University of Alberta

USING DESCRIBERS TO SIMPLIFY SCRIPTEASE

by

Neesha Desai

A thesis submitted to the Faculty of Graduate Studies and Research
in partial fulfillment of the requirements for the degree of

Master of Science

Department of Computing Science

Abstract

A high level of programming knowledge is needed in order to script a video game. This prevents video game design from being accessible to non-programmers. ScriptEase is a tool that was designed to solve this problem. While ScriptEase has been shown to be accessible to 10th grade English students there remains areas for further simplification. This thesis focuses on changing the way authors set options within ScriptEase by introducing a new technique called *Describers*. *Describers* allow authors to adapt plain English sentences to provide a description of each option. A user study was conducted that compared *Describers* against the original technique of using definitions. The participants were able to complete significantly more statements and showed a preference for the *Describer*. Simplifications such as the *Describers* will lower the entrance bar for an author. The underlying structure of the *Describer* can be used to simplify creating conditional statements.

Acknowledgements

First, thanks to my supervisor, Duane Szafron, for helping me refine my ideas and encouraging me.

Next, thanks to everyone involved in the ScriptEase research group for their help, support and suggestions.

Thanks to the Department of Computing Science at the UofA for giving me so many opportunities to get involved.

To Tom Johnson, thnks for sponsoring my study and allowing me to run my user study with participants from psych 104/10.

Thanks to Mike Carbonaro for providing insight into understanding my user study results and Walter Bischof for helping with the stats.

To all the grad students I've hung out with, you know who you are, thanks for the support, putting up with venting, and making the experience so worthwhile.

Finally, thanks to my family for all their support over the years.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Video games are a multi-billion dollar industry in North America. In 2007 alone, consumers spent $18.85 billion on games and consoles [1]. Video gaming is a fast-growing industry that is quickly catching up to the movie industry [17, 1] (or exceeding depending on what counts as movie profits). This rapid growth can be explained by a changing consumer market that includes more older gamers and more female gamers, and an increase in length and interactivity of the games that has improved their replay and sequel value [17, 1].

Since the demand for video games has been growing each year, the video game industry has been struggling to keep up. For example, *Grand Theft Auto IV* [10] took three and a half years to develop, cost $100 million to make, making it one of the most expensive video games to date, and involved over 1000 people [12]. Another game, *Assassin's Creed 2* [22] took about 2 years to develop and involved a development team of about 450 [13]. However, even with the long development times and high cost of development, there are multiple areas where the industry continues to cut corners when creating new games. A popular example is in the scripting of non-player characters (NPCs) or, more accurately, the lack of scripting. It is not uncommon to see characters stare at a wall, walk into objects, or repeat the same lines.

Video games are created by teams that usually number in the hundreds like *Assassin's Creed 2*, but may number into the thousands (*Grand Theft Auto IV*), with each person having a specific role, such as programmer, graphic artist, voice actor, story author, tester, marketer and sound engineer. However, there are some inherent flaws in this process. Specifically, the story author must communicate ideas clearly to the programmer. If there is some miscommunication the game may end up very different from what the author had in mind. Most story authors are unable to program, and therefore must rely on the programmers to interpret this communication correctly. Only after a programmer has implemented a concept can a story author test it and provide feedback. This process creates a large bottleneck in creating video game content related to the story.

1.1 Video Game Content Problem

Although top-of-the-line graphics can initially help sell a game, it is the content that makes a video game a success. However, the process of developing this content is currently a large bottleneck. Video game content is usually created by designers, including story authors, and then implemented by programmers. This means that designers have to communicate their ideas to programmers, wait for a programmers to implement them, test the ideas, and then go back to the programmers to request any changes and start the process over again.

Imagine a scenario in a game where the player character (PC) is in a room in a castle. In the room there is a locked heavy door, a chest containing a potion, a locked lever and a troll. In the chest is a Potion of Death Armour that will help the PC to defeat the troll. Defeating the troll will unlock the lever and pulling on the unlocked lever will unlock and open the door and allow the the PC to exit the room.

This scenario is simple to understand and many video game toolsets can allow a non-programmer to easily assemble the objects into a story module. However, if the module was then played, nothing would happen when the PC killed the troll. This is because no scripts have been written and attached to the various objects, specifying what should happen and when. For example, the troll requires a script that details the actions that need to happen when it dies; in this case, the lever should be unlocked. The same goes for the lever, which requires a script detailing that the door in the room will open when the lever is pulled. A programmer is usually needed to write these scripts.

However, learning to script a video game requires a steep learning curve and cannot be quickly taught to designers.

1.2 ScriptEase

ScriptEase [20, 15] is one solution to the scripting problem. It allows non-programmers to design their own video game modules for BioWare's *NeverWinter Nights* video game. This has been verified by experiments where ScriptEase was successfully used by Grade 10 English students who were able to make their own modules [9, 21, 8, 3, 4, 5]. ScriptEase works by supplying a pattern catalogue that authors choose from. ScriptEase has four types of patterns (encounter, dialogue, behavior and quest) but, for the purpose of this research, we focus on encounter patterns.

1.2.1 ScriptEase Encounter Patterns

An encounter pattern is triggered by a specific event and consists of a set of actions to execute in response to that event. An example encounter pattern would be the PC pulling a lever to open a door. The pulling of the lever is the event that triggers the pattern and the action is to open the door. Events are typically caused by the PC's interaction with the game environment such as pulling a lever, opening a chest or killing a troll.

Within ScriptEase, encounter patterns are generalized so that they may apply to a number of game objects. Instead of calling the pattern *Pull lever - open door* the equivalent encounter pattern in ScriptEase is called *Placeable use - toggle door.* This generalization allows the designer to be creative in the object chosen to toggle the door while still being able to keep the pattern catalogue small. For example, this pattern could also be used to have a door close when the PC looks inside a chest. The pattern catalogue would have grown exponentially if the patterns were object specific and become a burden to use.

For a pattern to be generalized, the pattern cannot be pre-assigned to specific game objects. Instead, all game objects required by the patterns become options, with the pattern specifying the specific type for each option. For example, in the *Placeable use - toggle door* pattern there are two options: a placeable and a door. The types are conveniently *placeable* and *door*. Options can usually be set in one of three different methods.

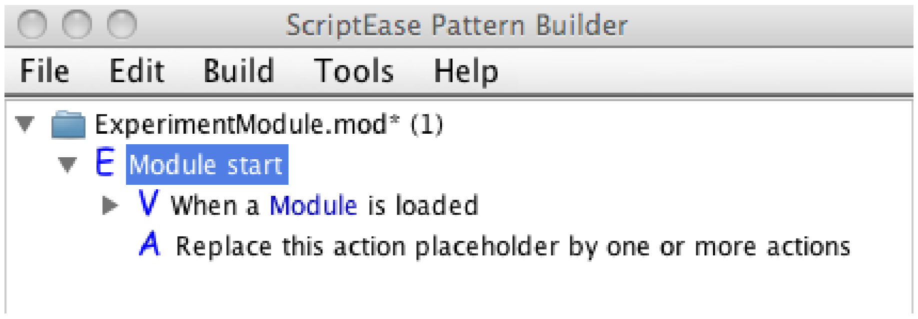
1. The author can choose from a drop down list that contains the names of the options that have been previously set within the encounter. Selecting an option name binds the current option to the value of the option whose name was selected.
2. The author can use a game object picker to choose from the objects in the module or when appropriate enter a specific string/number.

The third method is more complicated. There are times when neither method provides enough flexibility and the author must first create one or more definitions and then set the option to the definition.

Each encounter consists of one or more action atoms. These action atoms also have their own set of options like encounters for generalization. However, some of the options are linked by the designer to the options that are set for the encounter. For example, the encounter pattern has an option specifying a door that the author sets to be the Metal Door. When creating the encounter pattern, the designer wants the door specified at the encounter level to open and includes an action atom that does so. Instead of requiring the author to bind the same door to a new option for the action atom, the designer can pre-set the option to link to the same object selected at the encounter level. This decreases the number of options the author must set.

Once a pattern is chosen, the author can adapt the pattern by adding or removing actions. For example, the author could add an additional action that has the PC speak a one liner when the lever is pulled. Additional action statements will contain more options. All of these options must be set by the author.

Let us expand on the scenario of the troll in the castle. Instead of the room consisting of one chest that contains the potion, assume there are now five chests spread about the room, but only one contains the potion. This scenario can be created by simply adding the additional chests to the room and placing the potion inside a specific chest. However, if the same chest is always used, the location would be obvious to the player if the game was played a second time or the PC died and the player restarted from a saved game. To make the scenario more rewarding the author might want the potion to appear in a random chest. This changes the decision of what chest to place the potion in from being predetermined to being decided while the game is being played.



ExperimentModule.mod* (1)
E Module start
V When a Module is loaded
A Replace this action placeholder by one or more actions

Figure 1.1: The Module Start encounter pattern in ScriptEase.

We can turn this into a ScriptEase encounter pattern by using the *Module start* encounter pattern, which contains only a placeholder action, as shown in Figure 1.1. The author must replace the placeholder by one or more actions. In this case, the author wants the module to randomly choose a chest and place the potion inside. To do so the author would start by choosing to add a new action, specifically the action *Give an item to a creature/placeable* as shown in Figure 1.2. The author now has three options to set: who provides the item, what the item is, and who receives the item. The first two can be set using the picker to choose the Blue Chest (in this case, no one actually gives the item to the chest so we can use any object) and the Potion of Death Armour. The receiver is the tricky one as it needs to randomly choose between the five chests in the room. At this point, the author can no longer easily set the option value and must use definitions to describe the chosen chest.

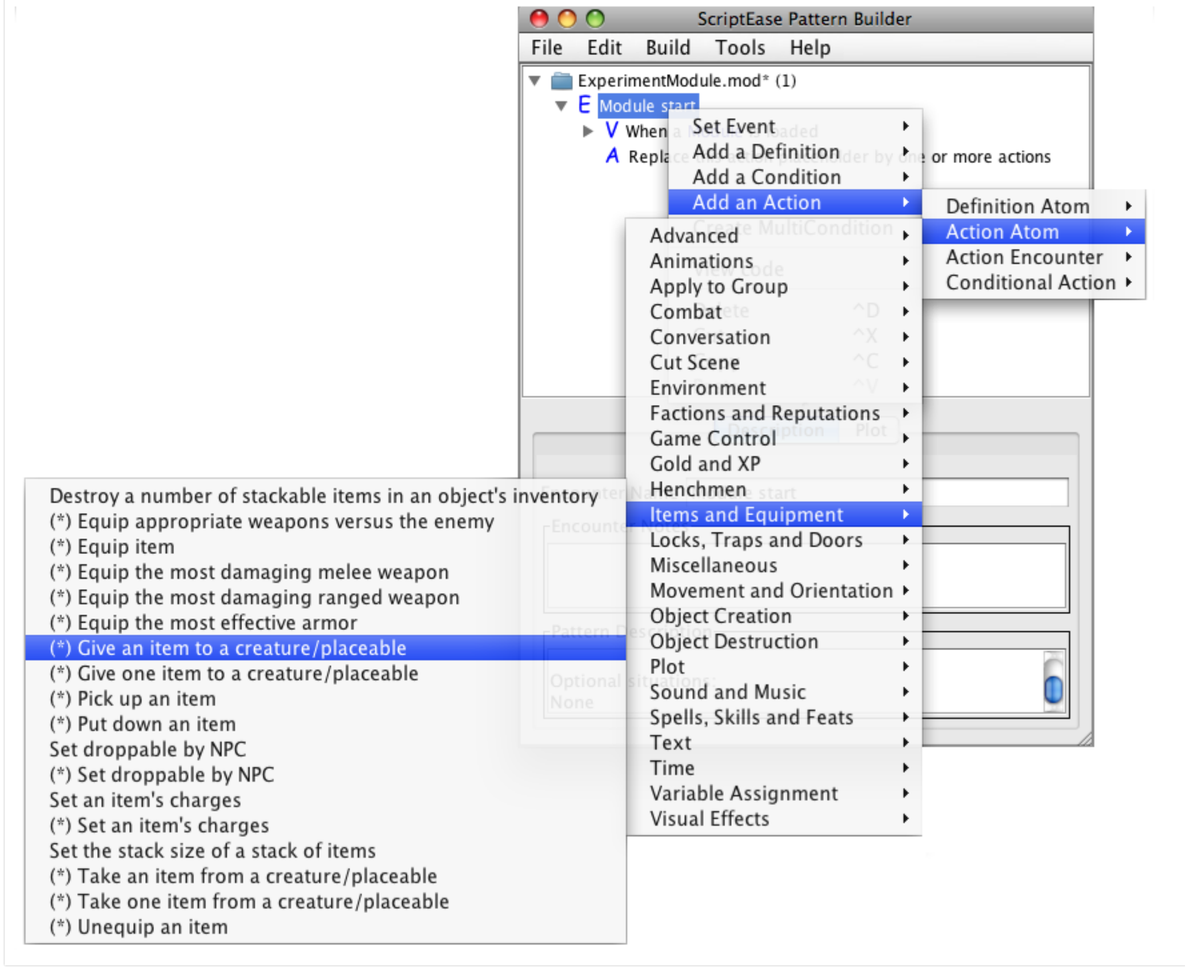
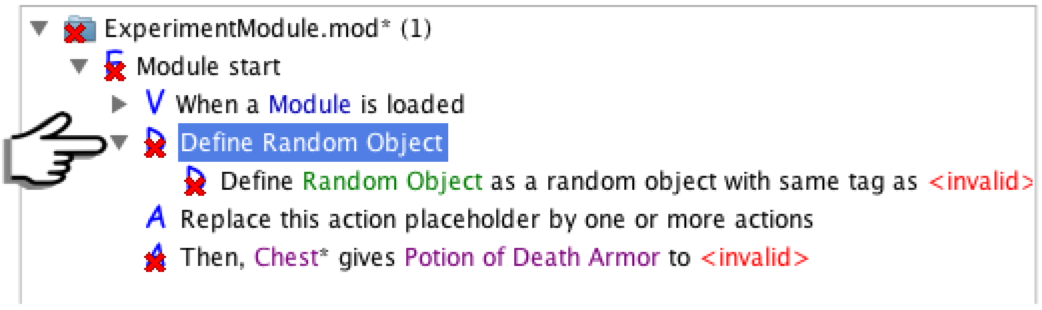


Figure 1.2: Adding an action to an encounter pattern in ScriptEase.

1.2.2 Definitions in ScriptEase

Definitions are used when the author needs to either compute an object/value that changes while the game is running, such as a the nearest creature to a door, a roll of the dice, or a creature's statistic like hit points. They are also used when the author wants to compare values (who has more hit points? or is the PC's charisma greater than 15?). Definitions provide the author with more flexibility in assigning options as multiple definitions can be linked together to compute more complex objects/values. Definitions can be added to an encounter at any time and all the definitions within an encounter are listed within a definition block as shown in Figure 1.3. The definitions are available to all the actions within the encounter and can be used to set options.



ExperimentModule.mod* (1)
Module start
When a Module is loaded
Define Random Object
Define Random Object as a random object with same tag as <invalid>
Replace this action placeholder by one or more actions
Then, Chest* gives Potion of Death Armor to <invalid>

Figure 1.3: The definition block in an encounter pattern.

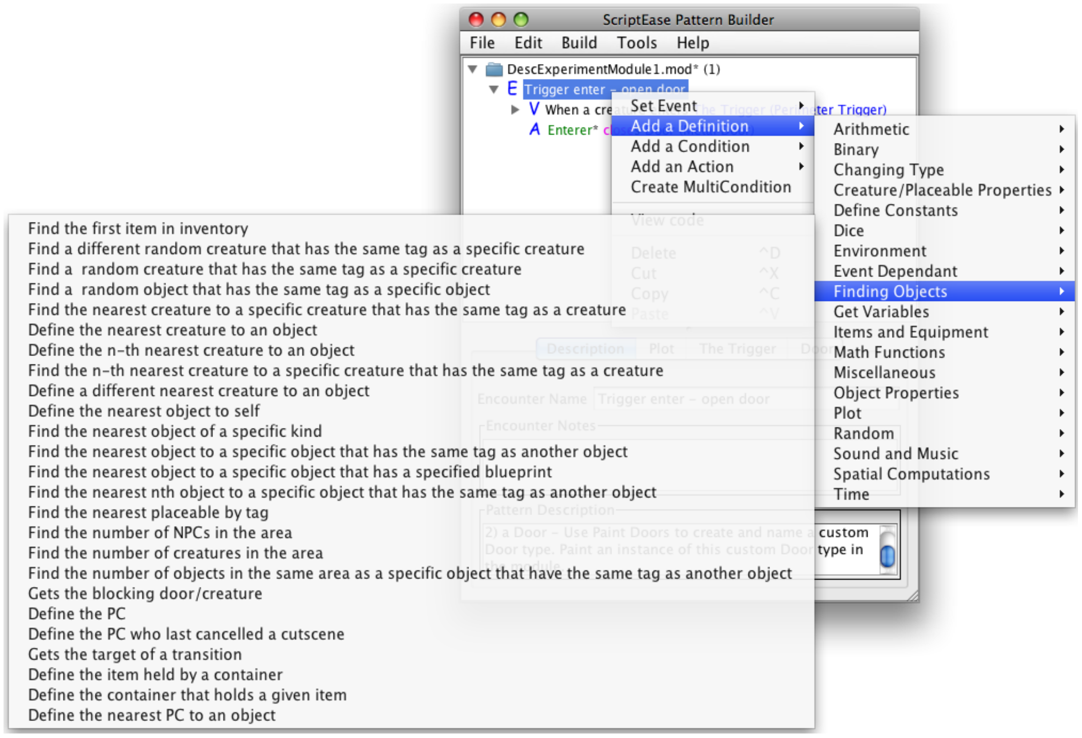
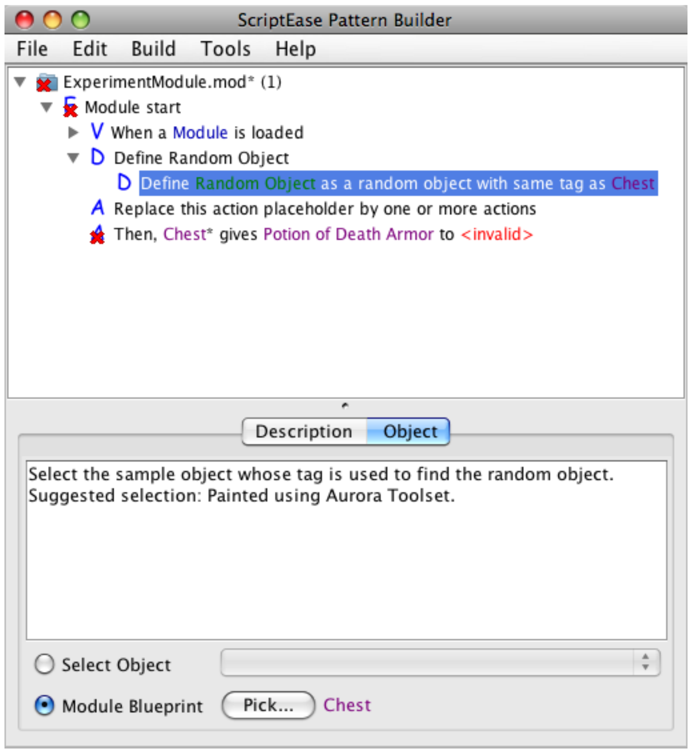

ScriptEase Pattern Builder
File Edit Build Tools Help
DescExperimentModule1.mod* (1)
Trigger enter – open door
Set Event
Add a Definition
Add a Condition
Add an Action
Create MultiCondition
Arithmetic
Binary
Changing Type
Creature/Placeable Properties
Define Constants
Dice
Environment
Event Dependant
Finding Objects
Get Variables
Items and Equipment
Math Functions
Miscellaneous
Object Properties
Plot
Random
Sound and Music
Spatial Computations
Time
Find the first item in inventory
Find a different random creature that has the same tag as a specific creature
Find a random creature that has the same tag as a specific creature
Find a random object that has the same tag as a specific object
Find the nearest creature to a specific creature that has the same tag as a creature
Define the nearest creature to an object
Define the n-th nearest creature to an object
Find the n-th nearest creature to a specific creature that has the same tag as a creature
Define a different nearest creature to an object
Define the nearest object to self
Find the nearest object of a specific kind
Find the nearest object to a specific object that has the same tag as another object
Find the nearest object to a specific object that has a specified blueprint
Find the nearest nth object to a specific object that has the same tag as another object
Find the nearest placeable by tag
Find the number of NPCs in the area
Find the number of creatures in the area
Find the number of objects in the same area as a specific object that have the same tag as another object
Gets the blocking door/creature
Define the PC
Define the PC who last cancelled a cutscene
Gets the target of a transition
Define the item held by a container
Define the container that holds a given item
Define the nearest PC to an object

Figure 1.4: Definitions categories and examples in ScriptEase.

To add a new definition to an encounter pattern the author must first select from various categories of definitions such as binary (are two spells the same?), math functions (adding two numbers together) and object properties (a creature's hit points) and then within the definitions in a category as shown in Figure 1.4. Like encounters and actions, some definitions have their own set of options. For example, the definition to add two values together requires two numbers as options. Also like encounters and actions, these options can be set using the same methods, including using other definitions. However, while actions are aware of all the definitions in the encounter, definitions only have knowledge of the definitions that are listed before them in the definition block.



ExperimentModule.mod* (1)
Module start
V When a Module is loaded
D Define Random Object
D Define Random Object as a random object with same tag as Chest
A Replace this action placeholder by one or more actions
Then, Chest* gives Potion of Death Armor to <invalid>

Select the sample object whose tag is used to find the random object.
Suggested selection: Painted using Aurora Toolset.

Select Object [✓ Module Blueprint] Chest

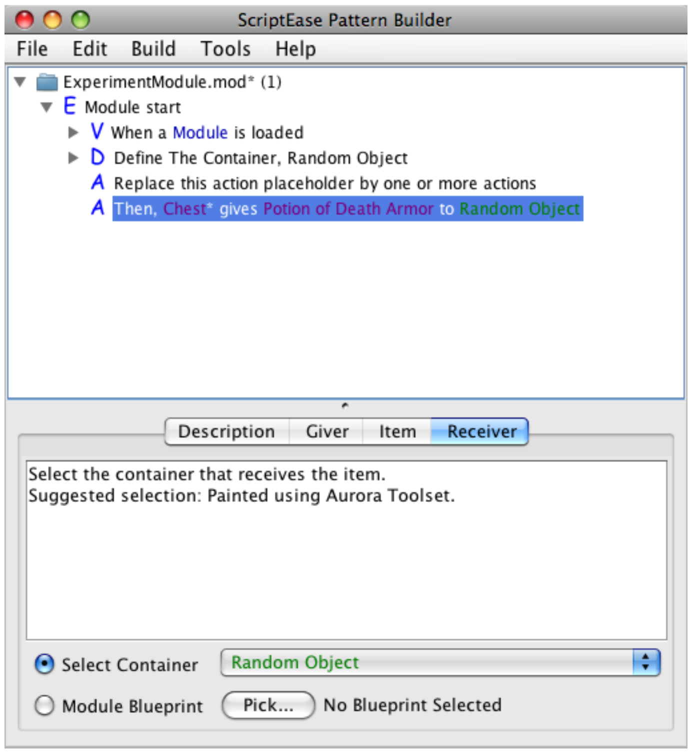
Figure 1.5: Complete definition in ScriptEase.

Let's return to the example of placing a potion in a random chest. We need to find a definition that will randomly choose a single object from a set of objects. Looking in the *Finding Objects* category, we can use the single definition *Find a random object that has same tag as a specific object*, as shown in Figure 1.4. This definition has one option, the object whose tag we are trying to match. Figure 1.5 shows the complete definition after setting the option to: Chest. Now, the original action can be finished by setting the Receiver option to the "Random Object" definition as shown in Figure 1.6.

1.2.3 Difficulty using Definitions

The current structure of definitions within ScriptEase can cause difficulties for the author due to the volume of information and the rules the author must adhere to when using them.

Because all the definitions are available at any time, the author must always sort through a long list of definitions, including those that do not make sense based on what the author is trying to accomplish. As ScriptEase contains hundreds of definitions, this can quickly


ScriptEase Pattern Builder

File Edit Build Tools Help

- ExperimentModule.mod* (1)
  - E Module start
    - V When a Module is loaded
    - D Define The Container, Random Object
    - A Replace this action placeholder by one or more actions
    - A Then, Chest* gives Potion of Death Armor to Random Object

Select the container that receives the item.
Suggested selection: Painted using Aurora Toolset.

Select Container: Random Object

Module Blueprint: Pick... No Blueprint Selected

Figure 1.6: Set option to definition in ScriptEase.

become overwhelming.

Once the author selects the correct category of definitions, there are many definitions that are similar but have subtle differences. For example, look at the following two definitions.

- Find the nearest object to a specific object that has the same tag as another object
- Find the nearest n-th object to a specific object that has the same tag as another object

The second definition contains the extra word *n-th*. This addition changes the definition from returning the nearest object as the first definition does, to returning the object that is n-th many away from the specific object, where n is an integer greater than or equal to 1. And even then, if the author chooses an *n-th* value of 1, the definition is now semantically equivalent to the first. When looking down a list of definitions that are all somewhat similar it is easy to miss small distinctions like the "n-th" and choose the wrong one or even try to use them interchangeably.

Some other difficulties with definitions include linking multiple definitions together and matching option types. All of these problems may confuse the author and discourage the use of definitions. My research contribution is to address these issues using a new mechanism called *Describers*.

1.3 Contributions and Outline

Definitions allow the author greater expressibility when implementing a story. However, the current level of difficulty in using definitions may discourage authors from using them and can limit the author's originality. I claim that *Describers*, a new method for setting options, would be easier to use and preferred by authors, thereby allowing authors to create more original stories with less effort.

The next chapter takes a step back and looks at the related work that has already been done on making video game design accessible to the non-programmer. This includes a look at Scratch, created at MIT and Alice, created at CMU, as well as tools used by various video game companies, like BioWare, that allow authors to create their own modules.

The primary contribution of this thesis is *Describers* and this technique is discussed in detail in Chapter 3. *Describers* work by allowing the author to choose a description category. Each category represents multiple definitions presented as a single describer line which the author can modify as needed.

A user study was conducted using 1st year Psychology students to compare the *Describer* against using definitions, the results of which are presented in Chapter 4. The user study showed that the participants were able to complete more tasks while using the describers and that they overwhelmingly chose the *Describer* over using definitions.

Finally, Chapter 5 concludes this work with a summary and a look at future work such as condition statements.

Chapter 2

Related Work

Tools to allow both programmers and non-programmers easier ways to create their own video games started appearing in the late 1990s. Since their inception these tools have become more and more popular to the point that there are now "games" where the gamer is really buying a game creation tool rather than a game. Two examples are *Spore* by EA [14] and *Kodu* by Microsoft Research [19].

To help gain acceptance many of these tools have a community of some sort, where authors can share their creations, ask questions, and get help. For *NeverWinter Nights*, in its first 3 months there were over 1000 user created modules uploaded to the community site [11], a site that is still active today. This community extends the life of a game since gamers interested in playing new scenarios can access more content and authors interested in designing games have a huge field of players to test and comment on their creations.

The tools generally fall into two categories: a non-programmer-friendly research motivated tool or an add-on for a commercial game. The first category includes programs like *Alice* and *Scratch* while the second category includes tools like the Aurora Toolset for *NeverWinter Nights* and the TES Construction Set for *Oblivion* by Bethesda [2]. However, this distinction is becoming blurred by tools like ScriptEase (research motivated, but applies to a commercial game) and *Kodu* a commercial game designed to enable non-programmers to create their own games.

In the following sections I examine two programs designed specifically for non-programmers, Alice and Scratch, and then at the characteristics of tools included with commercial video games.

2.1 Programs for Non-Programmers - Alice and Scratch

ScriptEase is not the only environment that has been designed with non-programmers in mind, nor is it the only one designed at a University. Specifically, Alice [6] (and Storytelling Alice [7]) from CMU and Scratch [16] from MIT are similar environments. However, there are a couple of big differences between them and ScriptEase.

ScriptEase currently generates code for *NeverWinter Nights* modules, which means that ScriptEase does not need to implement graphics for the game module as the graphics are all supplied by *NeverWinter Nights*. Since *NeverWinter Nights* is a commercial game the graphics are of fairly high quality. Even though *NeverWinter Nights* is now seven years old, the graphics are still of much better quality than Alice or Scratch. Alice and Sctrach have built-in graphics designed specifically for their tool and contain a library of images the author can use. Scratch's library consists of 2-D images of fairly poor quality, while Alice's library is more extensive, containing 3-D images and landscapes. The types of images in both libraries also indicate a younger target audience than *NeverWinter Nights*, with many images of cute animals and school related items. However, similar to *NeverWinter Nights*, the author can create and import new objects.

Scratch and Alice were both designed to help teach programming concepts. Unlike ScriptEase, which hides the scripting from the authors (although it is available for them to view), these programs require the authors to direct the action and interaction of objects by using syntax similar to a programming language. The next section describes their syntax in detail.

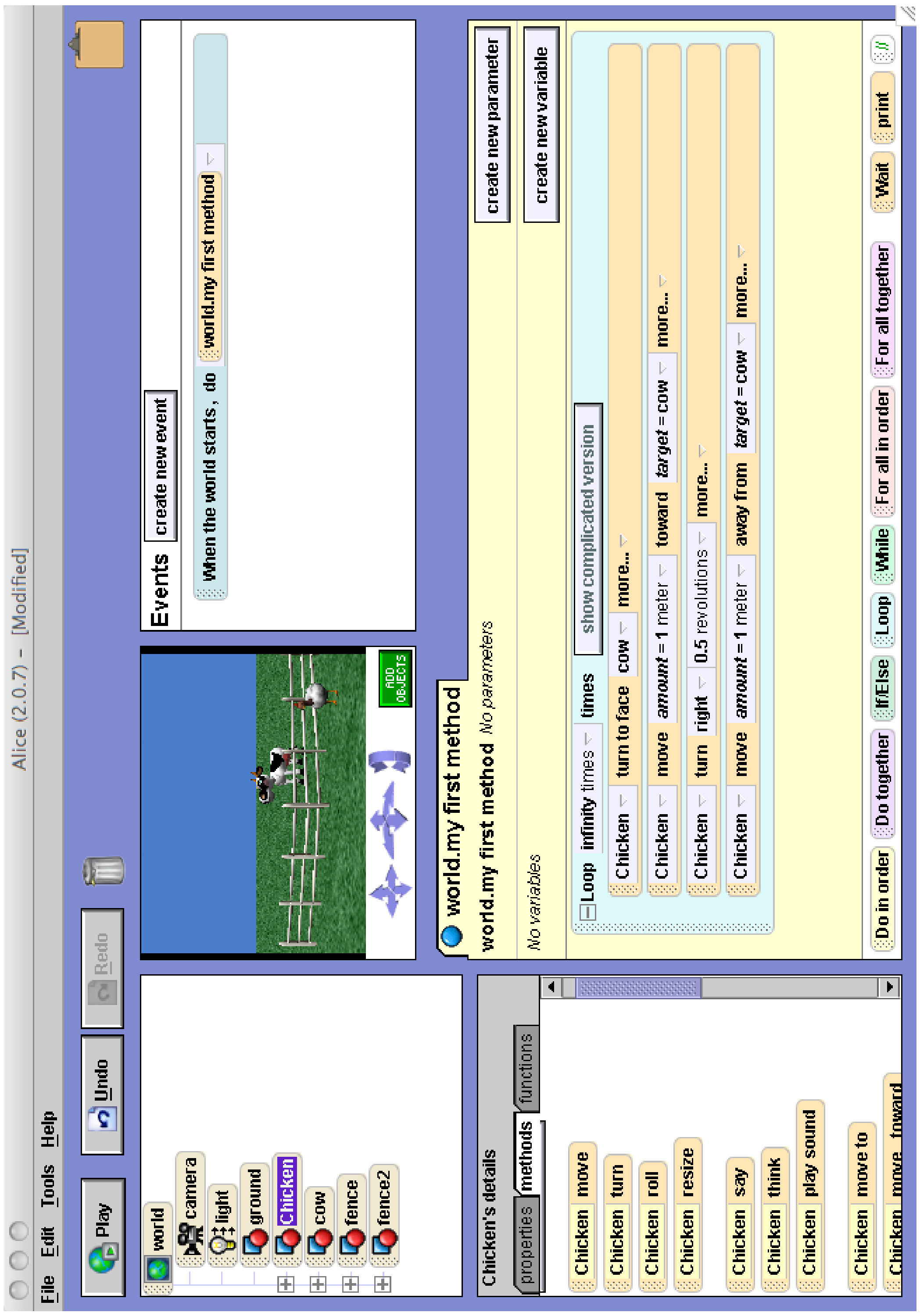
2.1.1 Alice and Storytelling Alice - CMU

Alice is probably one of the longest existing tools designed to allow non-programmers to create their own video games. Alice 2.0 was initially released in 1999, although modifications and updates have occurred since. A new beta version of Alice 3.0 is currently available. While Alice was the original, there is now a second sister program called Storytelling Alice that aims to lower the entry bar and make the tool more accessible to younger authors. One way Storytelling Alice attempts that is to focus on creating stories or movies versus interactive games. Because the actual usage of both tools is almost identical, I have just presented the information on Alice.

To use Alice, the author must first populate the game world by choosing from the library of objects, or creating their own and importing them into the program. Once the world has been populated with objects, the author can construct scripts to control how the objects act and react during game play. An example screenshot of Alice is shown in Figure 2.1. Scripting is done by dragging predefined lines of "code" into a main code window which represents a single method on a world. Multiple methods can be created and they may include parameters and variables. Methods can be invoked by game events or other methods. For example, in Figure 2.1, the *When the world starts* event invokes the *my first method* method on the world object. All objects in the game world, including the ground, camera, and lights, come with a preset group of properties, methods and functions. In addition, a set of control statements exist: Do in order, Do together, If/Else, Loop, While, For all in order, and For all together. Alice also includes the ability to print (which prints to a sort of log screen that appears below the game screen), comment the code, and to wait (which stops game play for a specified period of time).

Authors build their game world by selecting pre-built lines from a code library and adding them to a method to build up a sequence of actions. For many of the code segments, the author must set parameter values which are similar to the options an author sets in ScriptEase. After dragging a code segment onto a method, if there is a parameter, a menu immediately pops up requesting the author to select from a set of possible bindings. For example, if the author chooses to have a chicken move towards the cow, they would choose the *Chicken move toward* method. This code segment requires two pieces of information from the author: how far to move and towards what game object. The author may select from some preset distances of 1/4, 1/2, 1 and 2 meters or enter a value. Next the author selects what game object the chicken should approach, in this case the cow. Once the required parameters have been set, the code segment appears as a sentence; for example, *Chicken move 1 meter toward the cow more...*, where *Chicken*, *1 meter* and *cow* are all buttons allowing the author to change these parameters at any time. The *more...* is also a button that produces a menu of other parameter values that can be associated with the code segment, but that don't need to be specified as they have defaults. For the *Chicken move toward* code, these optional parameters include *as seen by* (an object that controls the camera) and *duration* (how long the action lasts). A loop in which the chicken turns and walks towards the cow for one meter before turning away is shown in Figure 2.1.

However, Alice does not provide an easy mechanism for creating complex descriptions. Suppose we have a room with two doors and a person, Bob, pacing the room. After 5 seconds of pacing, we want Bob to move to the closest door and open it. Alice does not provide a predefined code segment to find the closest object to another object. Instead,



```
When the world starts, do  world.my first method
```

```
world.my first method
world.my first method  No parameters
No variables
Loop infinity times times   show complicated version
    Chicken turn to face cow more...
    Chicken move amount = 1 meter toward target = cow more...
    Chicken turn right 0.5 revolutions more...
    Chicken move amount = 1 meter away from target = cow more...
```

Figure 2.1: Example of an Alice project.

the author must calculate the distance from Bob to each door, determine which number is smaller, and then have Bob walk to the correct door. If the author requires a similar calculation for Mary, the author must recreate a similar segment of code for Mary, since Alice does provide the ability for an author to use the same method on multiple objects (there are no classes). The only alternative to copying and pasting code is to use a method on the world and pass in Bob or Mary. Methods on the world are essentially static methods. In a sense, functions in Alice are similar to definitions in ScriptEase, and methods in Alice are similar to action atoms in ScriptEase. Alice also contains Events which can be used to trigger some methods but not all. There are two types of methods in Alice, instance methods on objects that cannot be shared and global methods on the world.

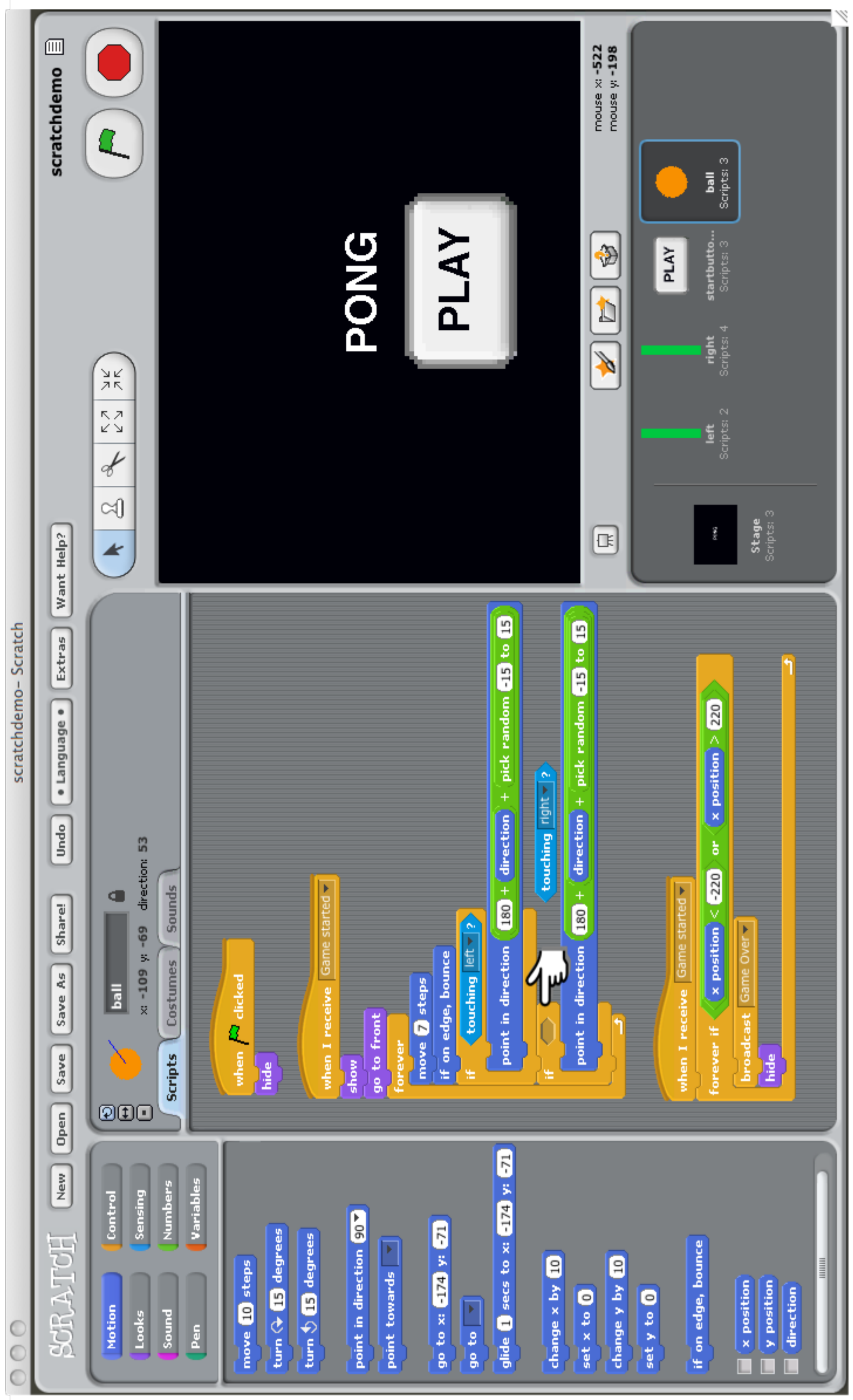
ScriptEase has always aimed to provide an experience where authors see plain English sentences whenever possible, just as Alice does. Alice's code segment structure where options are accessible within a sentence and the more button that allows an author to add more information to a sentence are not available in the current version of ScriptEase. However, these two features heavily influenced my design of *Describers* as I found them great ways to request information from the author without losing the context of what the information is describing.

2.1.2 Scratch - MIT

Scratch is a relatively recent video game design program that was first available in 2007, although initial research dates back to at least 2003 [16]. Scratch is designed for authors who are 8 years old or older and has an active user community. Scratch's website allows authors to upload their creations to share with others in the Scratch community which supports their slogan of *imagine - program - share*. Like Alice, Scratch was designed to teach programming skills and therefore uses a syntax that mirrors actual code. An example screenshot of Scratch is shown in Figure 2.2.

To program with Scratch, an author selects puzzle-like pieces of code of different shapes that can snap together if they can be connected. Unlike Alice, Scratch provides no hint to the author that a parameter must be set or how to set it. Control statements, like loops, just provide an empty space indicating the author can insert code at that location as shown in (Figure 2.2), with the only indication of what can be inserted identified by the shape of the hole.

Unlike Alice, which allows the author to create very distinct methods, all code for a single object in Scratch is placed in the same code window. The code segments do not need



```
when [flag] clicked
hide

when I receive [Game started]
show
go to front
forever
    move (7) steps
    if on edge, bounce
    if <touching [left]?>
        point in direction ((180) + (direction) + (pick random (-15) to (15)))
    if < >
        point in direction ((180) + (direction) + (pick random (-15) to (15)))

<touching [right]?>

when I receive [Game started]
forever if <<(x position) < (-220)> or <(x position) > (220)>>
    broadcast [Game Over]
    hide
```

Figure 2.2: Example screenshot of Scratch. The hand points to an example empty space in an if statement. The missing piece of code is on the right side of the hand.

to be connected, allowing for multiple methods to be created. Figure 2.2 shows three code segments on a single object. Each method must have an event as the first component.

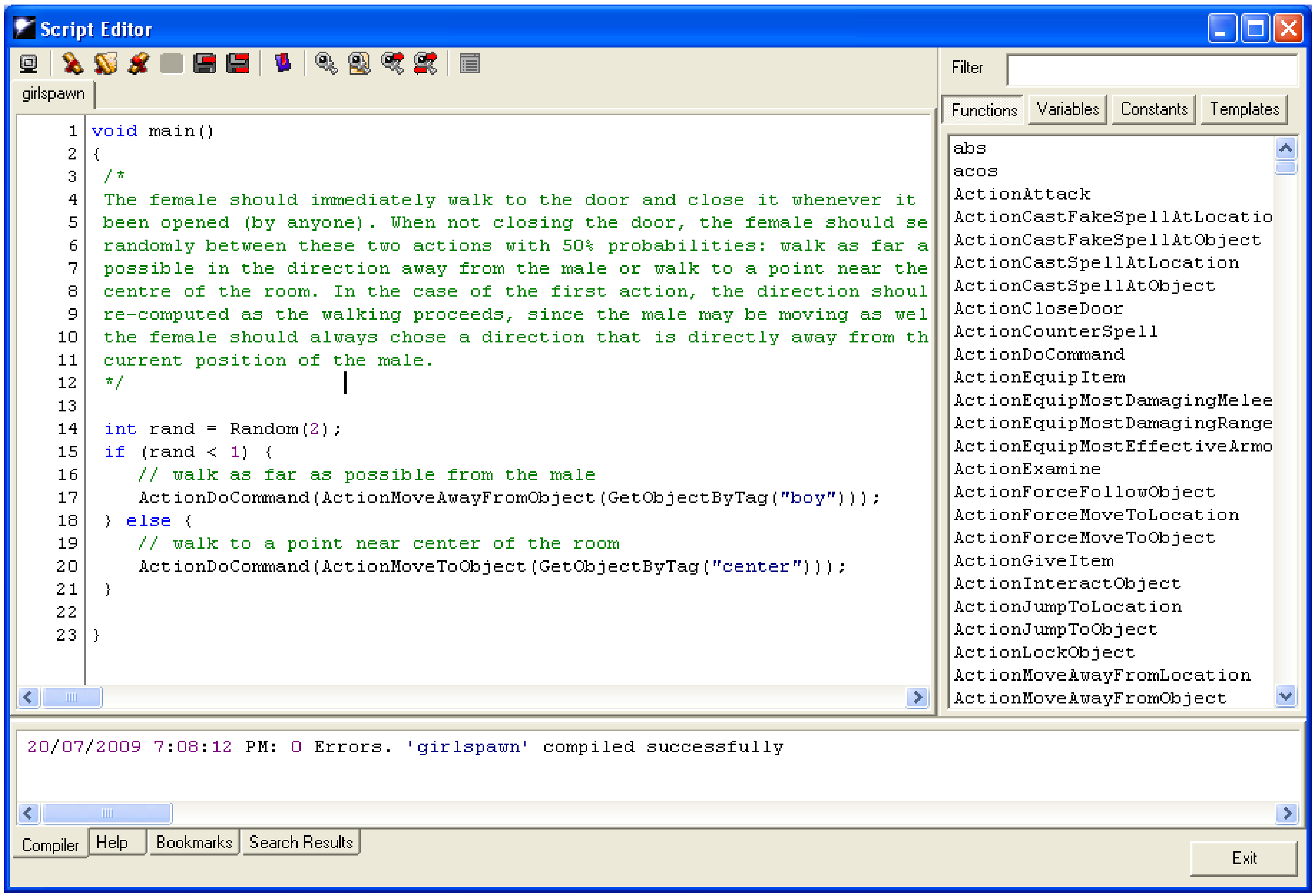
Scratch has two techniques to share information. The first technique allows a game object to broadcast a text message that other game objects can receive. This can be used to pass control between various blocks of code. The second technique is by using variables. Authors can make both global and local variables. However, as a variable must be bound to a number and the broadcast messages are text which cannot be parsed, Scratch strictly limits the amount and type of information the author can pass between code blocks.

While Scratch provides an easy to learn interface, the actual ability of the author is quite limited on multiple fronts. As already mentioned, Scratch limits the amount and type of information that can be shared between code blocks. The graphics provide no support for 3D images, and creating a slide-scrolling 2D game is extremely difficult. Scratch also comes with a preset collection of code segments with no way for the author to create new segments. The limited code library allows an author to quickly learn everything that can be done in Scatch. However, this also means that the author may quickly outgrow Scratch as the author's programming knowledge increases. Scratch limits an author's complex control of game creation by limiting the scope of programming.

2.2 Video Game Included Tools

At the same time Alice was first being introduced at the end of the 1990s, video game companies were starting to include the ability for players to modify and create their own content for games. *NeverWinter Nights* was one of the first games that heavily promoted both its included toolset and a community of players formed to share their modules.

Today there are many games that include tools and communities where amateur authors post their own content that other players comment on and play. However, even as the use and production of these tools has increased, the actual ease of use of the tools has not. Many times the tools were designed to aid in the construction of the original game and are included as an extra. Since these tools were not intended for consumers they are difficult to use. The tools usually rely heavily on a scripting language designed for the game. These scripting languages are usually specific to a single game, but may be used across sequels and often have a syntax that is similar to C and Java. This provides a roadblock for non-programmers who want to create modules for multiple games as they need to learn multiple scripting languages and cope with the quirks of each language. For example, *Neverwinter*



```
void main()
{
 /*
 The female should immediately walk to the door and close it whenever it
 been opened (by anyone). When not closing the door, the female should se
 randomly between these two actions with 50% probabilities: walk as far a
 possible in the direction away from the male or walk to a point near the
 centre of the room. In the case of the first action, the direction shoul
 re-computed as the walking proceeds, since the male may be moving as wel
 the female should always chose a direction that is directly away from th
 current position of the male.
 */

 int rand = Random(2);
 if (rand < 1) {
    // walk as far as possible from the male
    ActionDoCommand(ActionMoveAwayFromObject(GetObjectByTag("boy")));
 } else {
    // walk to a point near center of the room
    ActionDoCommand(ActionMoveToObject(GetObjectByTag("center")));
 }

}
```

```
abs
acos
ActionAttack
ActionCastFakeSpellAtLocatio
ActionCastFakeSpellAtObject
ActionCastSpellAtLocation
ActionCastSpellAtObject
ActionCloseDoor
ActionCounterSpell
ActionDoCommand
ActionEquipItem
ActionEquipMostDamagingMelee
ActionEquipMostDamagingRange
ActionEquipMostEffectiveArmo
ActionExamine
ActionForceFollowObject
ActionForceMoveToLocation
ActionForceMoveToObject
ActionGiveItem
ActionInteractObject
ActionJumpToLocation
ActionJumpToObject
ActionLockObject
ActionMoveAwayFromLocation
ActionMoveAwayFromObject
```

```
20/07/2009 7:08:12 PM: 0 Errors. 'girlspawn' compiled successfully
```

Figure 2.3: Example of NWScript code.

Nights uses the scripting language NWScript. An example of NWScript code can be found in Figure 2.3.

With these game-specific scripting languages, the authors must rely heavily on the game's community for support to learn the language and tool. Unlike more common programming languages such as Java and C++, multiple books are not being published on these game scripting languages and examples are not prevalent on the internet.

When learning the language, the authors suffer another setback when it comes to finding and eliminating bugs. Video games are usually designed to fail silently so as to not interrupt the game play. This makes it difficult when playing through a scenario to figure out when and why it failed or if it even did. The tools where scripting is done usually have very limited warnings for errors. Bugs that other programming environments would catch are often missed.

There are benefits to using these integrated tools such as the community support and the tight coupling between the tool and the game. Because ScriptEase is not integrated with *NeverWinter Nights* or the Aurora Toolset it allows the authors to script a module but does not provide support for designing and placing objects. Tools like the Aurora Toolset allow the author to do everything all in one place but, of course, this means the author must use the scripting language of the game. Figure 2.4 shows a screenshot of the Aurora Toolset.

Another benefit of these tools is the interest they create within the games community for the game. As mentioned before, this ability for the players to add more content to their game can greatly extend a game's life. It can also provide an interactive and exciting workspace where the author can learn to program. It's not hard to see that learning to program in a visually enriched environment will be more engaging than the typical "Hello World" program taught to first year computer science students.

Therefore, while ScriptEase may not be an all-in-one tool like the Aurora Toolset, it makes up for this by not requiring authors to learn a scripting language and by shortening the learning curve. However, ScriptEase can be made even easier by further simplification. My research involves simplifying how options are set on actions and encounters by creating a new method called *Describers* which is presented next.

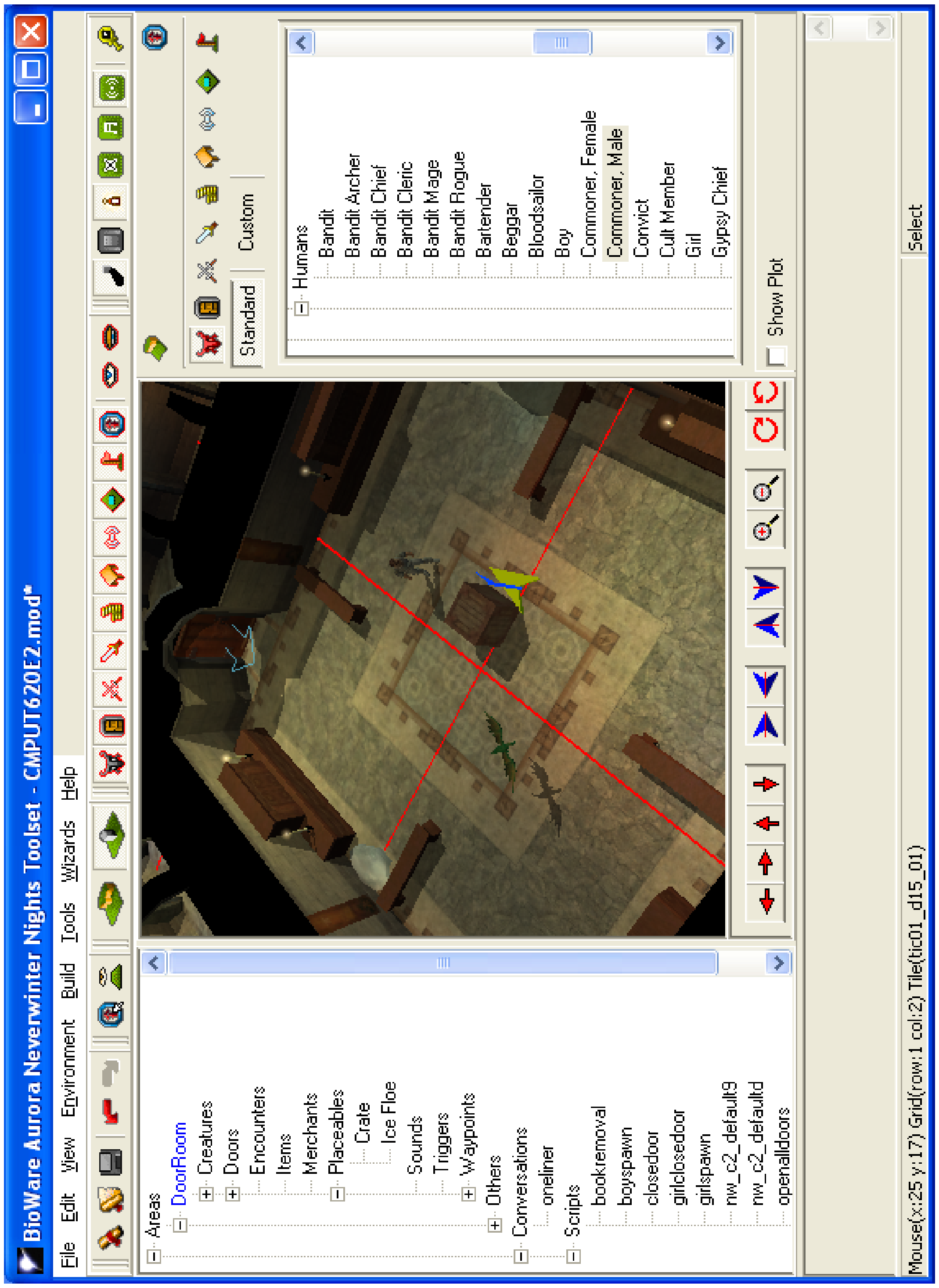


Figure 2.4: Screenshot of the Aurora Toolset.

Chapter 3

Describers

The use of definitions was one of the main areas of ScriptEase identified as more complicated than necessary. Definitions are necessary when an author wants to access an object's properties (find a creature's hit points), combine values (do arithmetic to compute a creature's heading), or set option values to be calculated at run time (find the nearest creature to a door). Ultimately, definitions are used to set options or to provide the condition for a ScriptEase if construct. To use definitions, an author has to sort through a lengthy list of definitions to find the one that matches the goal.

To simplify how an authors finds and uses definitions, I developed a new way to set option values. This new process uses *Describers* to allow the author to start with an English sentence and modify it to describe the option they want to set.

This chapter will discuss the problems with definitions, what *Describers* are and how they work, how knowledge is represented within *Describers* and, finally, how *Describers* solve the definition problems.

3.1 Definition Problems

Describers were developed to solve the problems inherent in definitions. While examining the pattern catalogue I discovered three problems with definitions as listed below.

1. Content overload.
   - Option types.
   - Similar definitions.
2. Multiple workspaces.
3. Linking definitions.

I then developed *Describers* with the intent of finding solutions to these problems.

3.1.1 Content Overload

When the author needs to create a new definition the entire definition catalogue is always available and visible. The author sees all the definition patterns whether or not they relate to the goal for the current definition. ScriptEase does attempt to focus the author by grouping the definitions into multiple categories as shown in Figure 1.4. However, even after the definitions are grouped there is still an overwhelming amount of data presented to the author with about 20 groups averaging 15 definitions per group.

When an author is looking for a definition there are two concerns. First, the type of object being defined must match the type of the option being set or be binary to be used as a condition. Second, the semantics of the definition must match the goal. In ScriptEase the definitions are grouped by semantics. This means the author must continue to sort the data to find those that contain the correct definition type.

Option Types

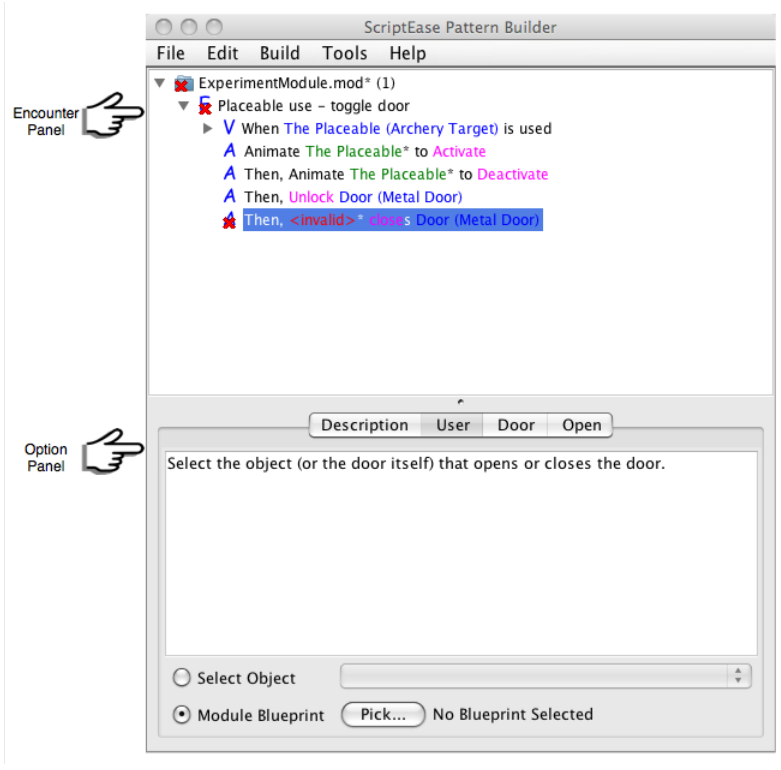
In order to assign a definition to an option, the type of the defined object must either be the same as the option type, or one of the option type's subtypes. For example, an option of type *object* can be set to a defined object of type *creature* because a *creature* is a subtype of *object*. This does not work both ways, as an option of type *creature* cannot be set by a definition of a generic *object*. Since ScriptEase has no knowledge of the option or condition type when a definition is being constructed it is up to the author to remember the goal type.

Similar Definitions

Within the definition categories there are many definitions that look almost identical, but contain small differences. An example of this can been seen in Figure 1.4. For example, there are many definitions to find the nearest object, each with subtle differences. When an author is searching for a definition it is important to notice these so that the correct definition can be selected.

3.1.2 Multiple Workspaces

When using definitions the author must shift focus from the option panel as shown in Figure 3.1 to the encounter panel to create the new definition. The author must add the appropriate definitions to the definition block which can be seen in Figure 3.2. Most definitions have their own options. For example, the definition *random object* shown in Figure 3.2 has an option called object that describes the tag the random object must match. In addition to

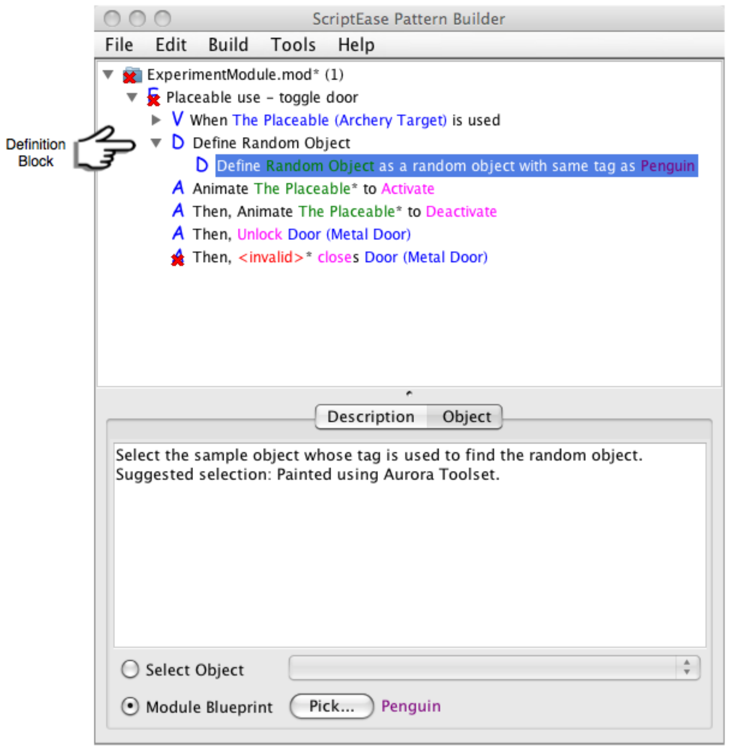

Encounter Panel
Option Panel
ScriptEase Pattern Builder
File Edit Build Tools Help
Description User Door Open

ExperimentModule.mod* (1)
Placeable use – toggle door
V When The Placeable (Archery Target) is used
A Animate The Placeable* to Activate
A Then, Animate The Placeable* to Deactivate
A Then, Unlock Door (Metal Door)
A Then, <invalid>* closes Door (Metal Door)

Select the object (or the door itself) that opens or closes the door.

Select Object
[✓ Module Blueprint] Pick... No Blueprint Selected

Figure 3.1: ScriptEase window showing the encounter panel on top and the option panel on the bottom.



ExperimentModule.mod* (1)
Placeable use – toggle door
V When The Placeable (Archery Target) is used
D Define Random Object
D Define Random Object as a random object with same tag as Penguin
A Animate The Placeable* to Activate
A Then, Animate The Placeable* to Deactivate
A Then, Unlock Door (Metal Door)
A Then, <invalid>* closes Door (Metal Door)

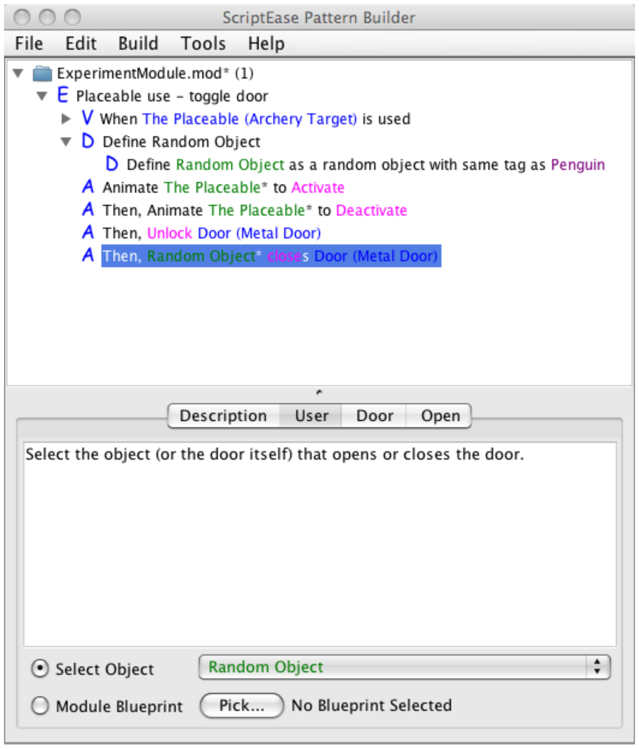
Select the sample object whose tag is used to find the random object.
Suggested selection: Painted using Aurora Toolset.

Select Object
[✓ Module Blueprint] Pick... Penguin

Figure 3.2: Example of the definition block containing a single definition.

setting options in definitions, the author may need to link multiple definitions together. An example of definition linking is given in section 3.1.3.

Once the definition is complete the author must move back to the original option or condition panel and use the select object button to set the option to the completed definition as shown in Figure 3.3. When switching from the option panel, the author must remember the type of the option they are setting. Once the author has finished creating the definition, the original option still must be linked to the new definition in the option panel.



ExperimentModule.mod* (1)
E Placeable use - toggle door
V When The Placeable (Archery Target) is used
D Define Random Object
D Define Random Object as a random object with same tag as Penguin
A Animate The Placeable* to Activate
A Then, Animate The Placeable* to Deactivate
A Then, Unlock Door (Metal Door)
A Then, Random Object* closes Door (Metal Door)

Select the object (or the door itself) that opens or closes the door.

Select Object: Random Object
Module Blueprint: No Blueprint Selected

Figure 3.3: Example of setting an option to a definition.

3.1.3 Linking Definitions

Forward chaining requires that the author reach the goal by building from the small details towards the overall goal. For example, the author wants to determine how much gold to award a PC for completing a task. To provide a scaleable adventure the author wants to award the PC an amount of gold that is equal to the PC's level times one hundred. To complete this task the author must create some definitions. This task will require the author to use two different definition patterns, one that finds the PC's level and one that multiplies two numbers together. To start, the author would need to create the definition *Define The Level as <invalids>'s level* and set the creature option (currently *invalid*) to the *PC*. Next, the author needs to create the definition *Define Product as 0 times 0.* In this definition the author needs to set the first number (currently *0*) to the PC's level by using the known list to reference the first definition. The second number can be set by choosing to enter a constant and using the keyboard to enter one hundred.

If the author thought about this problem in the opposite order, i.e. first noting they needed to multiply two numbers together and second that they needed to find the PC's level, they would run into difficulty. The definition that multiplies two numbers together would be unable to "see" the definition of the PC's level because of scope.

Scope determines which definitions are available to the other definitions within a definition block. Specifically, a definition can only reference the definitions that are listed before it. In the gold award example, if the author decided to create the two definitions in the reverse order, the product definition would not be able to see the PC's level definition because of scope. The author does not receive any information as to why the definition is not available in the known objects list and it would be up to the author to trouble-shoot the problem.

It is common for an author to link multiple definitions. When linking is necessary, it is important to understand both the types of the objects being defined and the scope of the individual definitions. In addition, if definitions are linked together within the definition block, the author must remember to set the original option to the final definition.

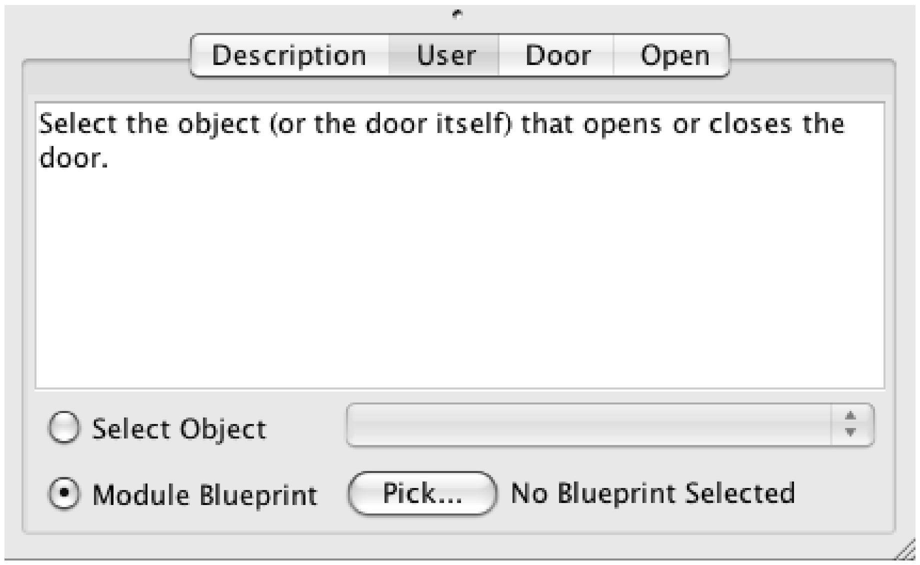
Describers provide a practical solution to these problems. At the end of this chapter we will revisit this list to see how the *Describers* solved each of the three problems.

3.2 Describers - What are they and how do they work?

Describers were developed to solve the shortcomings of definitions. However, it was important to retain the expressive power of definitions while preventing authors from catchable errors, such as incorrect type matching.

The main concept of *Describers* is to have the author make a series of guided decisions. Based on each decision, a template is provided that the author can modify as needed.

In the original version of ScriptEase when the author needs to set an option, a panel similar to the one shown in Figure 3.4 is presented. This panel allows the author to select a previously defined object from the known objects list by using the *Select Object* button. For example, if the action requires the creature who entered a trigger, this dynamic creature



Select the object (or the door itself) that opens or closes the door.

( ) Select Object

(•) Module Blueprint   Pick...   No Blueprint Selected

Figure 3.4: The panel for setting an option in the original ScriptEase.

will be placed in the known list by the trigger event. Alternatively, the author could use the picker to select a game world object statically included in the module or set a numerical or string value, or select from an enumerated list. For example, if the action requires the Grumpy Dwarf who was placed behind the bar it can be selected using the picker. If the author wants a dynamic object that does not have a previous definition, a definition must be created. For example, if the author wants the action to be performed by the creature that is currently nearest to a specific door, a new definition must be added to compute this creature. Unfortunately, if an author needs to create a definition, the focus must move away from this panel as discussed in section 3.1.2. With *Describers* I wanted to avoid the author having to move between various panels by keeping all the information required by a single action in the same panel.

When an author needs to set an option using *Describers* the panel looks like the one in Figure 3.5. The author still retains the ability to set the option using any of the three methods used in the original version of ScriptEase: known objects, picking, and new definitions. These three methods have been renamed in the *Describer* to *Recall it* (the known objects), *Pick it* (the picker) and *Describe it* (definitions). This order represents the relative difficulty of using each method, with recalling being the easiest and describing the most difficult. To access these choices with a *Describer*, the author clicks on the <*invalid*> text in the panel and is given a menu to choose from as shown in Figure 3.6

The *Recall it* and *Pick it* options work the same way as with the original version of ScriptEase. The *Recall it* option presents a hierarchical menu of known objects. The *Pick it*

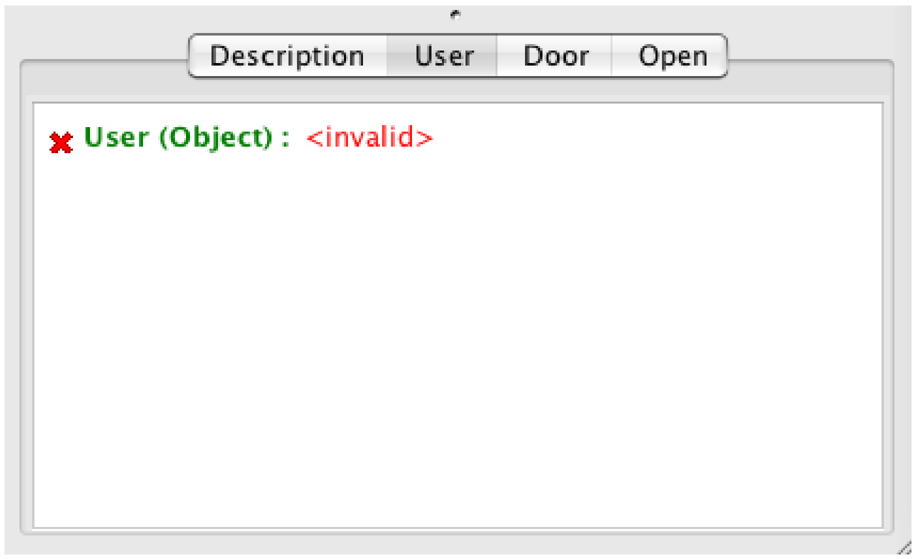


Figure 3.5: The initial describer panel.

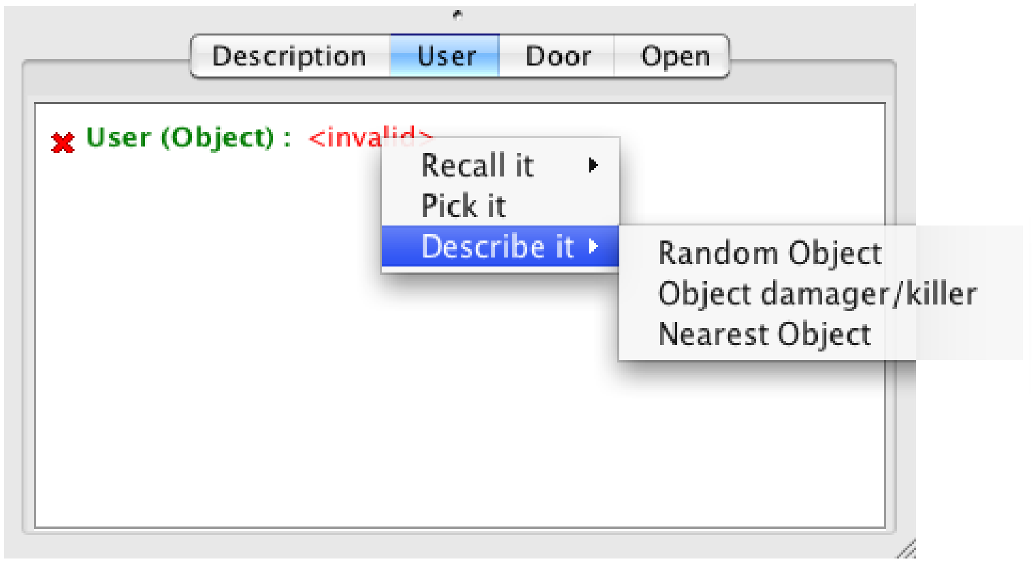
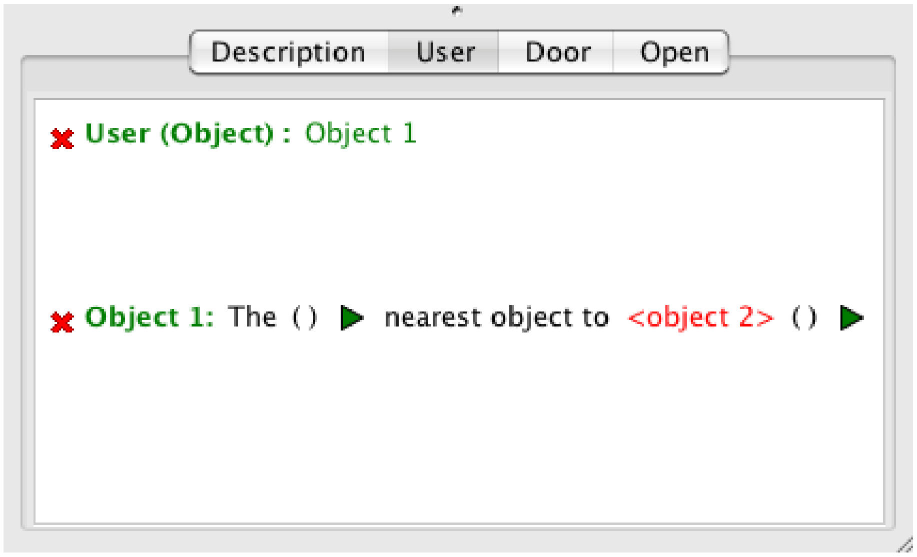


Figure 3.6: The menu an author sees when choosing to set an option.

option opens the picker dialog. However, suppose the author wants to select an object that cannot be recalled or picked.

For example, suppose the author wants to describe the option as the nearest object to *Grumpy Dwarf*. The author would choose to *Describe it* and select the *Nearest Object* category from the menu. The panel would be updated and now look like Figure 3.7. The line *The () nearest object to <object 2> ()* is known as a *describer line*. Every describer line is preceded by a label (in this example *Object 1*), which is used to indicate how/where the line fits within the description. A single description can consist of multiple describer lines.



User (Object) : Object 1

Object 1: The ( ) nearest object to <object 2> ( )

Figure 3.7: Updated panel after the author chose to *Describe it* with the category *nearest object*. The first line is the option line while the second is an example of a describer line.

At this point there are five things to note within Figure 3.7. The following five subsections will discuss what each of these means to the author. First, that the describer line starts with a label, *Object 1*. Second, the use of colored text: the label is green and the describer line option, *<object 2>*, is red. Third, that the text on the first line, the option line, (*<invalid>*) has changed to *Object 1* which is the same as the label. Fourth, within the describer lines there are sets of parentheses followed by a green arrow. And, fifth, that both lines begin with a red x.

3.2.1 Describer Labels

Each describer line the author creates is given a default unique label. The labels are created based on the type of object being described by the line. A number is added to make the label unique. The labels are used to link where a describer line fits within the overall description. An option that has been set by describing is updated with the label of the describer line it created. For example, the label *Object 1* was updated in the option line when the author chose the description that transformed Figure 3.6 to Figure 3.7.

The default labels are not particularly informative to the author. However, the author can change the labels to something more meaningful. In the above example the author chose to use the *Nearest Object* category, so the author may want to change the label from *Object 1* to *Nearest Object*. If the author clicks on the label at the start of a describer line, a single menu option appears, *Change line label*, as shown in Figure 3.8. Once selected, Figure 3.9 shows the popup window that appears. Here the author enters the new label. Finally, once the author clicks OK, the label on the main panel is updated with the new label, as shown in Figure 3.10. The label is updated in two locations within the main panel: the start of the describer line and the option line.

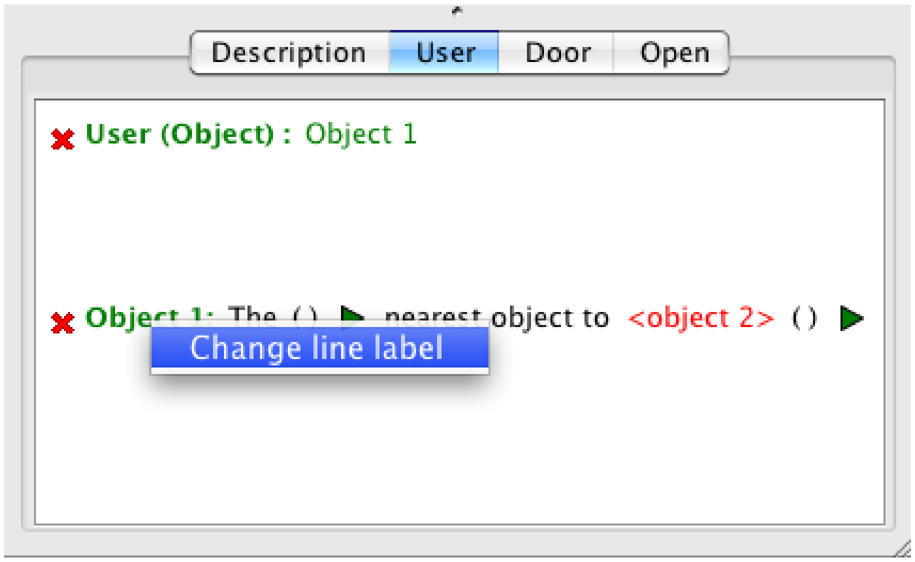


Figure 3.8: Menu to change a describer line's label.

Labels also have a secondary use that is not as apparent as matching a describer line to an option. The label of each describer line is added to the known objects list so the author can reuse each describer line.

3.2.2 Text Colors

Text colors have always been used in ScriptEase to highlight information. Specifically, colored text represented options the author can set. The different colors were used to represent how an option was set, such as by a definition, selected from the known objects list, chosen

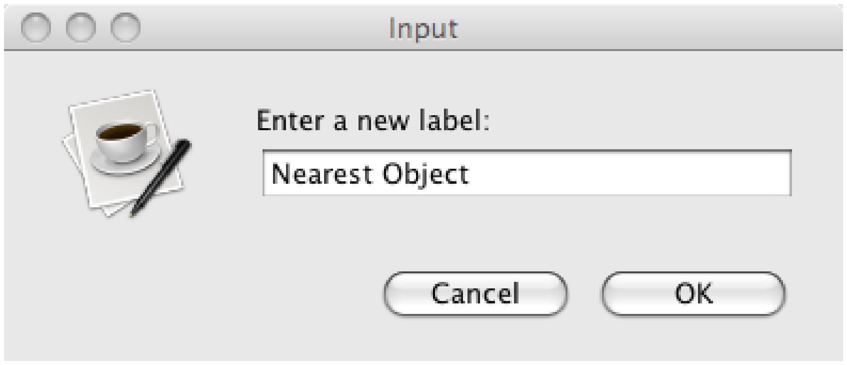
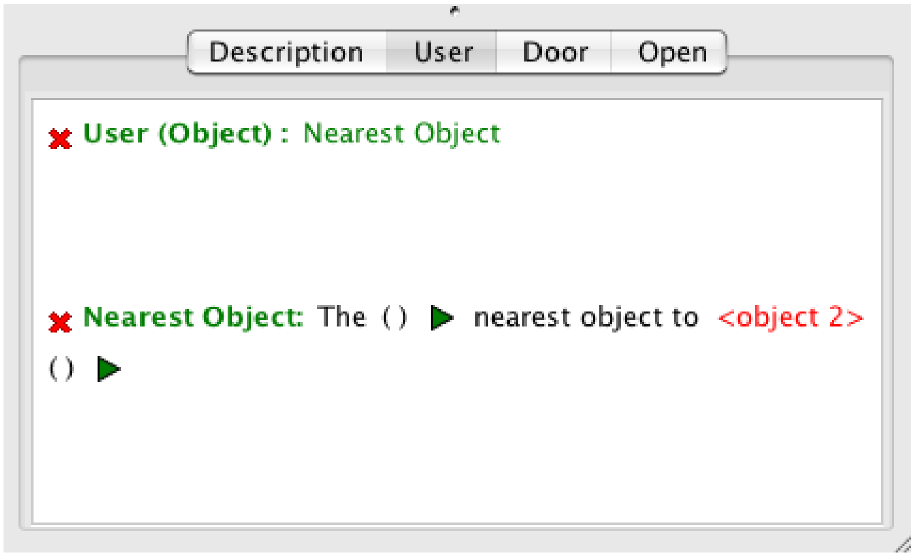


Figure 3.9: Popup window allowing the author to change a describer line's label.



User (Object) : Nearest Object

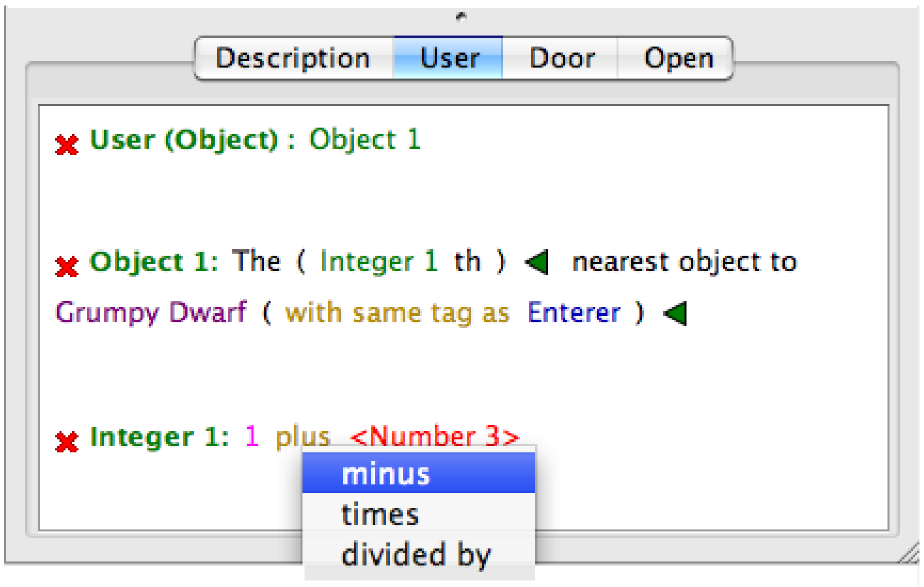
Nearest Object: The ( ) ▶ nearest object to <object 2>
( ) ▶

Figure 3.10: Example with new describer line label *Nearest Object*.

by using the picker, or selected from an enumeration of constants.

In the *Describer*, colors are used to identify not only how an option was set, but also to inform the author how to interact with the describer line. The following list of colors explains the meaning of each color, while Figure 3.11 shows an example set of describer lines displaying all the colors.

- Red - an option that needs to be set.
- Orange - a choice point in the describer line. When clicked on, the author sees a small set of choices that change the meaning of the describer line.
- Green - a label for a describer line. Green text within a describer line indicates that an option was set by choosing to *Describe it* and guides the author to the label of the created describer line that follows.
- Blue - an option set by choosing from the *Recall it* list.
- Pink - an option set by choosing from the *Pick it* list, where the picker either allowed the author to manually set a value (number or string) or choose from an enumeration (like from a list of effects or the state of a door).
- Purple - an option set by choosing from the *Pick it* list, where the author selected an actual static module object (a door, creature, trigger, etc).



User (Object) : Object 1

Object 1: The (Integer 1 th) nearest object to Grumpy Dwarf (with same tag as Enterer)

Integer 1: 1 plus <Number 3>

Figure 3.11: Example describer lines displaying the various text colors used. The displayed menu lists the alternative choices to *plus*.

3.2.3 Setting Options / Linking Describer Lines

Describer lines may themselves contain one or more options that the author needs to set. To set an option in a describer line the author is shown a similar menu to the one presented when the author first chose to set the action option: *Recall it*, *Pick it*, and *Describe it*. If the author chooses to *Recall it* or *Pick it* the result is inserted into the describer line, replacing the option's label and updating the option's text color. An example of this can be seen in Figure 3.11 where *Grumpy Dwarf* was chosen using *Pick it* and *Enterer* was chosen from *Recall it*.

Should the author choose instead to *Describe it* a submenu is displayed showing a list of categories. The category choices may be different from when the author chose the initial describer line as the list is based on the type of the option in the describer line rather than the type of the option in the original action. Once a choice is made, a new describer line is added to the panel and the option's text is changed to green and updated to the label of the new describer line.

Figure 3.11 contains two describer lines with the labels: *Nearest Object* and *Integer 1*. The first label, *Nearest Object* is also found on the option line while the second label, *Integer 1* is repeated within the first describer line. The matching text is used to indicate the connection between two pieces of information. The *Nearest Object* label found in the option line indicates the object to which the original action option was set, in this case a describer line, and it directs the author to the describer line with the matching label, *Nearest Object*. Within the first describer line, the author decided to further describe *Integer 1* as specified by the green text. Once again, the label text, *Integer 1*, directs the author to the matching describer line.

To understand the description presented in Figure 3.11, the author starts by reading from the first describer line. Whenever the author reaches a green label, they move to the describer line indicated by the label's text and continue reading from there. After reaching the end of a describer line, the author returns back to the label in the previous describer line and continues reading. For example, the description in Figure 3.11 can be read: *The (5 plus <Number 5> th) nearest object to Grumpy Dwarf with same tag as Enterer*.

3.2.4 Arrows

Each describer line is actually a representative of multiple plausible definitions. The author can access the various potential definitions by modifying the describer line. One way the author can modify a describer line is by adding optional phrases which are represented by a pair of parentheses and a green arrow. The green arrows provide the mechanism for adding a phrase. A right pointing arrow indicates that there is an option phrase that can be inserted in the parentheses at the arrow's location. Figure 3.12 shows an example menu for expanding a describer line. Once an optional phrase has been added, the green arrow changes to point to the left. A left pointing arrow follows an existing optional phrase and can be used to either remove the current phrase or to change it, as shown in Figure 3.13.

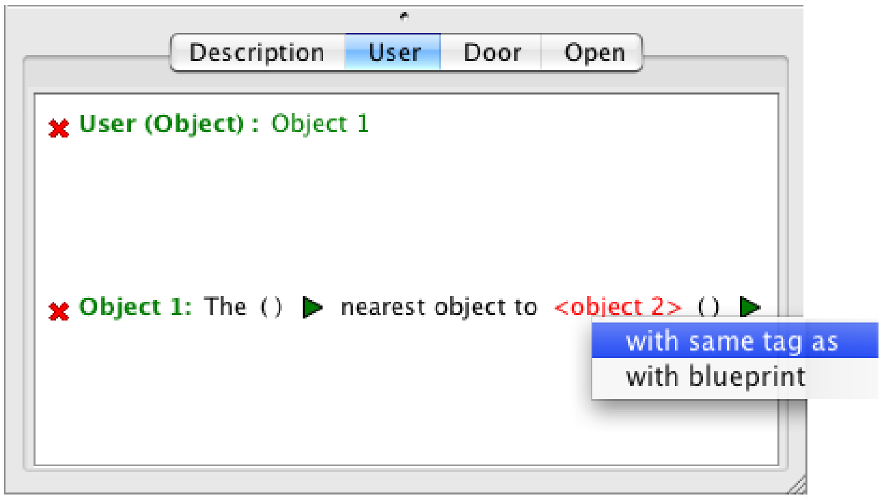


Figure 3.12: The menu for adding an option segment.

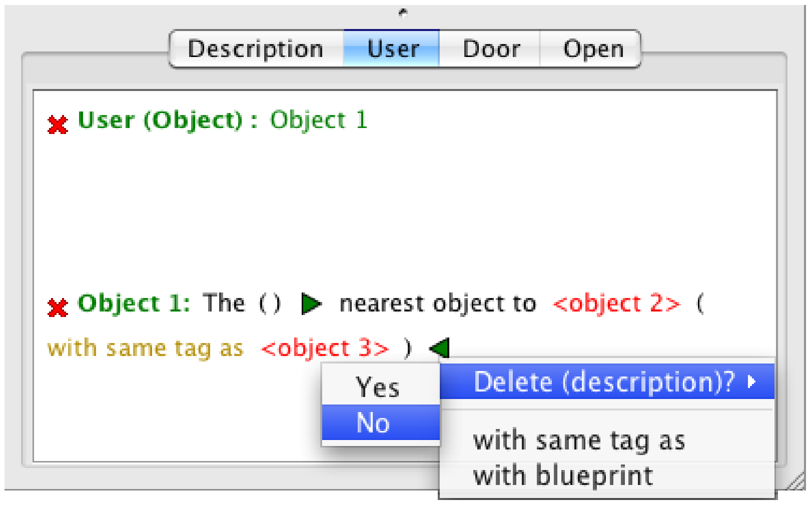
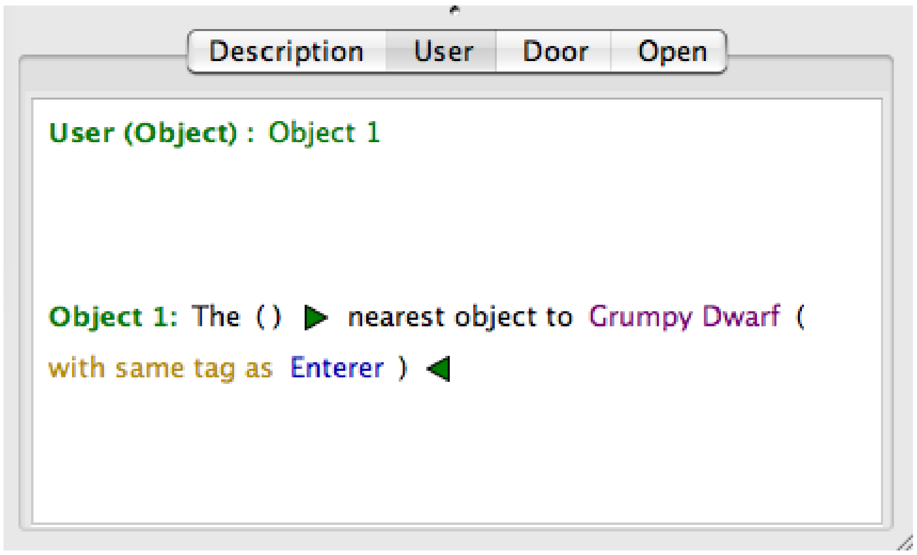


Figure 3.13: The menu is displayed by selecting a left pointing arrow, allowing the author to remove or change the optional segment.

3.2.5 Red X's

ScriptEase uses a small red x located on the left hand side of a line to indicate that a pattern is incomplete. I have extended this convention to apply to the describer panel where a red x indicates an incomplete describer line. Examples of red x's can been seen in most figures within this section including Figure 3.13. A describer line is incomplete if there remains a visible option that is not set, or if a describer line that was created to set an option within the current line is incomplete. Once all sections of the description are complete, all red x's are removed and the author knows the description is complete. An example of a complete description is shown in Figure 3.14.



User (Object) : Object 1

Object 1: The ( ) ▶ nearest object to Grumpy Dwarf (with same tag as Enterer) ◀

Figure 3.14: An example of a complete description.

3.3 Knowledge representation - Graphs / Categories

In order to allow the author to select an initial definition and then modify it for context, the way the definition knowledge was represented within ScriptEase needed to change. Definitions with similar information needed to be found and the similarities noted and a new method for displaying and storing this information needed to be created. For example, there are five definitions that return a nearest creature in ScriptEase. To display the single describer line shown in Figure 3.14 requires all five of these similar definitions to be combined into a single template.

3.3.1 Similarities

The describer uses two techniques to reduce the amount of information the author has to view to set an option. The first technique was to show only definitions whose type matches the option's type. This can either be a direct match (*doors* only show *doors*) or the definition type can be a subtype of the option type (*doors* are a subtype of *objects*). Since a container is either a *placeable* or a *creature*, a container option can use a placeable or creature description. The second technique was to replace a set of similar definitions with a single template. The first step in identifying the templates is to sort the definitions into categories based on similarities of function. For example, look at the following list of definitions. I have modified the definition names slightly from the original ScriptEase terminology to highlight the similarities.

- Find different random creature with same tag as a specific creature
- Find random creature with same tag as a specific creature
- Find nearest creature to an object with same tag as another creature
- Find nearest creature to an object
- Find n-th nearest creature to an object
- Find n-th nearest creature with same tag as another creature
- Find different nearest creature to an object

There are two definitions containing *n-th*, two containing *different*, five with *nearest creature*, two with *random* and four containing *with same tag as*. One way to group these definitions is to construct two categories: nearest creatures and random creatures. In fact, by simply looking at this list it is possible to find other definitions that would make sense but are not listed, such as *Find random creature* and *Find different random creature*. This allows a designer to find feasible definitions that otherwise may have been missed. Within a group of sentences it is easy to note the areas that are similar and where the differences are. For example, within the nearest creatures group all lines contain *Find* and *nearest creature*, while only some of the sentences contain *n-th*, *different* and *with same tag as another creature*. Within those differences it can be further noted that *n-th* and *different* are never found in the same sentence, but both can be paired with *with same tag as another creature*. This kind of analysis allows us to identify a single abstract description that spans a large number of existing definitions and potentially missing definitions.

3.3.2 Graphs

By noting the similarities between groups of definitions it was possible to turn each group into an abstract description represented by an acyclic directed graph. Two example graphs representing the definitions presented in the last section can be seen in Figures 3.15 (nearest creatures) and 3.16 (random creatures). The black nodes in the graph represent text, the red nodes are options that need to be set by the author, the green dotted nodes are optional nodes and the orange nodes (not shown) are choice nodes.

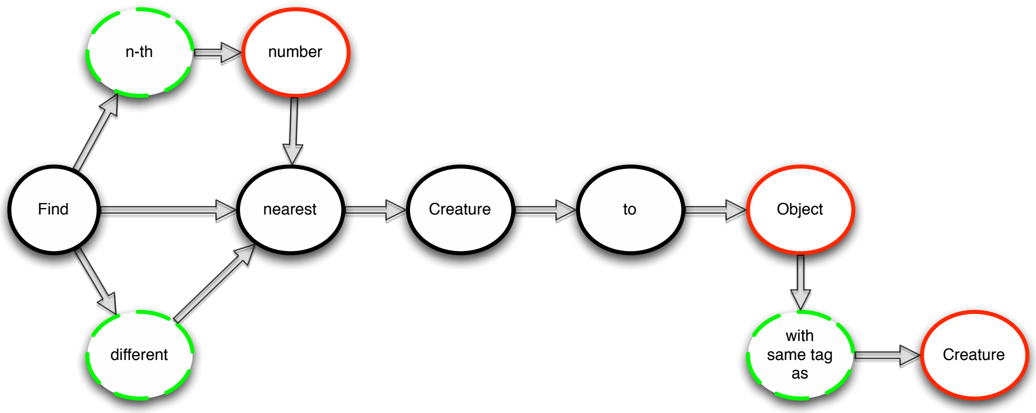


Figure 3.15: Tree of creature nearest definitions

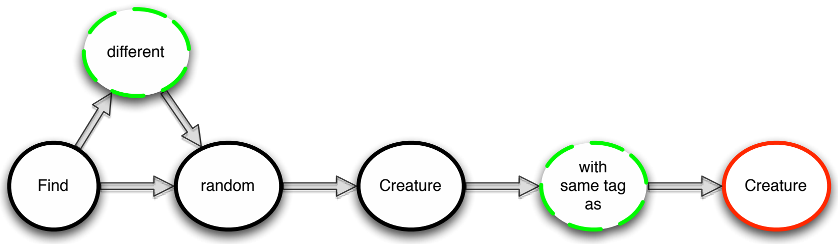


Figure 3.16: Tree of creature random definitions

This abstraction of definitions greatly reduces the amount of information visible to the author. Each graph/group was given a name. When the author decides to use *Describe it* to set an option, a description category must be selected. The category the author chooses has a direct mapping to a single graph. Once a category is chosen, the author is given a default description, also known as the initial describer line. This line is created by beginning at the start node of the graph. Each graph has one start node which is the only node that has no edges entering it. The describer line is generated by starting with the start node and repeatedly moving to one of the next nodes that is either black or red. The line finishes when there are either no more nodes to move to, or the only remaining nodes are green. If there is more than one node to choose from (see Figure 3.17), the generator selects the first node in the list created by the describer pattern designer. In the case of Figure 3.15, the line would be *Find nearest creature to object*, and in Figure 3.17 the initial line would be *The creature who entered the module module*.

The optional points in the graph, the green nodes, indicate where in the describer line the green arrows should appear and what menu choices the author will have. In Figure 3.15 the first green arrow would represent two choices for the author: *n-th* or *different*.

The red nodes are the options that the author will need to set. The text within the nodes indicates the type of the object that the node represents.

3.3.3 Choice Nodes

Some graphs contain choice nodes, which are locations in the graph where there are two or more valid paths from a node and exactly one path must be chosen. The Enterer/Exiter graph in Figure 3.17 is an example where the author must decide whether to select *enterer* or *exiter*. A second choice node occurs in this graph where the author must choose between *module*, *area*, and *perimeter*. The nodes where these choices exist show up in the describer line as orange text, as stated in section 3.2.2 on text colors. Figure 3.18 displays an example view of this graph as a describer line, with the menu for changing the second choice node from *module* to *area* or *perimeter* displayed.

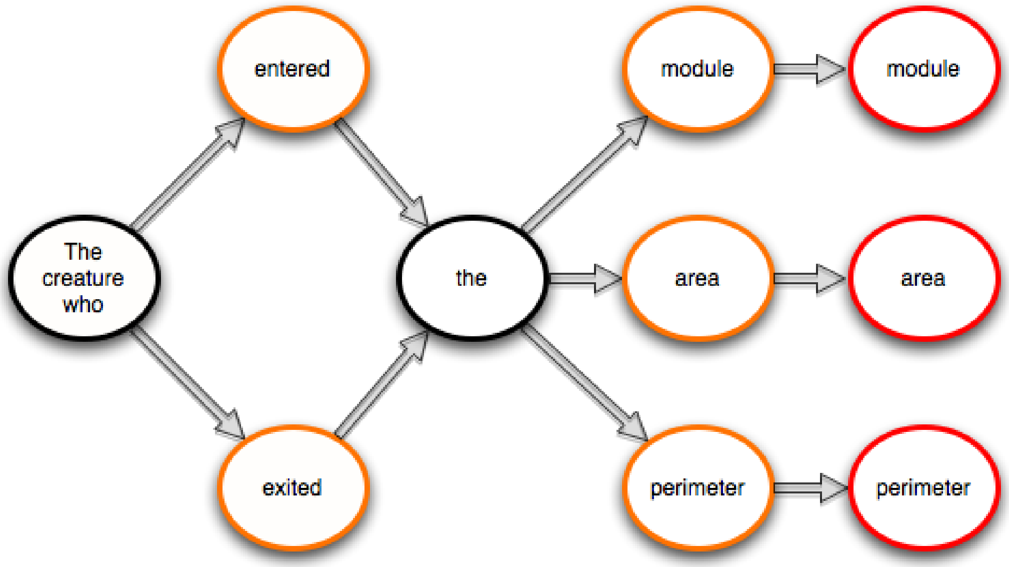
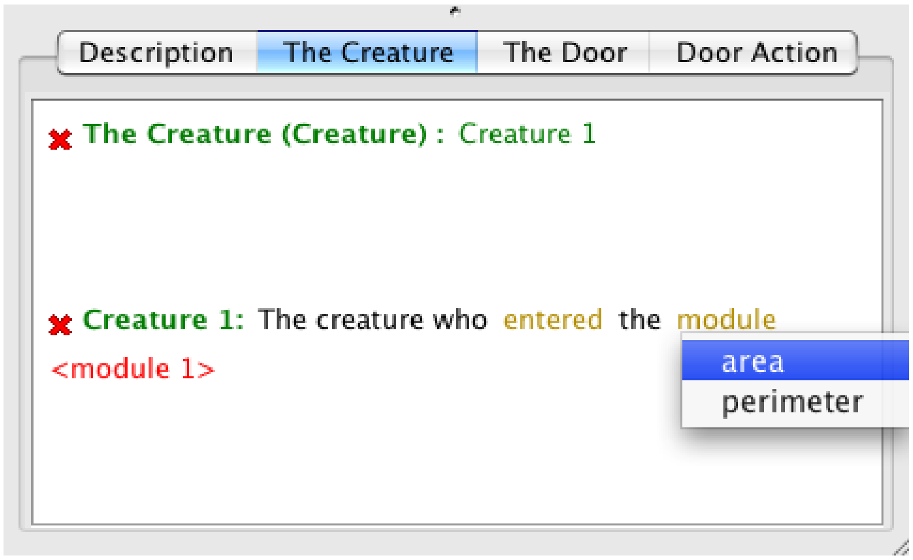


Figure 3.17: Tree of enterer/exiter definitions



The Creature (Creature) : Creature 1

Creature 1: The creature who entered the module <module 1>

Figure 3.18: Describer line representing enterer/exiter graph. The menu shows the options for changing module.

3.3.4 Categories

Grouping definitions together based first on definition type and then on similarities allowed us to reduce the total count of definitions without reducing the number of expressible definitions. In fact, the process of creating these graphs can discover definitions that were initially missed. The graphs also allow new definitions to be included as alternative paths through existing description graphs, without substantially increasing the information the author sees.

Each graph is given a name to describe the overall context of the graph. The names for the three graphs presented in this section are *nearest creature*, *random creature*, and *enterer/exiter*. These names are presented to the author when the author chooses to *describe it*. By referring to a description name the author can quickly find the appropriate category and customize the generated describer line to suit the story.

3.4 Using Describers to solve the Definition problems

Now that we understand how the *Describers* work, let's recall the original three problems with definitions and examine how *Describers* can be used to solve them.

3.4.1 Content Overload

In section 3.1.1 I showed the volume of information the author is exposed to every time they need to create a new definition. Since the definitions were created separately from where the options are set, ScriptEase had no information as to the required option type and, therefore, ScriptEase had to show all definitions. With *Describers*, ScriptEase now has the ability to display only the describer lines (definitions) that match the correct option type. We took this one step further by using graphs to represent data as discussed in section 3.3, allowing us to combine multiple definitions into a single description line. This representation means that the authors need only see a list of categories, each one representing a single graph, instead of all the individual definitions. The ability to modify a describer line allows an author to express all the definitions. For example, there are forty-one definitions for type *creature*. These can be grouped into 10 graphs.

Option Types

As mentioned in section 3.4.1, ScriptEase can use *Describers* to access the option type when the author decides to use *Describe it*. This allows the describer lines to be filtered based on the option type. This knowledge continues when the author decides to set one of the describer line options with a description of its own. The choices for the new describer line are based on the current option the author is trying to set. This allows the author to focus on definitions relevant to what they are working on instead of being overwhelmed by all the possible definitions.

Similar Definitions

Describers work by grouping similar definitions together into a single category. The author then chooses a category to work from and can modify the describer line as needed. Since all similar definitions have been grouped together the author does not need to worry about singling out the definition they want.

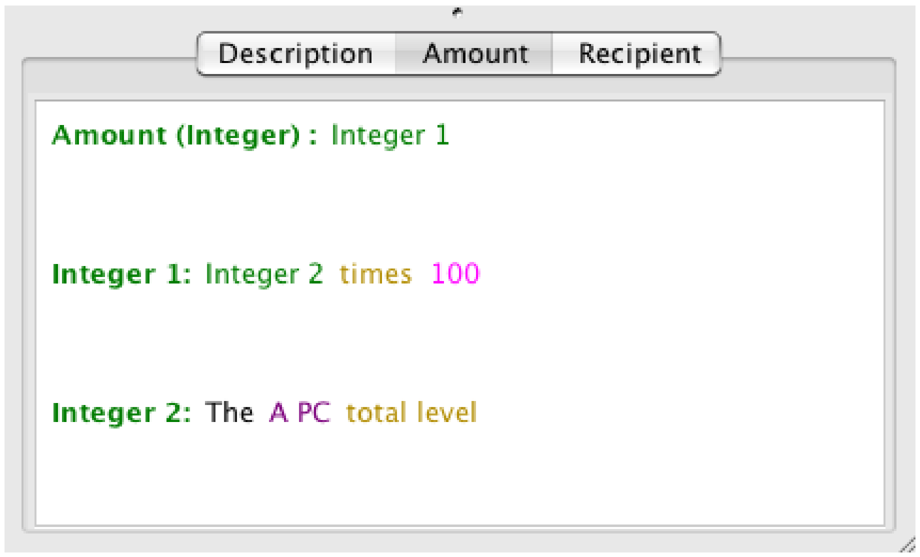
3.4.2 Multiple Workspaces

With definitions, if the author can't use *Recall it* or *Pick it* to set an option in the option panel, the author must add a definition to the definition block in the encounter panel. With *Describers*, the author can use *Describe it* to set an option in the option's panel. In addition, if a describer line also needs to have an option set, its new description follows the describer line in the option panel. The author can set any options, modify them, or replace them with a new description. Since all of this is done within the same option panel, unlike with definitions, the author always knows the context in which they are working.

3.4.3 Linking Definitions

Definitions required the author to use forward chaining where the author starts with the entire game state and must construct a series of definitions that maps this game state to the option required. *Describers* instead require backward chaining, where the author works from the overall goal towards the small details (current game state).

Let's use the same example as in the earlier section 3.1.3. The author wants to determine how much gold to award a PC for completing a task. The PC will be awarded an amount of gold that is equal to the PC's level times one hundred. In the *Describer* the author starts by choosing to *Describe it* and selects the category *Math - Add, Subtract, Multiply, Divide, Negate*. The panel now shows the initial describer line <*number 1*> *plus* <*number 2*>. The author would then need to click on *plus* and change it to *multiply*. Then the author would set <*number 1*> by choosing to *Describe it* and the category *Creature's Statistics*. The new describer line would appear below the first: *The* <*creature 1*> *gold*. In this new describer line the author needs to set the creature to the *PC* and change *gold* to *total level*. Finally, the author needs to set <*number 2*> by choosing to pick it and entering 100. Figure 3.19 shows the final result.



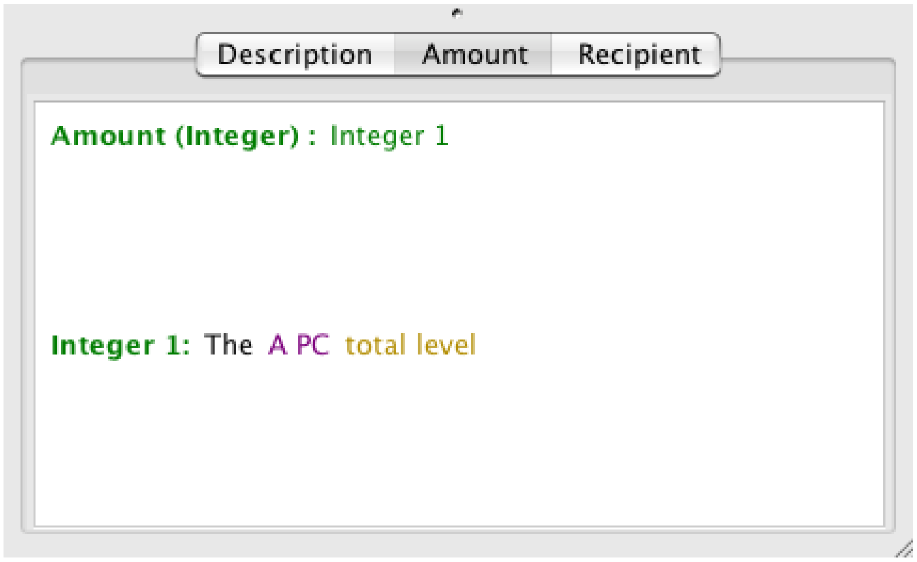
Amount (Integer) : Integer 1

Integer 1: Integer 2 times 100

Integer 2: The A PC total level

Figure 3.19: Describer displaying the complete description for awarding gold to a PC that is equal to the PC's level times 100.

This design permits the author to work in only one direction, thereby preventing the author from creating definitions before they are needed. In the example, suppose the author had decided to start by describing the PC's level (which is valid since level matches the option type, number) first. Once the author had set the parameters in the resulting describer line, the description would be complete. The author would be unable to include the describer line that multiplies the two numbers together as shown in Figure 3.20. By being unable to continue, direct feedback is given to the author that the description being created cannot be completed with the initial describer line chosen.



Amount (Integer) : Integer 1

Integer 1: The A PC total level

Figure 3.20: Describing the level to early.

The author may need to link multiple describer lines together, as in the example of computing the reward amount, to create the complete description they have in mind. When setting an option within a describer line the author always has the choice to *Describe it* or *Recall it* or *Pick it*. This exists for all options within any describer line. If the author chooses to *Describe it* a new describer line is created. Because all of the work is done within the same workspace there is an immediate link between each new describer line and the option that created it. This means the author does not need to create the description and/or describer lines and then link them back to the option being set.

Once we had a working prototype of the *Describers* it was time to compare them with definitions. The next chapter discusses the results of the user study we conducted comparing the two tools.

Chapter 4

Study Evaluation

In order to evaluate the *Describer*, I ran a user study in which the participants used a modified version of ScriptEase containing *Describers* and the original version of ScriptEase containing definitions. For the study, we referred to the two versions of ScriptEase as the *Describer* and the *Definer*. The goal of the study was to determine which method the participants found easier to use, preferred over all, and which method was more efficient.

4.1 The User Study

The user study was run at the University of Alberta in the Department of Computing Science. This study received ethics approval from the Arts, Science and Law Research Ethics Board under number 2040. The study lasted for one hour. All participants used both methods: the *Describer* and the *Definer*. Each participant was given two different test scenarios and used each method on a different scenario. Each scenario consisted of five statements (ScriptEase actions), with the first statement including a walkthrough on how to use the specific method required for that scenario.

The object of the study was to compare the performance of the participants using the two different methods. Were they able to complete more statements using one method than the other? Did they prefer one method? Was there a transfer of knowledge (learning) between the first method they used and the second? Was there a difference between female and male participants on tool preference?

4.1.1 The Participants

The participants in this study were from the University of Alberta's Psychology 104/105 classes in the Winter 2009 semester. The study's participants were undergraduate university students from 18 to 22 years of age (mean and median of 19). In exchange for participating in the study, the students received course credit. Unfortunately, not all of the participants were able to successfully complete the first statement using both methods. Since the first statement was a walkthrough, data from these students was eliminated from the study. In addition, some participants failed to record all the information required in the study or had technical problems with their data files. After screening out these participants, there were 49 complete data records from the original 83 participants.

Of the 49 participants there were 33 females and 16 males. Only four participants had any prior experience with programming and 33 of the 49 played video games a minimum of once a month. Ninety percent used the computer daily, while the other 10% used it several times a week. The participants ranged from 1st to 3rd year students (mean: 1.7, median: 1).

4.2 Test Scenarios

There were two different test scenarios designed for this experiment designated A and B and they both consisted of five statements. Each statement described an action such as: *The chest spawns a penguin near the block of ice that is x closest to the pc. X is determined by a roll of a 4 sided dice*. In order to complete each statement, the participants would open a specified module in ScriptEase. The module contained an action that represented the statement. The participants were required to set all of the options for the action to make it match the statement description. The first statement contained a walkthrough that introduced one of the two methods to the participants. This walkthrough showed the participants exactly how to set each option of the action to match the statement, using one method, either the *Describer* or the *Definer*.

I wanted the participants to use a different set of statements for each method so that the mechanism they used to set an option using the first method would not bias the way they tried to set the same option using the second method. However, I also didn't want to make any method inherently easier by providing a set of statements specific to that method. To prevent this, I created two different scenarios but had two groups use scenario A with the *Describer* and scenario B with the *Definer* while the other two groups used scenario A with the *Definer* and B with the *Describer*, resulting in the four different groups of participants. Two of the groups used the *Definer* first and two of the groups used the *Describer* first to prevent one method from having an advantage over the other. This ordering created the four groupings listed in Table 4.1. The table also includes the number of participants in each group.

Group Number	1st Method	1st Scenario	2nd Method	2nd Scenario	Participants
1	Definer	B	Describer	A	11
2	Describer	A	Definer	B	14
3	Describer	B	Definer	A	12
4	Definer	A	Describer	B	12

Table 4.1: The four different user study groups.

This approach determined that I needed four test scenario booklets, two for each scenario with the walkthrough individualized for each method. I kept the first statement in both scenarios identical in order to keep the introduction to the two methods consistent. Two of the four test scenario booklets can be found in Appendix A representing both scenarios and methods (Describer A and Definer B).

Each action statement was saved in its own module file using a name such as 32ADesc4.mod where the 32 was replaced by a participant number, A referred to the test scenario (A or B), Desc referred to the method (Desc for *Describer* or Defn for *Definer*), and 4 referred to the target statement, from 1 to 5. After completing a statement the participants were instructed to save their module and open the next one.

4.2.1 Experimental Setup

Upon arrival, each participant was randomly assigned to one of the four groups. This assignment was done by having the students fill the room starting from the front to the back. The computers had been set up ahead of time such that the four groups were cycled.

The study started with a brief introduction to the participants and the participants were asked to sign a participant consent form. They were informed of their participant number and reminded to use it on all of the forms they completed. I then told them that it was important to record the time they finished each statement. The participants were instructed to start with the top test scenario booklet (which was also numbered) and that they would have twenty minutes to work on the statements. They were told to focus on correctness over quantity. At this point they were instructed to start.

After the first twenty minutes, the participants were told to stop and save their work. An assistant and myself then switched each computer to the alternate method and opened up the next module for them. The participants were again reminded to record the time for each statement and given 20 minutes to work on the second method. The first statement of the second method was also a walkthrough.

At the end of the second twenty minutes, the participants were once again instructed to stop and save their work. I then had them complete the survey found in Appendix B. The survey was designed to collect three types of information. First, demographic information about the participant including age, gender, year of study, experience with video games and use of the computer. Second, participant preferences between the two methods based on ease of use, intuitiveness, speed and overall preference. And third, to provide any written comments and feedback.

Group 1	Group 2	Group 3	Group 4
Definer B	Definer B	Definer A	Definer A
1	2	1	1
1	2	1	1
1	2	1	1
1	2	1	1
1	1	1	1
1	1	2	1
3	1	1	1
1	1	1	1
1	2	1	1
1	1	3	1
1	1	1	1
	2	1	1
	3		
	1		
Describer A	Describer A	Describer B	Describer B
2	1	2	3
2	1	2	1
2	1	2	3
2	2	3	1
3	1	3	3
2	1	3	4
3	1	1	5
2	3	2	4
1	2	4	2
2	1	3	1
1	1	1	4
	2	2	4
	2		
	1		

Table 4.2: Number of completed statements.

Once they finished the survey they received a debriefing document (found in Appendix C) that explained the purpose of the study and how to contact me if they had any further questions.

4.3 Module Results

To interpret the module results of the user study I ran a 3-factor ANOVA test, using the R statistical analysis software [18], to compare the main effects and interaction effects of the order, method and scenario on the number of statements the participants were able to complete. Table 4.2 contains the raw data and Table 4.3 presents the results of the test. The immediate conclusions that can be made from the table results are that all three factors produce statistically significant main effects on the number of statements the participants were able to complete. These results indicate that students were able to complete more statements using the *Describer* method than the *Definer* method. The results also indicate that students completed more statements using the second method used (order) than the first method which shows that learning took place from one method to the other. Finally, students were able to complete more statements for scenario B than for scenario A.

	Df	Sum Sq	Mean Sq	F value	Pr(>F)
Order	1	4.500	4.500	7.7398	0.0065803
Method	1	20.088	20.088	34.5502	6.933e-08
Scenario	1	8.295	8.295	14.2678	0.0002843
Order:Method	1	0.535	0.535	0.9195	0.3401833
Order:Scenario	1	0.008	0.008	0.0142	0.9055538
Method:Scenario	1	2.641	2.641	4.5419	0.0358010
Order:Method:Scenario	1	0.025	0.025	0.0426	0.8368916
Residuals	90	52.327	0.581		

Table 4.3: A 3-Factor ANOVA test comparing the interaction effects of the order, scenario and method on the number of completed statements.

The other conclusion from these results is that there is a significant interaction effect between the method and scenario. This interaction effect is clear in Figure 4.1. The participants were able to complete almost an equal number of statements on the *Definer* for both scenarios. However, the participants able to complete considerably more on scenario B than scenario A when using the *Describer*.

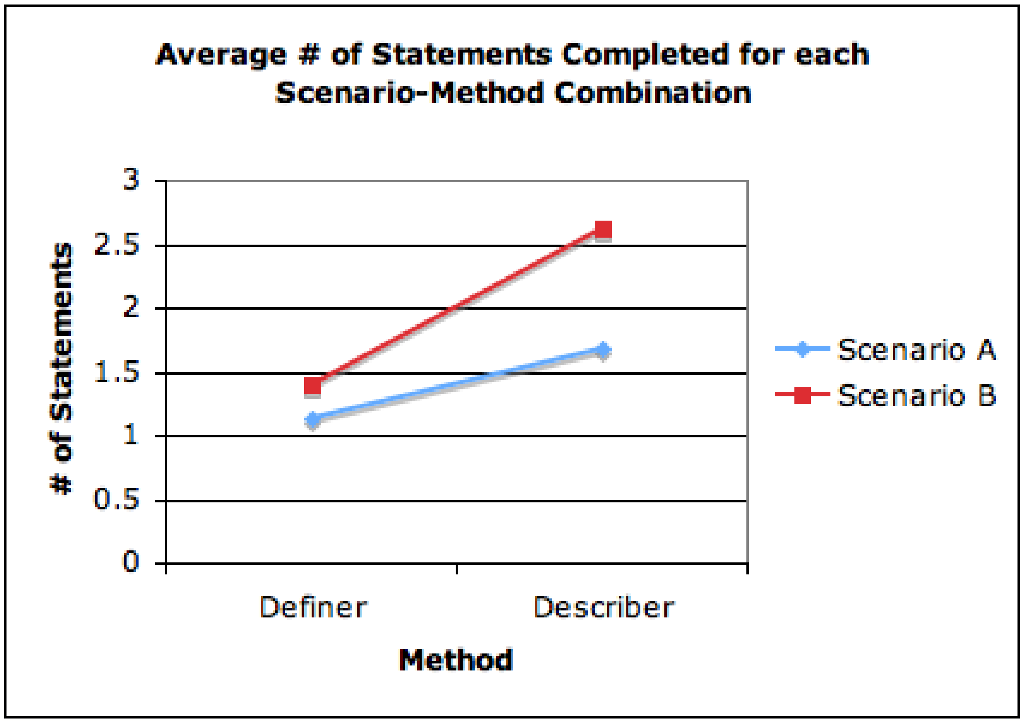


Figure 4.1: Graph showing the average number of statements participants were able to complete on each method by scenario.

4.3.1 Time

As part of the study, the participants were asked to record the time they started each method, and then the time they finished each statement. Because few participants were able to complete more than one statement using the *Definer*, I only analyzed the time it took participants to complete the first statement using each method. The average times per group are presented in Table 4.4. The four groups produced very similar average times on the first method. The improvement in time was also similar between the groups.

I ran a second ANOVA test to compare the three factors (order, scenario and method) against the time. The results of the ANOVA are shown in Table 4.5. These results show that the order had a statistically significant influence on the amount of time it took the participants to complete the statement. Apparently, students learned enough about setting options in statements when they used the first method to significantly reduce the time it took to complete a similar statement using the second method. However, the actual method and scenario had no significant influence.

It is unfortunate that there were not enough participants able to complete more statements using the *Definer*, since comparing the times for the later statements would provide

Group Number	1st Method	1st Scenario	Average Time (min:sec)	2nd Method	2nd Scenario	Average Time (min:sec)
1	Definer	B	10:24	Describer	A	7:22
2	Describer	A	10:17	Definer	B	7:43
3	Describer	B	11:20	Definer	A	6:49
4	Definer	A	10:30	Describer	B	7:27

Table 4.4: The average time by each group on completing the first statement using both methods.

	Df	Sum Sq	Mean Sq	F value	Pr(>F)
Order	1	248.39	248.39	30.5018	3.568e-07
Method	1	0.94	0.94	0.1155	0.7348
Scenario	1	6.56	6.56	0.8057	0.3719
Order:Method	1	0.27	0.27	0.0338	0.8546
Order:Scenario	1	0.01	0.01	0.0007	0.9794
Method:Scenario	1	0.14	0.14	0.0172	0.8960
Order:Method:Scenario	1	5.46	5.46	0.6700	0.4153
Residuals	85	692.19	8.14		

Table 4.5: The average time by each group on completing the first statement using both methods.

more information about how quickly the participants were able to grasp one method versus the other. However, as the participants were able to complete statistically more statements using the *Describer* than using the *Definer*, I can conclude indirectly that the speed with which participants learned to effectively use the *Describer* must have been faster than the speed they learned to use the *Definer*. Direct evidence of this result will require another study.

4.3.2 Participants vs. Themselves

For a final analysis of the results, I looked at how many statements the individual participants completed using the *Describer* and compared it to the number they completed using the *Definer*. The results are presented in Table 4.6. The four participants (8%) who completed more statements using the *Definer* were all part of Group 2. About 55% of the participants completed at least 1 more statement using the *Describer*, and just over 37% completed the same. Of the participants who completed more using one method versus the other, they were almost 7 times more likely to do so using the *Describer* than the *Definer*.

# more statements	-1	0	>0	+1	+2	+3	+4
# of participants	4	18	27	14	7	5	1

Table 4.6: Number of participants who created x many more statements using the *Describer*.

4.4 Survey Results

Part of the study required the participants to fill out a survey providing demographic information along with rating the two methods over four characteristics: Easier, Faster, More Intuitive and Overall Preference. The participants were asked to rank each of the four characteristics on the scale shown in Table 4.7.

Rank	Definer Better	Definer Slightly Better	Same	Describer Slightly Better	Describer Better
Value	1	2	3	4	5

Table 4.7: Rankings used on Survey and the associated numerical value used in evaluating.

The results of the survey are presented in four parts by grouping the participants: everyone, gender, gamers, and programmers.

4.4.1 Everyone

The results of all 49 participants are summarized in Table 4.8. The first five rows show the number of participants who choose each ranking for the four characteristics. The following three rows group the totals of the Better and Slightly Better for each method. The third last row provides the mean which was calculated by assigning each ranking a value from one to five. The associated rank and value can be seen in Table 4.7. A mean of 3 would rate the two methods as equal, while a mean > 3 leans towards the *Describer* and < 3 leans towards the *Definer*. The final 2 rows are the results of a T-Test which was calculated with an expected mean of 3.

For all four characteristics the mean of the results was above 3, in support of the *Describer*. However, only three of the four results produce statistically significant results: Easier, Faster and Overall Preference. The strongest value comes from the Overall Preference characteristic with a mean of 3.71. The fourth characteristic, More Intuitive, produced a p-value of 0.07698, which is not below the 0.05 value needed for 95% confidence.

Survey Answer	Easier	Faster	More Intuitive	Overall Preference
Describer Better	15	18	10	21
Describer Slightly Better	17	12	14	8
Same	7	7	12	11
Definer Slightly Better	0	5	8	3
Definer Better	10	7	5	6
Total Describer	32	30	24	29
Total Definer	10	12	13	9
Total Same	7	7	12	11
Mean	3.55	3.59	3.33	3.71
T-Test t-value	2.6444	2.8726	1.8073	3.5729
T-Test p-value	0.01103	0.006045	0.07698	0.0008159

Table 4.8: Survey results of all participants (everyone).

4.4.2 Gender

The next step I took was to break the results down based on gender. There were 33 female and 16 male participants.

Females

The results of the female population produced even stronger z-values towards the *Describer* then the overall results. As Table 4.9 shows, all four categories produced statistically significant p-values. In fact, the number of participants who preferred the *Definer* went from a range of 17-27% for everyone down to a range of 12-21% for females. Once again, the strongest characteristic was the Overall Preference.

Survey Answer	Easier	Faster	More Intuitive	Overall Preference
Describer Better	10	11	6	14
Describer Slightly Better	12	8	11	5
Same	6	7	9	10
Definer Slightly Better	0	4	5	2
Definer Better	5	3	2	2
Total Describer	22	19	17	19
Total Definer	5	7	7	4
Total Same	6	6	9	10
Mean	3.67	3.61	3.42	3.82
T-Test t-value	2.8611	2.6347	2.1257	3.8018
T-Test p-value	0.007379	0.01287	0.04124	0.0006091

Table 4.9: Survey results of female participants.

Males

There were only 16 male participants and none of the data was able to produce statistically significant confidence values as shown in Table 4.10. The strongest characteristic for the males was Faster, with none of the males ranking the two methods as equal. However, the most interesting result was the characteristic More Intuitive, where the males were spread almost equally across all five ranks. The total results of the rankings do appear to give the *Describer* an edge in Easier, Faster and Overall Preference, but more data is needed to be able to give a conclusive result.

Survey Answer	Easier	Faster	More Intuitive	Overall Preference
Describer Better	5	7	4	7
Describer Slightly Better	5	4	3	3
Same	1	0	3	1
Definer Slightly Better	0	1	3	1
Definer Better	5	4	3	4
Total Describer	10	11	7	10
Total Definer	5	5	6	5
Total Same	1	0	3	1
Mean	3.31	3.56	3.13	3.5
T-Test t-value	0.7346	1.3147	0.3333	1.1677
T-Test p-value	0.4739	0.2084	0.7435	0.2611

Table 4.10: Survey results of male participants.

4.4.3 Gamers/Non-Gamers

I then decomposed the results based on whether or not the participant played games. Those who stated they played at least once a month were categorized as gamers and everyone else as non-gamers. There were 33 gamers and 16 non-gamers.

Gamers

Did gaming experience change the participants' feelings? Table 4.11 would indicate it did not. Once again there were clear statistically significant results in the Easier, Faster and Overall Preference categories.

Non-Gamers

The 16 non-gamers did not produce any statistically significant results as shown in Table 4.12. Interestingly, the strongest characteristic was Easier and was borderline confident.

Survey Answer	Easier	Faster	More Intuitive	Overall Preference
Describer Better	11	13	7	15
Describer Slightly Better	13	7	10	7
Same	1	5	7	5
Definer Slightly Better	0	4	6	3
Definer Better	8	4	3	3
Total Describer	24	20	17	22
Total Definer	8	8	9	6
Total Same	1	5	7	5
Mean	3.58	3.64	3.36	3.85
T-Test t-value	2.1177	2.5525	1.6445	3.6129
T-Test p-value	0.04207	0.01567	0.1099	0.001025

Table 4.11: Survey results of participants who play video games at least once a month.

The two lowest results were More Intuitive and Overall Preference which are also the results where the total number of participants who preferred the *Describer* was less than half. The Easier and Faster results do appear to lean towards the *Describer* but more data is needed to provide any conclusive results. It is important to note that non-gamers are not a target group of ScriptEase, as it is highly unlikely for a non-gamer to be involved with designing a video game. However, the results from non-gamers can still provide interesting insights as they will present a different view.

Survey Answer	Easier	Faster	More Intuitive	Overall Preference
Describer Better	4	5	3	6
Describer Slightly Better	4	5	4	1
Same	6	2	5	6
Definer Slightly Better	0	1	2	0
Definer Better	2	3	2	3
Total Describer	8	10	7	7
Total Definer	2	4	4	3
Total Same	6	2	5	6
Mean	3.50	3.50	3.25	3.44
T-Test t-value	1.5811	1.3284	0.7746	1.1634
T-Test p-value	0.1347	0.2039	0.4506	0.2628

Table 4.12: Survey results of participants who do not play video games.

4.4.4 Programmers/Non-Programmers

Finally I looked at the results based on prior programming experience.

Programmers

There were 4 participants who had any programming experience. Because of the low number of programmers, I am unable to make any claims about the two methods or run a T-Test. I have presented the data in Table 4.13, which shows that the four participants all leaned heavily towards the *Describer*. More participants are needed before the results can be analyzed. I expected that the programmers might see no difference between the two methods because they are two ways of accomplishing the same task. For programmers, both methods can be viewed as a way to set function arguments.

Survey Answer	Easier	Faster	More Intuitive	Overall Preference
Describer Better	2	2	0	4
Describer Slightly Better	2	2	3	0
Same	0	0	0	0
Definer Slightly Better	0	0	1	0
Definer Better	0	0	0	0
Total Describer	4	4	3	4
Total Definer	0	0	1	0
Total Same	0	0	0	0

Table 4.13: Survey results of participants with programming experience.

Non-Programmers

Of the 49 participants, 45 had no programming experience. The results for the non-programmers are shown in Table 4.14 and are very similar to the everyone group results.

Survey Answer	Easier	Faster	More Intuitive	Overall Preference
Describer Better	13	16	10	17
Describer Slightly Better	15	10	11	8
Same	7	7	12	11
Definer Slightly Better	0	5	7	3
Definer Better	10	7	5	6
Total Describer	28	26	21	25
Total Definer	10	12	12	9
Total Same	7	7	12	11
Mean	3.47	3.51	3.31	3.6
T-Test t-value	2.1062	2.3304	1.6132	2.8657
T-Test p-value	0.04092	0.02443	0.1139	0.00636

Table 4.14: Survey results of participants who do not have programming experience.

4.5 More Data

The study provided conclusive results in almost all aspects. However, due to the inconclusive results from the survey regarding intuitiveness, the lack of male participants and the lack of programmers, I intend to add data to this study by repeating the experiment with more participants. Adding participants should lower the variance. Obtaining more programmers may involve finding participants from a wider pool.

Chapter 5

Future Work and Conclusion

In this chapter I present a summary of the work presented in this thesis followed by some suggested future work. I then finish with a few concluding remarks.

5.1 Summary

This dissertation discusses the development of a new technique, *Describers*, for setting options in ScriptEase. *Describers* were created to solve the three problems identified with the current technique of using definitions.

1. **Content overload** - all definitions are available at all times which means the author needs to be aware of the option type and the definition's return type. The *Describer* limits the amount of information the author sees in two ways, by only displaying descriptions that match the option type and grouping similar descriptions.

   Option types - descriptions are filtered such that only those that match the type of the option being set are accessible for setting an option.

   Similar definitions - similar descriptions are grouped together to form acyclic directed graphs so that an author can choose a category (mapped to a single graph) and then, by modifying the resulting statement, can access the various descriptions.
2. **Multiple workspaces** - instead of the author needing to leave the option panel to create one or more definitions, all the work is done in the option panel which allows the author to work in the context of the option being set.
3. **Linking definitions** - linking multiple definitions requires the author to make sure the definitions have been correctly ordered in the definition block and to correctly match definition return types with option types. Since all work in the *Describer* remains in a single panel, ScriptEase can do the description to option type matching for the

author by filtering the descriptions. The author can connect multiple descriptive lines together by choosing *Describe it* when setting an option.

The efficiency and desirability of the *Describers* was tested in a user study where the participants tried to complete a set of statements using the *Describer* and using an original version of ScriptEase referred to as the *Definer*. The results were conclusive that the *Describer* was more efficient than the *Definer*. The participants were able to complete an average of 2.14 statements using the *Describer* compared to the 1.27 using the *Definer*. The participants overall preferred the *Describer*, as did the females and gamers. The results of males, non-gamers and programmers were inconclusive.

5.2 Future Work

As always, finishing one phase of research produces new questions and paths for future work. For my work with *Describers* there are two main paths that I believe should be explored next: more user studies and further simplification.

5.2.1 More User Studies

The results we gained were excellent, but there are still unanswered questions. We know that females preferred the *Describer* but were unable to make any conclusions about what males prefer. The same can be said for programmers as we had such a small number of participants with any programming experience.

While females produced a conclusive result on which tool was more intuitive, and the result for gamers was right on the edge of statistically significant, it was not significant for everyone. I believe that another study designed to focus specifically on evaluating this characteristic is needed. This also suggests that there may be some other part of ScriptEase that authors are finding difficult to use that is affecting how intuitive the participants found the methods. A new study would be able to look into this in more depth.

One of the early motivations for further simplification was to make ScriptEase more accessible to younger students. As mentioned in the related works (Chapter 2) both Scratch and Alice are aimed at younger students, placing them in a different category from ScriptEase. ScriptEase has been successfully used by Grade 10 students before [9, 21, 8, 3, 4, 5], but has never been used in a study with authors younger than Grade 10. A new study with Grade 6 or 7 students trying to use a version with *Describers* would allow us to see how accessible ScriptEase is for younger authors.

5.2.2 Further Simplification

While simplifying ScriptEase with the creation of *Describers* is a large step forward towards simplifying ScriptEase, it is not the only area that can benefit from simplification. For my research, I implemented *Describers* to set options for encounters. However, options also need to be set in Behaviors and Quests patterns, and *Describers* should also be implemented for these patterns. Another area for simplification exists in creating conditions within patterns.

While programmers understand if-statements and how to create the conditional statement, this is a task many non-programmers find extremely difficult. However, these same non-programmers understand how to phrase yes-no questions and this type of question can be easily translated into conditional statements behind the scenes. I believe that the underlying structure of *Describers* can be transferred easily over to solving the problem of creating conditional statements by having the authors describe yes-no questions.

5.3 Concluding Remarks

As the video game industry continues to grow, the bottleneck created by content creation will become a larger strain. Tools can help ease the scripting bottleneck by aiding designers who are usually non-programmers. Video games often include tools to allow authors to create their own levels and modules but require the authors to learn a scripting language specific to the game. ScriptEase is one of the few available tools that attempts to bridge this gap.

My thesis looked at how to further simplify ScriptEase and make it even more accessible by changing the way options are set. The new technique, *Describers*, was then shown through a user study to be much more efficient and was preferred over the original method of using definitions. Further research through more user studies and future simplification will continue to make video game design more accessible to non-programmers and eventually may eliminate the content creation bottleneck while providing more engaging games.

Bibliography

[1] Eric Bangeman. Growth of gaming in 2007 far outpaces movies, music. Website, 2009. `http://arstechnica.com/gaming/news/2008/01/growth-of-gaming-in-2007-far-outpaces-movies-music.ars`.

[2] Bethesda. Oblivion. Website, 2009. `http://www.elderscrolls.com/home/home.php`.

[3] Mike Carbonaro, Maria Cutimisu, H Duff, Stephanie Gillis, Curtis Onuczko, Jonathan Schaeffer, A Schumacher, Jeff Siegel, Duane Szafron, and Kevin Waugh. Adapting a commercial role-playing game for educational computer game production. In *GameOn North America*, 2006.

[4] Mike Carbonaro, Maria Cutimisu, Matthew McNaughton, Curtis Onuczko, Thomas Roy, Jonathan Schaeffer, Duane Szafron, Stephanie Gillis, and Sabrina Kratchmer. Interactive story writing in the classroom: Using computer games. In *Digital Games Research Association (DIGRA)*, pages 323–328, 2005.

[5] Mike Carbonaro, Maria Cutumisu, Harvey Duff, Stephanie Gillis, Curtis Onuczko, Jeff Siegel, Jonathan Schaeffer, Allan Schumacher, Duane Szafron, and Kevin Waugh. Interactive story authoring: A viable form of creative expression for the classroom. *Computers and Education, in press*, 2007.

[6] CMU. Alice. Website, 2009. `http://www.alice.org`.

[7] CMU. Storytelling alice. Website, 2009. `http://www.alice.org/kelleher/storytelling/index.html`.

[8] Maria Cutimisu, Curtis Onuczko, Duane Szafron, Jonathan Schaeffer, Matthew McNaughton, Thomas Roy, Jeff Siegel, and Mike Carbonaro. Evaluating pattern catalogs - the computer games experience. In *International Conference on Software Engineering (ICSE)*, pages 132–141, 2006.

[9] Maria Cutumisu, Curtis Onuczko, Matthew McNaughton, Thomas Roy, Jonathan Schaeffer, Allan Schumacher, Jeff Siegel, Duane Szafron, Kevin Waugh, Mike Carbonaro, Harvey Duff, and Stephanie Gillisr. Scriptease: A generative/adaptive programming paradigm for game scripting. *Science of Computer Programming*, 67(1):32–55, June 2007.

[10] Rockstar Games. Grand theft auto iv. Website, 2009. `http://www.rockstargames.com/IV/`.

[11] Yahoo! Games. August 30: Neverwinter nights press release - over 1,000 modules now available. Website, 2009. `http://www.videogames.yahoo.com/news-1113872`.

[12] GI GamesIndustry.biz. Gta iv: Most expensive game ever developed? Website, 2009. `http://www.gamesindustry.biz/articles/gta-iv-most-expensive-game-ever-developed`.

[13] GameZine.co.uk. Assassin's creed development team triples in size. Website, 2009. http://www.gamezine.co.uk/news/games/a/assassin-s-creed-2/assassin-s-creed-development-team-triples-in-size-\$1296733.htm.

[14] Electronic Arts Inc. Spore. Website, 2009. http://www.spore.com/.

[15] M. McNaughton, M. Cutimisu, D. Szafron, J. Schaeffer, J. Redford, and D. Parker. Scriptease: Generative design patterns for computer role-playing games. In *19th IEEE International Conference on Automated Software Engineering (ASE)*, pages 88–99, 2004.

[16] MIT. Scratch. Website, 2009. http://scratch.mit.edu.

[17] Otter. The movie industry vs. the gaming industry. Website, 2009. http://www.associatedcontent.com/article/1015720/the_movie_industry_vs_the_gaming_industry.html?cat=19.

[18] R Project. R project for statistical computing. Website, 2009. http://www.r-project.org/.

[19] Microsoft Research. Kodu. Website, 2009. http://research.microsoft.com/en-us/projects/kodu/.

[20] ScriptEase. Scriptease. Website, 2009. http://www.cs.ualberta.ca/~script.

[21] Duane Szafron, Mike Carbonaro, Maria Cutimisu, Stephanie Gillis, Matthew McNaughton, Curtis Onuczko, Thomas Roy, and Jonathan Schaeffer. Writing interactive stories in the classroom. *Interactive Multimedia Electronic Journal of Computer-Enhanced Learning(IMEJ)*, 7(1), 2005.

[22] Ubisoft. Assassin's creed 2. Website, 2009. http://assassinscreed.com/.

Appendix A

Test Scenario

Test Scenario B – Definer

Participant Number: ________

Instructions: Work your way through the below statements by using the describer to set the options according to the statement. When you open the module, the options that are to be set appear as <invalid>. After you complete each statement save the current file, record the time and open the next one according to the directions.

File Names: Each file is named using first your participant number, then the test scenario A or B, then Desc or Defn for describer or definer and finally the statement number. *Ex:* For example, **32BDesc4.mod** is the file for participant 32, on Test Scenario B, using the Describer for statement 4. Replace the 32 by your participant number.

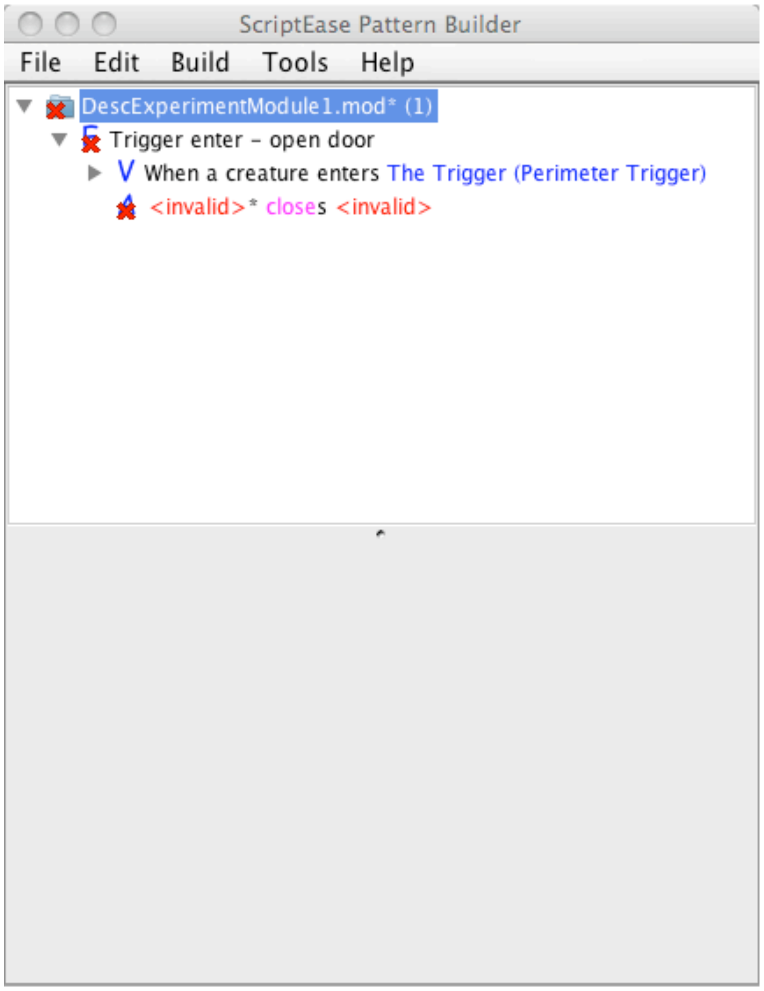
Focus on making sure your solutions are correct. It is not important to finish all parts in the time given.

The first statement includes a walk-through to introduce you to the tool.

Time Starting	

Statement 1: *When the trigger is entered, a random object with the same tag as Bob opens the metal door.*

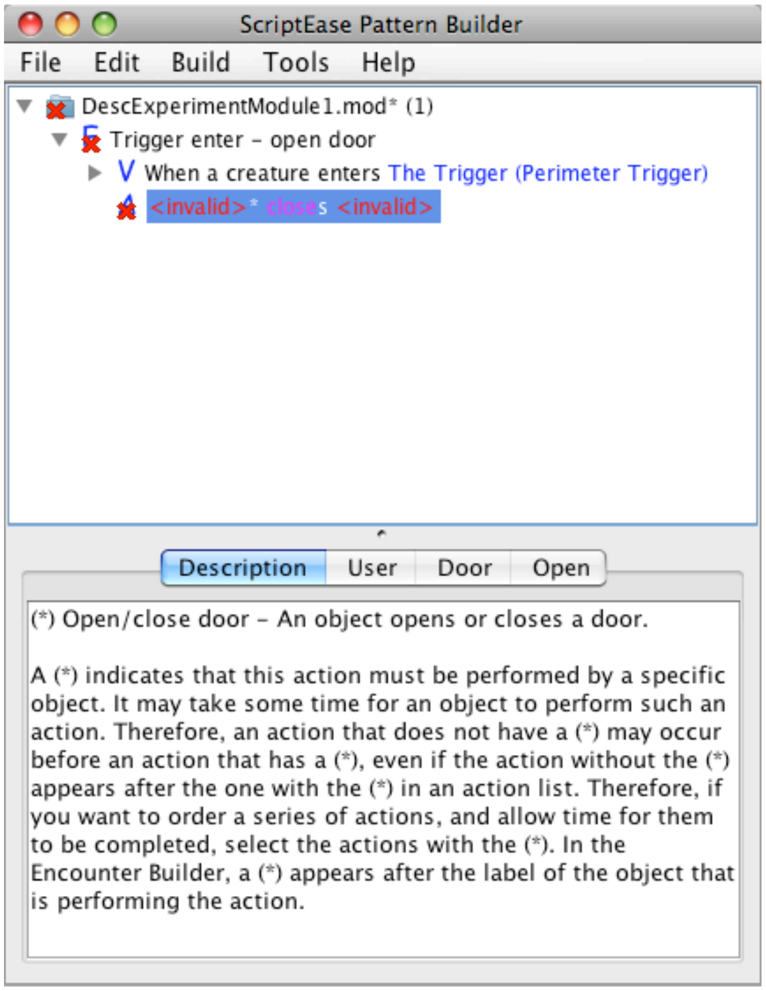
After opening the module and expanding the tree in the top frame ScriptEase should now look similar to the below picture:



DescExperimentModule1.mod* (1)
Trigger enter - open door
When a creature enters The Trigger (Perimeter Trigger)
<invalid>* closes <invalid>

The red x's are to let you know that there are remaining options that need to be set. You will notice that these red x's also appear in the describer pane on the bottom.

In the top frame, click on the line A <invalid>* closes <invalid> to bring it to focus. Notice that the bottom pane has changed and there are four tabs: Description, User, Door and Open.

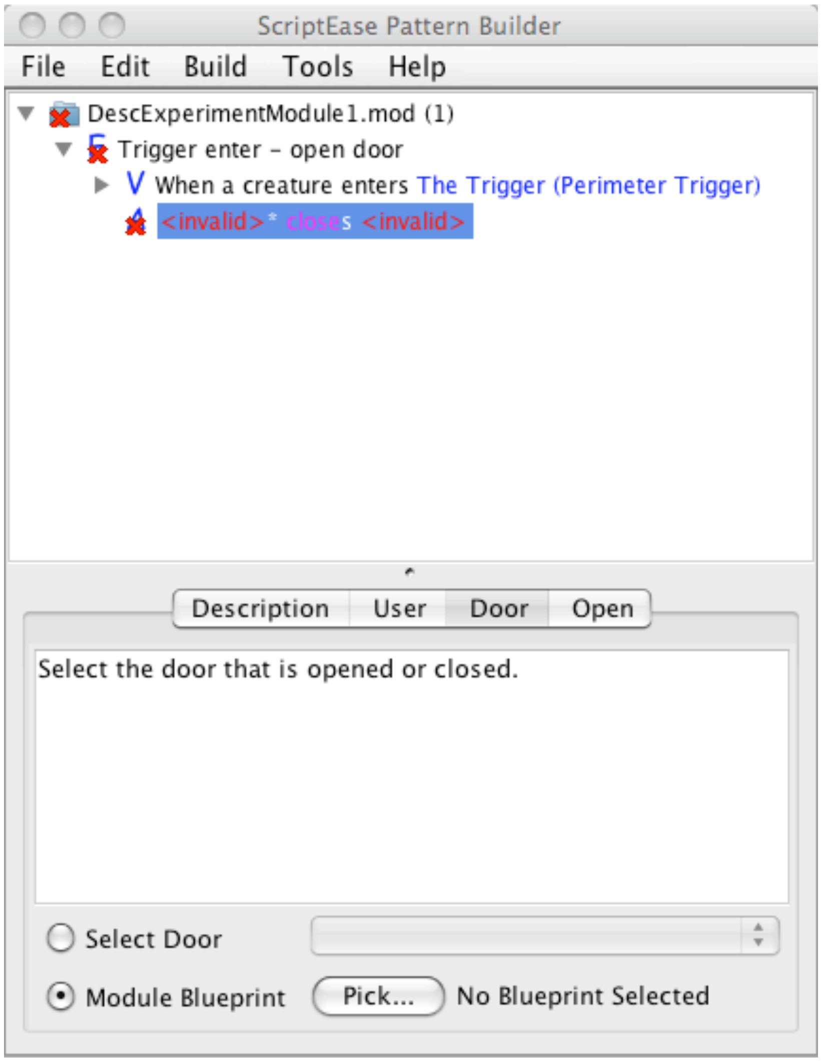

ScriptEase Pattern Builder
File Edit Build Tools Help
DescExperimentModule1.mod* (1)
Trigger enter – open door
When a creature enters The Trigger (Perimeter Trigger)
<invalid>* closes <invalid>
Description User Door Open

(*) Open/close door – An object opens or closes a door.

A (*) indicates that this action must be performed by a specific object. It may take some time for an object to perform such an action. Therefore, an action that does not have a (*) may occur before an action that has a (*), even if the action without the (*) appears after the one with the (*) in an action list. Therefore, if you want to order a series of actions, and allow time for them to be completed, select the actions with the (*). In the Encounter Builder, a (*) appears after the label of the object that is performing the action.

We need to change three things: set the User to a random object with tag Bob, set the Door to the metal door, and change the Open option from close to open. However, we are going to set them in a different order.

Let's start with setting the Door. If you click on the Door tab you will see there are two radio buttons at the bottom of the frame. The top says *Select Door* and the bottom says *Module Blueprint*. All options will provide two similar options for setting the with slight variations.

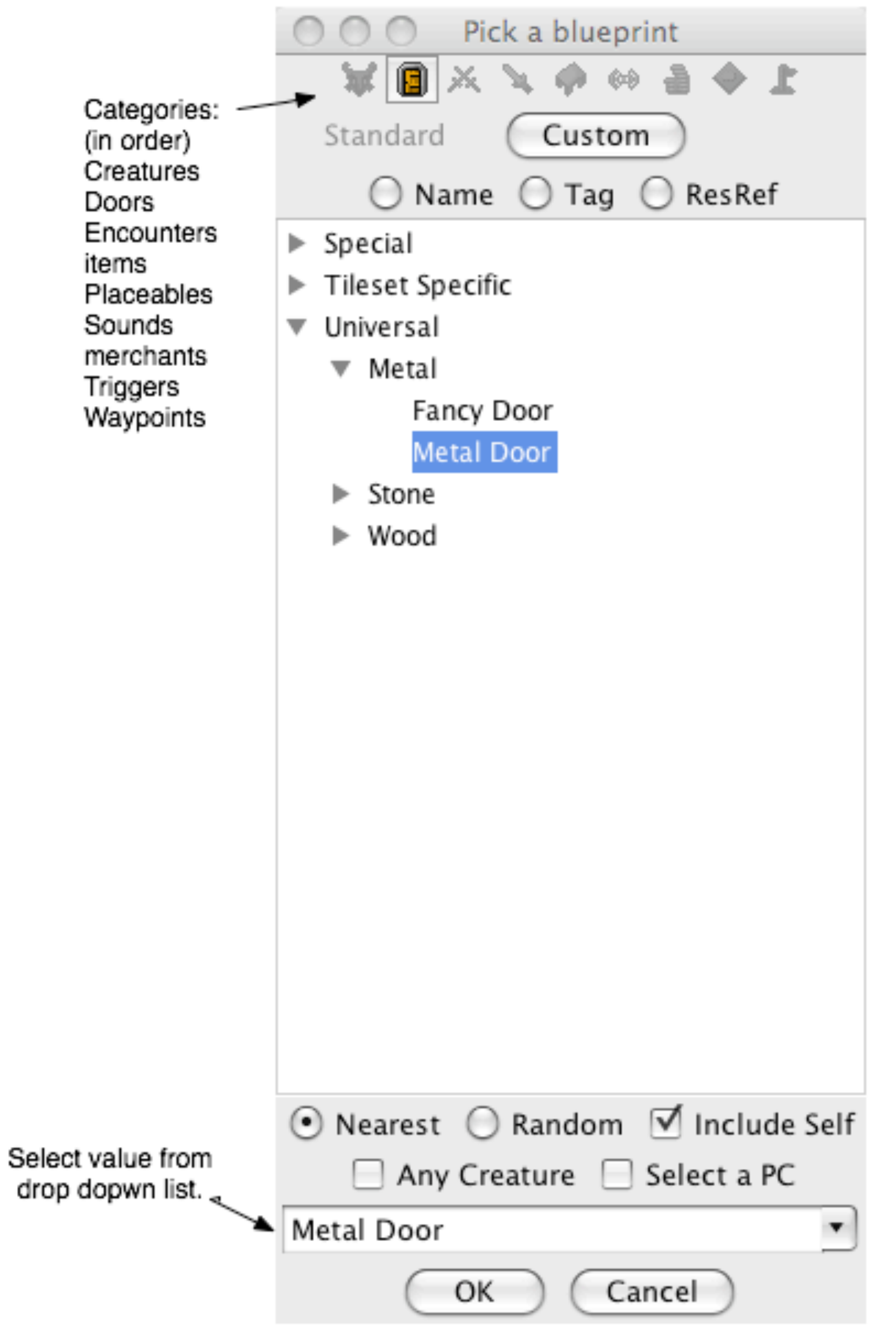

ScriptEase Pattern Builder
File
Edit
Build
Tools
Help
DescExperimentModule1.mod (1)
Trigger enter – open door
When a creature enters The Trigger (Perimeter Trigger)
<invalid>* closes <invalid>
Description
User
Door
Open

Select the door that is opened or closed.

Select Door [✓ Module Blueprint] Pick... No Blueprint Selected

The top radio button allows you to set the option by reusing an option you have previous set. If you click on the top radio button, you will notice you can now use the dropdown menu to the right. In it you will find one option, Door (Metal Door). By selecting the Door (Metal Door) the option is now set.

However, there is another way you can set this option, and that is by using the *Pick* button. If you click the *Pick* button the *Picker* (seen below) will open up and you can choose the door from there.



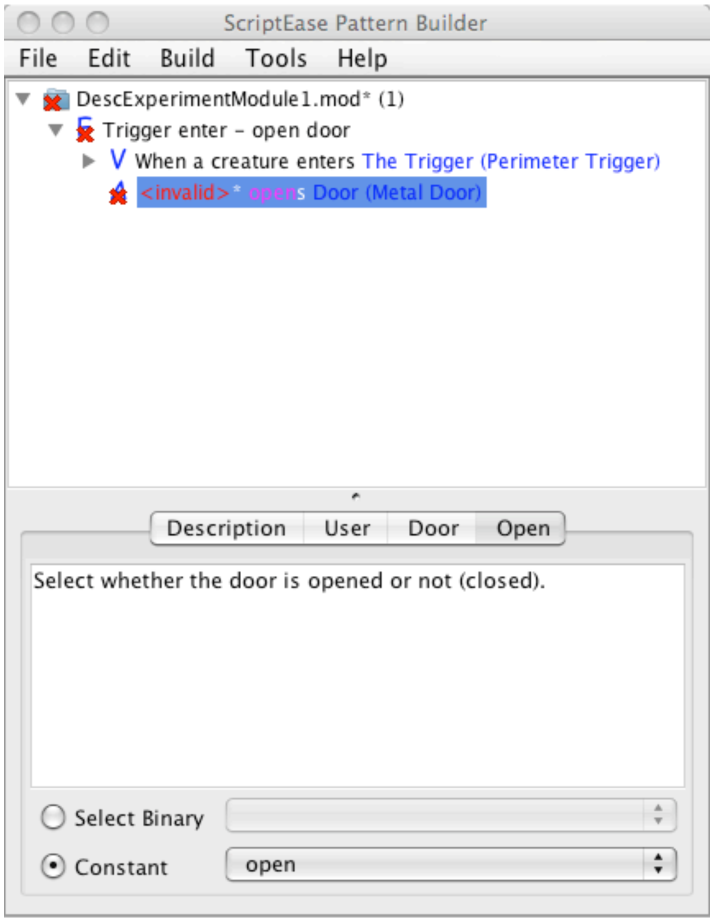
○ Name ○ Tag ○ ResRef

▶ Special
▶ Tileset Specific
▼ Universal
  ▼ Metal
      Fancy Door
      Metal Door
  ▶ Stone
  ▶ Wood

● Nearest ○ Random ☑ Include Self
☐ Any Creature ☐ Select a PC

Metal Door ▾

Next let's change *closes* to *opens*. If you click on the Open tab, it will look similar to what you originally saw on the Door tab. This time, instead of getting to use the *Pick* button at the bottom, you are setting a constant so there is a drop down menu with the two possible values – *open* and *close*. Since we want the Door to *open*, change the dropdown menu to *open*. Your screen should now look similar to the one below:

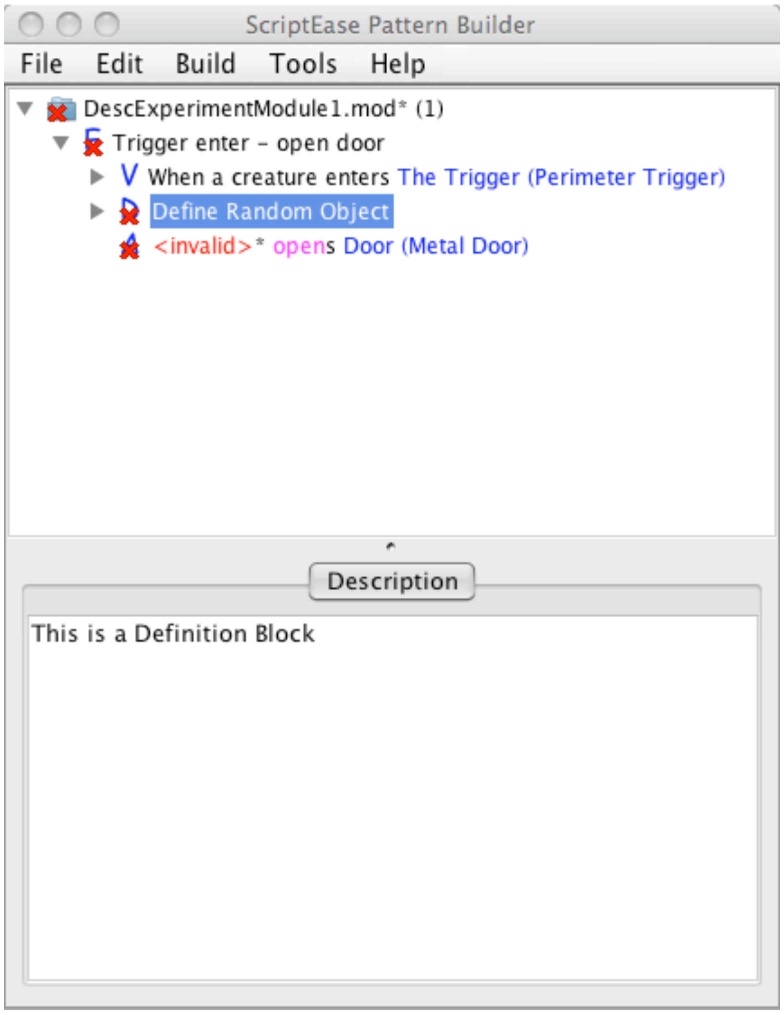

ScriptEase Pattern Builder
File Edit Build Tools Help
DescExperimentModule1.mod* (1)
Trigger enter – open door
When a creature enters The Trigger (Perimeter Trigger)
<invalid>* opens Door (Metal Door)
Description User Door Open

Select whether the door is opened or not (closed).

( ) Select Binary
(•) Constant: open

Finally, we need to set the User to be a random object. To do so, we will need to use a definition.

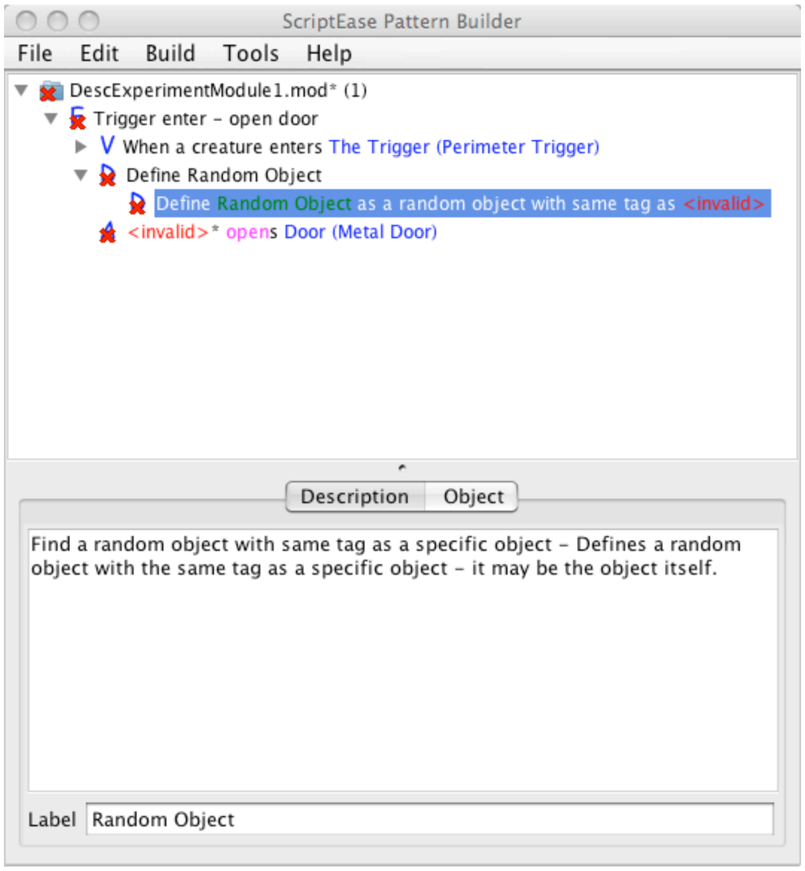
To add a definition, right click where in the top frame where it says “Trigger enter – open door.” From the menu that appears, choose *Add a Definition*, then in move to *Finding Objects* and select the definition *Find a Random Object that has the same tag as a specific object*, which should be listed third from the top. Your screen should now look like the picture below:



DescExperimentModule1.mod* (1)
Trigger enter – open door
When a creature enters The Trigger (Perimeter Trigger)
Define Random Object
<invalid>* opens Door (Metal Door)

This is a Definition Block

Expand the highlighted line *Define Random Object* and select the definition.


ScriptEase Pattern Builder

File Edit Build Tools Help

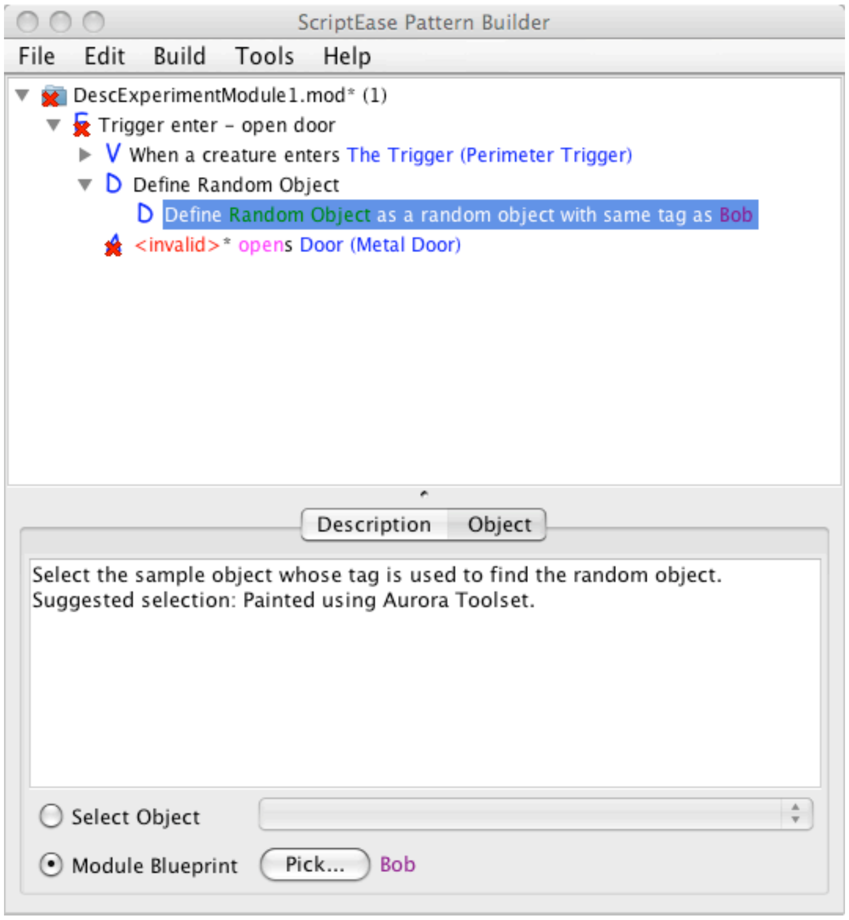
DescExperimentModule1.mod* (1)
Trigger enter – open door
V When a creature enters The Trigger (Perimeter Trigger)
Define Random Object
Define Random Object as a random object with same tag as <invalid>
<invalid>* opens Door (Metal Door)

Description | Object

Find a random object with same tag as a specific object – Defines a random object with the same tag as a specific object – it may be the object itself.

Label: Random Object

The bottom frame should look similar to what you saw earlier when you were setting the Door. This time we need to set the Object to be Bob. So click on the Object tab, and then use the *Pick* button to set the object to Bob.

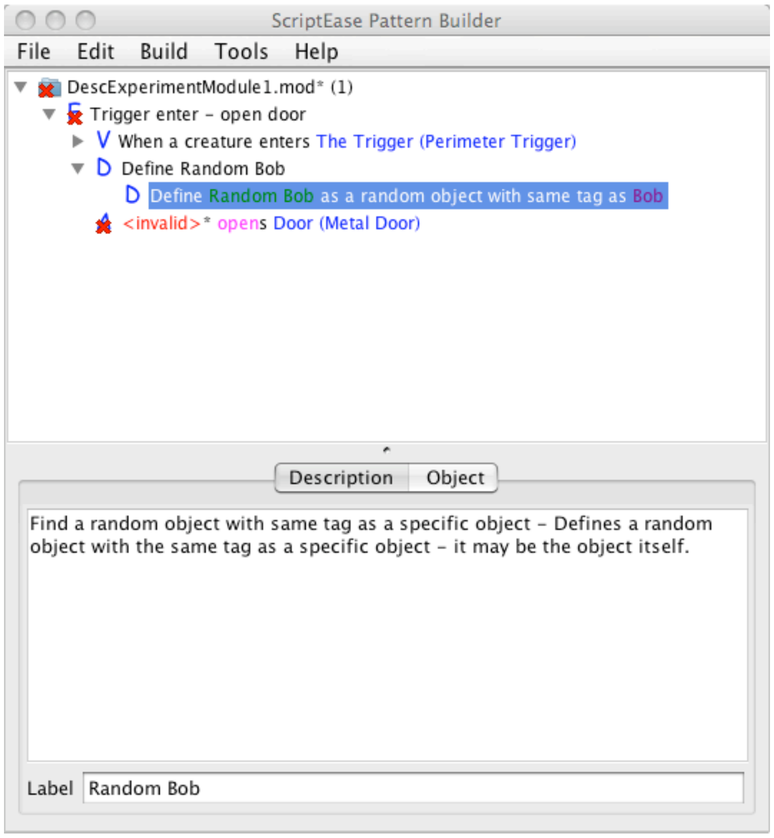


DescExperimentModule1.mod* (1)
Trigger enter - open door
V When a creature enters The Trigger (Perimeter Trigger)
D Define Random Object
D Define Random Object as a random object with same tag as Bob
<invalid>* opens Door (Metal Door)

Select the sample object whose tag is used to find the random object.
Suggested selection: Painted using Aurora Toolset.

Select Object
[✓ Module Blueprint] Pick... Bob

At this point, our definition is all set. However, the definition's name is *Random Object* which is only slightly descriptive. To make the name easier to recall move back to the Description tab. At the very bottom it says "Label" followed by a text field that specifies the definition's name – currently Random Object. Change the label to read *Random Bob* instead.

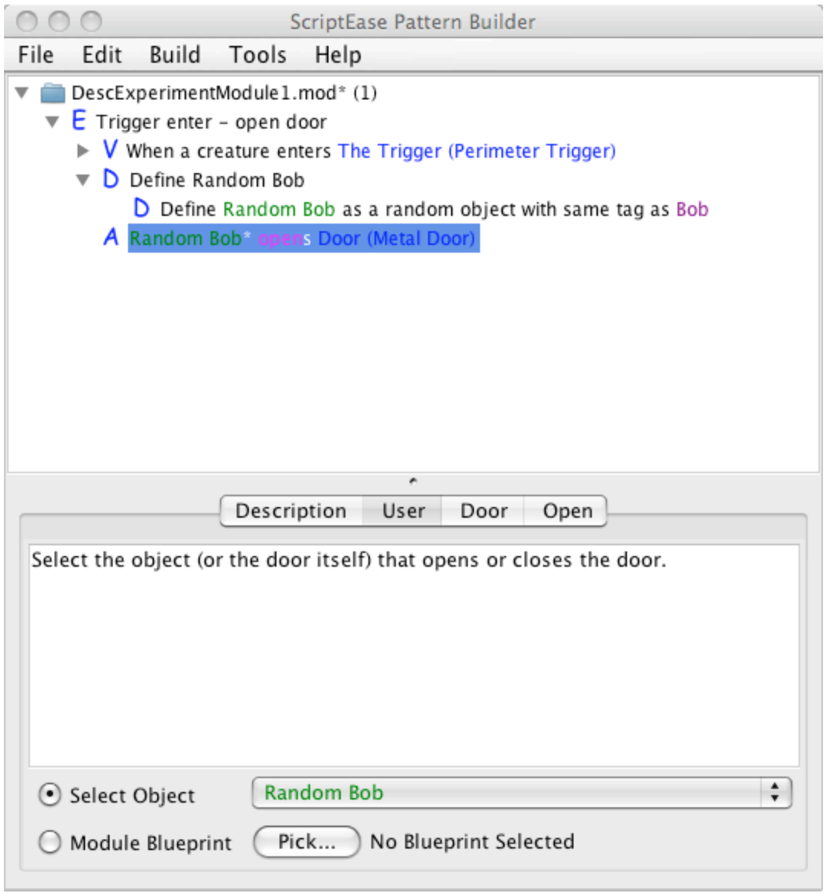


DescExperimentModule1.mod* (1)
Trigger enter – open door
When a creature enters The Trigger (Perimeter Trigger)
Define Random Bob
Define Random Bob as a random object with same tag as Bob
<invalid>* opens Door (Metal Door)

Find a random object with same tag as a specific object – Defines a random object with the same tag as a specific object – it may be the object itself.

Label Random Bob

Finally, we need finish by setting the User back in our original definition. So click on the line <invalid>* opens Door (Metal Door) and then click on the User tab. Since we have already created the definition of the user we want so we use the top radio button *Select Object*. Select the radio button, and then use the drop down menu to choose *Random Bob*.



DescExperimentModule1.mod* (1)
E Trigger enter - open door
V When a creature enters The Trigger (Perimeter Trigger)
D Define Random Bob
D Define Random Bob as a random object with same tag as Bob
A Random Bob* opens Door (Metal Door)

Select the object (or the door itself) that opens or closes the door.

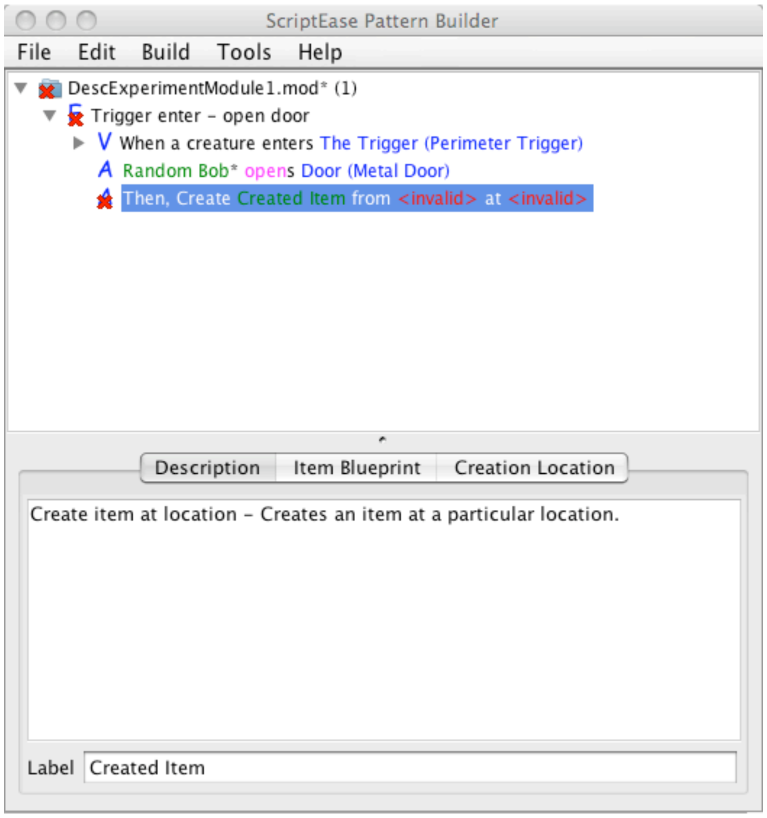
[✓ Select Object] Random Bob
Module Blueprint No Blueprint Selected

Now that you've finished working through the first statement, record your time. The rest of the statements do not provide a walkthrough. If you get stuck, please raise your hand and we will assist you.

Time Finished

Save module, and then open next module 32BDefn2.mod. (Change 32 to your participant number)

Statement 2: *Then, create a potion of Cure Critical Wounds 1.7m in front of the Enterer.*



DescExperimentModule1.mod* (1)
Trigger enter - open door
When a creature enters The Trigger (Perimeter Trigger)
Random Bob* opens Door (Metal Door)
Then, Create Created Item from <invalid> at <invalid>

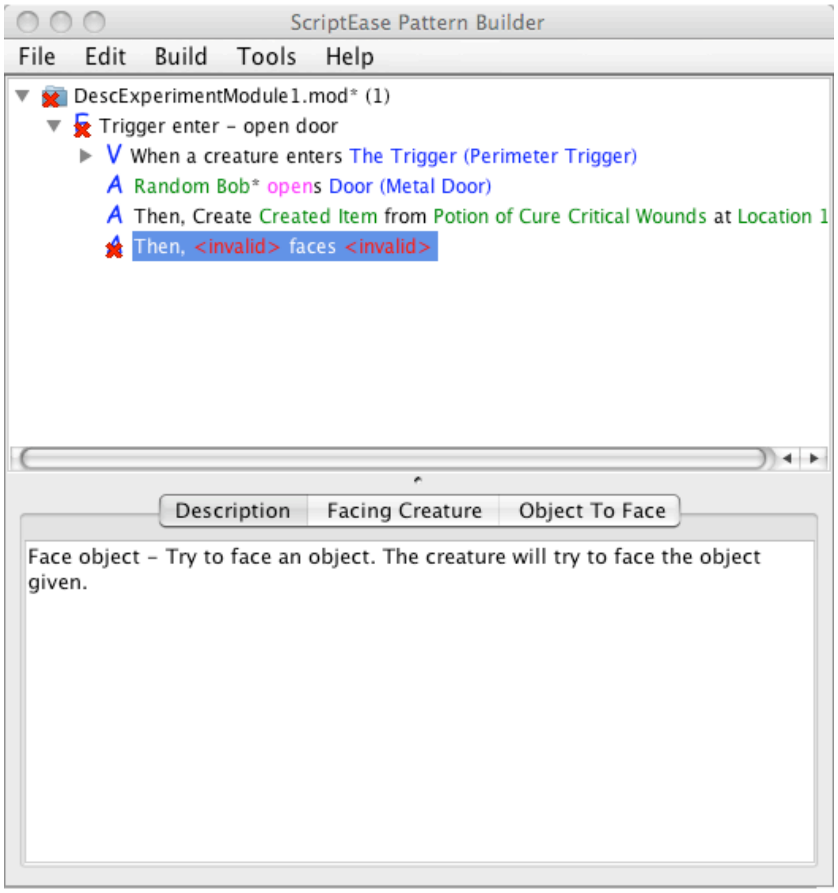
Create item at location - Creates an item at a particular location.

Label: Created Item

Time Finished	

Save module, and then open next module 32BDefn3.mod. (Change 32 to your participant number)

Statement 3: *Fred then faces the created item.*


ScriptEase Pattern Builder
File Edit Build Tools Help
Description Facing Creature Object To Face

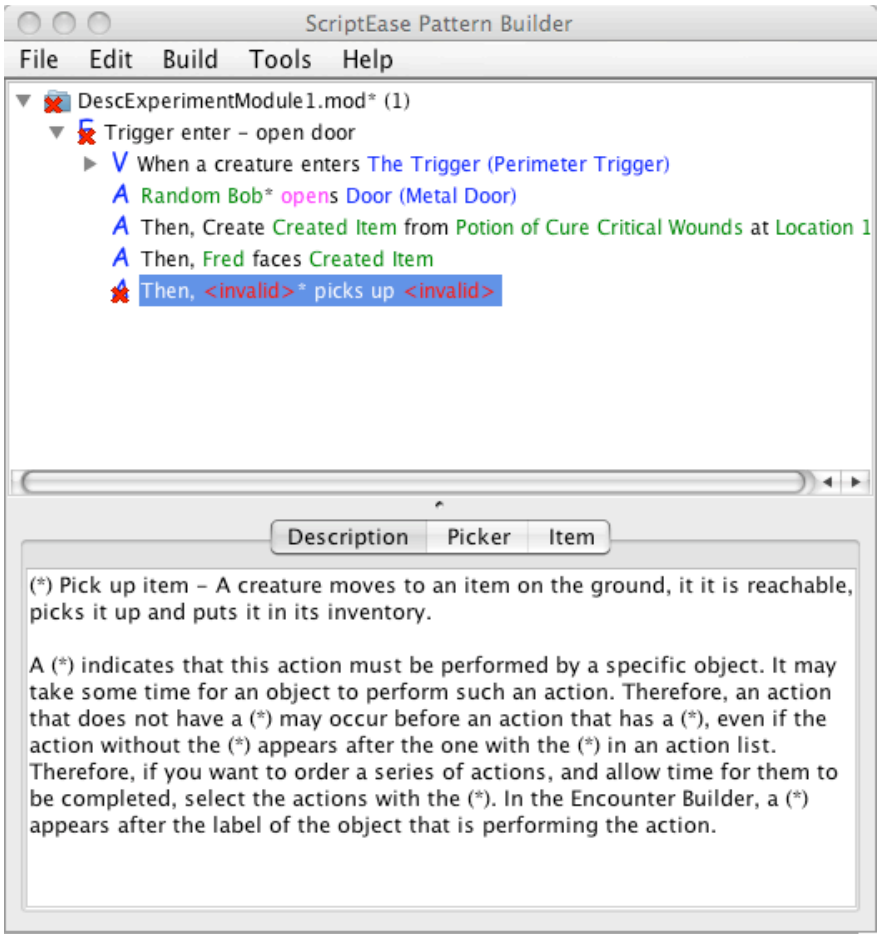
DescExperimentModule1.mod* (1)
Trigger enter – open door
When a creature enters The Trigger (Perimeter Trigger)
Random Bob* opens Door (Metal Door)
Then, Create Created Item from Potion of Cure Critical Wounds at Location 1
Then, <invalid> faces <invalid>

Face object – Try to face an object. The creature will try to face the object given.

Time Finished

Save module, and then open next module 32BDefn4.mod. (Change 32 to your participant number)

Statement 4: *Then, the x nearest creature to Fred with tag Polar Bear picks up the created item. X is determined by the roll of a 6 sided dice.*



DescExperimentModule1.mod* (1)
Trigger enter – open door
V When a creature enters The Trigger (Perimeter Trigger)
A Random Bob* opens Door (Metal Door)
A Then, Create Created Item from Potion of Cure Critical Wounds at Location 1
A Then, Fred faces Created Item
A Then, <invalid>* picks up <invalid>

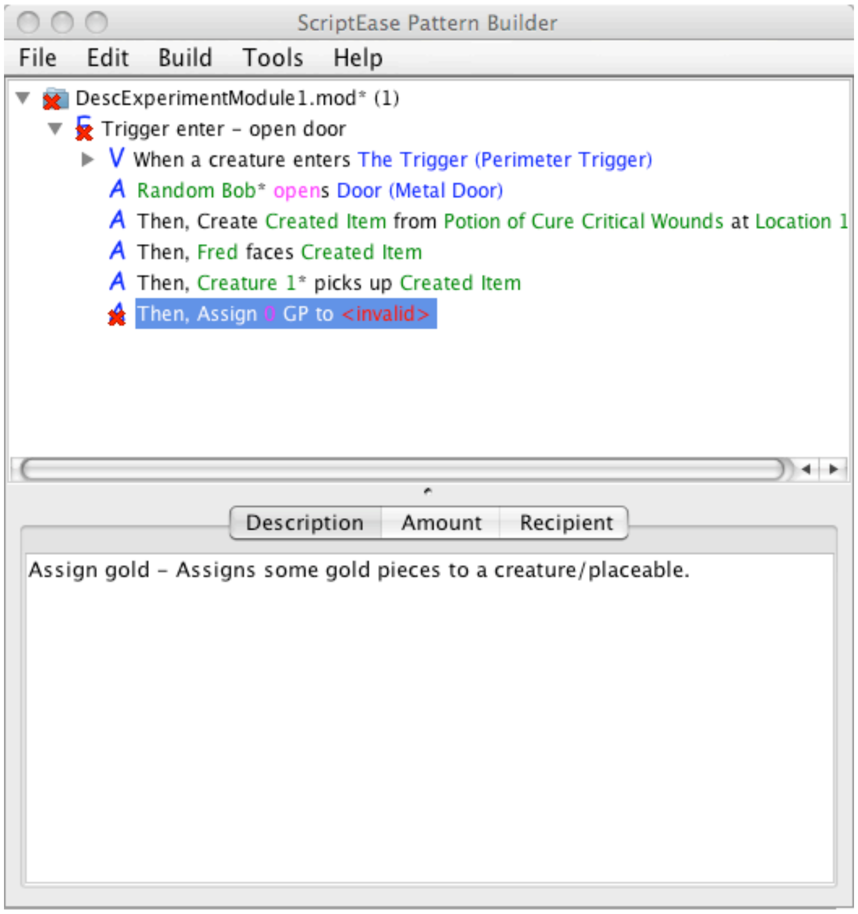
(*) Pick up item – A creature moves to an item on the ground, it it is reachable, picks it up and puts it in its inventory.

A (*) indicates that this action must be performed by a specific object. It may take some time for an object to perform such an action. Therefore, an action that does not have a (*) may occur before an action that has a (*), even if the action without the (*) appears after the one with the (*) in an action list. Therefore, if you want to order a series of actions, and allow time for them to be completed, select the actions with the (*). In the Encounter Builder, a (*) appears after the label of the object that is performing the action.

Time Finished

Save module, and then open next module 32BDefn5.mod. (Change 32 to your participant number)

Statement 5: *Then give the Enterer some gold. The amount of gold is equal to the current gold owned by the enterer plus their age. Name the gold amount Reward.*



ScriptEase Pattern Builder
File Edit Build Tools Help

DescExperimentModule1.mod* (1)
Trigger enter - open door
V When a creature enters The Trigger (Perimeter Trigger)
A Random Bob* opens Door (Metal Door)
A Then, Create Created Item from Potion of Cure Critical Wounds at Location 1
A Then, Fred faces Created Item
A Then, Creature 1* picks up Created Item
Then, Assign 0 GP to <invalid>

Description Amount Recipient

Assign gold - Assigns some gold pieces to a creature/placeable.

Time Finished

Save your module and then close ScriptEase.

Test Scenario A – Describer

Participant Number: ________

Instructions: Work your way through the below statements by using the describer to set the options according to the statement. When you open the module, the options that are to be set appear as <invalid>. After you complete each statement save the current file, record the time and open the next one according to the directions.

File Names: Each file is named using first your participant number, then the test scenario A or B, then Desc or Defn for describer or definer and finally the statement number. *Ex:* For example, **32ADesc4.mod** is the file for participant 32, on Test Scenario A, using the Describer for statement 4. Replace the 32 by your participant number.

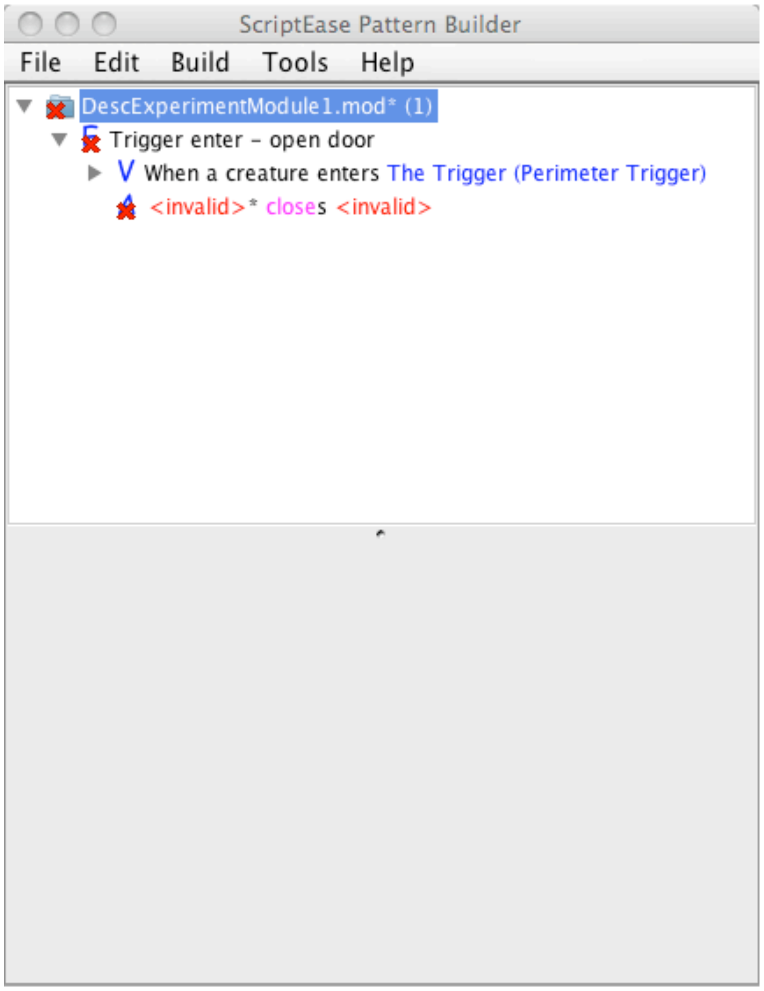
Even though you are recording your time, do not focus on trying to complete all the statements but on getting them correct.

The first statement includes a walk-through to introduce you to the tool.

Time Starting

Statement 1: *When the trigger is entered, a random object with the same tag as Bob opens the metal door.*

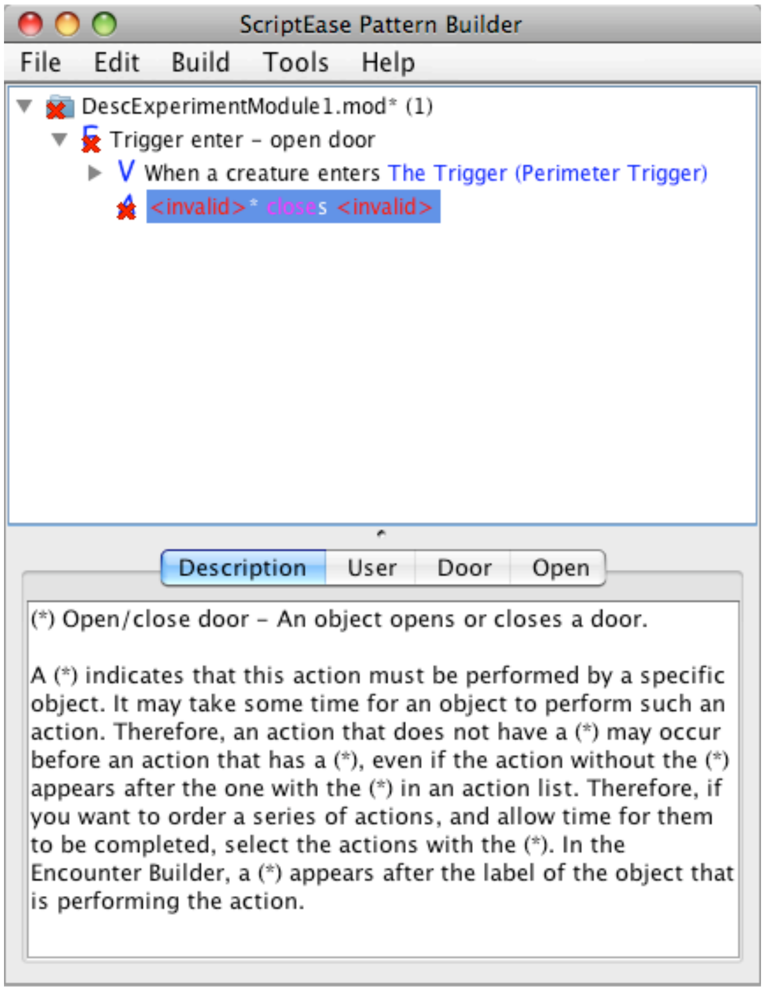
After opening the module and expanding the tree in the top frame ScriptEase should now look similar to the below picture:



DescExperimentModule1.mod* (1)
Trigger enter – open door
When a creature enters The Trigger (Perimeter Trigger)
<invalid>* closes <invalid>

The red x's are to let you know that there are remaining options that need to be set. You will notice that these red x's also appear in the describer pane on the bottom.

In the top frame, click on the line A <invalid>* closes <invalid> to bring it to focus. Notice that the bottom pane has changed and there are four tabs: Description, User, Door and Open.

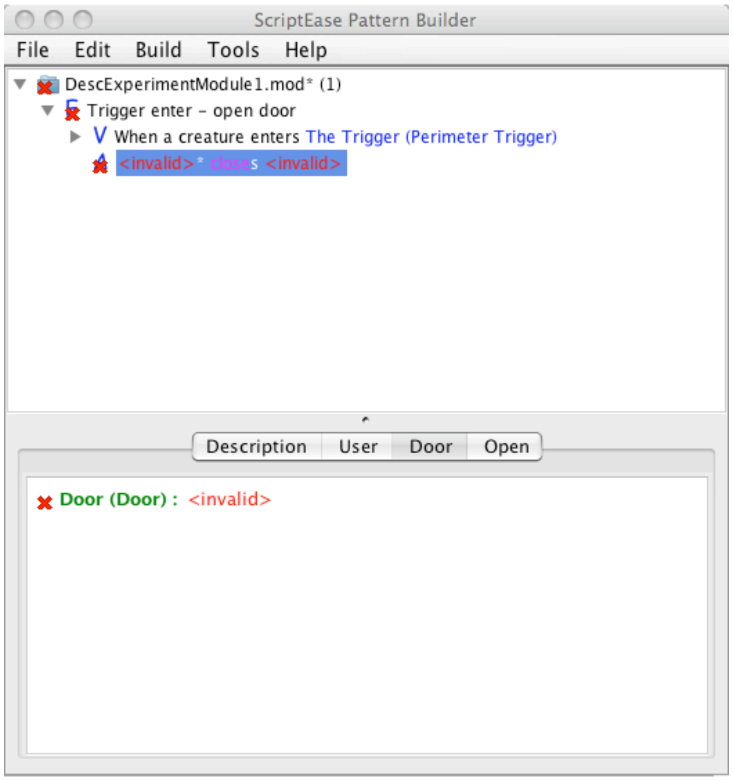

ScriptEase Pattern Builder
File
Edit
Build
Tools
Help
DescExperimentModule1.mod* (1)
Trigger enter – open door
When a creature enters The Trigger (Perimeter Trigger)
<invalid>* closes <invalid>
Description
User
Door
Open

(*) Open/close door – An object opens or closes a door.

A (*) indicates that this action must be performed by a specific object. It may take some time for an object to perform such an action. Therefore, an action that does not have a (*) may occur before an action that has a (*), even if the action without the (*) appears after the one with the (*) in an action list. Therefore, if you want to order a series of actions, and allow time for them to be completed, select the actions with the (*). In the Encounter Builder, a (*) appears after the label of the object that is performing the action.

We need to change three things: set the User to a random object with tag Bob, set the Door to the metal door, and change the Open option from close to open. However, we are going to set them in a different order.

Let’s start with setting the Door. If you click on the Door tab you will see xDoor (Door): <invalid> . The first part of the line Door (Door) specifies first the name of the option and then in brackets the option type. In this case, they’re both the same. The x means it hasn’t been set yet.



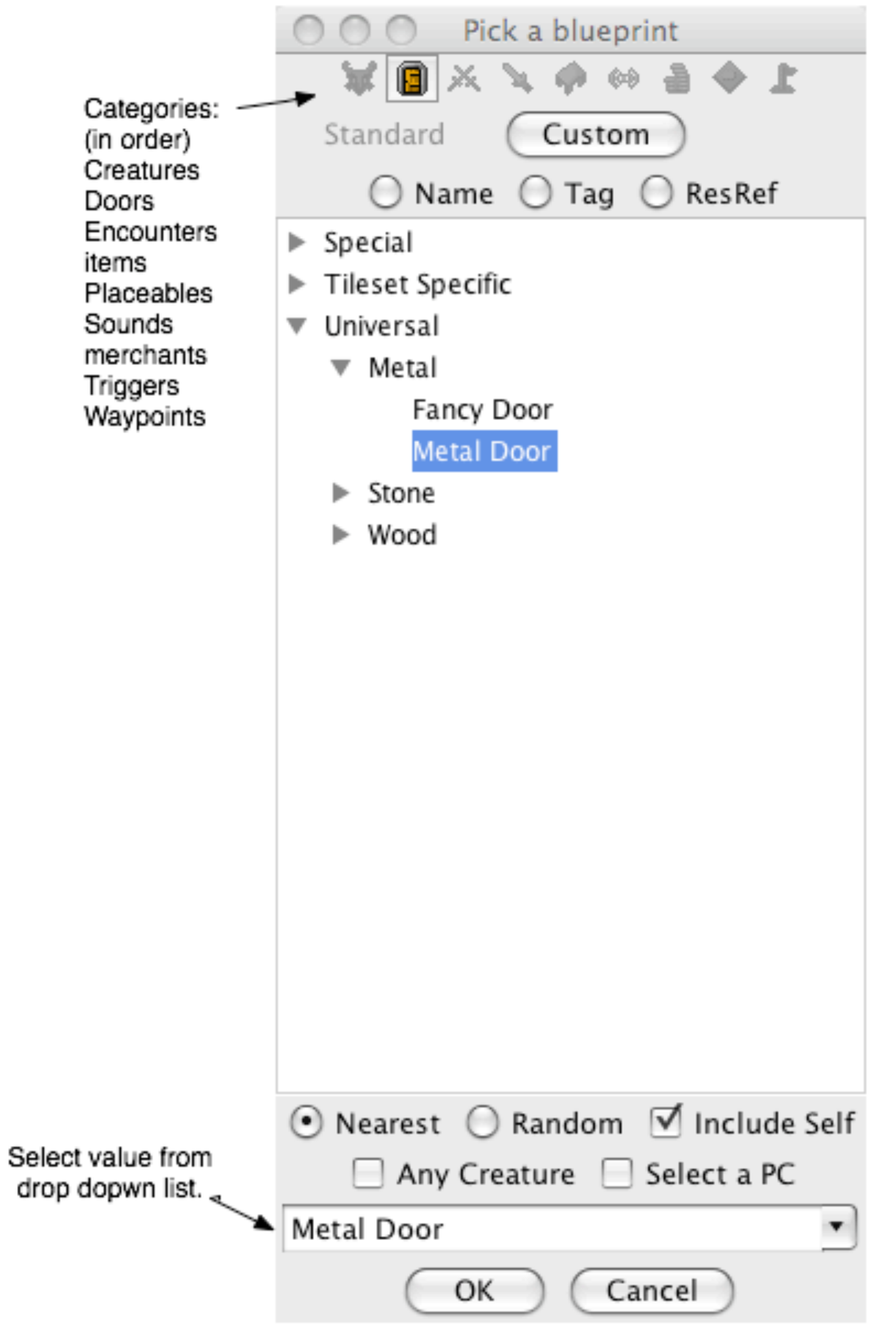
DescExperimentModule1.mod* (1)
Trigger enter – open door
V When a creature enters The Trigger (Perimeter Trigger)
<invalid>* closes <invalid>

Door (Door) : <invalid>

The <invalid> is where you start to set the option. If you change the Door from <invalid> in the bottom pane it will change in the top frame. Let’s click on <invalid> and try to set it. You will notice that you have three choices: Recall it, Pick it, and Describe it. You will always have these three choices when setting a option.

Recall it: This is used when you want to reuse an option you have previously set.
Pick it: If you want to choose a specific object directly (ie you know what creature, placeable, number, etc you want to use).
Describe it: This is used when you want a more complex option description. In this example we will use describe it to set the random object for the User.

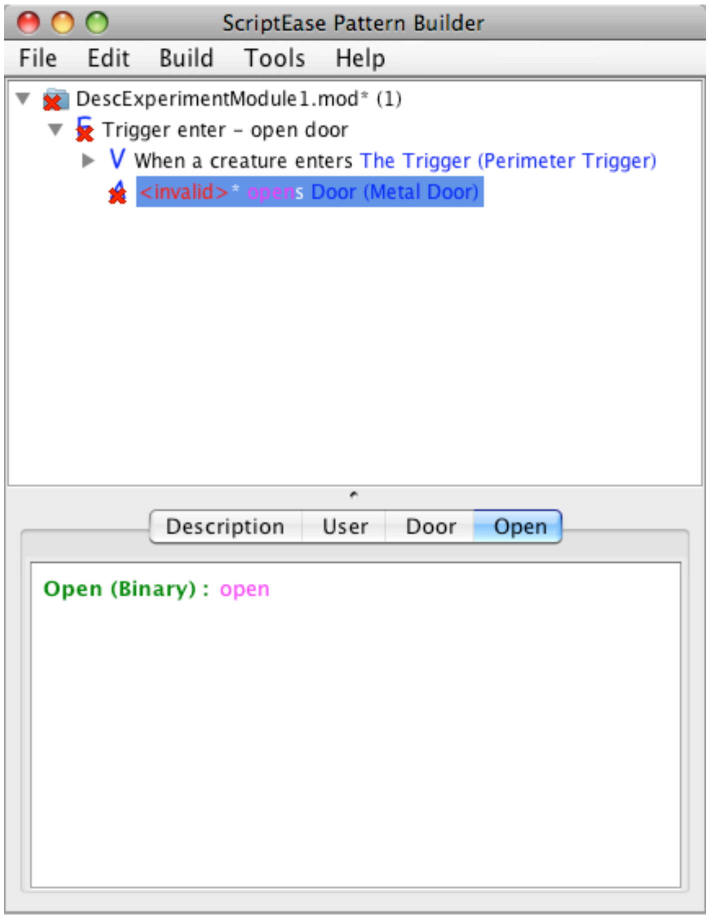
For the door, there are two ways we can set it. We can use *pick it* to choose the Metal Door in the drop down window (as seen below) or we can use *recall it* and choose “The Door (Metal Door).” Notice that if you use *pick it* the text in the top frame changes to “<invalid>* closes Metal Door” but if you use *recall it*, the top frame’s text becomes “<invalid>* closes The Door (Metal Door).”



[✓ Nearest] Random [✓ Include Self]

Metal Door

Next let's change *closes* to *opens*. If you click on the Open tab, it will look similar to what you originally saw on the Door tab. This time it says "Open (Binary): close". The option name is Open, the type is Binary and it is currently set to *close*. If you click on *close*, you will notice that this time you only have one choice, to pick it. The pick it window is different for this type then it was for the Door. You only get two options, *open* and *close*. Since we want the Door to *open* select *open* and click *ok*. Your screen should now look similar to the one below:

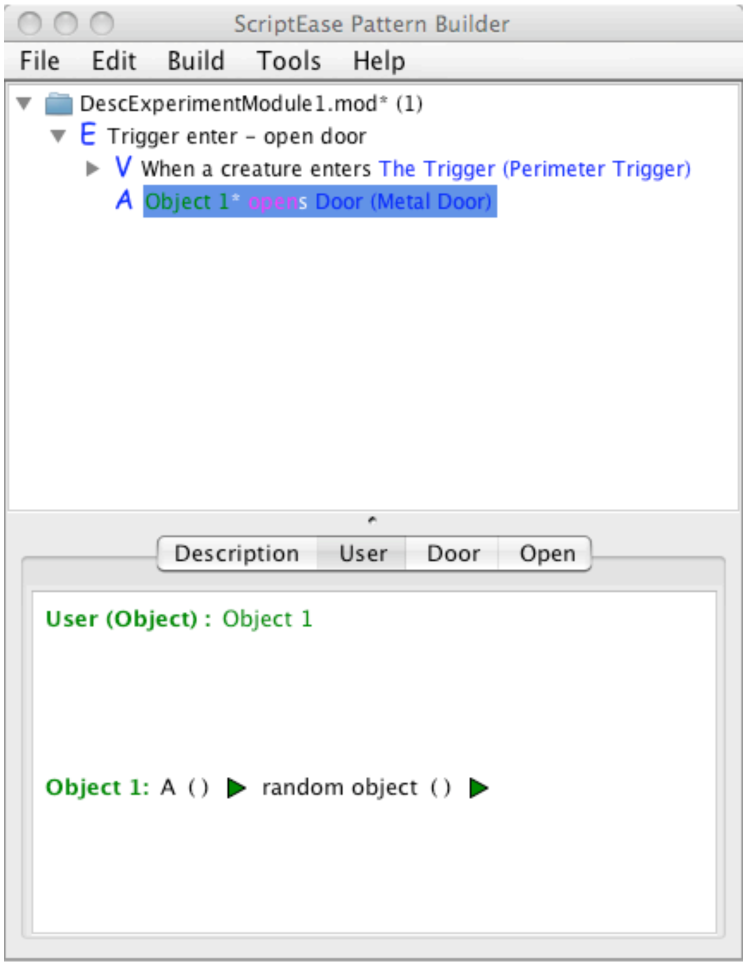


ScriptEase Pattern Builder
File Edit Build Tools Help
Description User Door Open

DescExperimentModule1.mod* (1)
Trigger enter – open door
When a creature enters The Trigger (Perimeter Trigger)
<invalid>* opens Door (Metal Door)

Open (Binary) : open

Finally, we need to set the User to be a random object. First click on User tab. Because we want to set the value to something more abstract – we don't know what creature, just that it's a random one – we need to describe the User. So this time after clicking on <invalid> choose to *describe it*. Here you will notice you have a set of choices similar to the *recall it* menu. Since we want a random object, let's choose that menu item. The bottom pane should now look like the picture below.

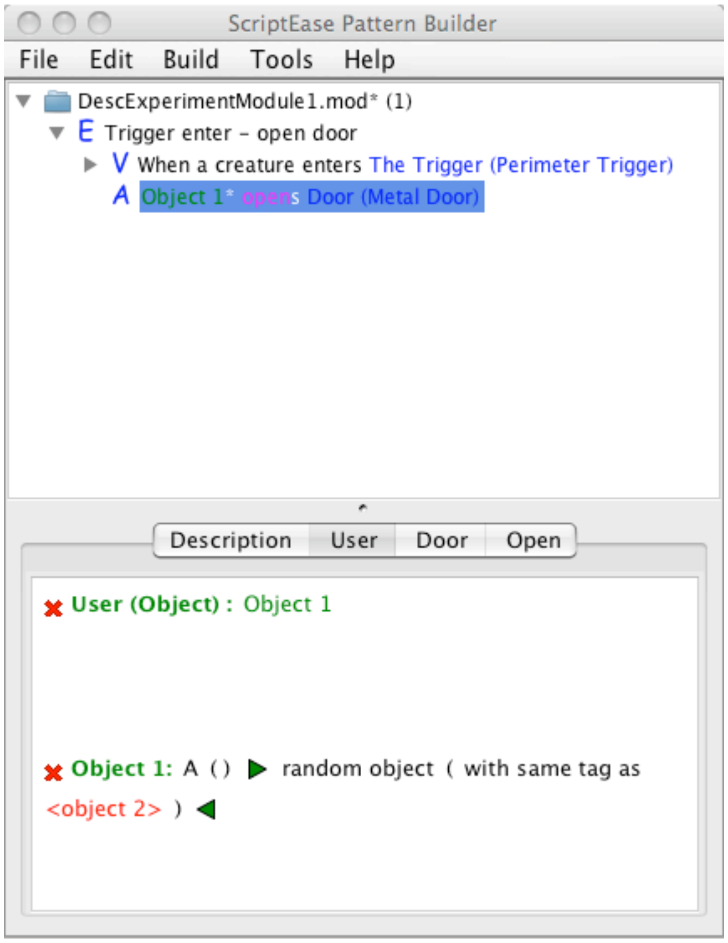


DescExperimentModule1.mod* (1)
E Trigger enter – open door
V When a creature enters The Trigger (Perimeter Trigger)
A Object 1* opens Door (Metal Door)

User (Object) : Object 1

Object 1: A () ▶ random object () ▶

We're not quite done, as the statement specified that the random object has to have the same tag as Bob. You should see two green triangles next to two sets of brackets. These green triangles allow you to expand the brackets to change the statement. If you click on them, you should notice that each shows an optional descriptive phrase *different* and *with same tag as*. Since we want to set the same tag, we will use the second triangle and add *with same tag as*. The line should now have expanded to look like below.

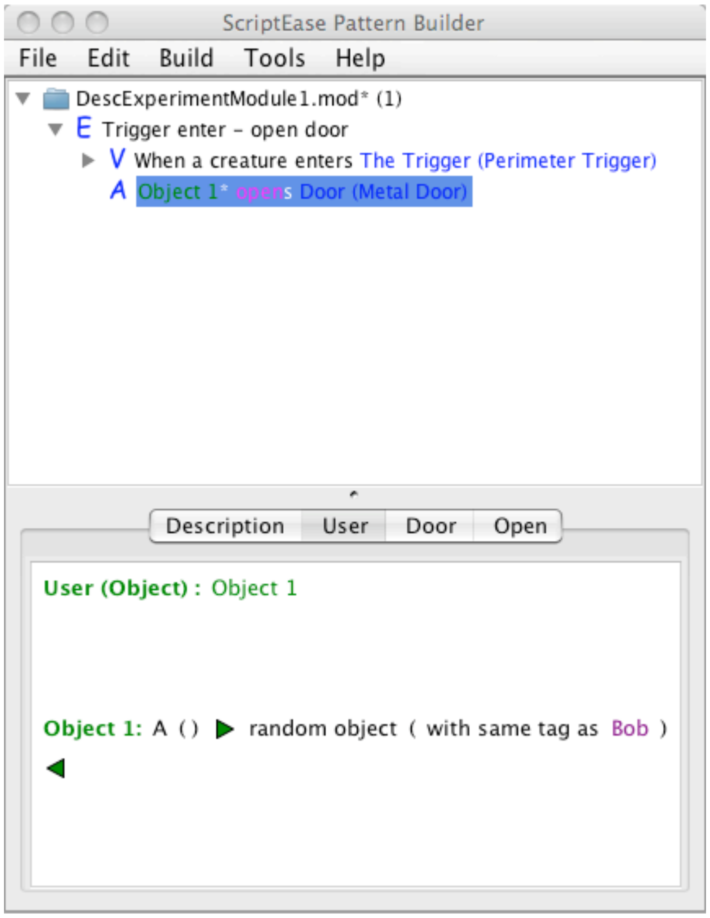

ScriptEase Pattern Builder
File Edit Build Tools Help
Description User Door Open

DescExperimentModule1.mod* (1)
E Trigger enter - open door
V When a creature enters The Trigger (Perimeter Trigger)
A Object 1* opens Door (Metal Door)

User (Object) : Object 1

Object 1: A ( ) ▶ random object (with same tag as <object 2>) ◀

The last thing we need to do is set the new option that appeared in the statement <object 2> Since we know we want the same tag as Bob, we can use *pick it* to find him in the creature section. The statement should be complete, and the red x's should have disappeared from both the bottom pane and the line in the top pane. It may continue to show the red x's on the first two parts of the tree, but if you select the very top line they should all disappear. If they do, then that means all options have been set and you are done.

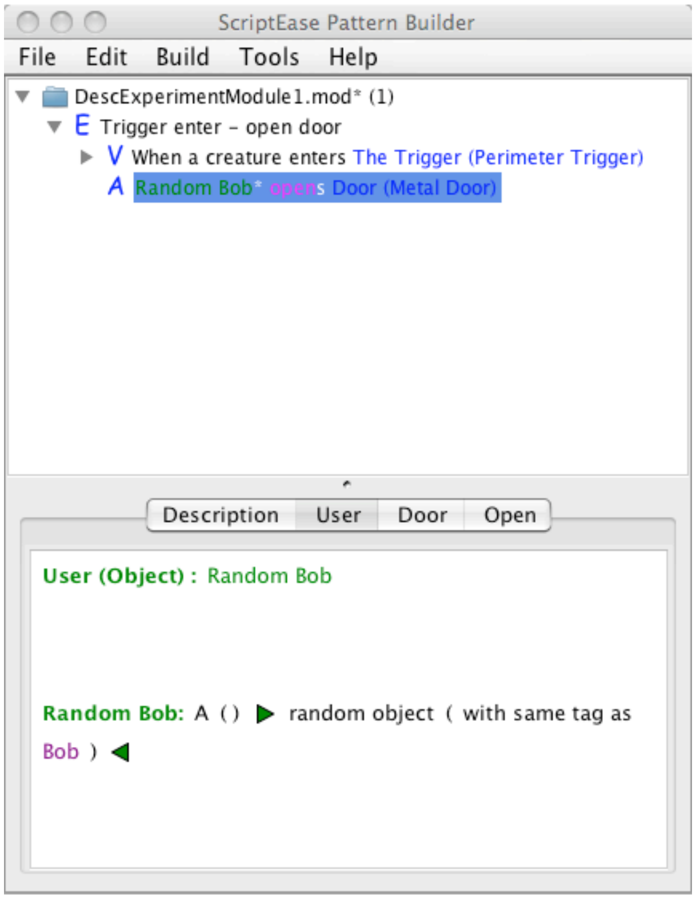


DescExperimentModule1.mod* (1)
E Trigger enter – open door
V When a creature enters The Trigger (Perimeter Trigger)
A Object 1* opens Door (Metal Door)

User (Object) : Object 1

Object 1: A ( ) ▶ random object (with same tag as Bob) ◀

Notice that the line in the top pane now reads Object 1* opens Door (Metal Door). *Object 1* is not very descriptive. We can change this text by clicking on the *Object 1:* line (second line) in the bottom pane. This will display a menu item to change the line label. Let's do so and set it to *Random Bob*. Click *ok* and now the line in the top pane makes more sense. "Random Bob* opens Door (Metal Door).



DescExperimentModule1.mod* (1)
E Trigger enter - open door
V When a creature enters The Trigger (Perimeter Trigger)
A Random Bob* opens Door (Metal Door)

User (Object) : Random Bob

Random Bob: A ( ) ▶ random object (with same tag as Bob) ◀

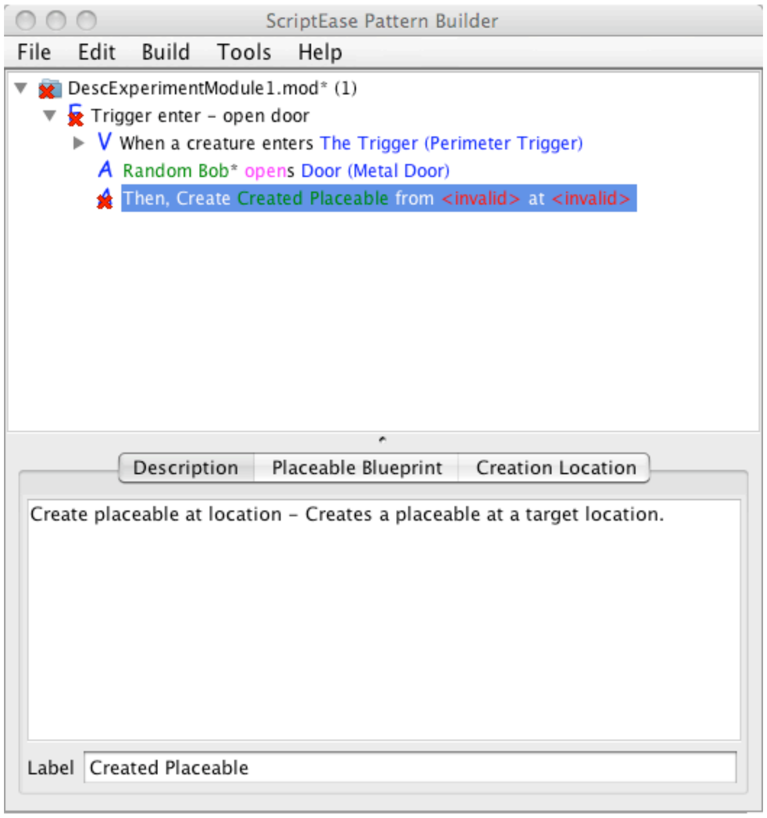
Now that you've finished working through the first statement, record your time. The rest of the statements do not provide a walkthrough. If you get stuck, please raise your hand and we will assist you.

Time Finished

Save module, and then open next module 32ADesc2.mod. (Change 32 to your participant number)

Statement 2: *Then, Bob creates a chest 2.5m in front of the bench. Name the location Chest Location.*

Note: Chests are found under placeables.



ScriptEase Pattern Builder

File Edit Build Tools Help

DescExperimentModule1.mod* (1)

Trigger enter – open door

When a creature enters The Trigger (Perimeter Trigger)

Random Bob* opens Door (Metal Door)

Then, Create Created Placeable from <invalid> at <invalid>

Description | Placeable Blueprint | Creation Location

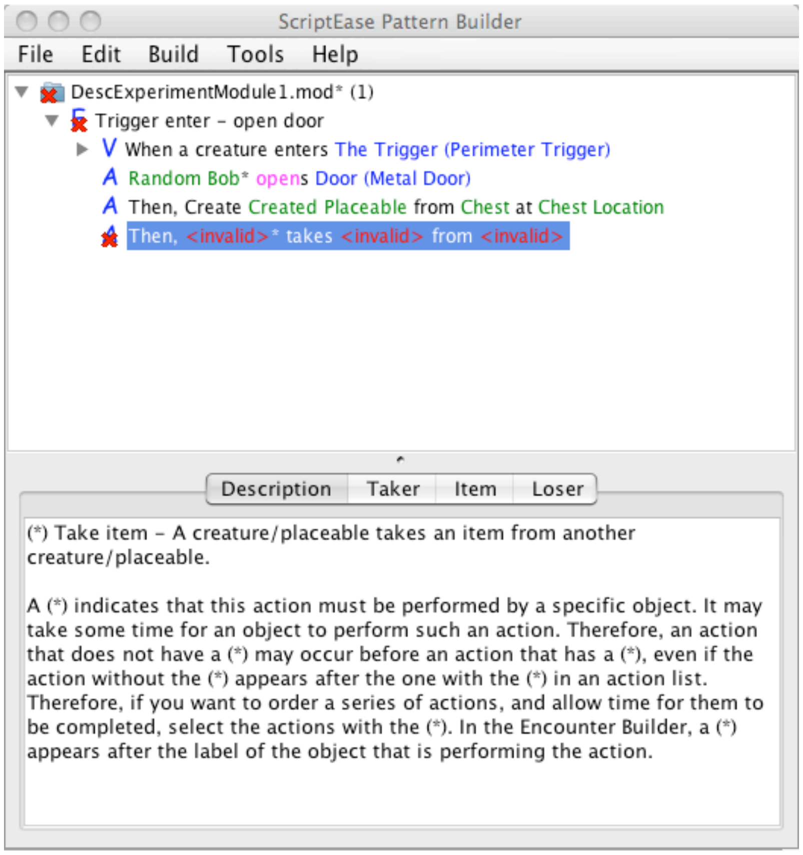
Create placeable at location – Creates a placeable at a target location.

Label: Created Placeable

Time Finished

Save module, and then open next module 32ADesc3.mod. (Change 32 to your participant number)

Statement 3: *Sally then takes a the Potion of Owl's Wisdom from the chest created in the previous statement.*


ScriptEase Pattern Builder
File Edit Build Tools Help
Description Taker Item Loser

DescExperimentModule1.mod* (1)
Trigger enter – open door
V When a creature enters The Trigger (Perimeter Trigger)
A Random Bob* opens Door (Metal Door)
A Then, Create Created Placeable from Chest at Chest Location
Then, <invalid>* takes <invalid> from <invalid>

(*) Take item – A creature/placeable takes an item from another creature/placeable.

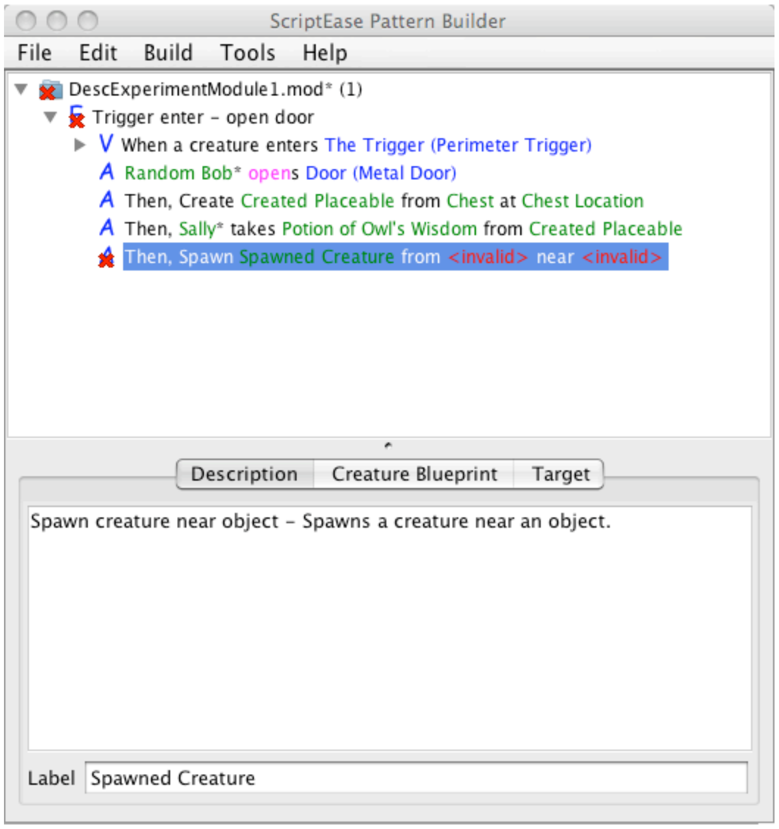
A (*) indicates that this action must be performed by a specific object. It may take some time for an object to perform such an action. Therefore, an action that does not have a (*) may occur before an action that has a (*), even if the action without the (*) appears after the one with the (*) in an action list. Therefore, if you want to order a series of actions, and allow time for them to be completed, select the actions with the (*). In the Encounter Builder, a (*) appears after the label of the object that is performing the action.

Time Finished	

Save module, and then open next module 32ADesc4.mod. (Change 32 to your participant number)

Statement 4: *Then, the chest spawns a penguin near the block of ice that is x closest to the pc. X is determined by a roll of a 4 sided dice.*

Note: Ice Blocks are found under placeables.



DescExperimentModule1.mod* (1)
Trigger enter – open door
V When a creature enters The Trigger (Perimeter Trigger)
A Random Bob* opens Door (Metal Door)
A Then, Create Created Placeable from Chest at Chest Location
A Then, Sally* takes Potion of Owl's Wisdom from Created Placeable
A Then, Spawn Spawned Creature from <invalid> near <invalid>

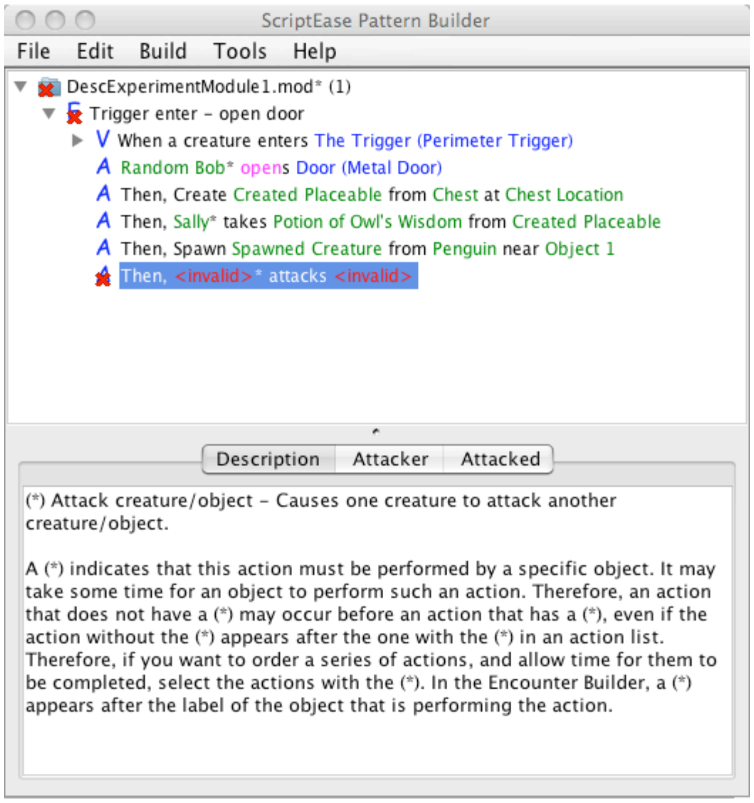
Spawn creature near object – Spawns a creature near an object.

Label: Spawned Creature

Time Finished

Save module, and then open next module 32ADesc5.mod. (Change 32 to your participant number)

Statement 5: *The spawned penguin then starts attacking the x nearest object to the Enterer with tag Chicken. X is the total number of npcs in the City Area minus 3. Name the attacked Chicken.*


ScriptEase Pattern Builder
File Edit Build Tools Help
Description Attacker Attacked

DescExperimentModule1.mod* (1)
Trigger enter - open door
When a creature enters The Trigger (Perimeter Trigger)
Random Bob* opens Door (Metal Door)
Then, Create Created Placeable from Chest at Chest Location
Then, Sally* takes Potion of Owl's Wisdom from Created Placeable
Then, Spawn Spawned Creature from Penguin near Object 1
Then, <invalid>* attacks <invalid>

(*) Attack creature/object - Causes one creature to attack another creature/object.

A (*) indicates that this action must be performed by a specific object. It may take some time for an object to perform such an action. Therefore, an action that does not have a (*) may occur before an action that has a (*), even if the action without the (*) appears after the one with the (*) in an action list. Therefore, if you want to order a series of actions, and allow time for them to be completed, select the actions with the (*). In the Encounter Builder, a (*) appears after the label of the object that is performing the action.

Time Finished

Save module and close ScriptEase.

Appendix B

Survey

Post-Study Questionnaire

Participant Number: ______

Age: ___

Gender: ___

Current Degree and Major: ____

Year of Study: ______

How regularly do you play video games? (check one)

Never □ Once a month □ Once a week □ Several times weekly □ Daily □

How many computer or console role-playing games (RPGs) have you played? (check one)

0 □ 1 □ 2 □ 3 □ 4-6 □ ≥7 □

Have you written computer programs before? (check one)

Yes - lots □ Yes - little □ No - Never □

How often do you use a computer? (check one)

Once a month □ Once a week □ Several times weekly □ Daily □

What do you use the computer for? (check all that apply)

Word processing (ex. Microsoft Office) □ Playing games □ Email □

Social Networking (ex. Facebook) □ Internet □ Multimedia (videos/music) □

For the following section you are comparing the two different versions of ScriptEase – the Describer and the Definer. For each question, use the scale to select which tool you think the statement apply to best and circle your choice.

Easier?
Describer Better … Slightly Better … The Same … Slightly Better … Better **Definer**

Faster?
Describer Better … Slightly Better … The Same … Slightly Better … Better **Definer**

More intuitive?
Describer Better … Slightly Better … The Same … Slightly Better … Better **Definer**

Overall Preference?
Describer Better … Slightly Better … The Same … Slightly Better … Better **Definer**

Comments (optional):

Appendix C

Debriefing Document

Debriefing

Is it more intuitive to describe versus define in ScriptEase?

Thank you for participating in this study. As I'm sure you're aware, video games are a growing market. More recently, video games have started providing tools to the player allowing them to become a designer and design their own levels. The more sophisticated tools allow the designer to create games with intricate storylines. However, in order to successfully use these tools, the designer usually needs to have a programming background as the control of the characters and objects in the game are in a series of scripts. This means that a lot of designers can't create their own games without going through a steep learning curve of teaching themselves to program.

ScriptEase has been designed with the designer in mind. The goal behind ScriptEase is to make video game design accessible to all users regardless of programming experience. However, there is only so much simplification that can take place before the amount of customization is impeded. This means that some areas of complexity have to be retained to allow the designer full control.

The describer and the definer are an example of such an area. When choosing what to attach a script too, a user shouldn't be limited only to the choices they can pick from a list. A fun game has dynamic capabilities where not everything is completely decided ahead of time.

For example, if the player character walks into a room that has three guards randomly wandering the room. In the middle of the room is a chest which the guards are guarding. If the player character starts to approach the chest, it makes more sense for the guard that is closest to the player character to approach and warn the player character then it does for the guard farthest away. In order to do this, the guard that is assigned to approach the player character should be chosen while the game is running, not ahead of time as we don't know which guard will be closest. To set this in ScriptEase we need to define or describe it instead of statically setting it to a specific guard.

At the end of the day, we are trying to find how to keep all the complexity while staying accessible to non-programmers. This experiment is looking at the Definer and the Describer. We want to see which version is easier to use to complete the same task. To do so all participants were given one of four scenarios:

1) Use the Definer with Test A, then the Describer with Test B
2) Use the Definer with Test B, then the Describer with Test A
3) Use the Describer with Test A, then the Definer with Test B
4) Use the Describer with Test B, then the Definer with Test A

The four scenarios are set up so that when we are analyzing the results, we can check to see if experiment A or B was inherently easier, and if the order of using the two tools made any difference in overall preference.

Our hypothesis is that a describer will be easier to use than a definer. We believe that the definer allows the user too many options making it easier to get lost and make mistakes when trying to define a statement. The describer on the other hand limits the set of choices the user sees based on the return type a particular option needs. This should help guide the user to the correct solution.

Once again, thank you very much for participating. If you have any questions about the study you can contact Neesha Desai via phone (780 492-3725) or email (neesha@cs.ualberta.ca) or Dr. Duane Szafron via phone (780 492-5468) or email (duane@cs.ualberta.ca).

If you have questions about your research participation, please e-mail your questions to the Research Participation Coordinator at rescred@ualberta.ca, or you may contact Dr. Tom Johnson (Director of the Research Participation Program) at 492-2834.