution efficiency, a second atomic action that generates code to destroy only a single object is also provided. Figure 8 also shows the code for this simpler atomic action, *SE_Ac_DestroyObjectWithAnimation*.

The second way that ScriptEase addresses the performance problem is to encourage designers to delete unused components instead of using run-time checking. For example, the designer can delete an unused situation from an instance of *disturb container – (specific item)*, as described previously (Figure 7).

Finally, ScriptEase solves the *evolution* problem by including two tools called the *Encounter Designer* (Figure 9) and the *Atom Designer* (Figure 10). A game designer without programming experience can use the Encounter Designer to edit an existing encounter pattern and save the result as a new encounter pattern or to create a new encounter pattern. A programmer can use the Atom Designer to create new basic atomic actions for ScriptEase, by writing them in NWScript. These new encounter patterns and atomic actions can also be exported and shared with other ScriptEase users.

Here is an example of using the Encounter Designer shown in Figure 9. When converting the manually written scripts from *Chapter One*, it was discovered that there was a need for three instances of the *disturb container – (specific item) toggle door* pattern described in Sections 3 and 4. Since this pattern did not exist in ScriptEase before *Chapter One* was converted, there were two choices. One choice was to create three instances of the general *disturb container – (specific item)* pattern and adapt each instance by adding the same set of three actions to unlock and open the nearest door, to destroy the item placed into the container and to give XP to the PC. However, since three separate instances are required and the idea is general enough to be used in other modules, we decided to make a new *disturb container – (specific item) toggle door* pattern.

Figure 9 shows the newly constructed pattern in the Encounter Designer. The procedure used to create this new encounter pattern was to first make a copy of the original *disturb container – (specific item)* pattern and then specialize it. An instance of the original pattern is shown in Figure 7. However, the original pattern did not contain the two grayed actions that are shown in Figure 7, since they were added during adaptation of that particular pattern instance. In Figure 9, the parameter tab of the lower pane reveals the encounter parameters. Since the pattern was copied from the *disturb container – (specific item)* pattern, two parameters were already defined, *The Container* and *The Item*. Two more parameters called *The Door* and *The XP* were added. To finish, the four actions shown in Figure 9 were added to both the *Add item* and *Remove item* situations of the new pattern

The Encounter Designer can also be used to create action encounters. The ScriptEase Atom Designer can be used to create action atoms, along with the event

atoms, definition atoms and condition atoms that have been shown in the figures in this paper. Figure 10 shows the *Assign XP* action atom from Figure 9 as it appears in the Atom Designer. Since the *Description* tab is selected, we can see the *Atom Name* and three other pieces of information for this atom. The *Function Name* is the name of the NWScript function that will be generated by ScriptEase for this atom. The *Generic Description* is displayed as a menu item in the ScriptEase Encounter Builder, so that the designer can select this atom as an action. The *Description Template* is used to display this action atom in the top pane of the Encounter Builder where the bracketed parameters (<>) are replaced by the actual parameter names as shown in Figure 9.

6. Using ScriptEase

In this section we present a summary of a case study, describe some experiences of ScriptEase users, and give some statistics about the modules they have constructed.

6.1 A ScriptEase Case Study

NWN was released with a single adventure that consisted of seven modules: *The Prelude*, *Chapter One*, *Chapter One Finale*, *Chapter Two*, *Luskan and Host Tower, Chapter Three* and *Chapter Four*. Each module is a self-contained file containing objects and scripts. To test the efficacy of ScriptEase, we replaced all of the *placeable* scripts by scripts generated from ScriptEase patterns. Since *The Prelude* and *Chapter One Finale* contain few scripts, we combined both of them with *Chapter One* into *Chapter One**. Summary statistics are shown in Table 1.

Table 1. ScriptEase pattern statistics.

Chapter	script calls	script templates	lines of code	pattern instances	pattern templates
One*	153	47	391	108	15
Two	112	40	279	104	14
Luskan	54	28	454	51	10
Three	127	50	669	118	15
Four	51	17	132	50	7
Total	497	182	1925	431	**24**

The second column of Table 1 shows the number of calls made to scripts by placeable objects. The third column shows the number of unique scripts that were referred to by placeable script calls.

The fourth column is the total number of non-comment lines of hand-written code that were contained in the placeable scripts for each module. All 1925 lines were replaced by generated code. In addition, the *Prelude* (part of *Chapter One**) and *Chapter Four* each contained an identical copy of a 1450 line script that created a special item whose characteristics depended on the characteristics of the PC. The original calls to these separate (but identical) scripts were replaced by calls to a single ScriptEase atom. This saving of 1450 lines was not included in the statistics. There were also two scripts in the *Finale* (part of *Chapter One**) that were simply converted to atoms and are not included (61 total lines) in the statistics either.

The fifth column shows the number of pattern instances that were used to replace all of these script calls. Every script call attached to a *placeable* object was successfully replaced by a pattern instance. The sixth column shows the number of patterns that were used in each module. However, the total for this column is not the sum of the column entries since many of the patterns were re-used between modules. The total (24) is the number of unique patterns used across all modules. The total for the script templates column is the sum of the column entries, since none of the hand-written scripts were re-used across modules. Note that multiple script templates were often replaced by a single pattern with multiple situations. For example, in *Chapter One* there is a *placeable* that has two scripts, one for *onOpen* events and one for *onDeath* events. These scripts are replaced by a single pattern that has two situations, one for *onOpen* and one for *onDeath*.

ScriptEase is very successful in removing error-prone *tag literals* and *state literals* from the code. A *tag literal* refers to an object created using Aurora. For example, in Figure 2, the tag `"M1Q5F03_M1Q5J1"` is used to obtain a particular door object. With ScriptEase, the designer does not type a tag literal. Instead the designer uses a pick dialog (Figure 4) to select the appropriate object. This approach significantly reduces program errors.

In NWScript, state can be stored in game objects using a pair of functions to set and get the state. For example, in Figure 2, the function call: `SetLocalInt(OBJECT_SELF,"NW_L_M1S1Opened", TRUE)` asks a chest object (`OBJECT_SELF`) to remember that it has been used to open a door. The function call: `GetLocalInt(OBJECT_SELF,"NW_L_M1S1Opened")` checks the remembered state. Many programming errors are caused by the *state literals* used in this state mechanism. ScriptEase can eliminate some of these errors by generating state literals from one-time patterns or from *plot tokens* (a high-level ScriptEase mechanism for remembering whether something important to the plot has happened yet). Table 2 shows the total number of tag literals and state literals that appeared in all modules that were attached to *placeables*, the number that were eliminated and the number that remained.

Table 2. ScriptEase literal elimination.

	original	eliminated	remaining
Tag literal defs	284	284	0
Tag literal uses	421	421	0
State literal defs	100	52	48
State literal uses	274	221	53

We differentiate between distinct definitions of a literal and the number of times it is used. Each use provides another chance for an error in the code. All tag literals were eliminated using parameter picking dialogs. A total of 26 state literals were eliminated using one-time patterns, 9 state literals were eliminated using plot tokens and 17 state literals were eliminated using other techniques, including removing useless code such as the computed integer, `nItemBase`, shown in Figure 2.

As the number of ScriptEase patterns grows, fewer and fewer new ones need to be created to construct new modules. Figure 11 shows the number of patterns required to generate the scripts for placeables in successive modules. The order of module creation was from left to right as shown in the figure. The first bar shows that *Chapter One** re-used 6 existing patterns and required 9 new ones. The second bar shows that *Chapter Two* re-used 9 existing patterns, re-used 4 patterns created for *Chapter One** and required 1 new pattern. In general, the percentage of new patterns required for each successive chapter tended to decrease as the case-study progressed (60%, 7%, 10%, 20%, 0%) with the exception of *Chapter Three,* which was the largest module.

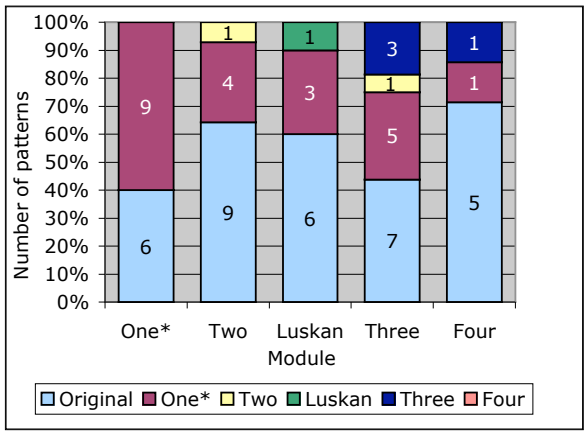

Figure 11 Pattern re-use by module.

Of the three new patterns created for *Chapter Three*, only one was re-used in *Chapter Four*. In fact, we could have refrained from creating the other two new patterns in *Chapter Three*. We could have adapted existing patterns instead. For example, we created the *Dead placeable – create placeable* pattern, which applies when a placeable is destroyed. This is a common idiom in CRPGs, so a new pattern was created. An alternative would have been to use the similar pattern called *Dead placeable – destroy objects* that was created for *Chapter One**. This pattern also applies whenever a placeable is destroyed. If this existing pattern was used instead of creating a new pattern, the instance of the existing pattern would have to be adapted by replacing the action.

What are good criteria for deciding whether to create a new pattern or adapt an existing (more general or more specific) one? A new pattern should be created when it can be re-used enough times to amortize its cost of creation (about an hour). In our experience, a pattern needs to be used about six times to amortize this cost (about 10 minutes to adapt an existing pattern). There are two kinds of re-use. Re-use due to multiple pattern instances in the module being constructed (*internal re-use*) and re-use in future modules (*external re-use*). The first is easy to determine. The second requires a good understanding of the domain. If a pattern is not general enough, it is likely that all of its re-use will be limited to multiple instances of internal re-use. In the case study, of the 14 new patterns we created, 6 were re-used externally in subsequent modules and 8 were only re-used internally in the same module. As we transformed modules, we could only guess which patterns we constructed would be re-used externally. External re-use of 43% is fairly good and in fact, the patterns that were not externally re-used may be re-used in other adventures later. Of course the three patterns created in *Chapter Three* had only *Chapter Four* for potential external re-use in the case study, so they had the highest probability of no external re-use during the case study.

In addition, of the 12 *placeable* /*container* patterns that were designed before the case study (for previously constructed modules), 8 were re-used during the case study (external re-use). In addition, one other existing non-placeable pattern (*basic conversation*) was re-used. It was more appropriate to generate several scripts that were attached to placeables in the hand-scripted code from this conversation pattern instead of from the placeable object.

6.2 ScriptEase Usage

ScriptEase was released to the NWN community on November 19, 2003 (http://www.cs.ualberta.ca/~script/scripteasenwn.html). At the time, BioWare called ScriptEase "the answer to our nonprogramming dreams" [2]. Since its release, ScriptEase has been downloaded more than 6000 times as of June 2004. Reaction from the community has been positive. The BioWare forum topics on ScriptEase contained 110 posts during this period. Several designers have sent us modules scripted entirely using ScriptEase.

Before ScriptEase was released, we performed a small user study to assess how practical it would be for non-programmers. We implemented part of an area known as *The Temple Ruins* from BioWare's previous CRPG called Baldur's Gate 2: Shadows of Amn [1]. We chose this area partially because it is in a commercial CRPG other than NWN and partially because it contains several interesting encounters. By specifying all of them in ScriptEase, we demonstrated its generality. We hired a high-school student to use ScriptEase for two 1-week periods. The student did not build any new atoms and did not design any new patterns. The student was familiar with NWN and the Aurora Toolset before the user study, but was not familiar with NWScript. After a week of beta-testing ScriptEase and reporting bugs (that we fixed), we requested that the student create *The Temple Ruins* module using ScriptEase. It took a little over one day, mostly taken up by one complex conversation involving several riddles with a Sphinx-like character. Not including this conversation, the student took approximately three hours to script the module and to debug it by play-testing. The implementation generated 51 scripts from 21 instances of 12 patterns (3 placeable patterns, 5 trigger patterns, 2 door patterns, 1 creature pattern and 1 conversation pattern). One of the ScriptEase authors had implemented the same area by hand using NWScript. It required over 700 lines of code, and despite being expert in the use of NWScript, three days were spent writing and debugging it.

In addition, just before our public release, a professional writer (non-programmer) was hired to write a tutorial for the release. She created and scripted an example module for the tutorial and reported that ScriptEase was easy to use for non-programmers.

7. ScriptEase Architecture and Generality

We have written a Java implementation of ScriptEase to support a specific CRPG, namely NWN. Picking a specific CRPG was necessary so that ScriptEase could be used by real developers and so that real experiments could be conducted to evaluate it and improve it. However, the idea of using generative design patterns for computer games transcends NWN and even CRPGs. In fact, the architecture of ScriptEase has been constructed so that it can be modified to support other CRPGs with minimal game-specific code. Figure 12 is an architectural diagram of ScriptEase that shows the NWN-dependent components on the left side and the NWN-independent components on the right side. Of approximately 41,000 lines of Java code, only about 15,000 are NWN-dependent.

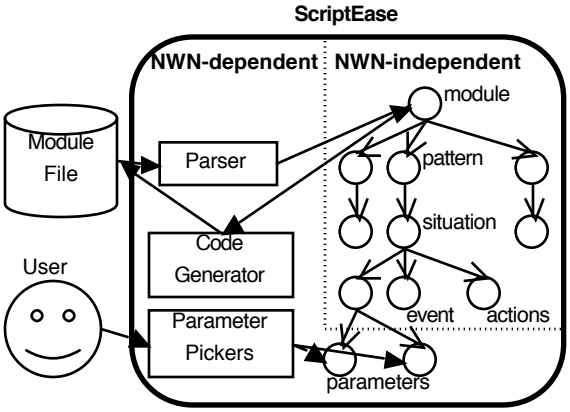

Figure 12 The ScriptEase architecture.

Table 3. Java code in the ScriptEase implementation.

	NWN-dependent lines / files	NWN-independent lines / files
Parsing	3,500 / 31	0 / 0
Parameter pickers	3,000 / 27	0 / 0
Code generation	2,000 / 10*	2,500 / 10* shared
NWN Types	6,500 / 51	0 / 0
Other	0 / 0	23,500 / 307
Total	15,000 / 119	26,000 / 307

Table 3 shows the breakdown of this code into NWN-dependent and NWN-independent components. Each game module is represented internally by a heterogeneous tree whose nodes represent events, situations, actions, atoms, types, parameter values and other support objects. Code for situation, action, atom and support nodes is game independent and distributed throughout the node classes. Event nodes need to exist for all CRPG games, although the exact event types are game-specific. We have abstracted the NWN-dependent information out of the event nodes so that a ScriptEase user can create events in ScriptEase itself. Therefore, the actual event types to support NWN were built using the ScriptEase Atom Designer and are not "hard-coded" into the implementation of ScriptEase. By doing this, the methods for the classes that implement event nodes are also NWN-independent.

The only NWN-dependent nodes are for types and parameters. Example types include placeable, creature, string and number. For example, each type knows the NWScript symbol that represents it, like `object`, `string` or `int`. Some parameters are game independent, such as numbers and strings. Others are NWN-dependent like journal entries, blueprints and conversation nodes. Each parameter has code for its parameter picker, such as the blueprint picker shown in Figure 4. Naturally, the parameter picker code for the

NWN-dependent parameters is NWN-dependent. In all there are about 13,000 lines of NWN-dependent code in 109 files for parsing, parameter picking and NWN-dependent type support.

To generate code, ScriptEase calls a `generateCode()` method on each tree node object recursively (using a visitor pattern). The control flow for code generation is determined by the patterns, situations, events and atoms that determine the structure of the tree, so this flow is game independent. However, the `generateCode()` methods ultimately emit NWScript code for operations like function invocations, opening new lexical scopes, and making forward declarations of support functions. The ScriptEase code that emits the NWScript code is NWN-dependent. To support another game, this code would either have to be replaced or abstracted into a family of code generation factories. There are about 5,500 lines of code in 10 files for code generation including about 2,000 NWN-dependent lines.

To support a CRPG game other than NWN, the following would need to be done.

1. Replace the NWN-dependent parts of the Java classes that represent type and parameter value nodes.
2. Write parameter picker code for the parameter nodes.
3. Write a file format parser for parsing the module file into the internal ScriptEase representation (essentially a symbol table).
4. Implement the `generateCode()` method for the new target scripting language.
5. Write new event, definition, condition and action atoms (in ScriptEase – not Java).

8. Conclusion

In this paper, we have shown how GDPs can be used to generate scripting code for CRPGs. Specifically, we have shown how the four inherent problems of GDPs can be handled in the CRPG domain by partially adapting GDPs before code generation. We have also shown how GDPs can be used to cope with the four major scripting problems faced by game developers. Our case study did not use an abstract or toy problem. It demonstrated the strengths of our approach on a real commercial computer game using real adventure modules.

9. Acknowledgement

This research was supported by research grants from the (Canadian) Institute for Robotics and Intelligent Systems (IRIS), the Natural Sciences and Engineering Research Council of Canada (NSERC), and Alberta's Informatics Circle of Research Excellence (iCORE). We thank the referees for suggestions that improved the manuscript. We especially thank our many friends at BioWare for their support and encouragement, with special thanks to Mark Brockington.

10. References

[1] Baldur's Gate 2: Shadows of Amn. Bioware Corp. / Black Isle Studios / Interplay, 2000. (http://www.bioware.com/games/shadows_amn).

[2] BioWare Corp. Nov. 19, 2003, http://nwn.bioware.com/archive/nwwed.html.

[3] J. Bosch. Design patterns as language constructs. JOOP, 11(2), pp. 18-32, 1998.

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

[5] J. Coplien and D. Schmidt, eds., Pattern Languages of Program Design. Addison Wesley, 1995.

[6] TogetherSoft Corporation. TogetherSoft Control Center tutorials: Using design patterns. www.togethersoft.com/services/tutorials/index.jsp.

[7] A. Eden, Y. Hirshfeld, and A. Yehudai. Towards a mathematical foundation for design patterns. Technical Report 1999-04, Dept. of Information Technology, University of Uppsala, 1999.

[8] G. Florijn, M. Meijers, and P. van Winsen. Tool support for object-oriented patterns. In ECOOP, Vol. 1241 of LNCS, pp. 472-495. Springer, 1997.

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

[10] R. Johnson and B. Foote. Designing reusable classes. JOOP, 1(2), pp. 22-35, 1988.

[11] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. Griswold. An overview of AspectJ. ECOOP, LNCS #2072, pp. 327-353. Springer, 2001.

[12] S. MacDonald, D. Szafron, J. Schaeffer, J. Anvik, S. Bromling, and K. Tan, Generative Design Patterns, Automated Software Engineering, September 2002, Edinburgh, UK, pp. 23-34.

[13] S. MacDonald. From Patterns to Frameworks to Parallel Programs. Ph.D. thesis, Dept. of Computing Science, University of Alberta, 2002.

[14] M. McNaughton, J. Redford, J. Schaeffer, and D. Szafron, Pattern-based AI Scripting using ScriptEase, AI 2003, Halifax, June 2003, pp. 35-49.

[15] ModelMaker Tools. Design patterns in ModelMaker. http://www.modelmakertools.com/mm_design_patterns.htm.

[16] D. Schmidt, M. Stal, H. Rohnert, and F. Buschmann. Pattern-Oriented Software Architecture: Patterns for Concurrent and Networked Objects, Vol 2. Wiley, 2000.