

UO-The Expanse Quest Package 6.5.2.32869

written by Raist

What is Quest?

Quest lets you make interactive story games. Text adventure games like Zork and The Hitchhiker's Guide to the Galaxy. Gamebooks like the Choose Your Own Adventure and Fighting Fantasy books. You don't need to know how to program. All you need is a story to tell. Your game can be played anywhere. In a web browser, downloaded to a PC, or turned into an app.

Website: <http://textadventures.co.uk/quest>

Tutorial: <http://docs.textadventures.co.uk/quest/tutorial/>

Tutorial (local): [Quest Tutorial](#)

Tutorial (Quest Compiler): [Quest Compiler](#)

Custom Libraries: [Libraries](#)

UO-The Expanse Quest Package:

<https://github.com/tass23/QuestPackage6.5.2.32869>

UO-The Expanse OQS: <https://www.uoexpanse.com/quests>

I was a big fan of Choose Your Own Adventure books when I was younger, and I got involved with Quest Text Adventures as a means to transcribe my books into CYA-style stories. I worked out a basic game to get a feel for the software, and had to decide if I wanted people to have to download the Quest app to play my games or if I wanted to convert them into an HTML file. I thought this would be a fun way to experience UO adventures, and after getting help from KVonGit, we ironed out some functions to work with the conversion tool QuestJS (now called Quest Compiler), and then I built the Offline Quest System for the UO-The Expanse website. After I released the edu Repack, I decided to also include the OQS as an optional tool for everyone to use.

This required me to build new libraries that streamlined the creation process by including a few spells, items, and Mobs. I highly recommend walking through the tutorial on the Quest website before stepping into the OQS.

The Quest tutorial is here: Tutorial:

<http://docs.textadventures.co.uk/quest/tutorial/>

Tutorial (local): [Quest Tutorial](#)

Tutorial (Quest Compiler): [Quest Compiler](#)

Custom Libraries: [Libraries](#)

UO-The Expanse Quest Package 5.8.6752 on Github:

<https://github.com/tass23/QuestPackage6.5.2.32869>

(does not include any completed games)

To play UO-The Expanse OQS: <https://www.uoexpanse.com/quests>

The Interface: Quest has a nice UI to handle all your creation needs, with QuestJS serving as the conversion tool you can use later for the HTML file. On the left side of the window are all the Objects, Functions, Commands, and Timers that are created for that game. On the right side is the detail pane for whatever is selected on the left. Notice the hierarchy of the overall game itself.

The *Game* is an Object that can have attributes, but everything is created *inside* the Game. Adding Libraries for Spells and Items are added to the Game, but exist outside of the Game file (libraries have to be stored in the same folder as the Game they are added to). An "expression" is a coded text, like *This box of crayons has 24 **+{object:crayons:crayon1}+** inside*. Otherwise this would just be a "messages" like, *This box of crayons has crayons inside*. Using expressions can make things players can interact with inside text blocks.

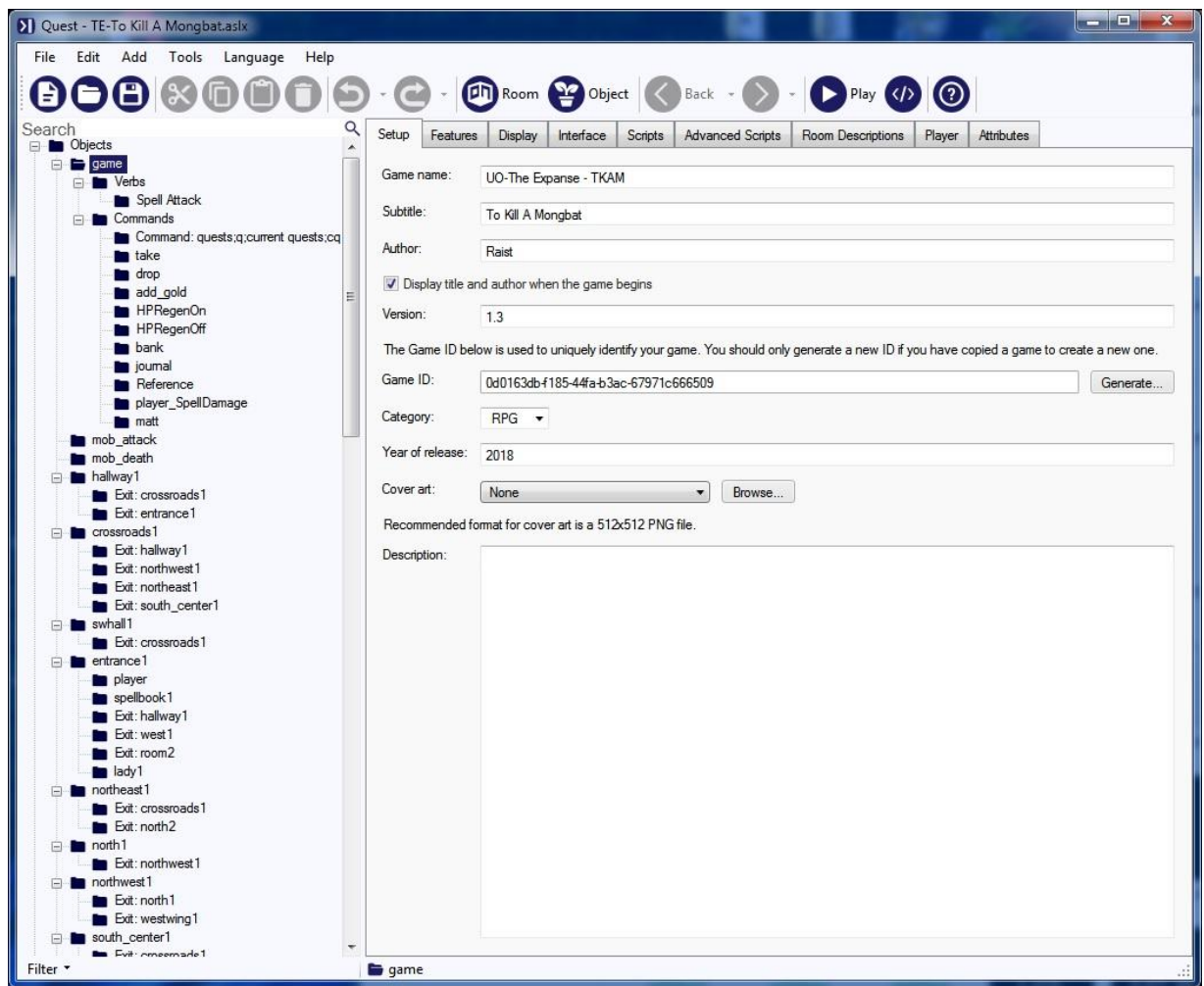
Objects vs Everything else An Object is something created when the game is launched that the player can interact with or that operates like scenery (you can *Look at* the Object, but you cannot pick it up). A Room is also an Object, but is a special type of Object that contains other Objects, like a box of crayons has individual crayon Objects inside the box.

Verbs are used to shortcut actions for players, like *Attack mongbat*, and are usually limited to an expression.

Commands are things the player types into the command bar, or when they use a hyperlink to interact with an Object in the game, and can be built with a script. Buy Object/Sell Object, that sort of thing.

Functions cause something to happen that the player does not control, but can cause. Like walking into a room, the player can use the light switch, and a function is turning on the light.

Timers are what you might expect; they use seconds to count time, and can be started when the Game starts or by a Command, a Verb, Script, or Function.



Rooms do not have to have exits, but if there is no way for the player to leave the room, they can't leave the room. Exits can be visible/invisible, locked/unlocked, have descriptions, or be "Look" only. Exits can be *one way*, meaning once you leave that room, you cannot go back.

Clones are copies of Objects. You cannot clone a Room, or command, or function, or verb, only Objects. When an Object is cloned, the original usually remains in the room it started in, and the clone would then go with the player. A "template" long sword the player buys, has specific attributes that are given to the clone that can be different from the template long sword. This is most evident with resistances on Objects for UO.

Turn Scripts are used every turn, which means after every time the player uses a Command. This is helpful for things like mob attacks that will happen automatically after every turn.

Libraries are used to store specific data that can be used in multiple games, like Spells, Items, Help files. Libraries can contain complex aspects like Game properties, Functions, Verbs, and Commands.

Object Property Tabs contain general information, and specific attributes for each Object in your game. Each of these tabs servers a different function for the Object.

Setup begins with basic information, name and alias of the Object, if the Object should be Scenery, Visible/Invisible, and the Look information.

Object handles Object aliases/abbreviations, specific Link Color, Generate/Disable automatic Verbs.

Inventory is where Take and Drop for the Object are controlled.

Verbs are where custom verbs are added with individual Assignments.

Features are specific actions the Object has, like Use/Give, is it a Container, is it Switchable, is it Edible, is it Wearable, does th Object light up the room, can the player become this Object, does this Object need initialization.

Use/Give only shows up if the Use/Give box is checked under Features, and sets up specific Uses and Gives.

Ask/Tell all Objects have Ask/Tell by default, so even a toaster can have dialogue.

Attributes houses various default variables, and custom variables of different types: String, Bool, Integer, Double, Script, String List, another Object, a Command pattern, String Dictionary, Script Dictionary, or Null.

Objects displays Objects *inside* this Object, like a container, but not all containers must be defined as a container. It depends entirely on the usage. A container like a Quiver would have arrows with a quantity and does not *have to be* a container, but something like a Backpack that can hold multiple types of Objects should be defined as a Container under the Features tab.

Room Property Tabs contain general information, and specific attributes for each Room in your game. While similar to the Object tabs, there are a few differences, such as:

Light/Dark which allows you to set a room as *initially* dark, and set up a description script of what happens when the room is dark.

Exits are as you might guess; ways to leave the room. A compass with 8 points offers 8 exits, plus In/Out and 2 None exits. Once an exit is defined for a Room, an *Object* is created for the Exit itself, and those have specific tabs.

Scripts can be defined as events that happen *Before entering, After entering, After leaving, Before entering for the first time, After leaving the room for the first time*, and finally *Turn scripts that happen while inside that room* (as opposed to *game* Turn Scripts).

Exit Property Tabs are limited to 3.

Exit being the general information about the Exit, Locked/Unlocked, Visible/Invisible.

Options defines the Exit as a Light Source or not.

Attributes houses the default and custom variables for that Exit.
Scripting in Quest can be done with JavaScript, or using the Quest interface.

CUSTOM LIBRARIES

These have all been combined into the TE-RPG_Lib, along with new functionality that is described inside the library file.

Spells Lib - Words of Power, based on UO, in the form of Commands. In general, players must type *spell name object*, where *spell name* is the *Command* and *Object* is the *Target* of the *Spell*. Includes a popup window for displaying the "Spellbook" with the Uses Remaining and the Words of Power. This popup is only available IF the player is carrying the Spellbook, and the same can be said for using the Words of Power (they are only available if the player is carrying the Spellbook). Spells/Words of Power: Each spell accounts performs a check against the spell target to determine if that spell can be cast on it (mob=True/False). The next check is for resistance. UO has the following resists/damage types: Physical Fire Cold Poison Energy Each spell fits into a primary damage category, i.e. Vas Flam is Fireball, so it does Fire damage. If a mob has Fire resist lower than X, the spell is cast, does damage, and the mob's health drops accordingly. Otherwise, the spell simply fails. This system could be adjusted to subtract a percentage based on the resist value, which would allow for more variations of damage. However, since each spell has a variable damage range, a single resist check functions more easily.

TEHelp Lib - The Expanse Offline Quest system complex HELP system.

TEItems Lib - Weapons, based on UO, complete with magical properties. Incorporated into the Save/Load system which creates hash codes for Items to be Copy/Pasted to a text file to save it. Weapon Room contains ALL available weapons. Default images loaded for each weapon and weapon types are defined. NOTE: Additional properties have been added for Game Rooms, Mobs and Player, including Status attributes.

TEMainSysLib - The Expanse Offline Quest System Main Library contains commands, functions and other universal data needed for the OQS to function. The following is a mandatory list of Attributes to be added to objects and which type of objects receive which Attributes. *Rooms*

1. cycle INT - Used for handling respawns and increasing difficulty of respawns.
2. map & map_zoom STR - Used for handling the Map library and displaying images for each room.
3. mlist STRLIST - Used for handling the list of available targets to attack in a room.
4. Scripts - OnExit: ClearScreen, Enter: targets
5. Add Supply Shop exits to Main Room

NPC Questers

1. avquests INT - Displays Total Available quests.
2. tquests INT - Displays Total Quests for that NPC.

Mobs

1. mob type INT - Used to determine respawns and target functions.
2. lootable INT - Used to generate Loot after death. Does not work with Dispelled mobs.
3. status STR - Displays current mob status in the mob window when LOOKed at.
4. desclive STR - Message to display when mob is Alive.
5. descdead STR - Message to display when mob is dead.
6. img_dead STR - Direct path online to dead mob image.
7. defense type INT - Used during combat for Resists.

TEMap_Lib - Map library to use generic code to call pre-defined attributes on Rooms (room.map and room.map_zoom). room.map image should be full size map of that room or the whole area. room.map_zoom should be a scaled section of the main map that matches that specific room or area. Change display text as needed to match each map. Construct one map library for each game.

NPC_Lib - Easy NPC. Allows for the player to select an additional helper to join them. Complete with STATS PANE visible on the Right side, under the player STATS. Choose from Mage or Archer classes.

QUEST LIBRARIES

Quest_Tracker_Lib -Used to create Quests and steps of Quests, as well as Rewards. Can be used to house multiple Quests.

Score_Lib - Used to define the Score of the game, broken down by each call of the IncScore function to increase or decrease the total Score. Offers two displays of Score Results.

This section is all about *How do I?* or *Here's what I did*. As Quest is coded in JavaScript, some people will be more or less comfortable scripting. Lean into the Quest interface until you feel more comfortable. Here are some examples taken from the Custom Libraries

SPELLS LIB Spells All Spells are Commands, and as such must be defined as a Command with the proper Command-related layout. The structure for a Command item is: Name -Pattern -Scope -Script The first Spell in the Spells_Lib is Vas Flam. This Command is executed when the player types it into the command bar, or use the Spellbook to select a Spell and then select a Target. The Command is built like this: Name: vas flam Pattern: vas flam #object# Scope: blank Script: (see - VasFlam.xml)

Timers All Timers count time in seconds, and can begin counting when the game starts, or they can be started by outside scripts, commands, verbs, objects, and functions. The structure for a Timer is: Name -Start timer when game starts - Interval -Script The first timer in Spells_Lib is poison_tick1.

As you can imagine this Timer is executed when a mob, or the player are poisoned. Since the player has a poison spell, In Nox, that Spell can start the Timer too. The Timer is built like this: Name: poison_tick1 Start timer when game starts: not checked Interval: 5 (seconds) Script: (see - Poison.xml)

NPC LIB Function InitUserInterface is used to build a display when the Game starts, before the player has a turn. The NPC_Lib was specifically built in order to add a character that would fight along with the player. The structure of the function is like this: JS.eval defines a span tag for an attribute on the NPC. s is used as a variable to build the actual Stats Pane. Each value of s is added to the next. JS.setCustomStatus (s) is called to process all the s entries. If a Stat/Attribute on the NPC is changed, a script is called to change the value in the Pane. To get all these values, we have to assign them somewhere, and we do that with each function that creates the type of NPC the player picks; Mage (function npcmage1) or Archer (function npcarch1). That structure goes like this: set an attribute on the NPC. JS.eval defines the SPAN tag for the attribute a Script is added for each attribute that is supposed to change during the Game. To set up the InitUserInterface, we begin with:

```
JS.eval ("$('#weapon-span').html('' + lady3.weapon.alias + '');")
```

Then the first s value that begins the HTML Table:

```
s = "<div id='status_div' style='padding:5px;padding-top:10px;border-radius:5px;border:grey solid thin;'>PARTNER"
```

Subsequent s values add the previous s value:

```
s = s+"<table width='100%'><tbody id='status_npc'>"
```

We call:

```
JS.setCustomStatus (s)
```

to process all the s values. Then the IF statement says if the script changedweapon is attached to the NPC, it must process the script changedweapon.

```

    if (HasScript(lady3, "changedweapon")) {
        do (lady3, "changedweapon")
    }

```

Lastly we secure the NPC Stats Pane location:

```

JS.eval ("$('#status_div').insertAfter($('#compassAccordion'))")

```

This finishes off the InitUserInterface function.

Next, we setup the Stats Pane and define the Span values for the HTML tags by using one of two functions: `npcmage1`, or `npcarch1`, for mage or archer respectively, which is chosen by the player when the game begins. We will use `MAGE` for the example. Since the Stats Pane must be generated before the Game starts, we have a conundrum, in that we can't get the correct Stats for the Pane until after the Game starts, so we must define "fake" values as placeholders inside `InitUserInterface` first. Now we can move into the correct values being updated after the player makes their choice. Firstly we set the attribute:

```

set (lady3, "weapon", spellbook2)

```

Then we use `JS.eval` to define the Span tag.

```

JS.eval ("$('#weapon-span').html('' + lady3.weapon.alias + '');")

```

Finally we define what `changedweapon` script does for this attribute:

```

lady3.changedweapon => {
    JS.eval ("$('#weapon-span').html('' + lady3.weapon.alias + '');")
}

```

Now when the Game begins, there is a Stats Pane on the Right side, under Places and Objects, which says `PARTNER: Martika`, and then all the values/attributes that we wanted to display are 0.

After the Game begins and the player has made their choices, all the "changed" scripts attached to each attribute are then called and each value is updated in the Stats Pane.

TEITEMS LIB

Items are Objects, but these particular Objects can be "equipped" by the player as weapons and that increases the player's damage, gives bonuses to resistances, hit chance, and Elemental attacks. However, if the player is not carrying their Spellbook, they can't cast Spells. Creating an Item the player can equip as a weapon follows normal Object structure, with these additions as attributes: pic type -img -addy -src -f_type -wtype -m_props -AosStrReq -AosMinDam -AosMaxDam -AttackChance -BonusDex -BonusHits -BonusInt -BonusMana -BonusStam -BonusStr -Chaos -Cold -DefendChance -Direct -Energy -Fire -HitCold -HitDispel -HitEnergyArea -HitFireball -HiteHarm -HitLeechHits -HitLeechMana -HitLeechStam -HitLightning -HitLowerAttack -HitLowerDefend -HitMagicArrow -HitPhysicalArea -LowerManaCost -Luck -MageWeapon -NightSight -Physical -Poison -ReflectPhysical -RegenHits -RegenMana -RegenStam -ResistColdBonus -ResistColdBonus -ResistEnergyBonus -ResistFireBonus -ResistPhysicalBonus -ResistPoisonBonus -SpellChanneling -SpellDamage -WeaponDamage -AttackChance2 -BonusDex2 -BonusHits2 -BonusInt2 -BonusMana2 -BonusStam2 -BonusStr2 -Chaos2 -Cold2 --DefendChance2 -Direct2 -Energy2 -Fire2 -HitCold2 -HitDispel2 -HitEnergyArea2 -HitFireball 2-HiteHarm2 -HitLeechHits2 -HitLeechMana2 -HitLeechStam2 -HitLightning2 -HitLowerAttack2 -HitLowerDefend2 -HitMagicArrow2 -HitPhysicalArea2 -LowerManaCost2 -Luck2 -MageWeapon2 -NightSight2 -Physical2 -Poison2 -ReflectPhysical2 -RegenHits2 -RegenMana2 -RegenStam2 -ResistColdBonus2 -ResistColdBonus2 -ResistEnergyBonus2 -ResistFireBonus2 -ResistPhysicalBonus2 -ResistPoisonBonus2 -SpellChanneling2 -SpellDamage2 -WeaponDamage2 -Wearable (true) -Wear_slots (lefthand) -Initialize (magical properties) All these properties are repeated on the player too. (see – AssassinSpike1.xml)

SaveObject Allows players to save Weapons that are part of the TEItems library. This means that any random Weapon that a player picks up from a corpse,

can be saved as a local text file, and then loaded back in a brand new game. SaveObject has a lot more code than LoadObject does, because SaveObject must grab *all* of the attributes and concatenate them (join them) along with their corresponding values. LoadObject takes the SaveObject and loads it based on an existing type of weapon. Like an AssassinSpike could be saved as MySpike, and loaded into any other game that allows Save/LoadObject. (see SaveObject.xml)

LoadObject Contains a check to see if we are loading an Object from saved data or not, then loads the text from the file, sets it to be Saveable again, sets it to scenery False, and moves the Object to the player. (see LoadObject.xml)

This Save/Load process can be automated using generic information, but the player still has to Copy/Paste the data, so it works best to **hide** the fields the player does not need to change and instruct them on the steps. This would also work for things the player does not carry, like scenery Objects, or Rooms, or Exits. Just remember: it works best when there is a **template** Object already in the Game.

Making Libraries One nice feature of Quest is being able to make custom libraries to save you time in the future. A library has a basic structure; it has an opening tag <library> and a closing tag </library> outside of everything else. *That's it.* One thing to watch for when making a library is repeated: Objects, Commands, Verbs, Functions. These things only need to exist in *one place*. Every aspect of a library is loaded into the Game it is used in, and that is why you want to avoid repeating things.

The downside to using a Library is that Quest cannot edit them. You have to use Notepad or Notepad++ to read the aslx file. However, you can build a *whole Game* in Quest, open the aslx file in whateverPad, add the opening and closing library tags and you're done! The next step is opening the Game the library will be added to, scroll to the bottom under *Advanced*, expand the +, go to Included Libraries, click the + on the Right side. This will open another window that allows you to select the library to include.

Libraries should exist in the same folder as the Game they are included with. Best practice, but if the libraries are stored in the same folder for all Games, that will work too.

If you edit a library while a Game is open in Quest that you are editing, Quest will display a message at the top of the window in a yellow bar saying: X.aslx has been modified outside of Quest, where X is the name of the library file that was modified, and tell you Reload because the library cannot be updated in the Game until it has been reloaded. If you Save and Close, when you reopen that Game, the library will be updated. Or you can Reload, but if making multiple changes, you will have to reload every time. Easier to Save and Close when all library edits have been made, or before the Game is opened in Quest.

At the top of a library file, but inside the opening <library> tag, you can include a Comment section to describe the purpose of the library by using this structure: <!-- This is your comment text. --> Anything inside the arrows can spread across multiple lines for as much as you need, provided that the closing arrow is *after* the last word of the comment.

Tutorial Introduction *(QuestTextAdventures.uk website)*

Introduction

Quest is a program for writing text adventure games and gamebooks (both of which are sometimes referred to as Interactive Fiction).

There are two versions:

- a web version which runs entirely in your web browser (Chrome, FireFox, Internet Explorer, Safari), without downloading any software.
- a Windows version, downloadable for Windows 7, 8, 10 and Vista.

This tutorial applies to both versions. The versions are fundamentally the same, although the Windows version of the game editor has a few more features.

What is a text adventure?

Text adventure games were the earliest type of computer game, from a time when computers could only display text - there were no graphics, so everything was described with text. You would play the game by typing commands with the keyboard such as “go north” or “hit troll”. Quest lets you make this kind of game - you can include graphics now though, and play the game by clicking hyperlinks instead of having to type everything.

Why create text adventures

Here are some reasons why interactive text games are great:

Interactive text games are easy to create You don't need to have a team of people creating graphics, music and sound effects. You don't even need any programming experience. If you've never created a game before, a text game is the easiest and quickest way to start. This doesn't mean that it's trivial - creating a good game, like creating a good novel, takes a lot of effort - but you don't need to have any special tools or expertise to start.

Interactive text games are accessible You don't need fast reactions to play a text-based game. In fact, you don't even need to be able to see - text-based games are one of the few types of games that the visually impaired can enjoy, using a screen reader to speak the text aloud. You don't need to have a particular type of computer - you can play a text game using nothing more than a web browser. All of this means that a text game can be played by just about anybody.

Using Quest, you can play and create text-based games, which can include pictures, sounds and video. To play some games which people have created already, see textadventures.co.uk.

If you have some time to spare, it's well worth watching the documentary Get Lamp (also on YouTube) - it's a brilliant telling of the history of text adventure games. (no longer available)

You can find another great introduction for beginners at Brass Lantern.(not updated since 2010)

Programming without Programming

Quest is a powerful system with a gentle learning curve - you can get started very easily without doing any programming at all, and build up from there. The point and click editor means there's no need to remember syntax, type in strange punctuation or even remember commands. But there is a lot of power underneath - a full programming language in fact. You never need to see any code to access the full power of Quest, but it includes a "Code View" feature so it's there if you need it.

[back](#)

Let's Begin

You cannot convert a gamebook into a webpage.

1. Creating a simple [text adventure](#)
2. Creating a simple [game book](#)

1. Creating a simple game

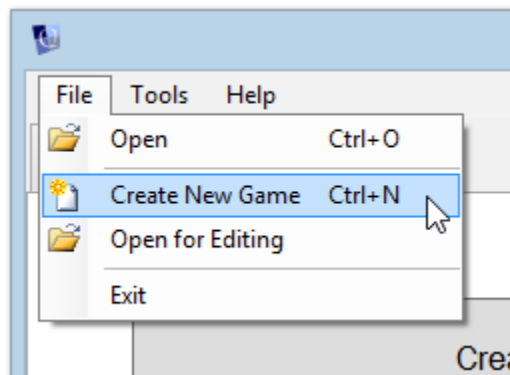
This tutorial is applicable to both the Windows desktop version of Quest, and the web version. Any differences in the two versions will be mentioned as we go along.

This tutorial guides you through creating your first text adventure game. If you want to create a gamebook instead, see [Creating a gamebook](#).

Creating a blank game

Windows desktop version

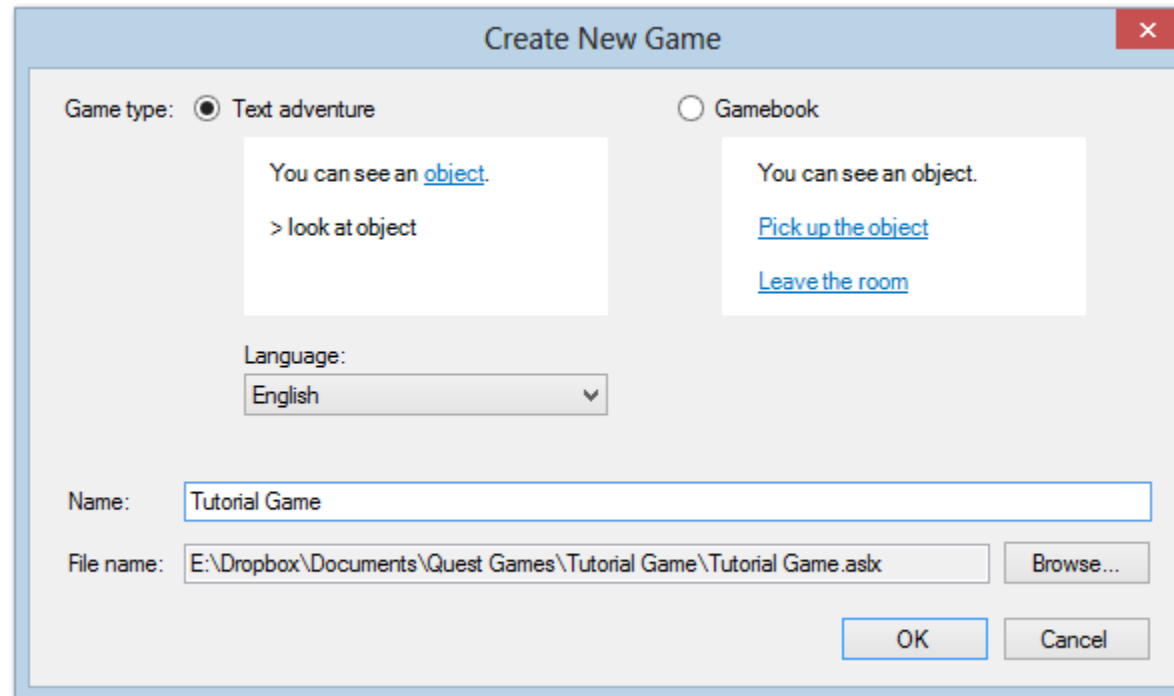
To create a new game, open Quest and click the File menu, then Create New Game.



[home](#)

Alternatively, you can switch to the *Create* tab and click the “Create a new game” button.

You’ll see a screen like this:



The screenshot shows a dialog box titled "Create New Game" with a red close button in the top right corner. Inside the dialog, there are two radio buttons for "Game type": "Text adventure" (selected) and "Gamebook". Below these, there are two preview windows. The "Text adventure" preview shows the text "You can see an [object](#)." followed by "> look at object". The "Gamebook" preview shows "You can see an object." followed by two links: "[Pick up the object](#)" and "[Leave the room](#)". Below the previews is a "Language:" label and a dropdown menu currently set to "English". At the bottom, there is a "Name:" label and a text box containing "Tutorial Game". Below that is a "File name:" label and a text box containing "E:\Dropbox\Documents\Quest Games\Tutorial Game\Tutorial Game.aspx", with a "Browse..." button to its right. At the very bottom right are "OK" and "Cancel" buttons.

Ensure that “Text adventure” is selected, and choose a language from the list - this tutorial will focus on creating a game in English, but the editor itself will look mostly the same whichever language you pick here.

Enter a name like “Tutorial Game”. Quest will create a folder and a game file for you. You can change where it puts the file by clicking the “Browse” button - it is recommended that you put your game file in its own folder.

[home](#)

Click OK and you'll see the main Editor screen:

Web version

To create a new game, log in to Quest. You'll see the "New game" form.

Create a new game

Game name:

Game type:

 ▼

Language:

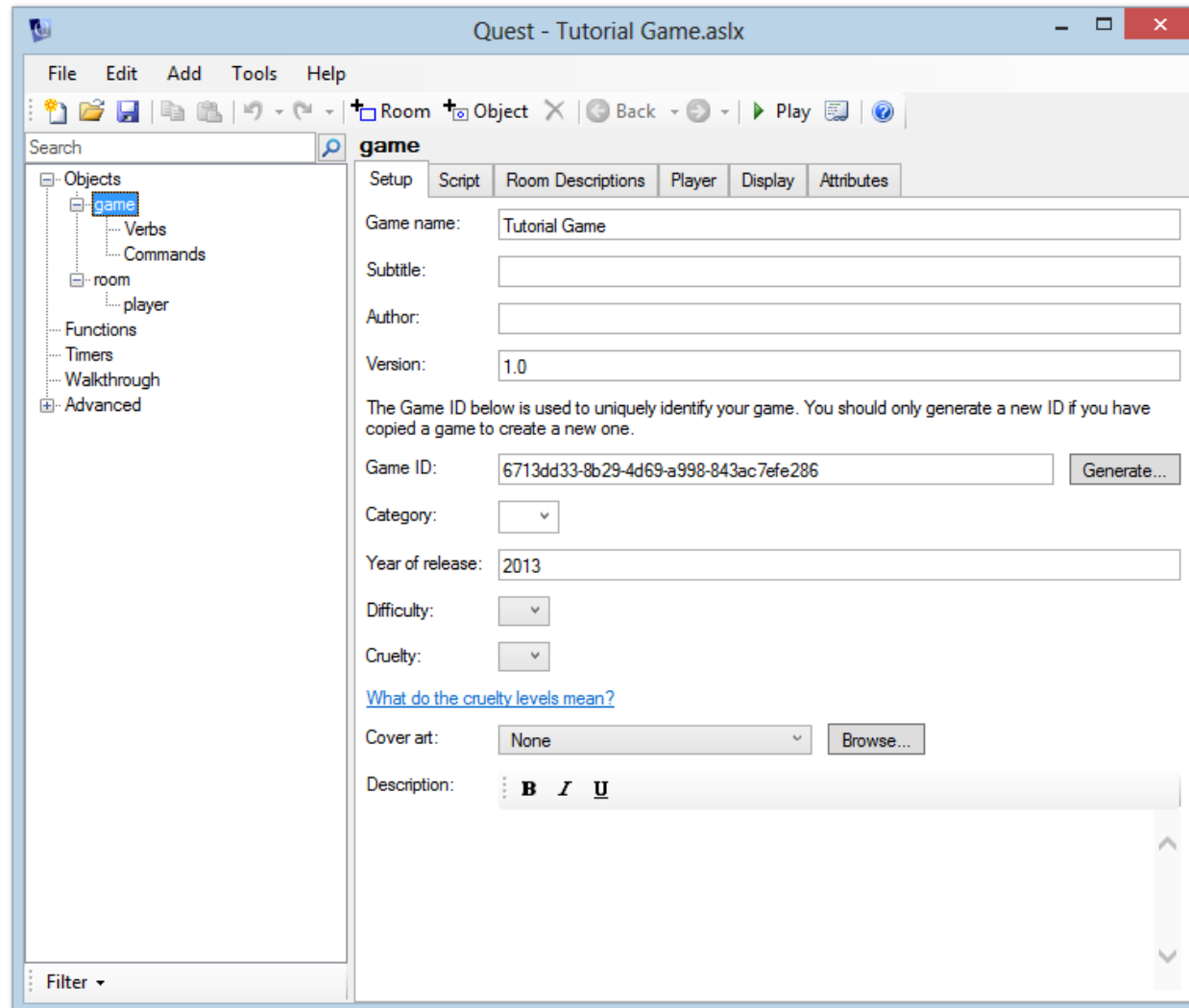
 ▼

As you can see, the options are the same in both versions, just laid out a bit differently.

Ensure that "Text adventure" is selected, and choose a language from the list - this tutorial will focus on creating a game in English, but the editor itself will look mostly the same whichever language you pick here.

Enter a name like "Tutorial Game" and click the "Create" button. Click the link which appears, and you'll see the main Editor screen.

The Editor Screen



[home](#)

Create a text adventure gaQuest - Tutorial Game

make.textadventures.co.uk/Edit/Game/5615

+ Room+ ObjectUndoRedoCopyPastePlayHelpSettingsSaved

Objects

- game
 - Commands
 - room
 - player
- Functions
- Timers

game

SetupScriptRoom DescriptionsPlayerDisplayPublish

Game name: Tutorial Game

Subtitle:

Author:

Version: 1.0

Category:

Year of release: 2013

Difficulty:

Cruelty:

What do the cruelty levels mean?

Cover art: Choose file

Description:

Submit a suggestion or bug

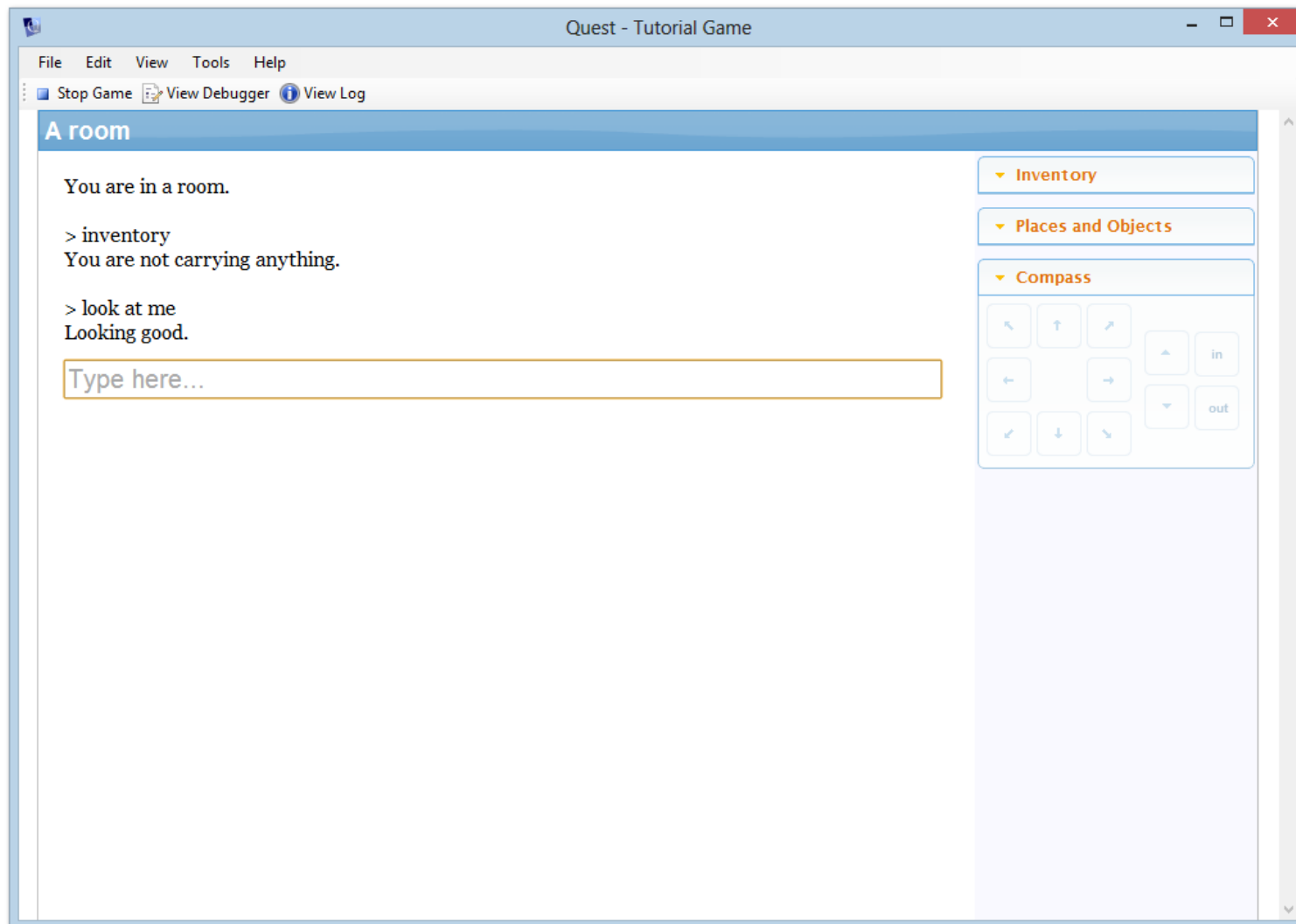
[home](#)

On the left is a tree showing you every element of the game. The “game” element is currently selected, so that’s what we can see in the pane on the right.

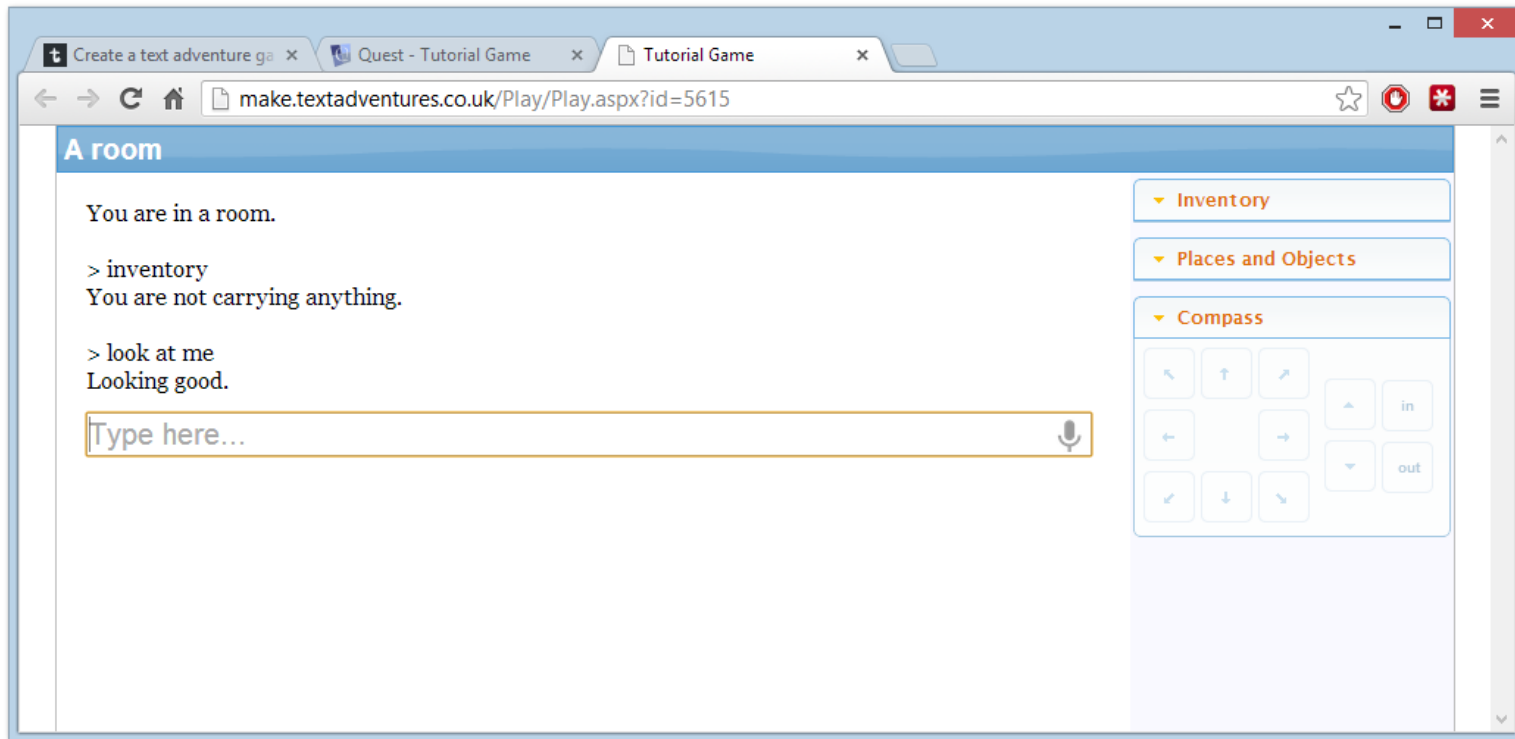
Quest has created a room called “room” for us, and inside this room is the “player” object, so that’s where the player will begin when you run the game. You can test the game by clicking the “Play” button. On the Windows version you can also select “Play Game” from the File menu or press the F5 key.

As you’ll see, it’s a pretty empty game at the moment. We can type some standard commands such as “inventory” to see that Quest comes up with some default responses, but that’s about all we can do at the moment.

[home](#)



[home](#)



In the Windows version you can go back to the Editor by clicking “Stop Game” in the top left of the screen, or you can also type “quit” or hit the Escape key.

If you are using the web version, you will probably have noticed that the game started in its own tab in your browser. Just close the tab when you have finished.

Simple Mode

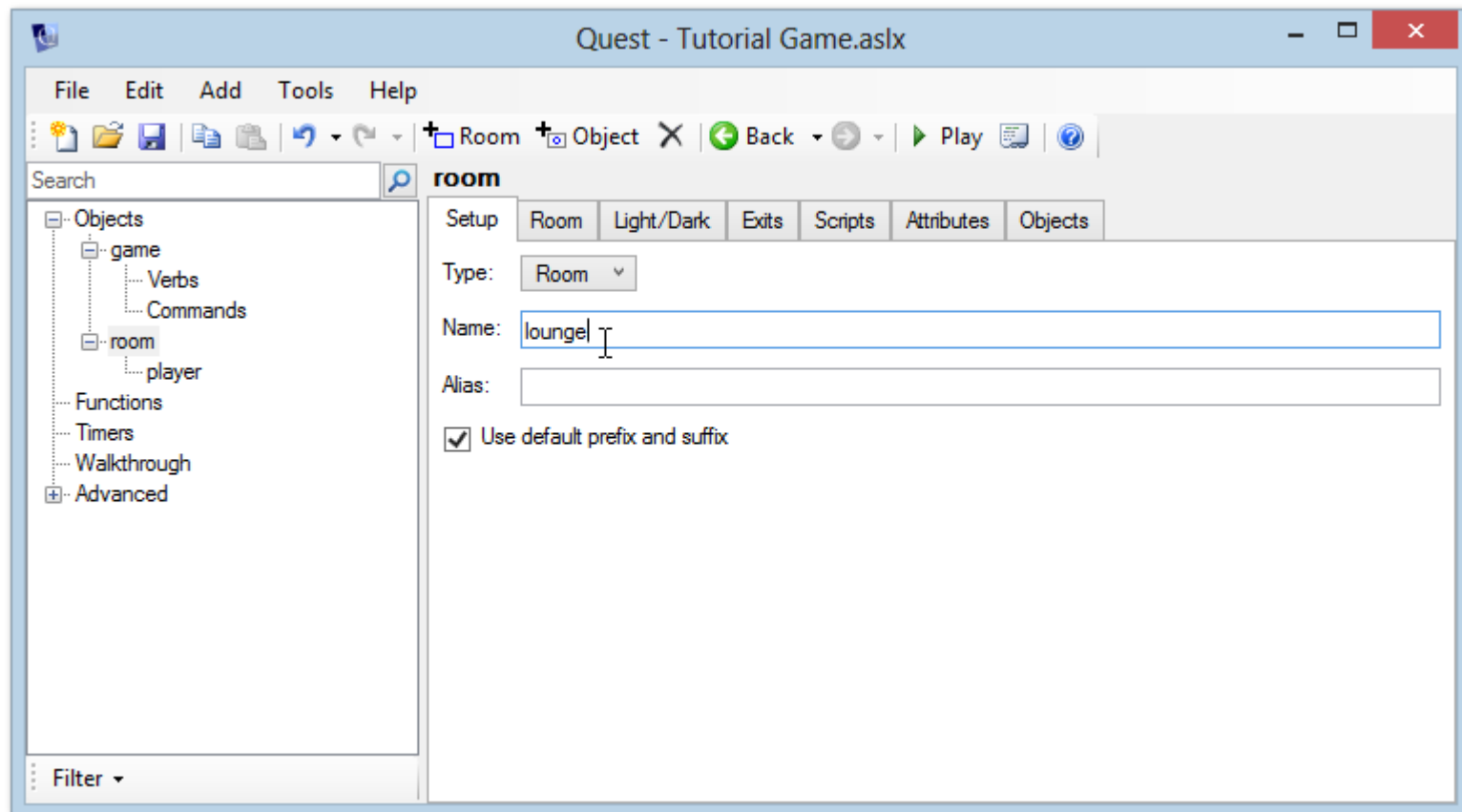
When starting out with Quest, you may find it easier to run in “Simple Mode”. This hides much of Quest’s more advanced functionality, but still gives you access to the core features.

You can toggle Simple Mode on or off at any time:

- on the Windows desktop version, you can toggle Simple Mode from the Tools menu.
- on the web version, click the Settings button at the top right of the screen.

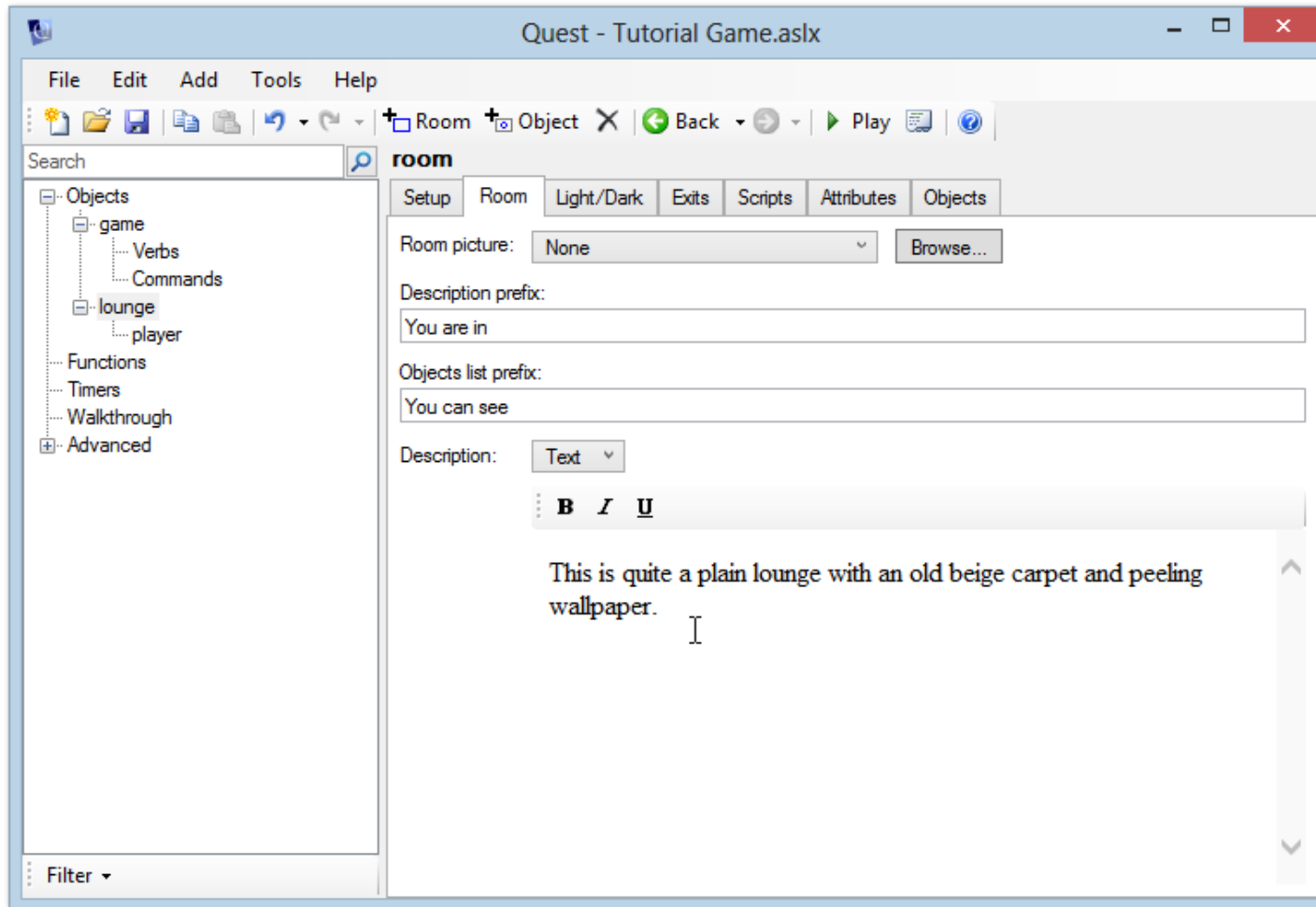
Setting up rooms

Quest created a room called “room”, which isn’t a very good name. In this tutorial game, we want to start in a lounge, so select “room” from the tree and change its name.



[home](#)

To create a room description, click the *Room* tab. Enter a Description in the text editor - something like “This is quite a plain lounge with an old beige carpet and peeling wallpaper.”



Let's add a second room to the game.

In the Windows desktop version, there are three ways you can do this:

- Click the Add menu, then click Room
- Click "Add Room" on the toolbar
- Right-click in the tree, then choose "Add Room"

In the web version:

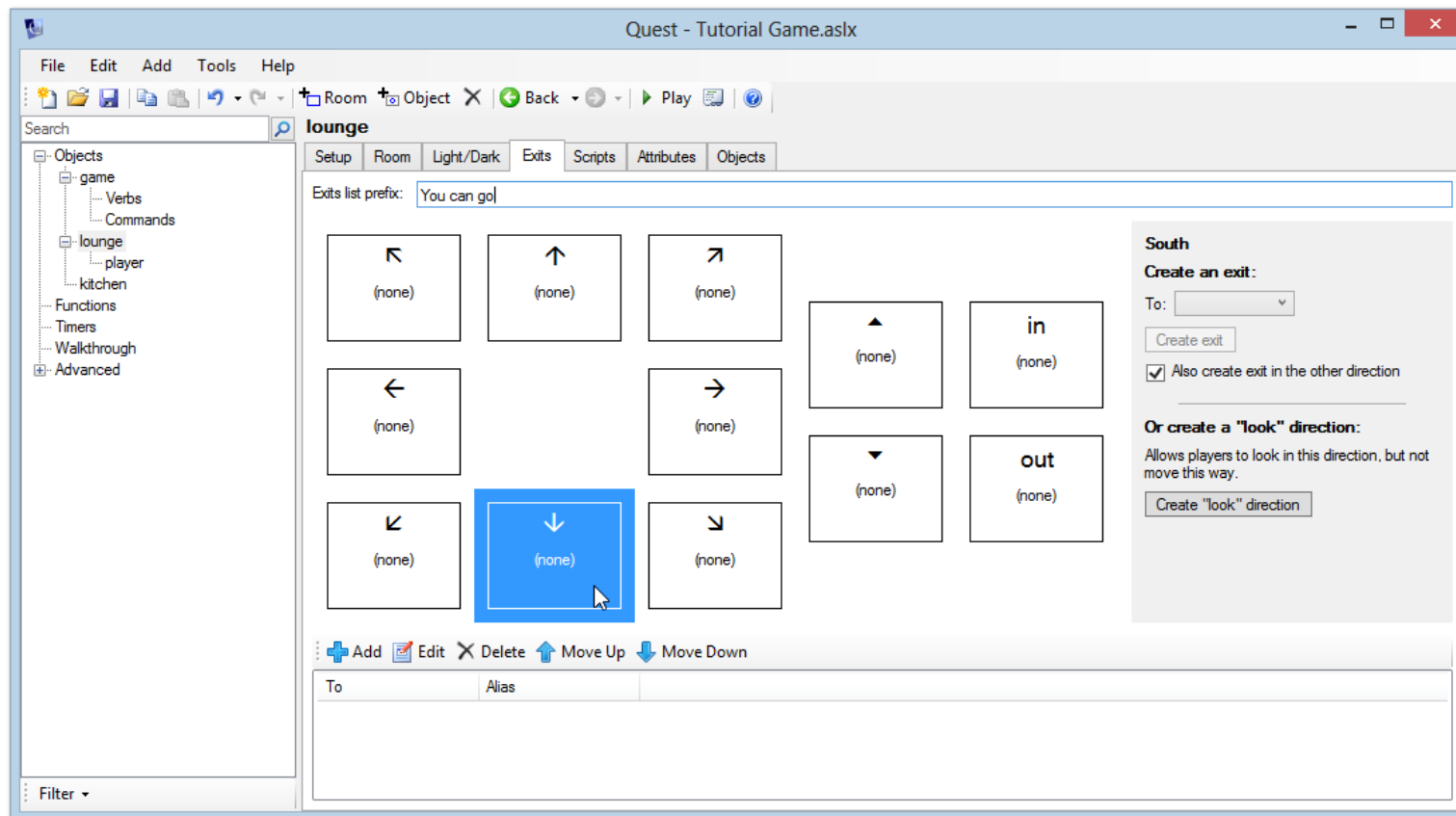
- Click the "+ Room" button at the top of the screen

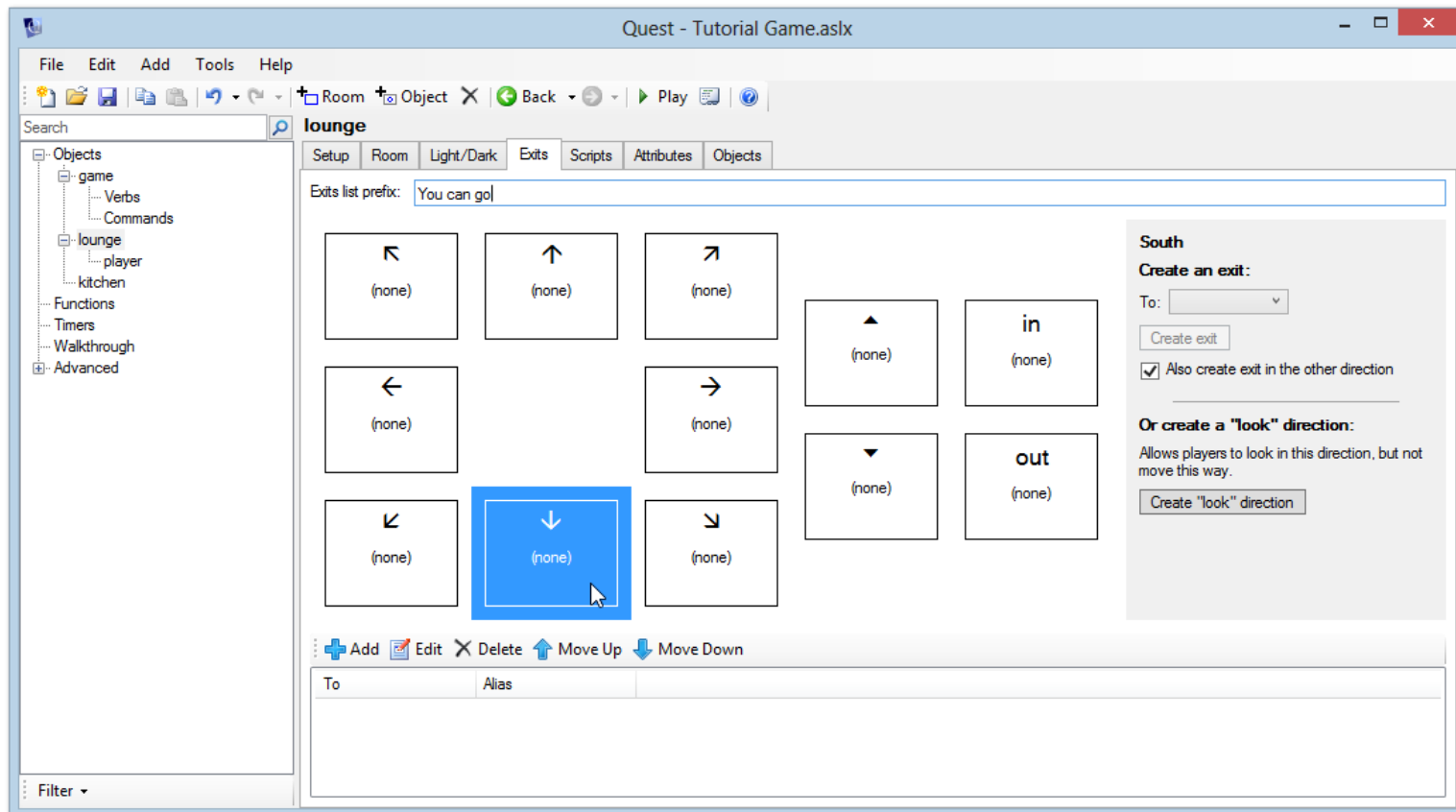
Add a room called "kitchen", and give it a description - use your imagination!

If you play the game at this point, you'll see the player is still trapped in the lounge, with no way out. To be able to get to the kitchen from the lounge, you need to add an exit.

Adding an exit

To do this, click back to the “lounge” room and go to the *Exits* tab. Then click the “South” exit:





When you click the “Create” button, actually *two* exits are created - one exit south from lounge to the kitchen, and another exit north from the kitchen back to the lounge. You can see both exits in the tree.

It is helpful to think of exits as “one way”. Each exit is “in” only one parent room (the “from” room), and points “to” one other room. That is why we have one exit in the lounge, pointing to the kitchen. A separate exit is in the kitchen, pointing to the lounge.

Exits, like every object in Quest, can have an alias, which is simply a way of displaying a particular name to the player. Notice how the two exits we just created have aliases of “south” and “north”. (We could give our exits any alias - it doesn’t have to be a compass direction. If we were setting a game on a ship for example, we might have exits with aliases like “port” and “starboard”.)

Play the game and verify that the player can go south and north between the lounge and kitchen.

Adding objects

Now let’s add some objects to the lounge, to give the player something to do.

A lounge is barely a lounge without a TV in it, so let’s add one now. With the lounge selected, you have four different ways of adding an object to the room. You can:

- Click the Add Object button on the toolbar
- Go to the *Objects* tab and click the Add button
- Click the Add menu and choose Object (Windows desktop version only)
- Right-click “lounge” in the tree and select “Add Object” (Windows desktop version only)

Use one of these methods to add an object to the lounge. A prompt will appear asking you to enter a name for the object. Enter “TV”. Leave the parent as “lounge” and click OK.

Object Names and Aliases

It is important to note the distinction between:

- the names that *players* can see and use to refer to objects
- the names that your Quest scripts use

Name: In order to avoid confusion, each object must have a unique name. So, if you have multiple televisions in your game, they must be given different names – like “TV1”, “TV2” and so on.

Alias: Of course, this wouldn't sound natural if these were the names that players saw, which is why Quest lets you set an alias. This is the name of the object that the player sees. In the example of multiple televisions, each of your TV objects could have an alias of "TV".

If you don't set an alias, players will see the object name – so you only usually need to worry about this if you want different objects to have the same displayed name.

So for now, we can leave the Alias box blank for the TV.

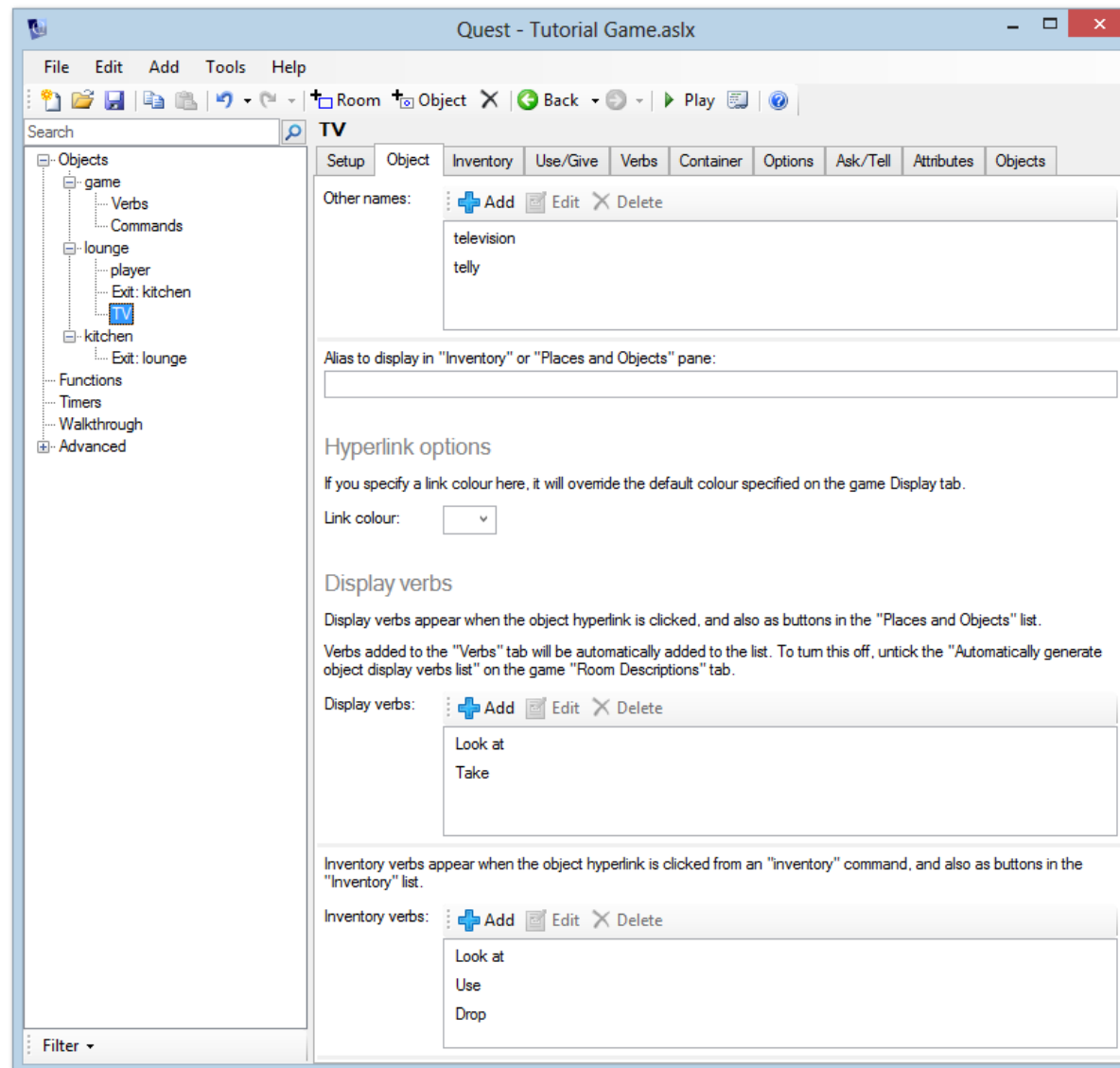
Other Names

If you go to the *Object* tab, you'll see an "Other Names" box. This lets you specify additional names that players can use to refer to this object. It is important to note that different players will have different ways of interacting with your game – many players prefer to use hyperlinks, but some prefer to type. You want to make it easy for Quest to understand what players type in, so you can add additional, alternative object names to ensure that happens. For example, for our TV object, some players might type in "look at television", and would reasonably expect that to work.

So, add "television" to the list of Other Names for this object. This will ensure that players can type in either "look at TV" or "look at television" to look at this object.

As an exercise, add any other alternative names you think that players might want to use.

[home](#)

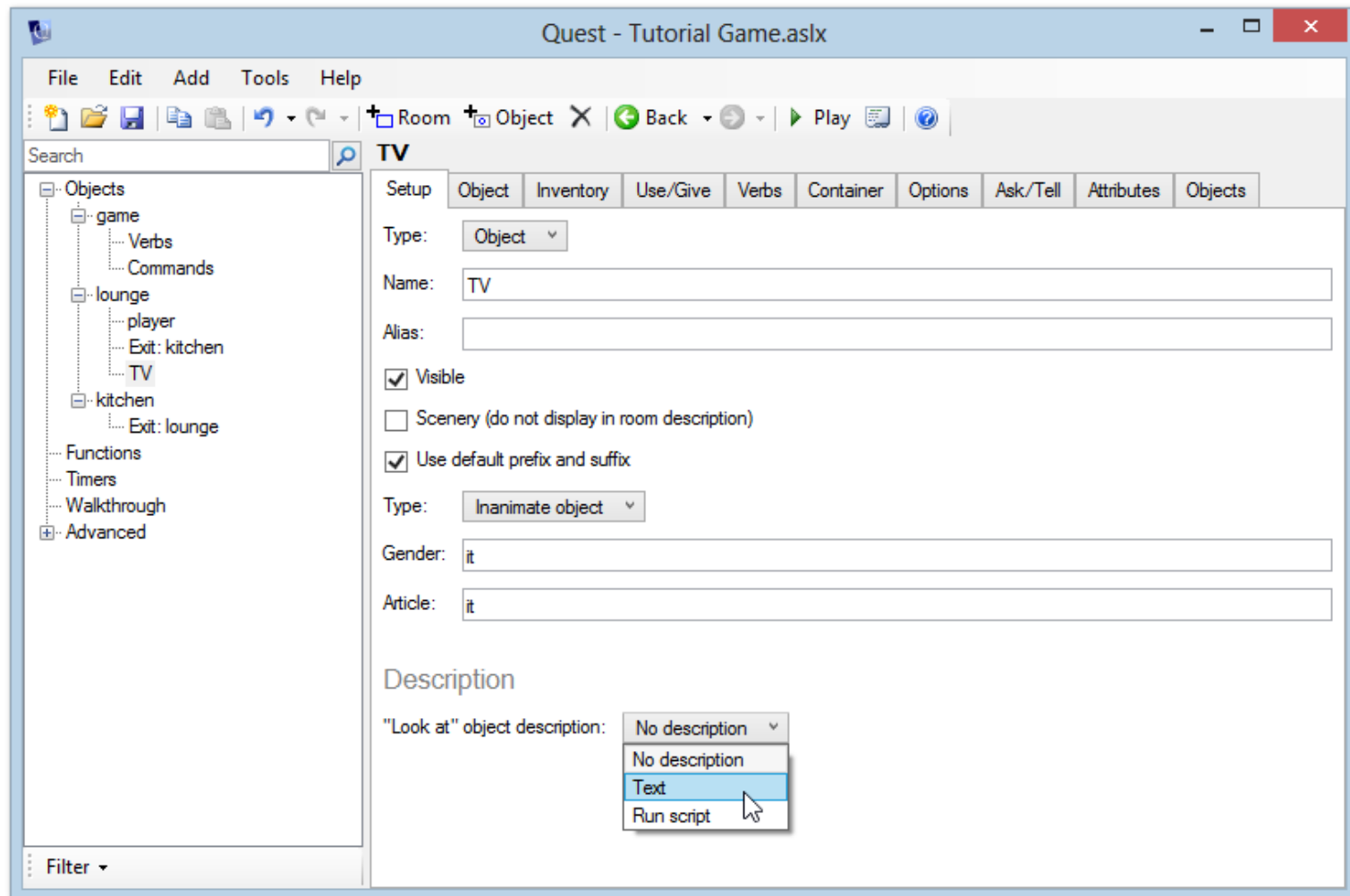


Description

If you run the game and look at the TV, you'll see that Quest doesn't have much to say on the subject - it says "Nothing out of the ordinary".

That's a bit boring - it's a sign of a bad game if you can't even be bothered to come up with descriptions for all your objects. We don't want to make a bad game, so let's add a description for this object. To do this, go to the Description drop-down in the bottom half of the object's *Setup* tab and select "Text".

[home](#)



[home](#)

Now enter the description. You could write something like “The TV is an old model, possibly 20 years old. It is currently showing an old western.”

Launch the game again and verify that it now shows you the description when you look at the TV.

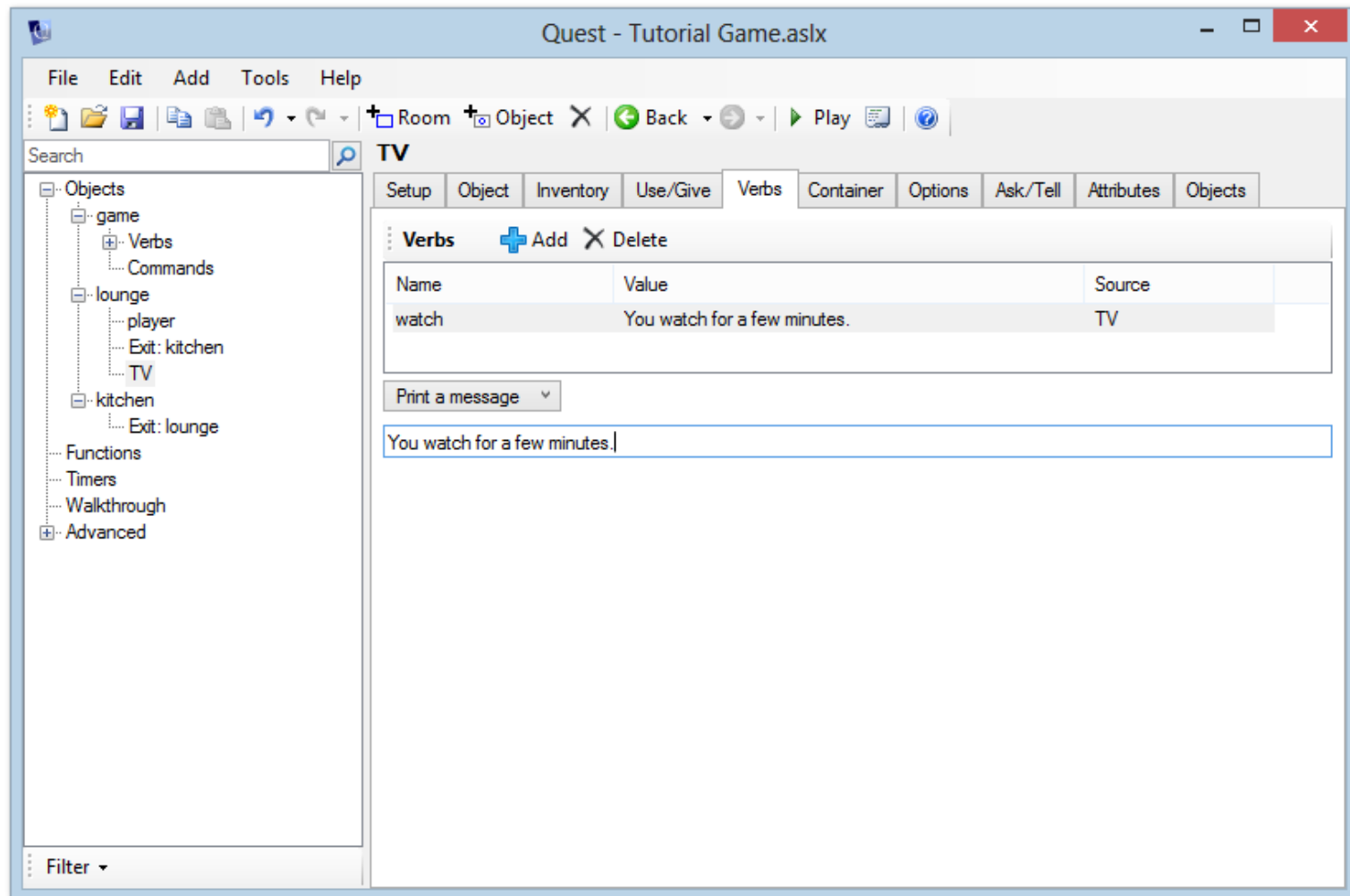
Adding a verb

It is a good idea to think about what kinds of things players might try to do to any objects in your game. In our example of the TV, it seems likely that a player might try to type “watch tv”, so it would be good if our game came up with a good response, rather than just saying it didn’t understand.

To do this, let’s add the verb “watch” to our TV object. As you should remember from school, verbs are “doing words”, and that’s what they are in Quest – verbs let you say what things can be “done” to your object.

Go to the *Verbs* tab, click the “Add” button and type “watch”. You can choose either to print a message or run a script when the player watches the TV. Enter a message. For example, “You watch for a few minutes. As your will to live slowly ebbs away, you remember that you’ve always hated watching westerns.”

[home](#)



Exercises

As an exercise, add the following objects to the lounge:

- A sofa. Give it a sensible description. Add a verb “sit on” so that the player can type “sit on sofa”. This should print a message like “There’s no time for lounging about now.”
- A table. Enter a sensible description.
- A newspaper. Enter a description, and add a verb “read” which will print an appropriate message.

Launch the game and verify that the objects you’ve just created have been set up and are working correctly – check that you can watch the TV, try to sit on the sofa, and read the newspaper.

We’re now on our way to making our first text adventure game. You may have noticed that, so far, we’ve only been walking around the game world and looking at things – we’ve not yet managed to interact with it and change it. We’ll start to do that in the next section, where we look at taking and dropping objects.

Next: Interacting with objects

Interacting with objects

Object Types

An object's *Setup* tab lets you choose the type of object. Select the "TV" object we created in the last section.

The first "Type" dropdown at the top of the screen lets you select from:

- Room
- Object
- Object and/or room

Rooms and objects are really the same thing in Quest - the option you select here simply lets the editor show you only what's relevant for the current object. If you wanted to create a cupboard inside a room that the player could get in, you might want to select "object and/or room" here - but in this tutorial, just leave this option set to the default.

The second "Type" dropdown lets you select from the following types:

- Inanimate object
- Male character
- Female character
- Inanimate objects (plural)
- Male characters (plural)
- Female characters (plural)

Whichever type you select, the object will behave in pretty much the same way, except that Quest's default responses will make a lot more sense if you set this correctly. This is because setting the type will update the Gender and Article (you can also override these manually).

- **Gender:** Usually "it", "he", "she" or "they". Quest uses this for sentences such as "*It* is closed", "*He* says nothing" and so on.
- **Article:** Usually "it", "him", "her" or "them". Quest uses this for sentences such as "You pick *it* up", "You can't move *her*" and so on.

Scenery

The “Scenery” option means that the object won’t be displayed automatically in the room description, or the “Places and Objects” on the right of the screen.

Why might we want to do this? Well, when we created our “lounge” description in the previous section, we wrote “This is quite a plain lounge with an old beige carpet and peeling wallpaper”. What if the player types “look at wallpaper”? Quest will reply “I can’t see that here”, which will be a bit strange.

Although the wallpaper isn’t an important object, we should still have a response for “look at wallpaper”. If we make it a scenery object, it’s “in the background” as far as the game goes, as it won’t appear in the “Places and Objects” list, or in the list of objects in the description of the room. We won’t be cluttering things unnecessarily, but we will still be providing responses for anything the player might reasonably type in.

So, create a new object called “wallpaper” and tick the Scenery box. Enter a description like “The horrible beige wallpaper hangs loosely on the walls.”

Launch the game and verify that although the wallpaper doesn’t explicitly appear in the description, you can still get a sensible response by typing “look at wallpaper”.

Exercise

Remember that we also mentioned the carpet in the room description as well. As an exercise, add this as another scenery object, and give it a sensible description.

Creating a Character

Let’s create our first character. He’ll be about as basic a character as you can get, and he won’t be the most talkative. This is because he’s dead. Well, you’ve got to start somewhere.

Create a new object called “Bob” and change his type to “Male character”. Give him a “look” description of “Bob is lying on the floor, a lot more still than usual.”

There is one other thing we need to do. If you run the game now, you’ll see the room description says “You can see a TV, a sofa and *a* Bob”. There’s only one Bob - well, in this game anyway - so we want to get rid of that “a”. Where did that even come from? The answer is the prefix.

Prefix and Suffix

A prefix and a suffix let you insert text before and after the object name when it's displayed in a room description. When you leave "Use default prefix and suffix" checked, the suffix is blank, and the prefix (for English games) is either "a" or "an" depending on whether the object name (or alias) begins with a vowel.

You can specify your own text by unchecking the box. Two new textboxes will appear, and for our "Bob" character you can just leave them blank, as we don't want any text added around our object name.

A quicker way of doing this is to select "Male character (named)" from the types list.

We will come back to Bob later in the tutorial, where we will make him a little more animated.

Taking the newspaper

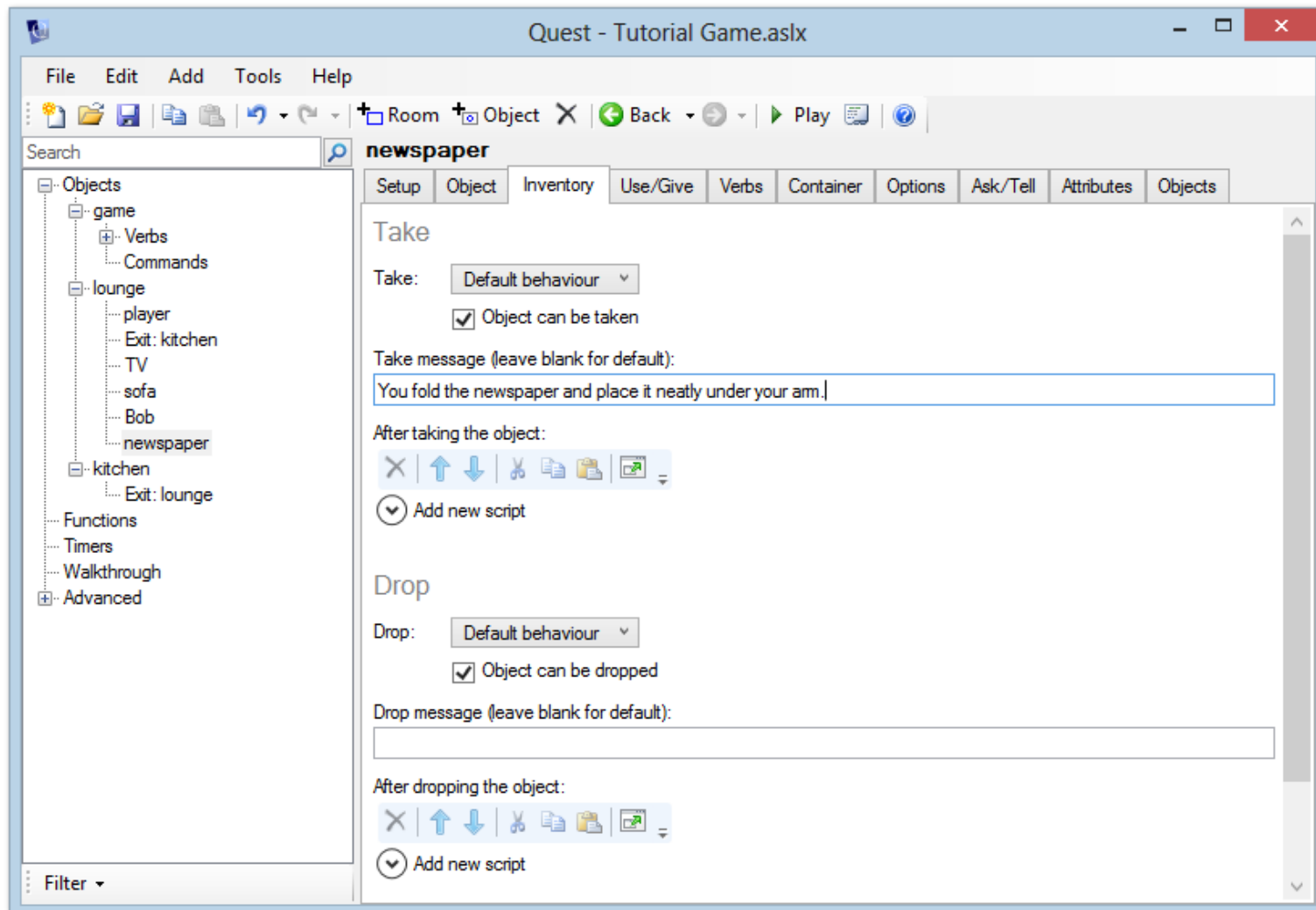
You should have added a newspaper object as an exercise at the end of the previous section. If you didn't, add one now. We're going to make this an object that the player can take.

This is very easy to do - simply go to the *Inventory* tab. You have a couple of different options for "Take":

- Default behaviour: You'll want to use this for most of your take-able objects. This option gives you the "Object can be taken" checkbox, and the ability to specify a "take message" which is printed when the player takes (or attempts to take) the object.
- Run script: If you want full control over what happens when the player attempts to take the object, choose this option. We will cover scripts later on in the tutorial, so don't choose this for now.

To let the player take the newspaper, we just need to tick the "Object can be taken" box. If you don't specify a message, you'll get a default message - "You pick it up". If you want something a bit more imaginative, you can enter a Take Message such as "You fold the newspaper and place it neatly under your arm".

[home](#)



The “Object can be dropped” box is ticked by default, so the player can take and drop the object as many times as they like. The options are the same as for “Take”, so you can specify your own drop message or completely customise the behaviour with a script if you like.

Switching the TV on and off

Let’s make it possible to turn the TV off.

Quest has a whole bunch of features built in that you can add to your objects. To keep them manageable, objects have a *Features* tab, and you can select the ones you want for each object. Ticking a feature here will make the appropriate tab display, and you can then go to that tab to turn the feature on, and set it up as you like. Settings on the *Features* tab only determine what other tabs are shown, they have no effect themselves on the object when playing the game.

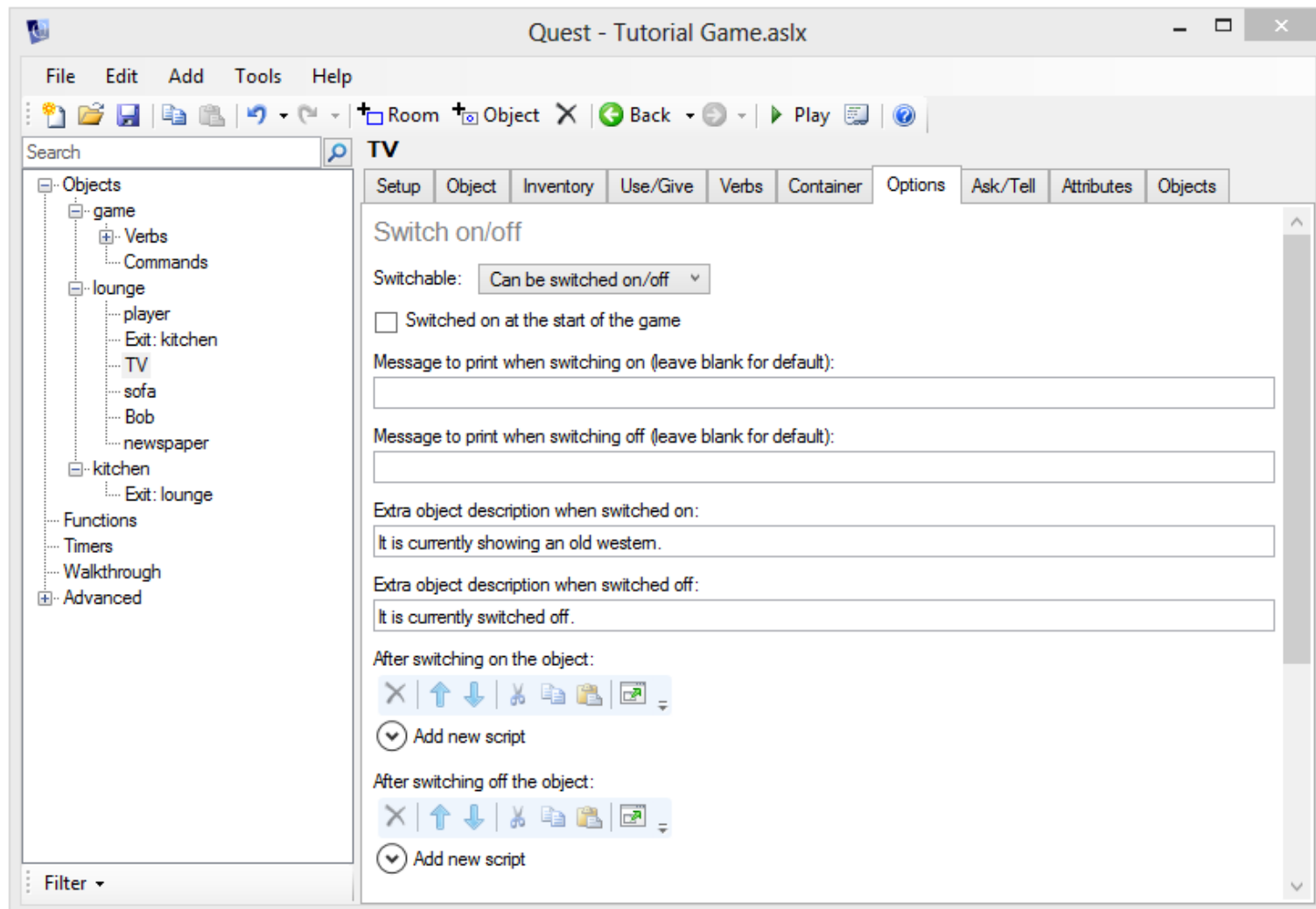
Switchable is one such feature, so the first step is to go to the *Features* tab, and tick “Switchable”. This will display the *Switchable* tab for the TV.

Now go to the *Switchable* tab. You’ll see a dropdown labelled “Switchable” - select “Can be switched on/off” and various options will appear.

The options should be fairly self-explanatory - you can choose whether the object is switched on when the game begins, and the text to print when switching on/off. Finally, you can choose some extra text to add to the object description. This lets you have show text depending on whether the object is switched on or off.

Enter some sensible text for these, for example as shown below.

[home](#)

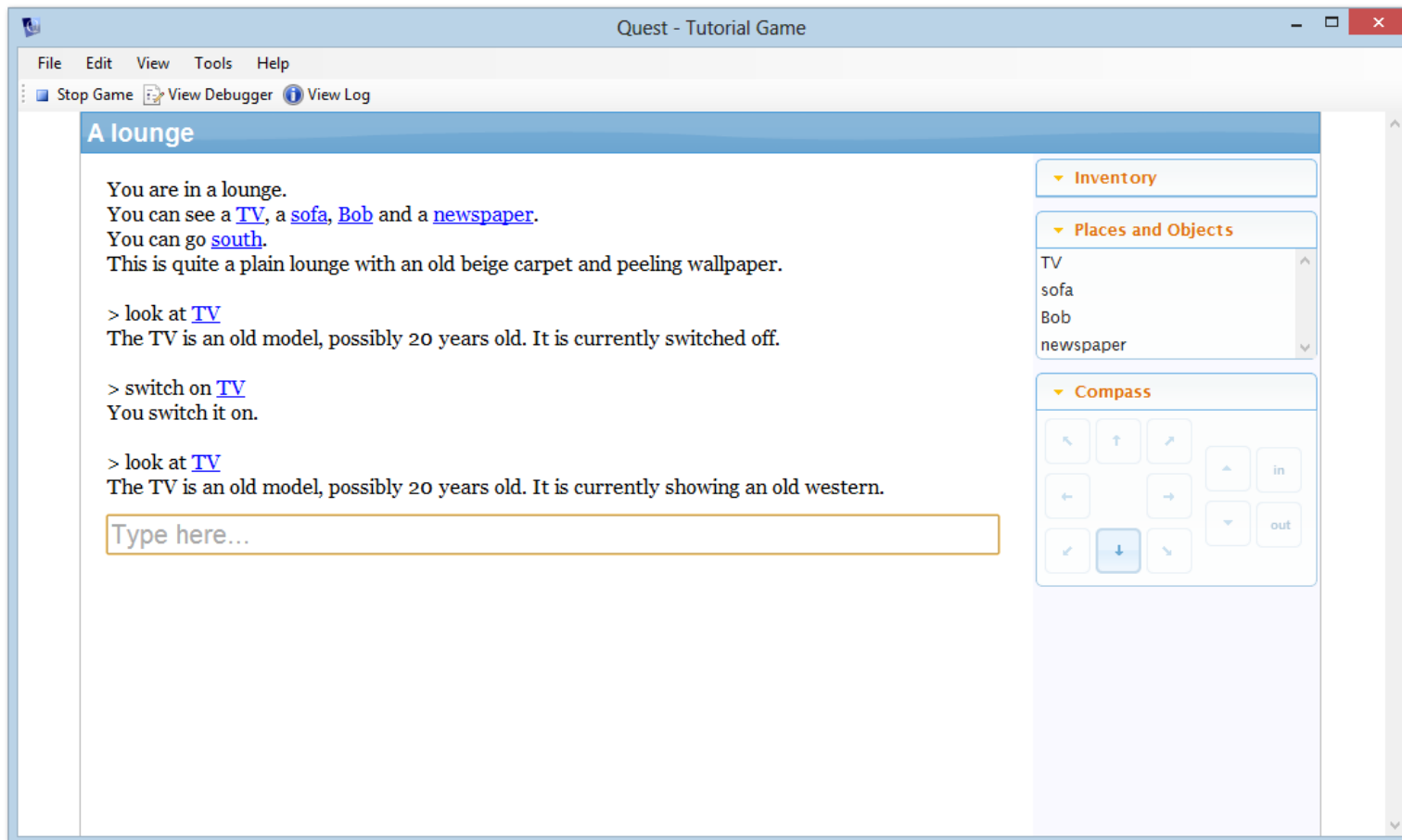


[home](#)

Go back to the Setup tab and change the “Look at” description so it just reads “The TV is an old model, possibly 20 years old.”

Now when you play the game, you get sensible behaviour for “switch on tv”, “switch off tv”, and alternative forms of the command.

[home](#)



[home](#)

Notice that by setting this object up as “Can be switched on/off”, two new options “switch on” and “switch off” have automatically appeared in the TV hyperlink menu.

Exercise

Add a “switchable” lamp to the lounge that is switched on at the start of the game.

Next: Anatomy of a Quest game

Anatomy of a Quest game

Every Quest game is made up of the following parts. Here are the main ones:

Elements

There are various types of element:

Objects

Objects are the basic building blocks of the game. Everything “physical” in the game is an object - that includes rooms and the player themselves. So in a simple game where the player is in a lounge, and there is a cat and a table in the lounge, there are four objects in total - the lounge, the player, the cat and the table.

Objects can contain other objects. This is done by setting the “parent” attribute. In our simple example, the lounge has no parent - it stands alone. The player is in the lounge, so the player’s parent is “lounge”. If the cat is sitting on the table in the lounge, then the cat’s parent is “table”, and the table’s parent is “lounge”.

Exits

Exits connect objects (usually rooms) together. The exit has a parent, so our simple game might have an exit from the lounge by setting the exit’s parent to “lounge”. Exits also have a “to” direction, so this exit might point to another room, such as the kitchen. Or, it might point to an object inside the same room, such as a cupboard - this would allow the player to go inside a cupboard in the room.

Commands and Verbs

Commands handle player input. They can exist globally, in which case the command will work everywhere. Commands can also exist inside a particular room, in which case that command will only work in that room. Commands have a pattern, such as `look at \#object\#`. When the player types something, it is compared to all the available command patterns. The best match is then used to process what the player typed in. So if the player typed “look at cat”, the “look at” command is matched, and it performs whatever action is necessary to print the description of the cat.

Verbs are a shortcut for commands. Many commands follow the same pattern, and it is easier to have the verb mechanism handle that for us, so we can concentrate on what makes ours special.

Game

The game itself is a special kind of object - it contains attributes such as the name of the game, options such as how to print room descriptions, and display settings.

Attributes

All element data (that is, all information about objects, commands, etc.) is stored in **attributes**. An element can have an unlimited number of attributes. Attributes can store things such as the object description, alternative object names, the behaviour when an object is taken, which objects can be used on the object, and much more. The attribute can be of many types:

String

A sequence of letters/numbers, for example “The cat is sitting quietly on the table”. Obviously, strings are very common in text adventure games!

Integer

A whole number, such as 1, 2, -3, 42 or 1 billion.

Script

One or more script commands, which are instructions for Quest to carry out. Everything that happens in a game is controlled by script commands. Script commands can print messages, move objects around, show videos, start timers, change attributes, and much more.

Scripts can be created by adding script commands using the user interface, or by typing code in “code view”. Behind the scenes, it is all the same, so you can flip between the two as you like.

Object

Objects can be attributes too. The “to” attribute of an exit holds an object, the destination of the exit. The “parent” attribute is also an object.

List

A list is an ordered sequence of things. Lists can contain strings, scripts or objects (though lists of objects are rarely attributes).

Dictionary

A dictionary is a look-up table of strings, scripts or objects. That is, a set of data where each item can be accessed by a string.

Libraries

Libraries are used to include common functionality in a game. There is a standard “Core” library that is included by default with all Quest games. This is made up of the elements above - commands, scripts and so on - and provides a lot of the standard functionality that players will expect in your game, such as the “look at” command, printing room descriptions, and so on.

Next: Using scripts

Using scripts

We'll now start to play with the real power behind Quest – scripts. Scripts let you do things within the game, change the game world, show pictures and more. With a script, you can print different messages or run other actions depending on the state of any object in the game.

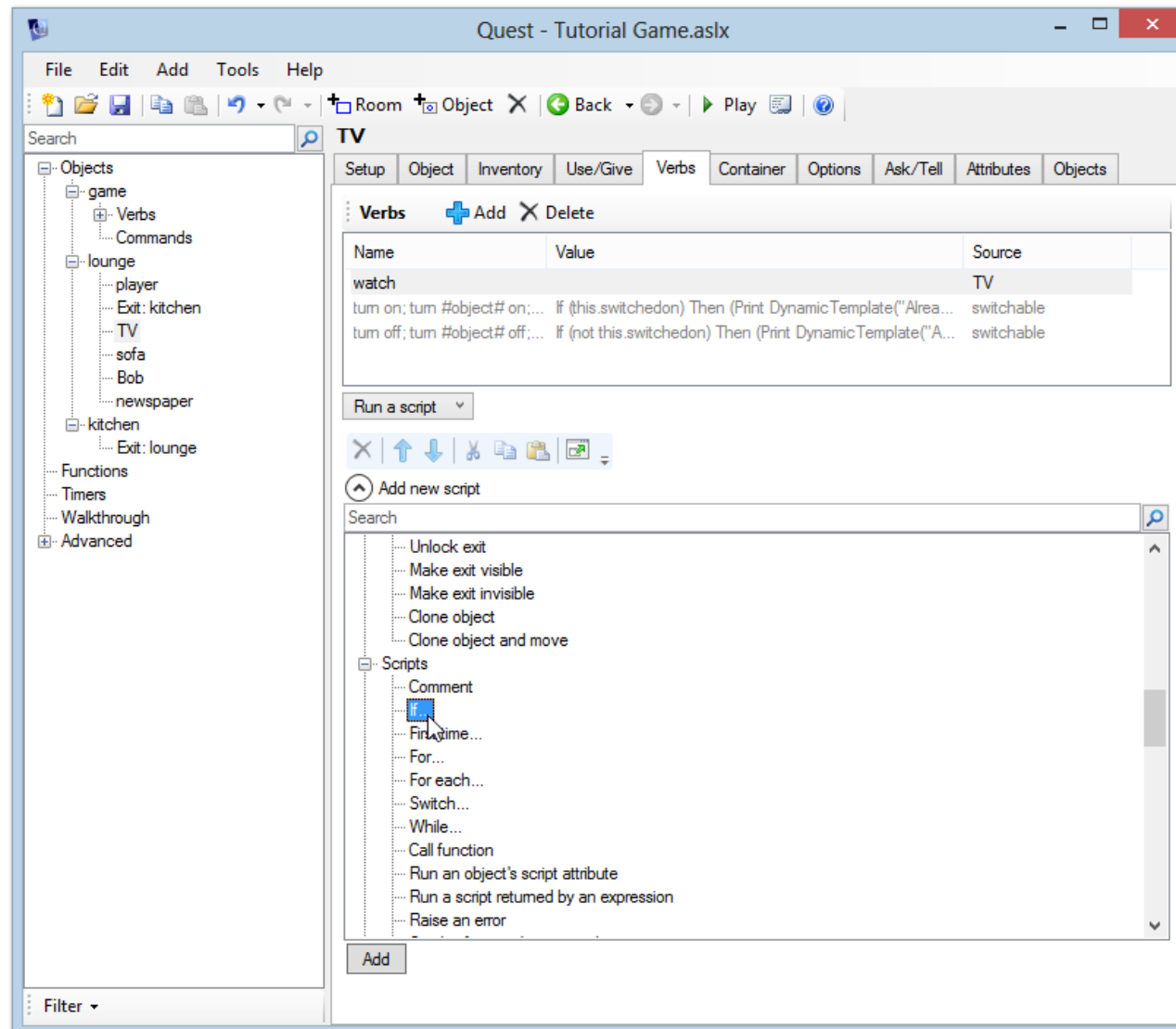
In this example, we'll use a script to customise the “watch” verb we added to the TV in the previous section. We want to update it to provide a sensible response depending on whether the TV is switched on or not.

Select the TV object and go to the *Verbs* tab. If you've been following all the steps in this tutorial, you should already have a “watch” verb which prints a message. If you've already got a “watch” verb, change it from “Print a message” to “Run a script” (or add a new “watch” verb if you don't already have one).

Click the “Add new script” header and you'll see a list of all the commands you can add to a script. The commands are in broad categories - Output, Objects, Variables and so on - but you can also find a command by typing in the Search box, if you don't know the category.

Go to the Scripts category and add the “If” command (you can click the “Add” button, or just double-click the command).

[home](#)

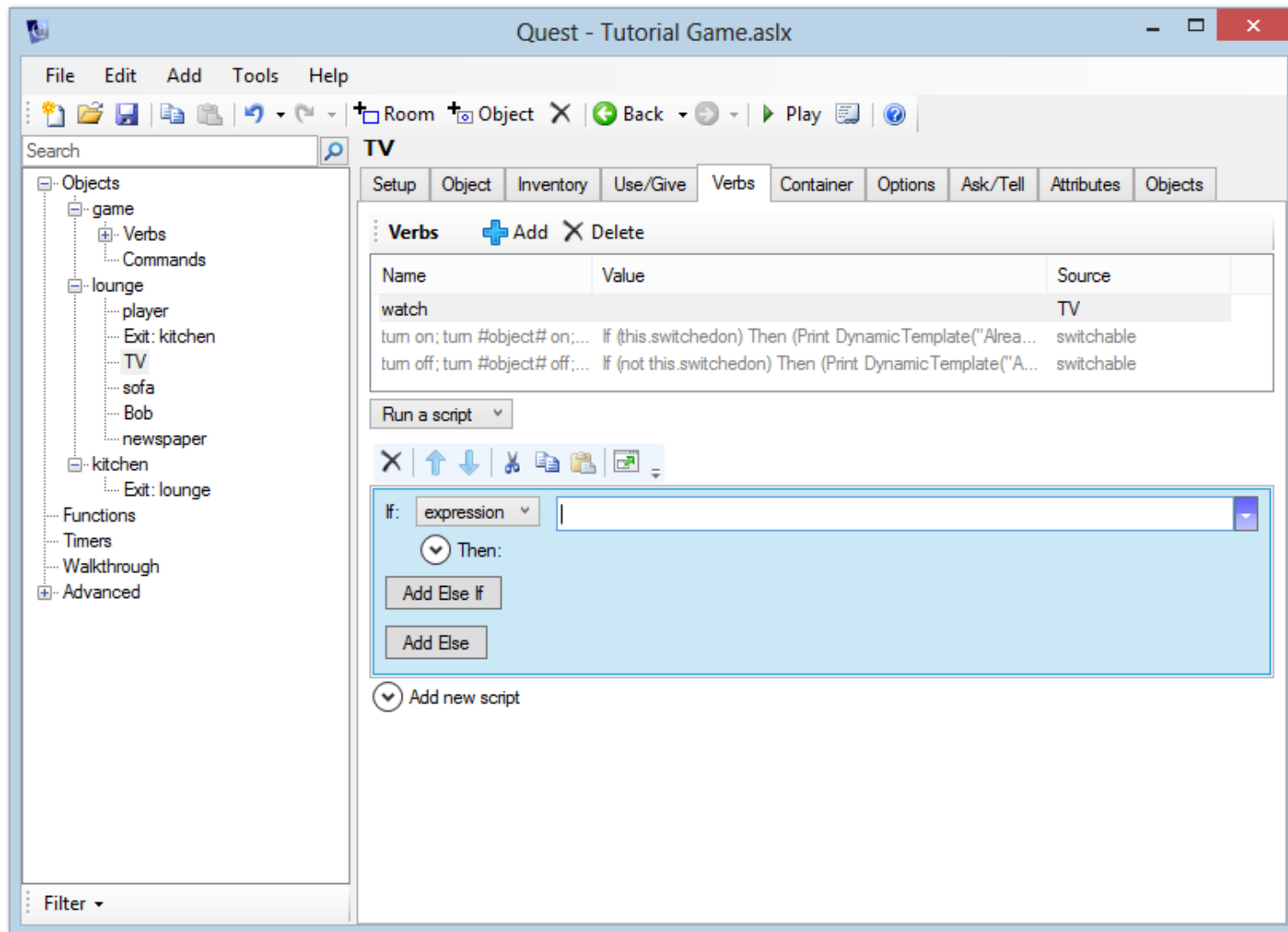


[home](#)

The “if” command is hugely powerful, because it lets us choose which script to run depending on a condition that we set.

After adding the command, you’ll see the following editor:

[home](#)

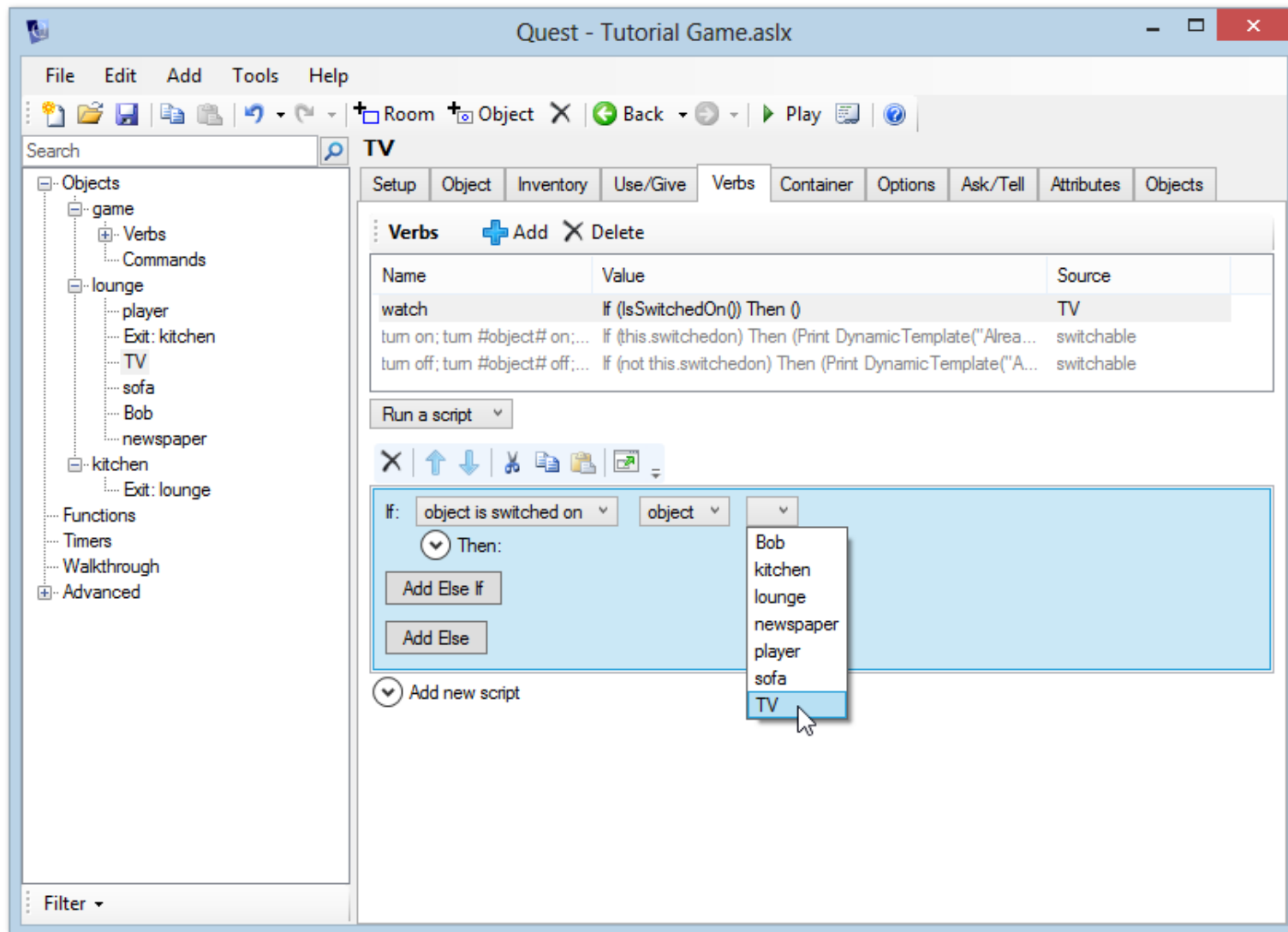


[home](#)

First, we need to add a condition. If you click the “expression” dropdown list next to the “If” label, you’ll see a list of conditions that you can add. Select “object is switched on”.

The editor template will then change, and next to the condition you will now see two more dropdown lists. Leave the first one set to “object”, and you’ll be able to choose an object from the second list. Select “TV”.

[home](#)



[home](#)

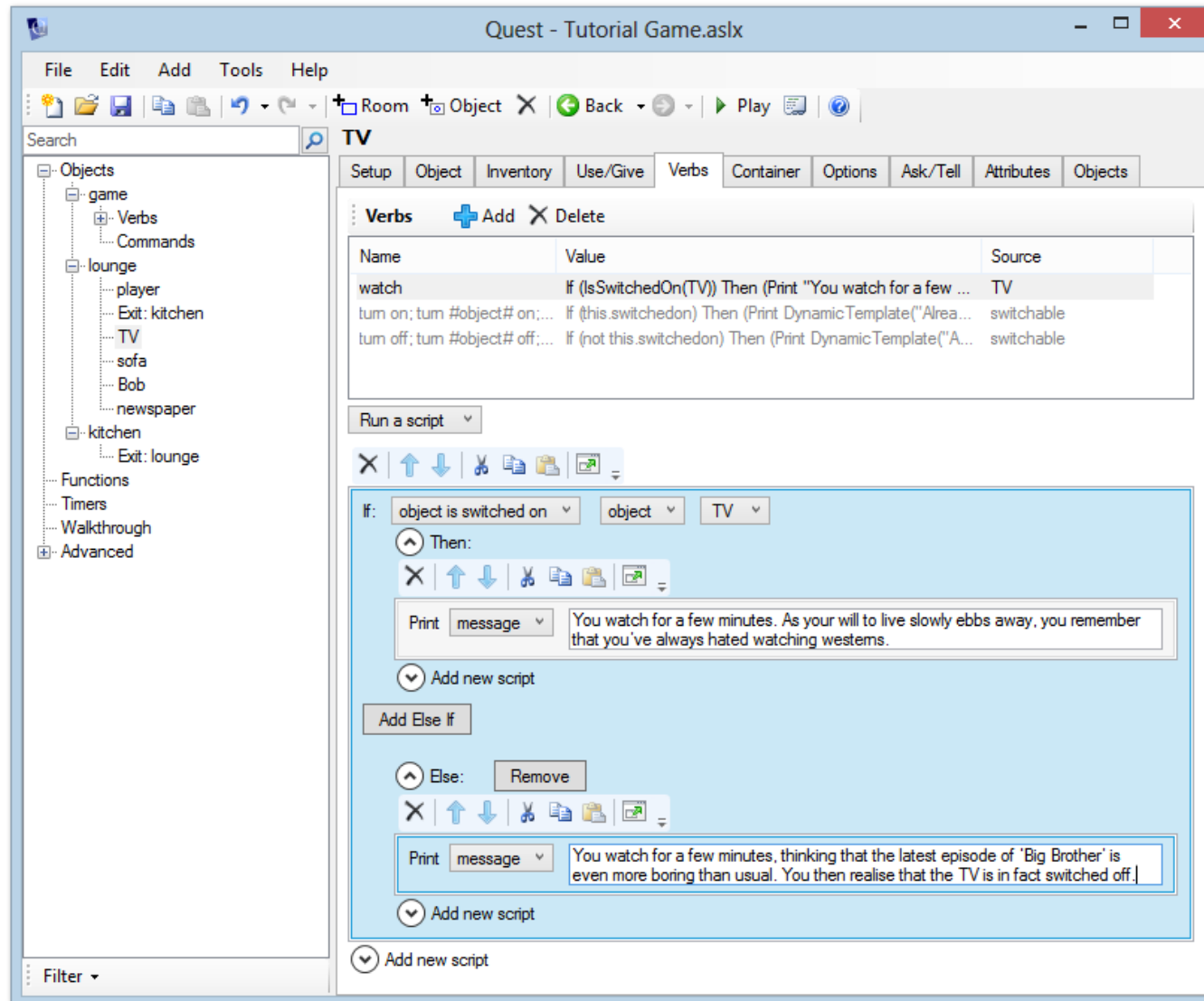
That's our condition added - now we just need to say what happens when the condition is met. Click the "Then" header and you'll see that you can add script commands here too. These script commands will *only* be run *if* the TV is switched on. Add a "Print a message" command.

This will be the text that will appear when the player types "watch tv" while the TV is switched on, so enter a message like "You watch for a few minutes. As your will to live slowly ebbs away, you remember that you've always hated watching westerns."

We're not done yet - what if the TV is *not* switched on? Fortunately we don't need to add a whole other condition - we can just add an "Else" script to the one we're working on. Click the "Add Else" button, then expand the "Else" header that appears. Add a "Print a message" command again, and this time add a message like "You watch for a few minutes, thinking that the latest episode of 'Big Brother' is even more boring than usual. You then realise that the TV is in fact switched off."

Your screen should now look like this:

[home](#)



[home](#)

Now would be a good time to play the game to test that it works properly. Switch the TV on and off, and verify that you get a sensible response when you type “watch tv”.

Next: Custom attributes

Custom attributes

We'll now start creating things in the kitchen, where we'll look at some more of Quest's features.

Enter a description like "The kitchen is cold and the stench of the overflowing bin makes you feel somewhat faint." As an exercise, add a scenery object called "bin" and give it a sensible description.

We're now going to look at **attributes**. Every time we've edited any aspect of an object or room so far, we've actually been editing an attribute. The prefix, description, "take" behaviour and so on are all attributes of an object. Whenever something changes in the game, it is a change in an object's attribute. When the TV is turned on or off, the "switched on" attribute is changing. Even when the player moves, this is actually just changing an attribute of the player object called "parent".

In this example, we'll store the weights of various objects by creating a new "weight" attribute. Later we will create a "weigh" command which will tell us the weight of **any** object.

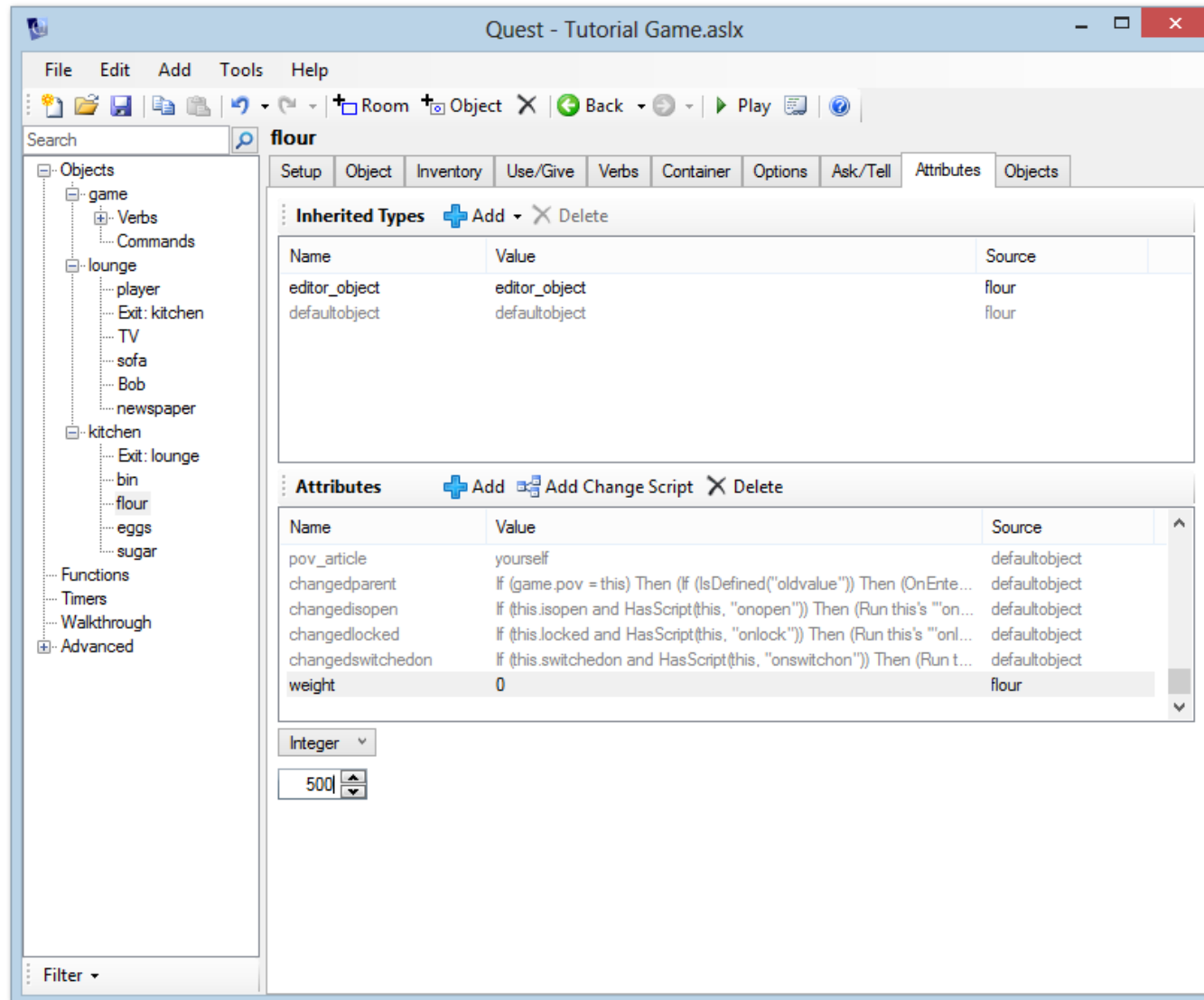
First, let us create a few objects we can weigh. Create three objects – flour, eggs and sugar. Make sure the object types are set correctly (either "inanimate object" or "inanimate object (plural)"). We'll use units of grams, so we'll say the flour has a weight of 500, the eggs have a weight of 250, and the sugar has a weight of 1000.

The Attributes Tab (Desktop Version Only)

If you are using the desktop version, you can click an object and select the Attributes tab - you'll see all the underlying data for the object. We can also use the Attributes tab to add our own custom data to any object.

So let's give the new objects weights. First we'll set the flour's "weight" attribute to 500. To do this, select the flour object and go to the Attributes tab. We'll look at "Inherited Types" later - for now, go to the Attributes table and click the Add button. Enter the name "weight". We want to use whole numbers for weight values, so select "Integer" from the list and enter the value "500".

[home](#)

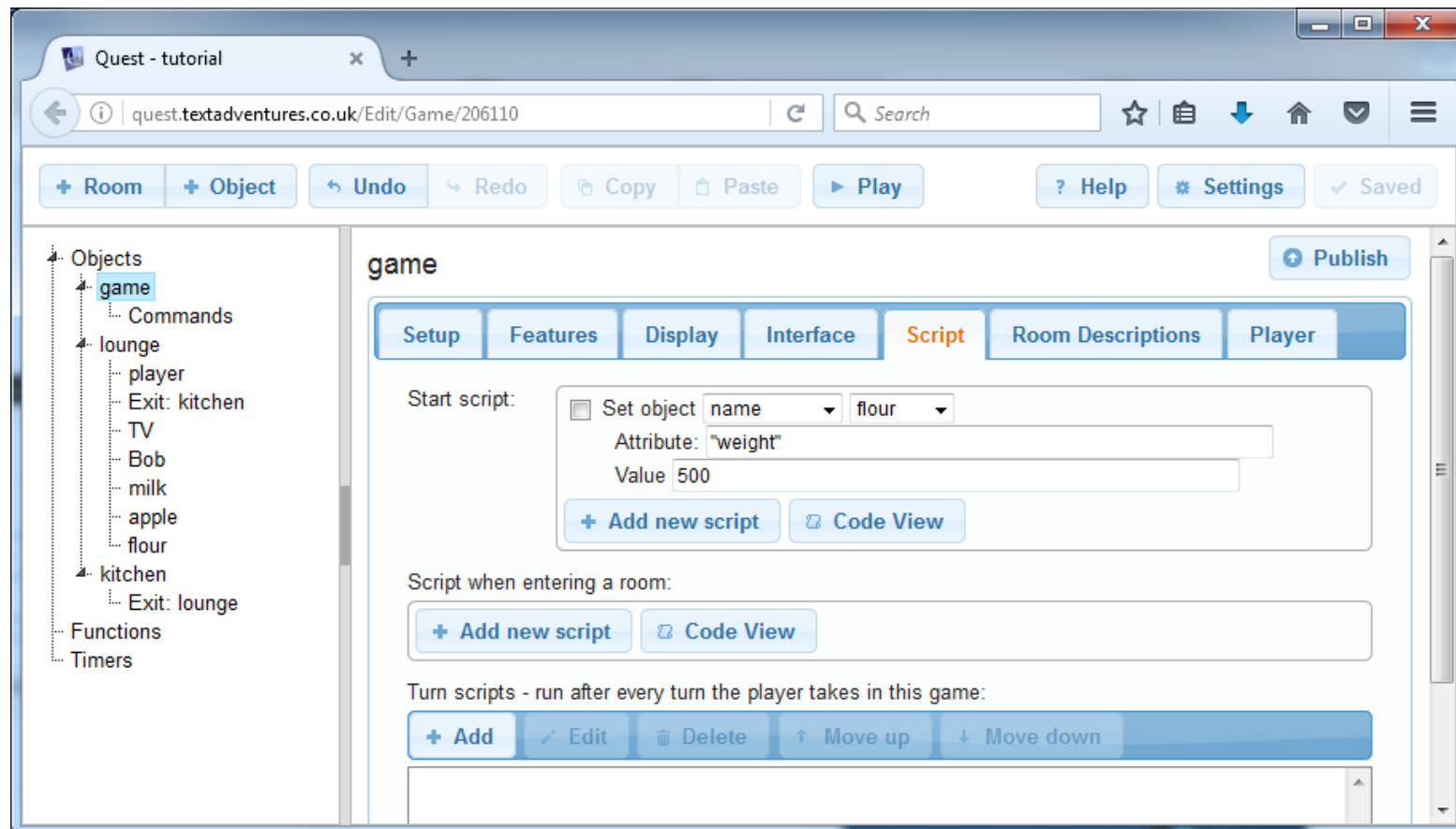


Alternatively...

The Web version currently has no attributes tab, so we will have to use an alternative approach. Go to the Script tab of the game object. The bit at the top is a script that will run when the game starts, so we can set attributes there (the disadvantage is that this will get pretty messy if you have dozens of objects with a few attributes each, but for a handful, it is okay).

Click “Add new script”, and select “Set a objects attribute (named by an expression)” from the “Variables” category. In the first line, keep “Name”, and in the other box, select the flour. For the next line, set the attribute name to “weight” (note that for once you need quotes here, but there are already provided). In the third line, for the value type “500”.

[home](#)



Now follow the same process to set the weights of the eggs and the sugar.

Reading Attributes

You can read an attribute from any script command by using an **expression**. Expressions let you perform calculations, run functions, and read the values of variables and attributes. To read an attribute, you use this form:

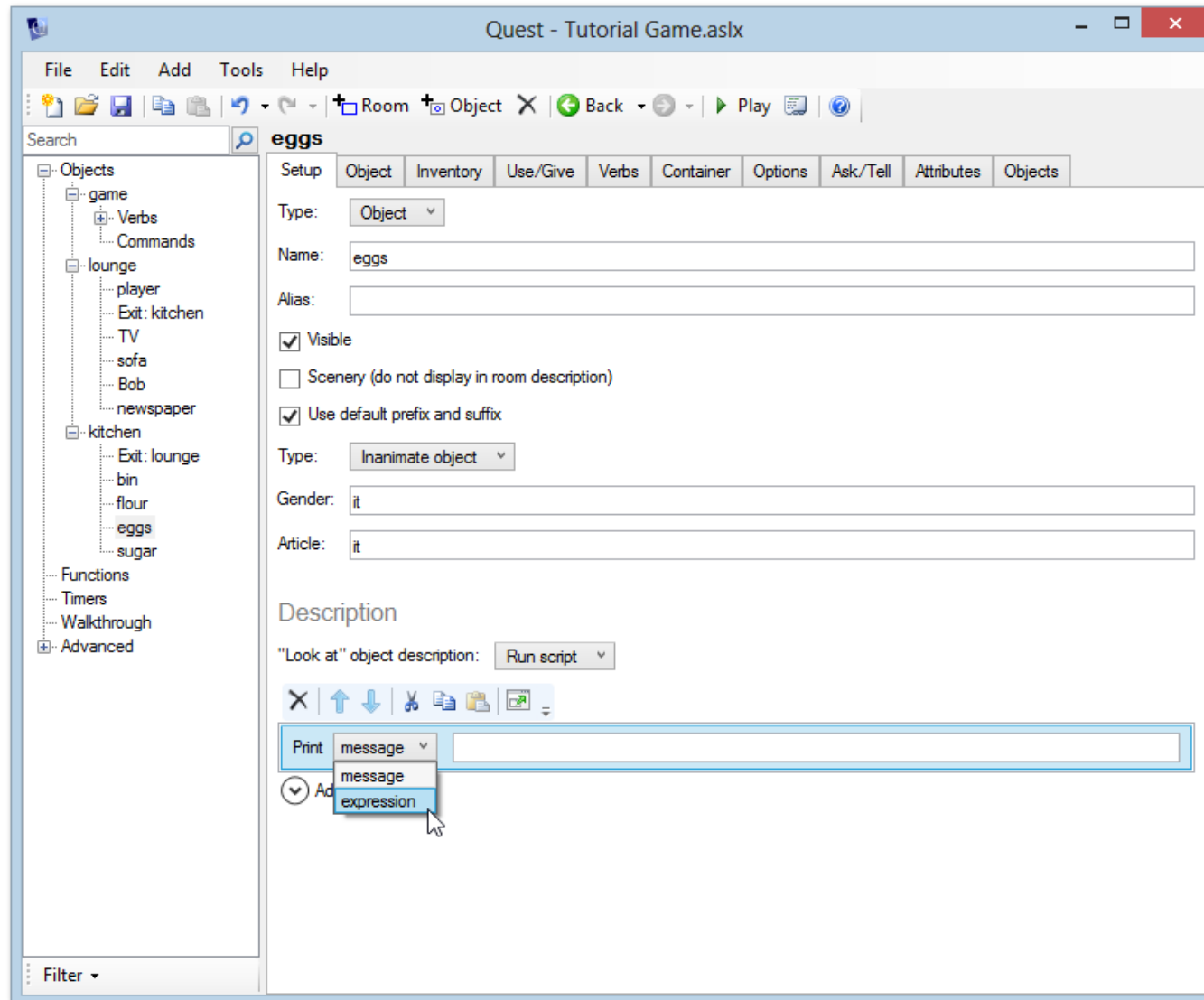
```
object.attribute
```

For example, to read the weight of the eggs, you would type:

```
eggs.weight
```

Let's update the "look at" description of the eggs, as an example. Select the eggs object, and then under the "Setup" tab, change the "Look at" description to "Run script". Add the "print a message" command. Now, instead of printing a normal message (which would never change, and can't read attributes), we want to print the result of an expression. So, click the "message" drop-down and select "expression" instead.

[home](#)



[home](#)

Enter this expression exactly, including the quotation marks in the correct place:

```
"A box of eggs, weighing " + eggs.weight + " grams."
```

This will insert the value of the “weight” attribute in our text.

Always make sure that your object and attribute names match the expression exactly - it is recommended that you always use lower-case object and attribute names, as these names are case-sensitive.

Launch the game and verify that the correct response is displayed – it should read “A box of eggs, weighing 250 grams.”

Of course, we could have just manually entered this into the description of the eggs anyway – we didn’t really need to use an attribute. The real power of this, though, is that you can easily change attributes while the game is running. We could change the “weight” attribute of the eggs for example, if the player used some of them to bake a cake (but, er, hopefully you’ll have some slightly more exciting ideas for things that players can do in *your* game). After the attribute is updated, our “look” description would automatically reflect the current weight of the eggs.

Next: Custom commands

Custom commands

In this section, we will add a **command** that lets the player “say” something, and another to “weigh” an object.

Note that we will *not* be using a **verb** here, as we have done before. Why not? Verbs are good when you want to have a *separate* response for each object, what we want is *one* script that will return information about *any object* in the case of “weigh”, or that has no object in the case of “say”. For this, we need to use a **command**.

Adding a Simple Command

Let’s add a simple command - “say”. This will let the player type conversation prefixed with the command “say,” for example “say hello”. Quest will respond with “You say ‘hello.’ We will also add the contextual text of “but nobody replies” as no-one is present in the game at this point.

To add a command:

- In the Windows desktop version, select “game” in the tree. Now, you can right-click on the tree and choose “Add Command”, or use the Add menu and choose “Command”.
- In the web version, select “Commands” in the tree (underneath “game”). Then click the “Add” button.

Enter the following text into the command pattern box:

```
say #text#
```

This pattern handles the player typing the command “say” followed by any text. The text following the “say” command is then put into a string variable called “text”.

For example:

- If the player types “say hello”, the “text” string variable will contain “hello”
- If the player types “say what a lovely day”, the “text” string variable will contain “what a lovely day”

We can read string variables within an expression, in exactly the same way as we read object attributes in the previous section.

Whenever the player types in a command that matches the command template, the command’s script will be run. Let’s now add a script to print the required response. Add a “print a message” command and choose “expression”. Then enter this expression:

[home](#)

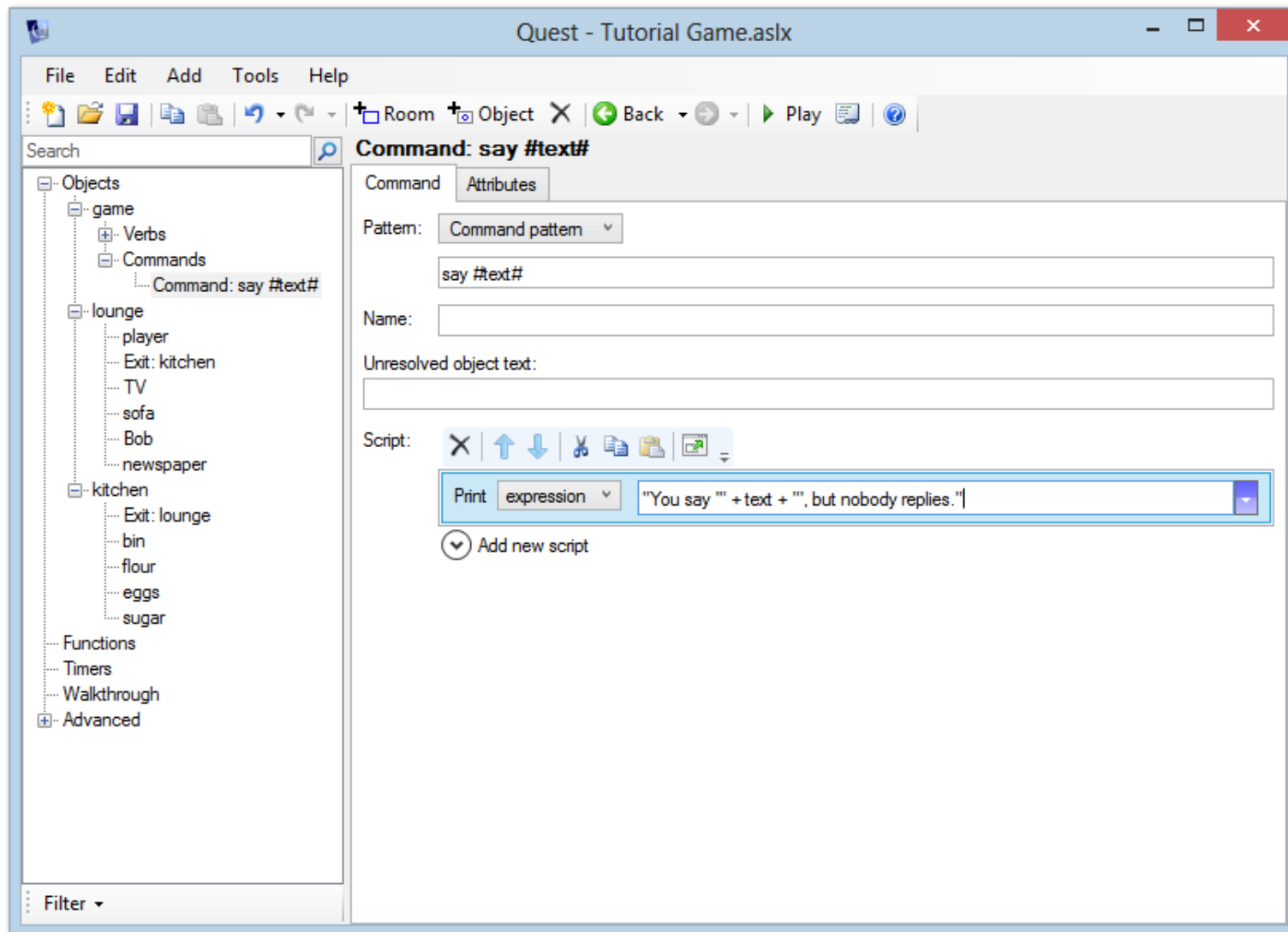
```
"You say \" + text + "\", but nobody replies."
```

What's with the backslashes in there? They let us use quotation marks inside an expression - if you put a backslash before a quote character, that quote character won't be interpreted as the end of a string.

But if you don't like the backslashes, you could use single quotes quite safely instead:

```
"You say ' + text + '", but nobody replies."
```

[home](#)



Launch the game and type in a few “say” commands to see that Quest responds correctly.

Alternative Command Patterns

You can easily add alternatives to a command pattern by separating them with semicolons. For example, we could adapt our “say #text#” command to deal with “shout” and “yell” by modifying the pattern to read:

```
say #text#; shout #text#; yell #text#
```

Adding a “Weigh” Command

We now know how to add a command that will process any kind of text the player enters. However, a lot of the time, our commands will be dealing with objects that the player can see. To handle objects correctly, just use the variable name “object”. So the “weigh” command’s pattern should be:

```
weigh #object#
```

If you want to create a command that uses multiple objects, you can call your variables “object1”, “object2” etc. - in fact anything starting with “object” will work.

The script we enter for the “weigh” command should respond “It weighs X grams”, where X is the weight of the item – as reported by its “weight” attribute.

We can read the weight attribute in the same way as before:

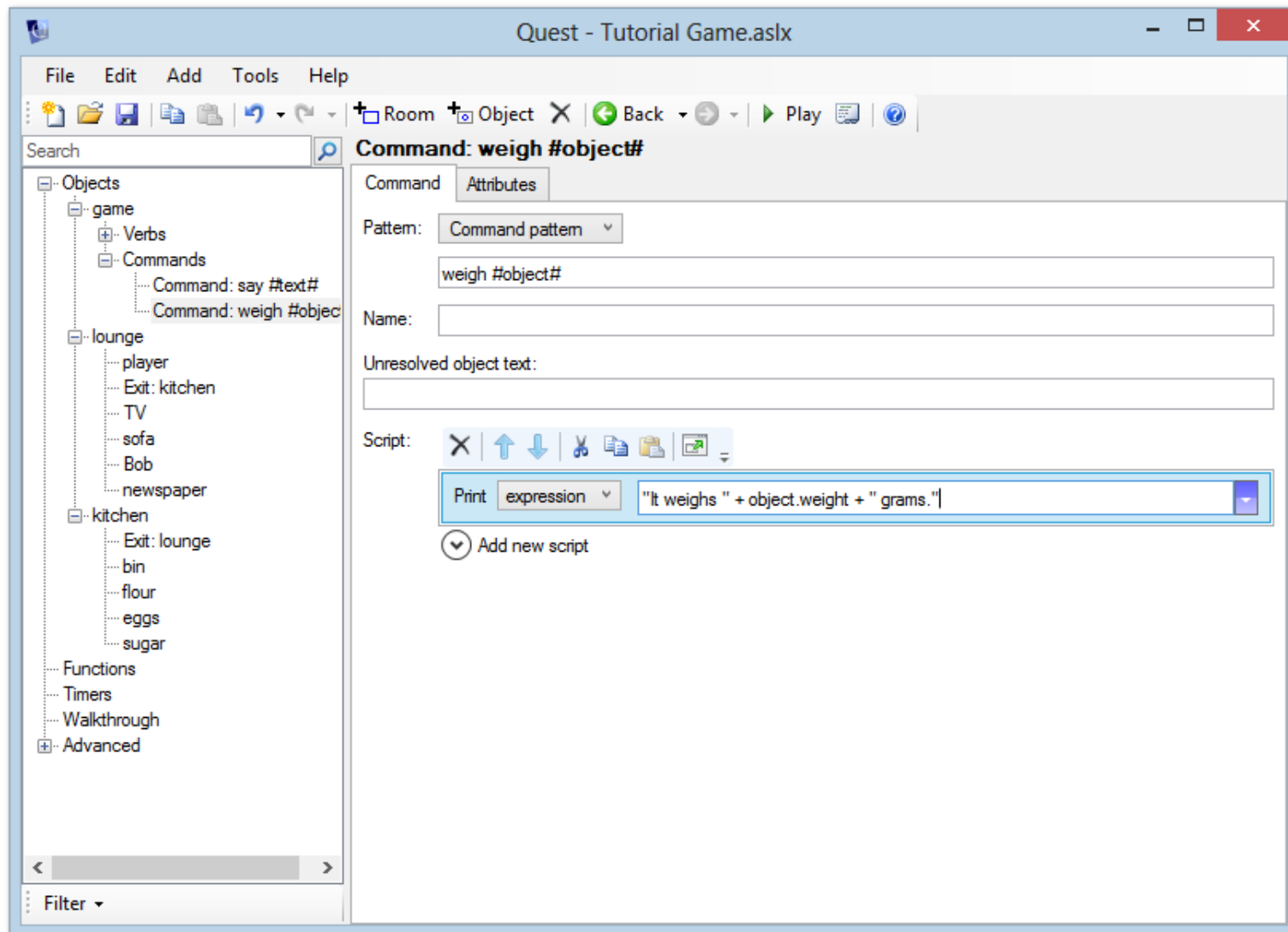
```
object.weight
```

Previously, we put an actual object name before the dot. This time, we’re putting a variable name there - so we’ll read the “weight” attribute of the object that the player entered.

Add the “weigh” command using the command pattern above, and add a “print a message” command to print this expression:

```
"It weighs " + object.weight + " grams."
```


[home](#)



[home](#)

Launch the game and go to the kitchen. See what happens when you type “weigh flour”, “weigh sugar” etc.

Now go back to the lounge. What happens when you weigh Bob?

Quest responds with “It weighs grams.” Why? Because he doesn’t have a “weight” attribute. Since we don’t want to have to enter a weight for every single object in the game, we’ll need to update our command so it checks for the existence of the “weight” attribute, and then prints the appropriate response.

Checking for an Attribute

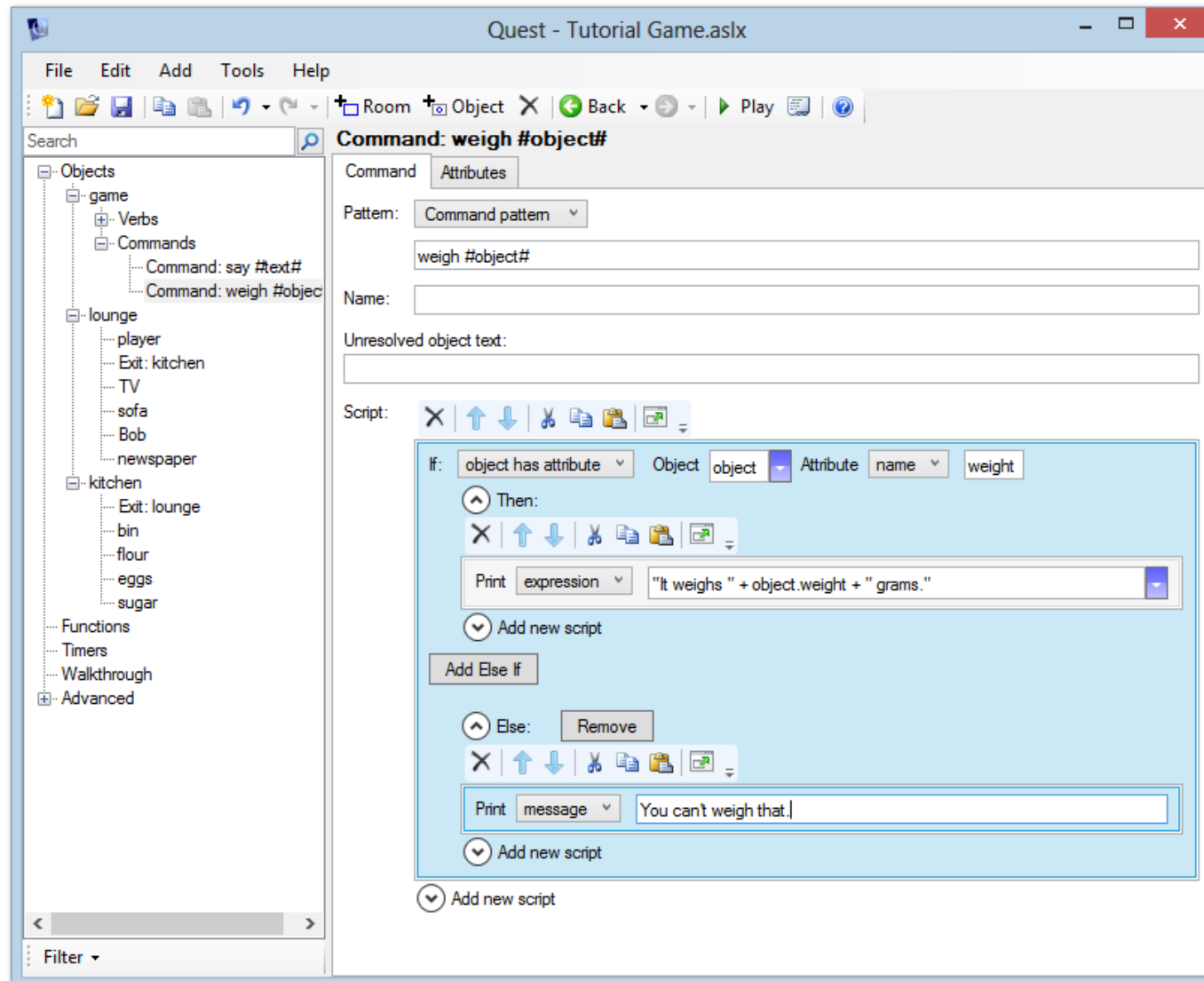
Go back to the script for the “weigh” command, select the existing “Print a message” command and click the “Cut” button on the Script Editor to move this to the clipboard.

Now, add a new “if” command. From the dropdown, select “object has attribute”. Enter the object expression “object”. For the attribute, leave “name” selected and type in “weight”.

Expand the “Then” script, and click Paste to restore the previous “print a message” script. For the “Else” script, print a message like “You can’t weigh that”.

The script should now look like this:

[home](#)



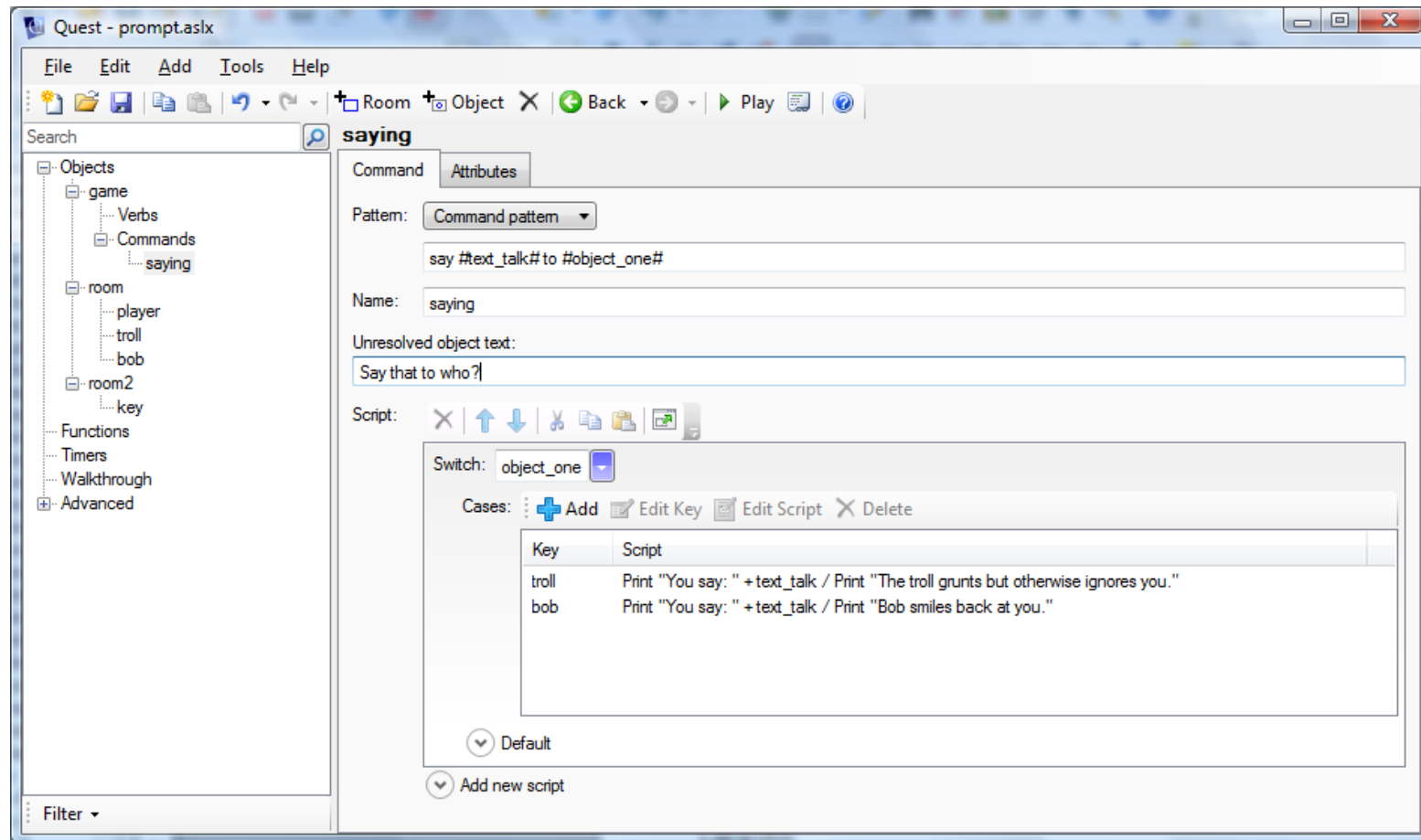
[home](#)

Launch the game and verify that you now get a sensible response for “weigh bob” and “weigh sofa” in the lounge (it should say “You can’t weigh that”) and that you can still weigh the items in the kitchen.

Additional Example (Advanced)

Quest can handle text and objects in the same command. Here the say command is extended to allow the player to specify who she is talking to.

[home](#)



The pattern you are using is this:

```
say #text_talk# to #object_one#
```

Quest will attempt to match #object_one# to an object present, and if it does then an object variable called “object_one” will be set to that object (if it cannot, Quest will output whatever you typed in the “Unresolved object text” box). The text part will match any text at all, just as before.

Suppose the player types:

```
say hi to troll
```

Quest matches “say” and “to” directly. It then matches “hi” to the text, so now the string variable “text_talk” is set to “hi”. Then it matches the object, as long as the troll is here, and sets the object variable “object_one” to the troll.

The script uses a switch command so you get a different response for different characters, and a default too.

You can set up commands with multiple objects just by giving them each their own name between the # marks.

Next: More things to do with objects

More things to do with objects

Giving and Using objects

After a player has taken an object, they can give that object to, or use it on, other objects in the game.

For example, after picking up some flowers, a player can give them to a character called “Susan” by typing “give flowers to susan”.

“Give” and “Use” are set up in exactly the same way – the only difference is whether the player types “give ... to ...” or “use ... on ...”.

In this example, we are going to revive the corpse of Bob in the lounge, using a heart defibrillator. First, we need to alter the setup of Bob so that we give a correct description whether he is dead or alive. To do this, we are going to use an **object flag**.

An object flag is simply a way of accessing a boolean attribute of an object (a boolean attribute can only be either “true” or “false”). You can use it to mark certain things as “done”, and is a common way to track the progress of a game.

It is good practice to give flags meaningful names, and to have them start off (or false). For Bob, we will call the flag “alive”.

So, using the defibrillator on Bob is going to add a flag called “alive” to Bob. In the “look at” description, we’ll check whether Bob has his “alive” flag set. If so, we’ll print “Bob is sitting up, appearing to feel somewhat under the weather”, and if not, we’ll print the old description “Bob is lying on the floor, a lot more still than usual.”

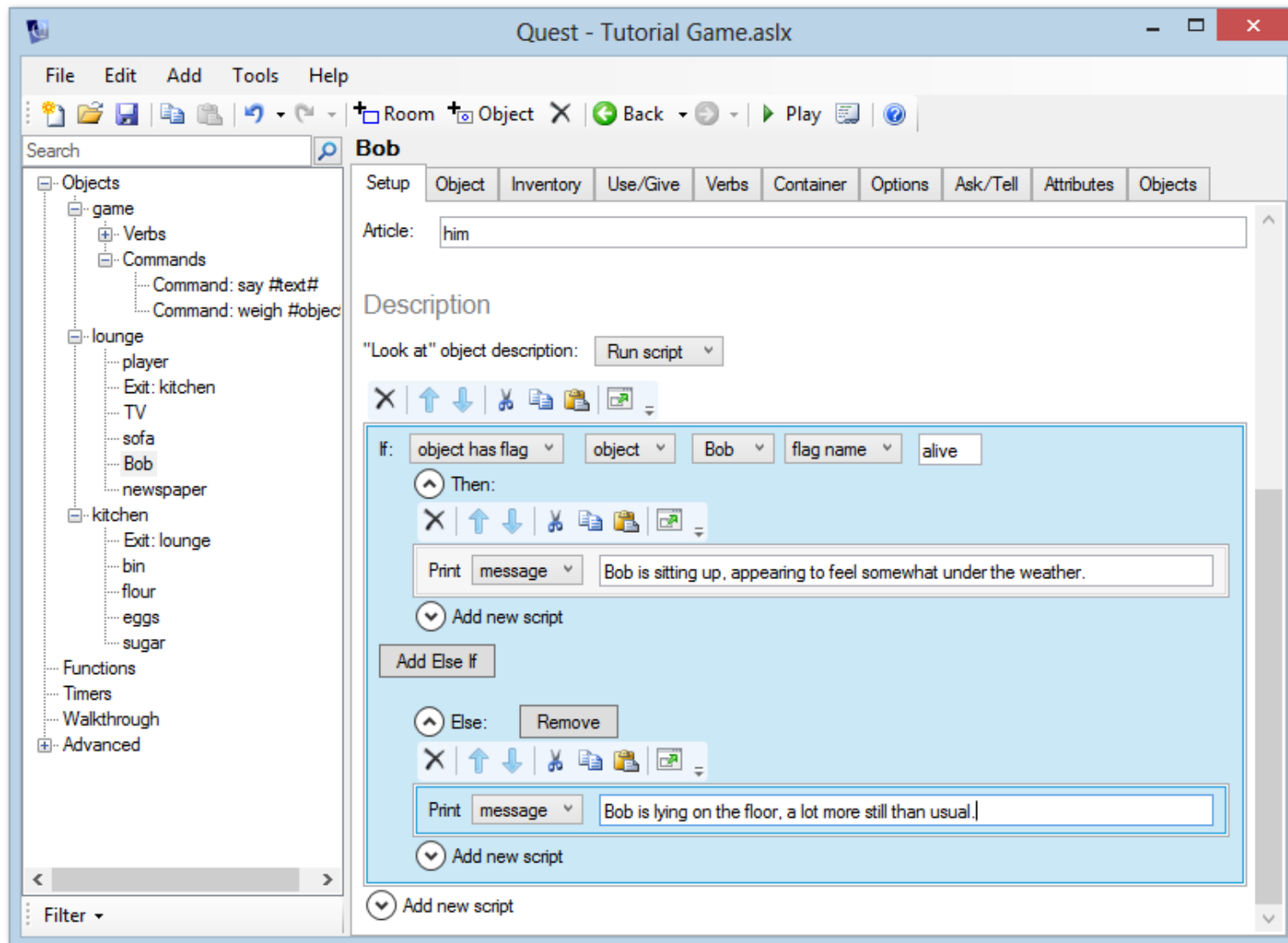
Updating the description

Getting the correct description to display is very similar to the “watch” example in the Using scripts section.

Change Bob’s “look at” description to “Run script”, and add an “if” command.

For the expression, choose “object has flag”. Then select “Bob” and enter the flag name “alive”. Add the descriptions above for “then” and “else” messages.

[home](#)



Using the defibrillator

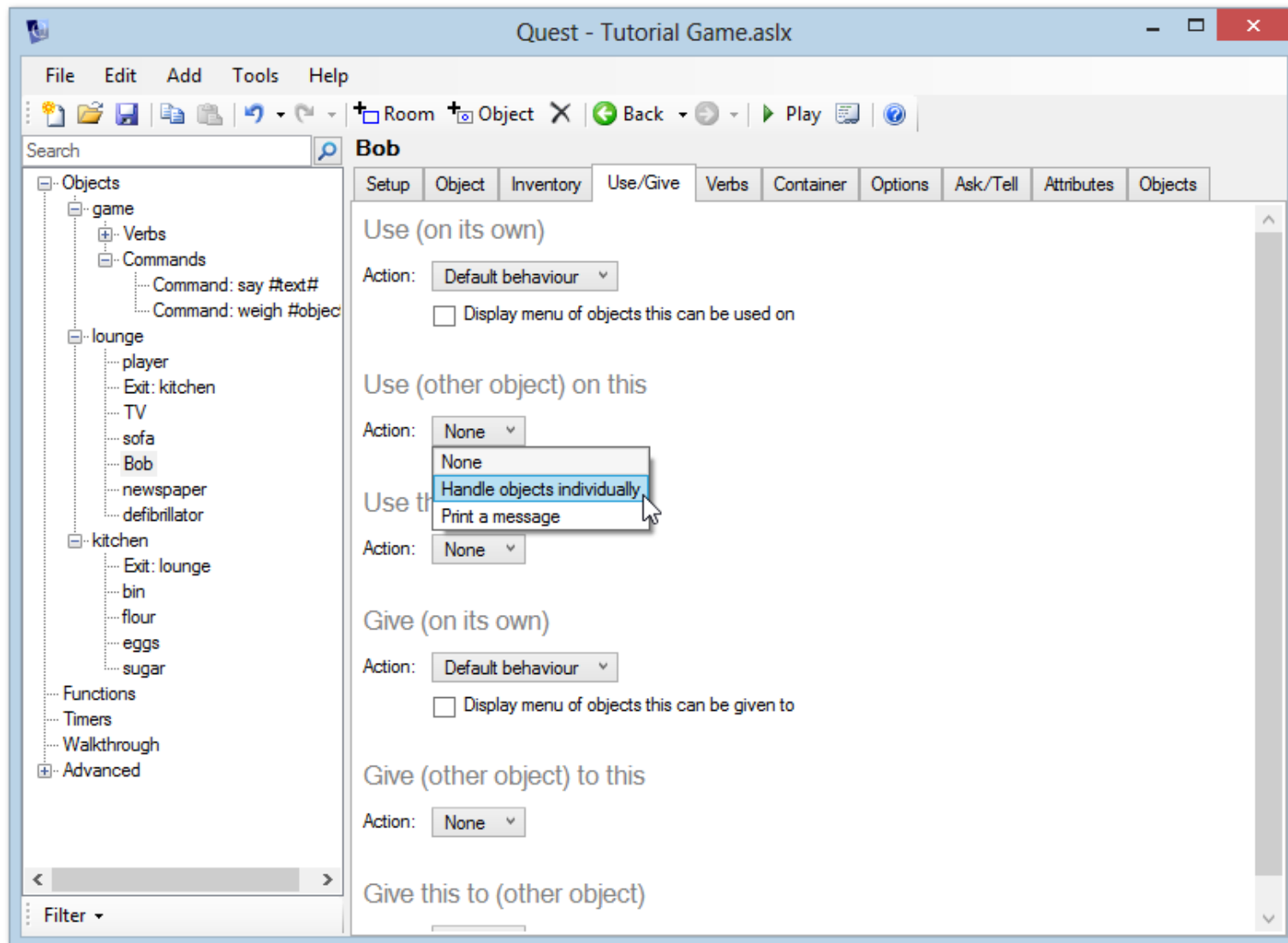
Now, add a “defibrillator” object to the lounge. Enter a description like “A heart defibrillator can magically revive a dead person, if all those hospital dramas are to be believed.”

Go to the *Inventory* tab and tick “Object can be taken”.

Now go back to Bob. We need to turn on “Use/Give”, so on his “Features” tab, tick the “Use/Give:...” check box. His *Use/Give* tab should now appear.

Go to the “Use (other object) on this” section of the new “Use/Give” tab, and choose “Handle objects individually”.

[home](#)



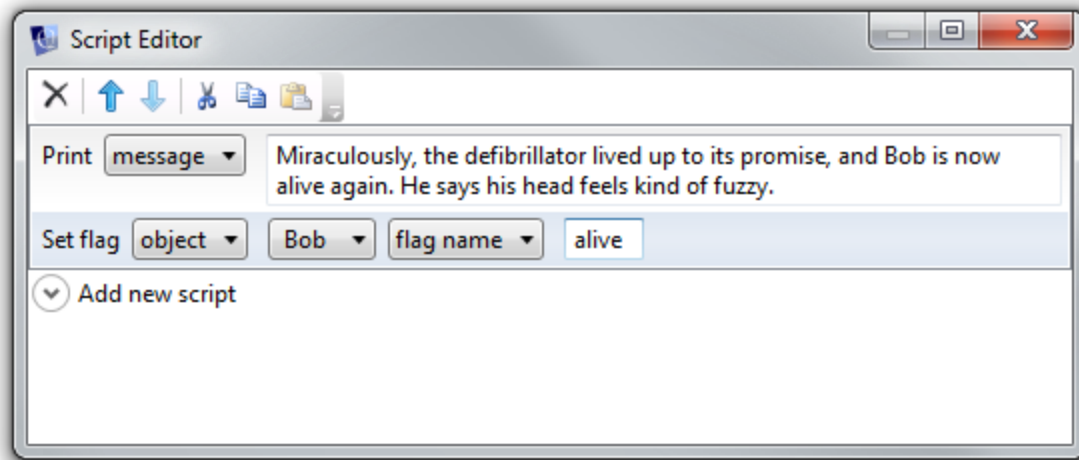
[home](#)

A list will appear in the “Use” section. Here you can add the objects that can be used on Bob, with a script for each. Click “Add” and type “defibrillator”.

A Script Editor window will now pop up. In this script, we want to print a message to tell the player what’s happening, and also set the “alive” flag on Bob.

So, add a “print a message” script and type something like “Miraculously, the defibrillator lived up to its promise, and Bob is now alive again. He says his head feels kind of fuzzy.”

Now add a “Set object flag” command. Choose Bob from the objects list, and enter the flag name “alive”.



Close the Script Editor window. Now run the game. Look at Bob, then type “use defibrillator on bob”, then look at Bob again. Verify that you see the correct text in each case.

Notice what happens when you type “use defibrillator on bob” a second time - you get the same response again. You should know how to fix this now - update your “use defibrillator on bob” script to check for the “alive” flag. Update this now (if you are struggling, just move on to the next section, where it will be revealed!).

Using Functions

It would be good if we could get the same effect just by typing “use defibrillator”. There are a couple of things we could do here, from the defibrillator *Use/Give* tab, under “Use (on its own)”:

- we can tick “Display menu of objects this can be used on”. This will allow the player to select Bob to use the defibrillator on.
- we can choose “Run script” from the Action list, to automatically defibrillate Bob.

The first option is definitely the easiest, but the second option allows us to demonstrate the use of functions.

Of course, we could simply copy the script we created in the section above, and paste it into the defibrillator’s “use this object (on its own)” script. However, if we then wanted to make an update to one script, we would have to update the other one as well.

The best way to resolve this is to make both our existing “use defibrillator on Bob”, and our new “use defibrillator” point to the same script. The way to do this is to set up a **function**.

Functions provide a way for you to set up scripts that can be called from anywhere in your game, so you don’t have to keep copying and pasting or re-entering the script.

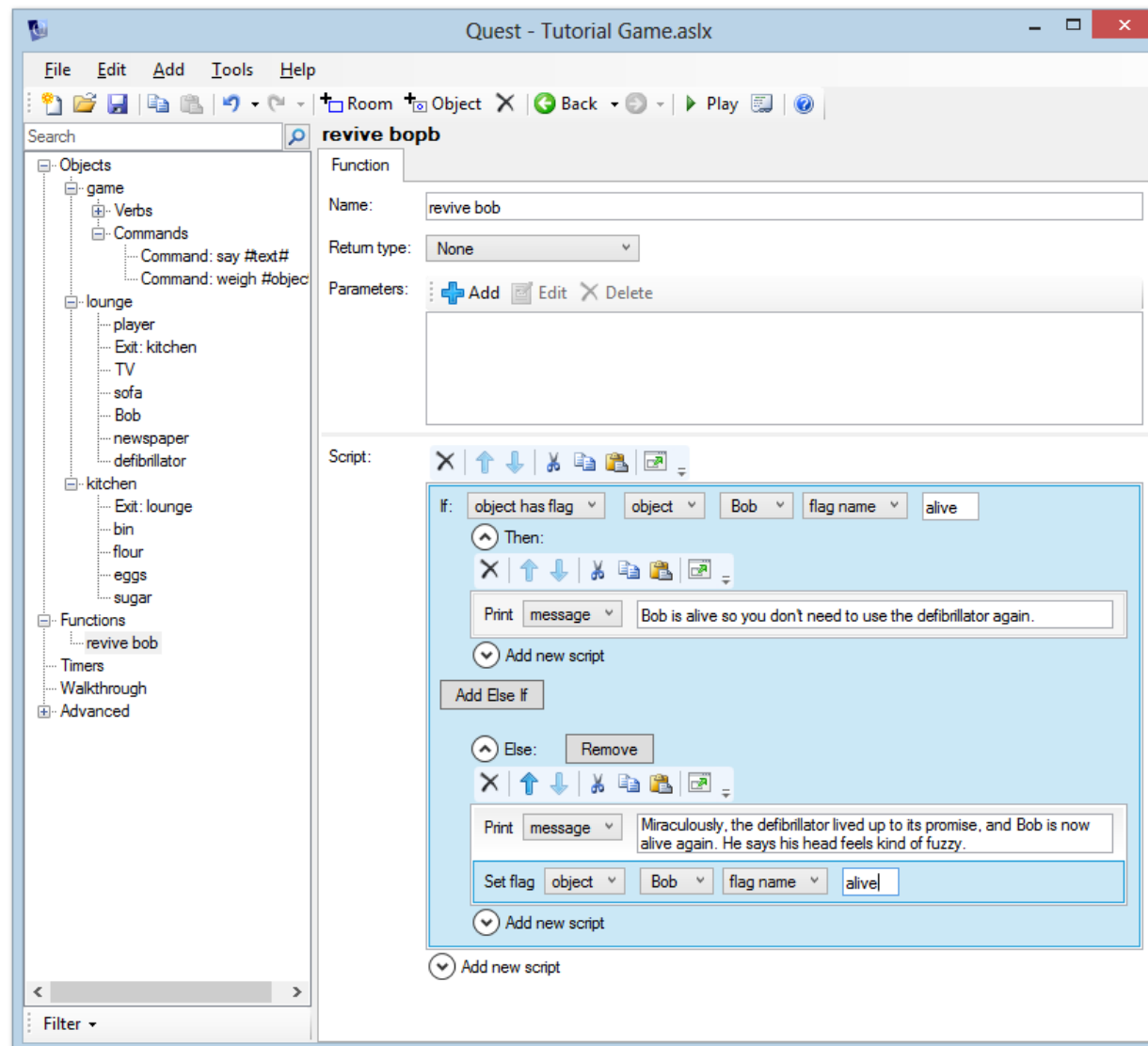
[home](#)

Let's create one now to store the script commands we use to resuscitate Bob. We can then set both "use defibrillator on bob" and "use defibrillator" to call this function.

First, go to Bob's *Use/Give* tab, and double click "defibrillator" in the "Use" table to bring up the Script Editor. Hold down the shift key and select all script lines, then click the Cut button to move this script to the clipboard. Now close the window.

Now add a new function (right click the tree, or use the Add menu), and call it "revive bob". For the script, click the Paste button.

[home](#)



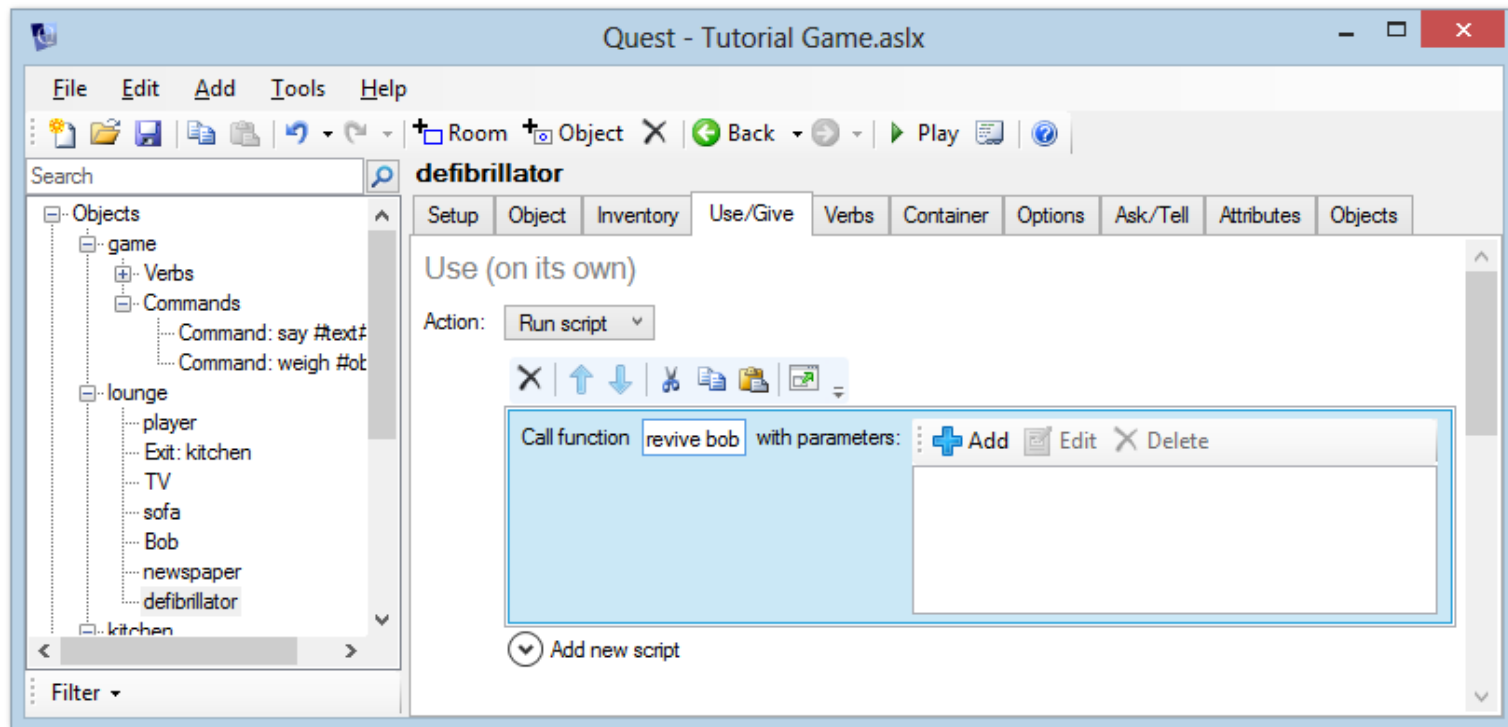
[home](#)

Now we just need to update “use defibrillator on bob” and create “use defibrillator”, to make them call this function. Go back to Bob’s use/give tab, and double-click to edit the defibrillator script again.

Add a “call function” script, and enter the name “revive bob”. Now close the window.

Now we’re kind of back where we started - if you run the game and type “use defibrillator on bob”, it calls the “revive bob” function and so we get exactly the same behaviour as before.

We just need to make “use defibrillator” call the same function now, so go to the defibrillator’s use/give tab, and in the “use (on its own)” section choose “Run a script”. Add a “call function” script, so that it also calls the “revive bob” function.



Launch the game now and verify you get the same response whether you type “use defibrillator on bob” or just “use defibrillator”.

Note that if you pick up the defibrillator and go to the kitchen, “use defibrillator” will still work. It would be pretty remarkable for a defibrillator to work at such a long range, so consider adding an “if” command to the “revive bob” procedure. You can select “player is in room” from the list of conditions to check whether the player is in the lounge before carrying out the defibrillation. If they’re not in the lounge, print a suitably sarcastic message.

Giving objects

Giving an object to a character works in exactly the same way as using objects on a character. Look at the “Give” sections - notice that you get the same options as for “Use”. You can add objects to it in the same way.

Ask and Tell

Ask and Tell work in the same way, so we’ll only cover “ask” here.

Click on Bob and go to the “Ask/tell” tab. As with “Use/Give” this will have to be turned on, but it is done globally, which means you only need to do it once for your whole game, and that it is done from the *Features* tab of the “game” at the top of the list on the left.

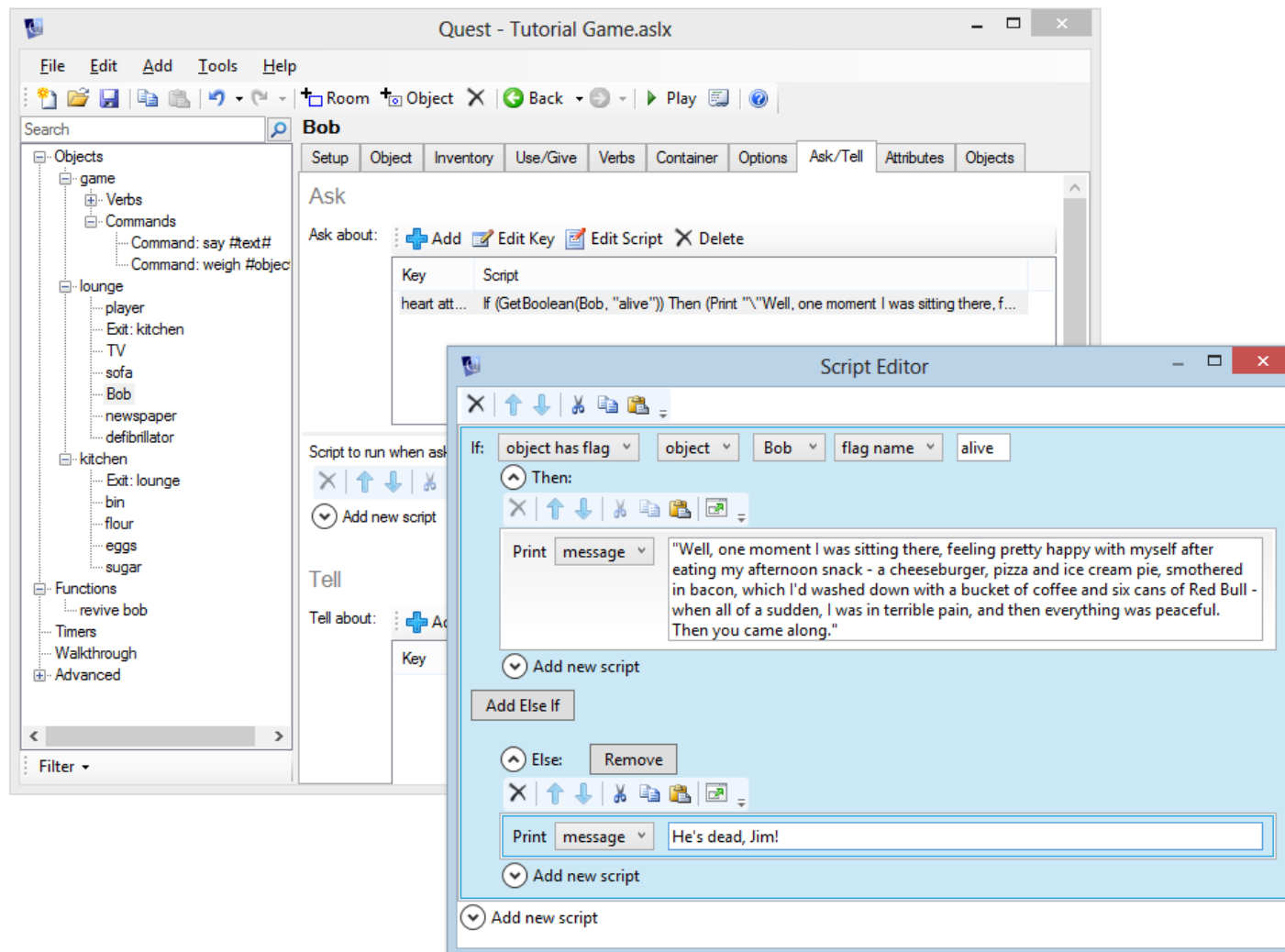
On Bob’s “Ask/Tell” tab you can add subjects to the list, and give a script for each subject. You can also give a script to run when the player asks Bob about something he doesn’t know about.

Let’s make Bob respond to a question about the massive heart attack he’s just amazingly

recovered from. Click the “Add” button and enter some topic keywords, for example “heart attack cardiac arrest”. When the player asks Bob about anything, this list of keywords is checked for matches in the player’s command. So the player could type “ask bob about heart” or “ask bob about cardiac arrest”, and that will match this topic.

A Script Editor will appear. It would make sense that the player can only ask questions of Bob when he’s been brought back to life, so the first thing to do is add an “if” command and check that Bob’s “alive” flag is set. If it is, he can say something like “Well, one moment I was sitting there, feeling pretty happy with myself after eating my afternoon snack - a cheeseburger, pizza and ice cream pie, smothered in bacon, which I’d washed down with a bucket of coffee and six cans of Red Bull - when all of a sudden, I was in terrible pain, and then everything was peaceful. Then you came along.”

[home](#)



Next: Using containers

Using containers

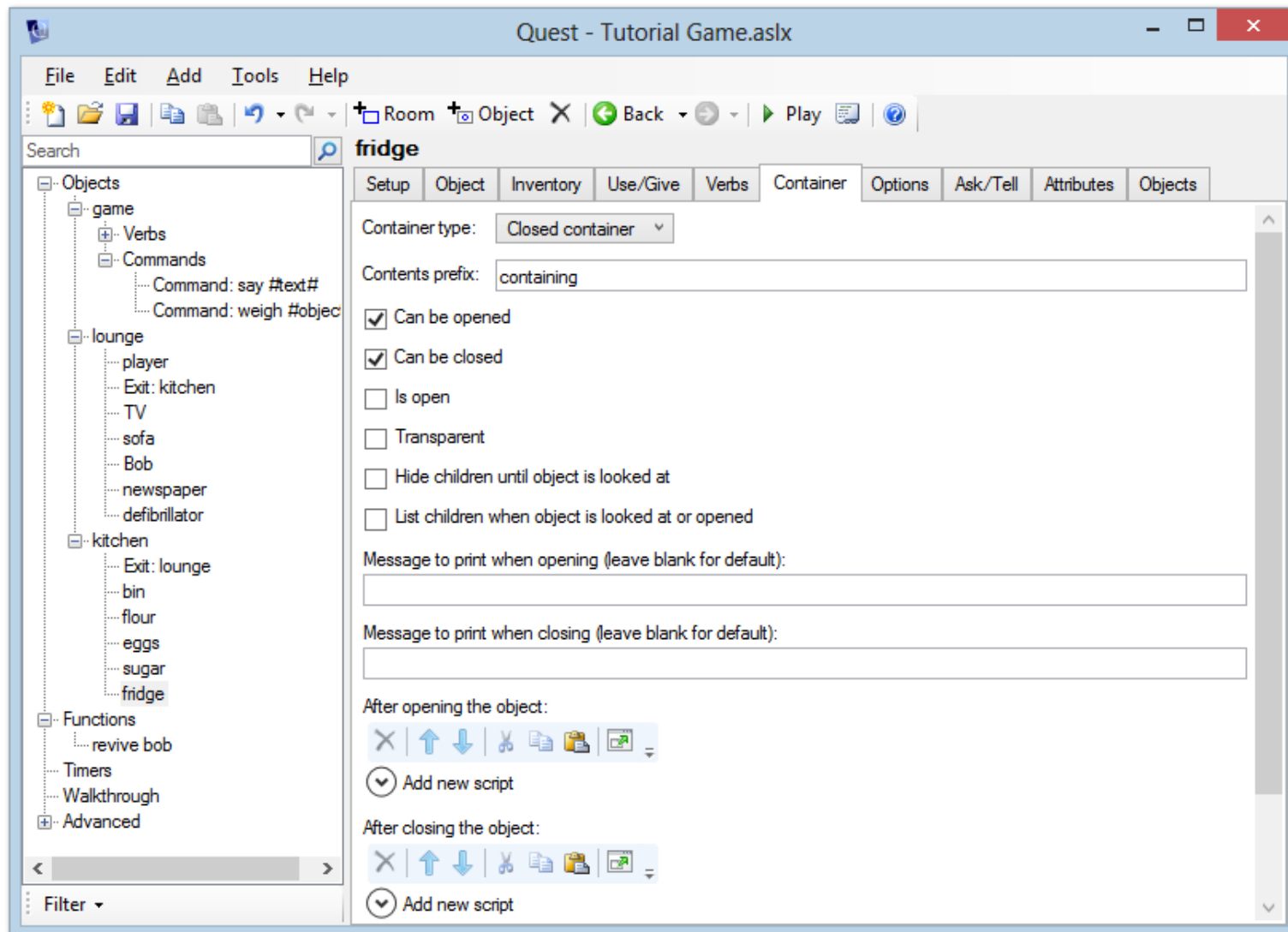
Containers are objects that can contain other objects. In this example, we'll create a "fridge" object in the kitchen, which contains several items of food and drink. The fridge is initially closed, so these items will only be visible once the player has opened the fridge.

Creating the Fridge

Create a "fridge" object in the kitchen and give it a description like "A big old refrigerator sits in the corner, humming quietly."

Now let's set the fridge up as a container. This is a feature, so first go to the *Features* tab, and tick "Container". Click the *Container* tab. By default, "Not a container" is selected. Change this to "Closed container". The Container options will now appear.

[home](#)



By default, the player can open and close the fridge. We're going to add some objects to the fridge in a moment, and it would be good if the contents were listed when the player opened the fridge, so tick the "List children when object is looked at or opened" option.

Adding objects to the fridge

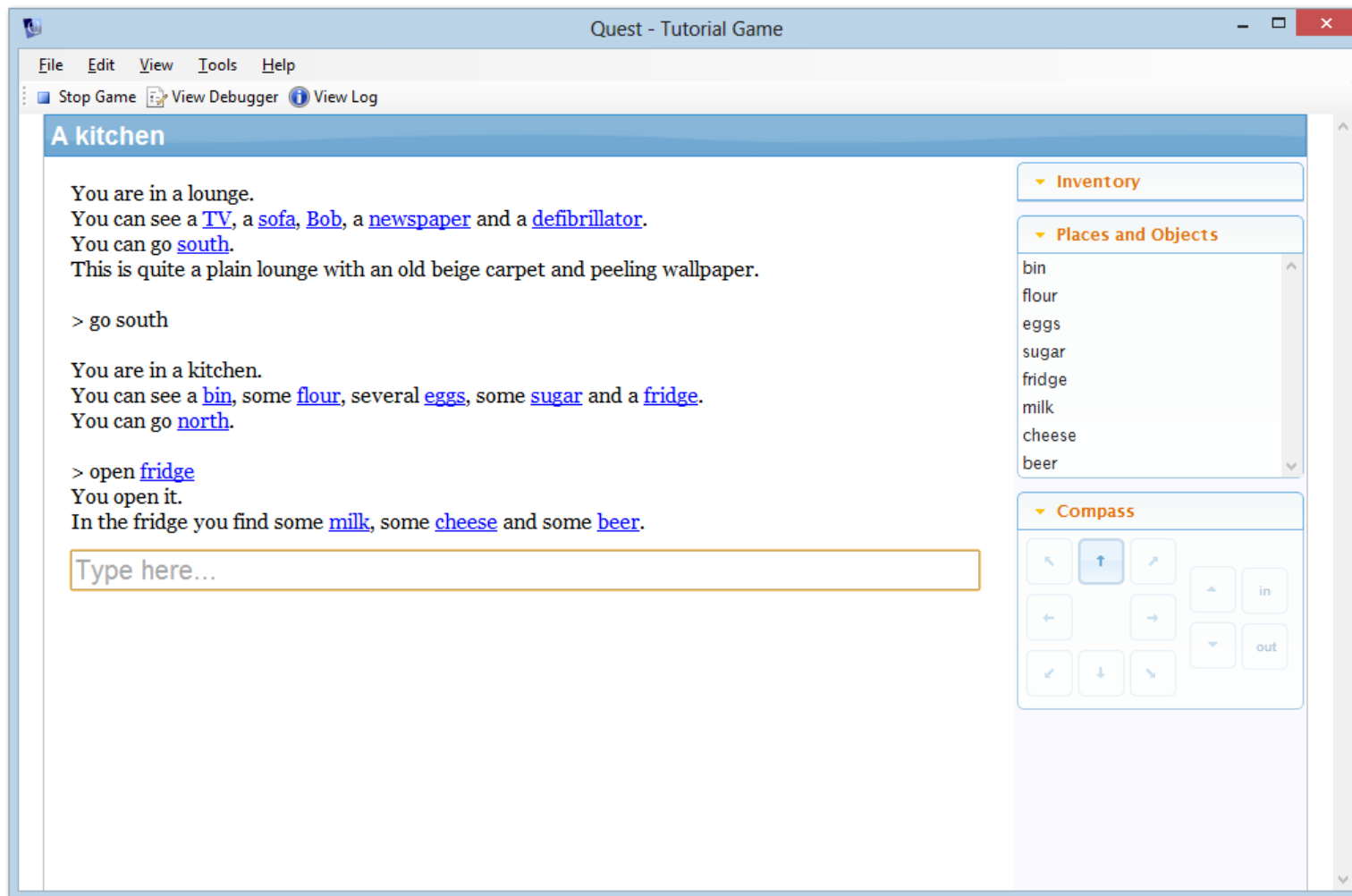
Now let's create some objects inside the fridge. To do this, we just create these objects as normal, but on the "Add object" window we set the parent to "fridge". Alternatively you can move objects. In the desktop version, just drag them around the left pane. For the web version, click on the "Move" button, towards the top right.

Add the following objects: milk, cheese, beer. Give each object a sensible description. The prefix for each object should be "some", so that the room description sounds natural. Allow each object to be taken.

Now run the game and go to the kitchen. Notice that you can't see the milk, and if you type something like "look at milk", Quest will tell you that it's not here. Now open the fridge, and the objects inside it will be revealed.

By setting the "List prefix" you can change the "It contains" text which appears before the list of objects.

[home](#)



Updating the description

In your “look at” description, you can check if the object is open by running a script. Add an “if” command and choose “object is open” - then you can print a different message depending on whether the fridge is open or closed.

When the fridge is open, you might print “The fridge is open, casting its light out into the gloomy kitchen”. When it is closed, you might print “A big old refrigerator sits in the corner, humming quietly”.

As an exercise, add a closed cupboard to the kitchen. Add a few items to the cupboard such as a tin of beans, a packet of rice etc. The player should be able to open and close the cupboard. When Quest lists the contents of the cupboard, it should say something like “The cupboard is bare except for ...”

Transparency

When you set the “Transparent” option, the player can see what objects are inside the container, even if it is closed.

Although the player can see what’s inside a transparent container, they still can’t take objects from it or put objects in it unless it is open.

Surfaces

Surfaces act very much like containers - they act as an always-open container, and objects that are on a surface are visible in a room description even before the player has looked at the surface. For this reason they’re a good choice for implementing things like tables. As an exercise, change the table object in the lounge to make it a surface (or create it if you haven’t already). Then move the newspaper so that it is on the table.

Lockable Containers

What if you don't want the container to be immediately openable? If it's part of a puzzle, you may want the player to have a particular "key" object before they can open it. To implement this, you can make the container lockable.

Let's create a small (and, admittedly, tedious) puzzle - we're going to put the defibrillator in a locked box. The player must get the key from the kitchen, unlock the box, and then take the defibrillator from the box before they can revive Bob.

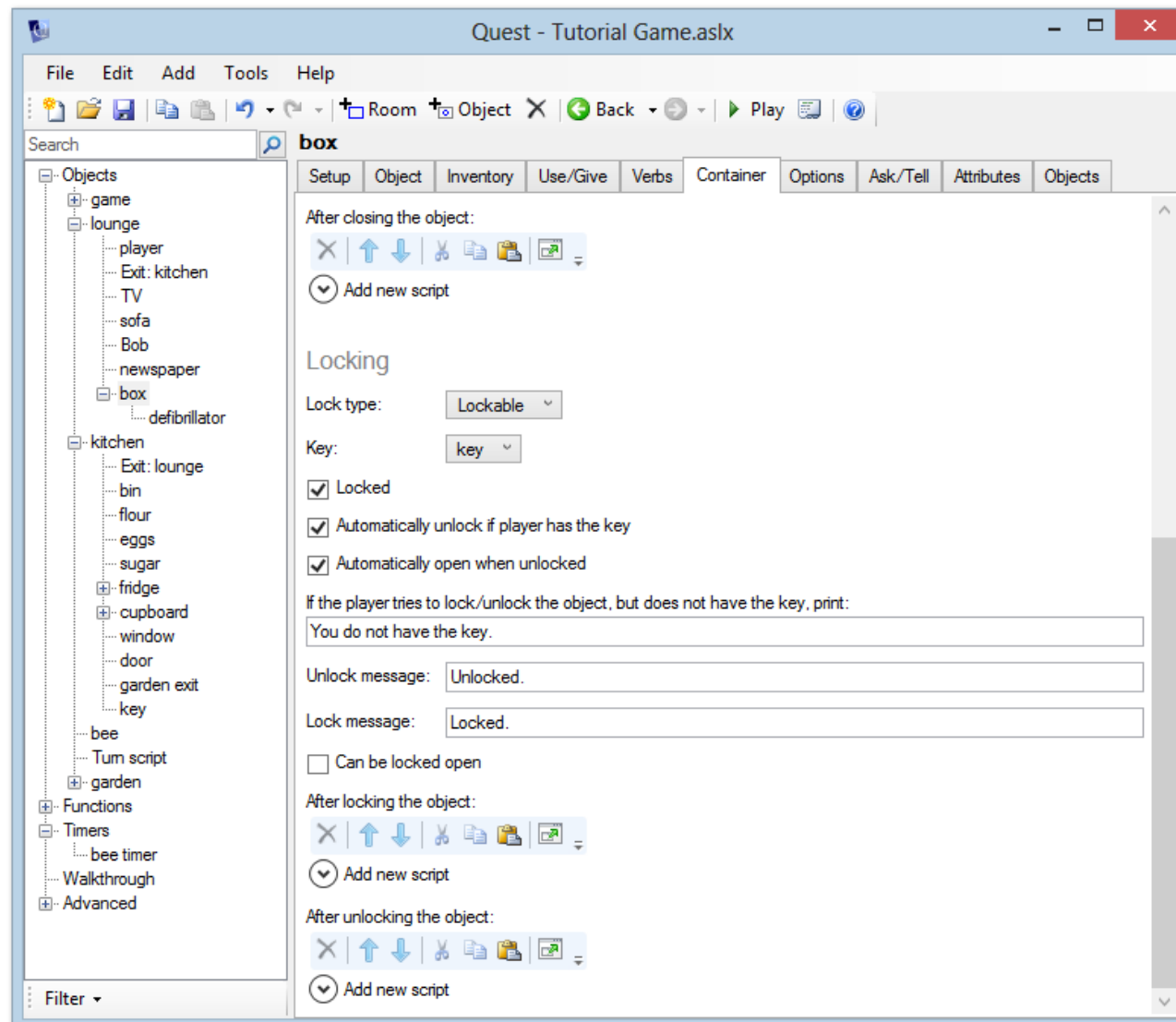
Please bear in mind this is probably the most boring puzzle imaginable. It is just an example. Don't use it as a guide for something that would make your game more exciting - it's up to you to think of interesting puzzles!

First, set up the objects:

- create a "box" object in the lounge. Make it a closed container.
- move the "defibrillator" object to the box (you can click and drag in the tree in the Windows version, or if you're using the web version you can select the defibrillator and then click the Move button in the top-right of the screen)
- in the kitchen, add a "key" object, and make it takeable.

Now, make the box lockable. Go to the Container tab and in the "Locking" section, choose "Lockable" from the lock types list. This will display the lock options. You can now choose the "key" object from the list.

[home](#)



[home](#)

By default we have the “Automatically unlock if player has the key” and “Automatically open when unlocked” options turned on. This is out of politeness to players really, as there’s no need to force them to jump through hoops and perform additional steps - if they’ve unlocked the object, it’s a fair bet they want to open it, and if they type “open box” before unlocking it, then if they have the key, there’s no point in forcing them to type “unlock box” first.

It might be a good idea to tick the “List children when object is looked at or opened” option, in the main Container options. Now your game output will look like something this:

```
> open box
It is locked.

> unlock box
You do not have the key.

> s
You are in a kitchen.
[rest of kitchen description snipped...]

> take key
You pick it up.

> n
You are in a lounge.
[rest of lounge description snipped...]

> unlock box
Unlocked.
You open it.
It contains a defibrillator.
```

Next: Moving objects during the game

Moving objects during the game

As your game unfolds and the player interacts with your world, you may want to bring additional objects into play, or remove others. In this example, we'll add a window to the kitchen. When the player opens it, a bee flies in. In the next section we'll make this bee quite irritating.

Creating a Hidden Object

First, let's create the "bee" object. We don't want this object to appear anywhere when the game starts, so create it outside of a room. If using the desktop version, you can set the parent to "none" when you add the new object, but it may be better to create a special room, perhaps called "nowhere" or "limbo" or "offstage", and keep all your hidden objects there. Give the bee a suitable description.

Bringing the Object into Play

Now, add a window object to the kitchen and give it a sensible description. We want to make this window openable, but it's not a container, as you can't put things in a window. We can't add "open" as a verb though, because the "open" command is handled by Quest's container logic. The solution is to go to the *Container* tab (via the *Features* tab, of course) and select "Openable/Closable" from the Container Type list. This provides basic functionality for opening and closing an object, but it doesn't do anything else.

Choose "Openable/Closable", and now add script commands to the "Script to run when opening object":

- Open object: window
- Print a message: "You open the window and a bee flies into the kitchen."
- Move object "bee" to "kitchen"

For the close script, you just need to add:

- Print a message: "You close the window."
- Close object: window

Launch the game and go to the kitchen. Open the window and verify that you can now look at the bee.

Checking if the Object is Already There

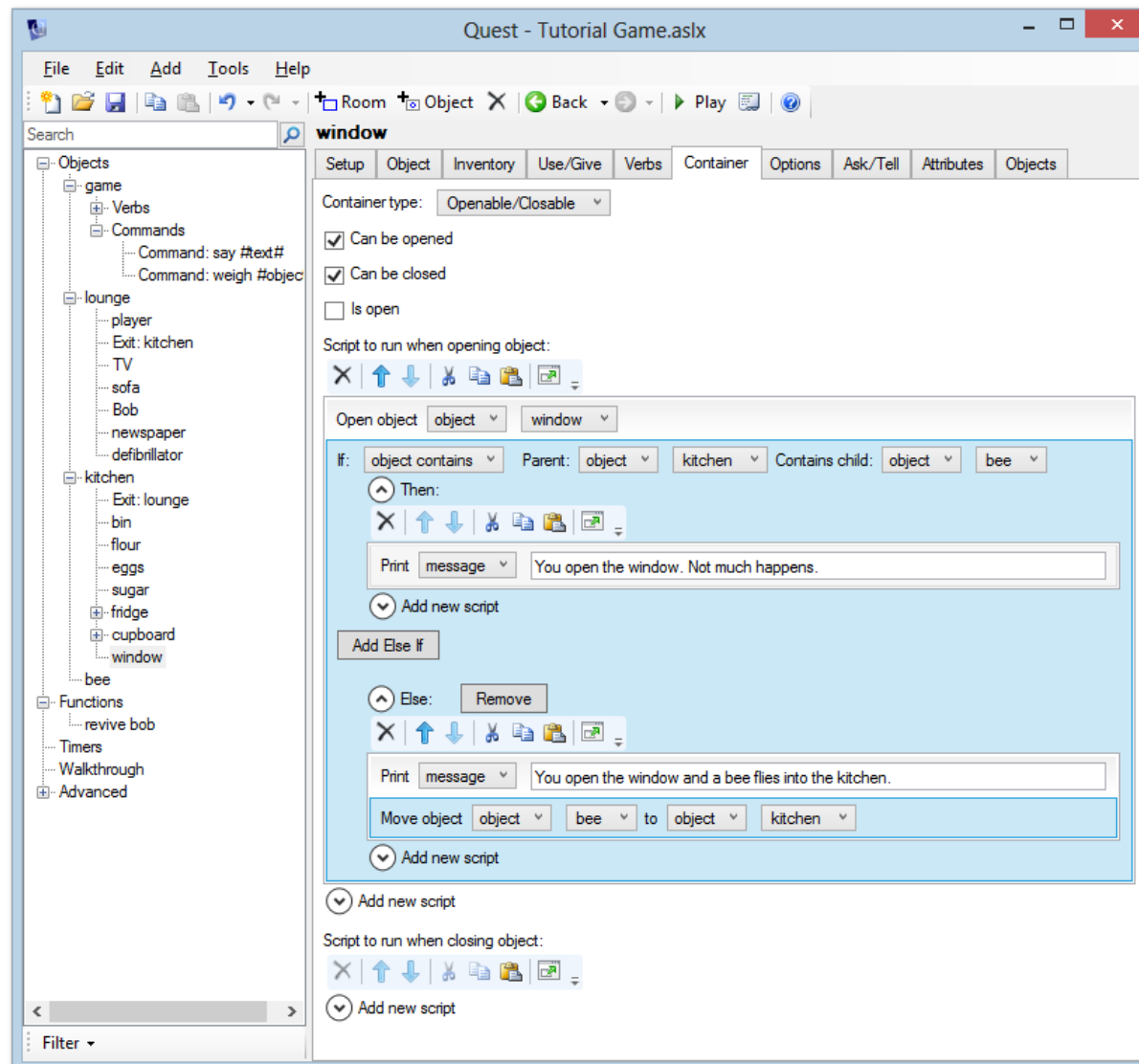
What if the player closes the window and then opens it again? They'll be told that the bee has flown in again, which doesn't make sense as it is already there.

One way to get around this might be to use an object flag, as we've done before. However it's even simpler just to check if the bee is in the kitchen. Add an "if" command and choose "object contains". Now you can select "kitchen" as the parent and "bee" as the child.

For the "then" script, print a message such as "You open the window. Not much happens."

Now cut and paste the existing "print a message" ("a bee flies in...") and "move object" to the "Else".

[home](#)



Removing an Object during Play

As well as bringing an object into play, you can also remove an object from play using the “Remove object” command from the Objects category. This will set the object’s parent to “null”, so you can always bring it back into play again later. To destroy an object entirely, use the “Destroy an object” command - the object will be completely removed from the game. It is more efficient to simply remove the object from play though - it is less work for Quest to simply unset the object’s parent than it is to remove *all* the object’s attributes and destroy it - so it is recommended that you use “remove” in preference to “destroy”.

As an exercise, add an “apple” object, with a sensible description. Add an “eat” verb to the object which will print a message saying “You eat the apple. Tasty.” and then remove the apple from play (though it is worth noting that items can be set to be edible via the Edible tab).

Next: Status attributes

Status Attributes

Often you will want the player to be able to see how they are doing at a glance, perhaps to see the score or health, or how much cash they have. This can be done with status attributes.

Status attributes must be set up as ordinary attributes first. You must then tell Quest that you want these particular ones to be shown in the interface. You can do this with attributes of the player or of the game object, but not anything else in the game. We will set up a score attribute on the player object.

Status Attributes - Desktop

Go to the *Attributes* tab of the player object. In the lower box, click “Add”, then type “score” and set it to an integer. Then go to the upper box, marked “Status Attributes”, click Add.

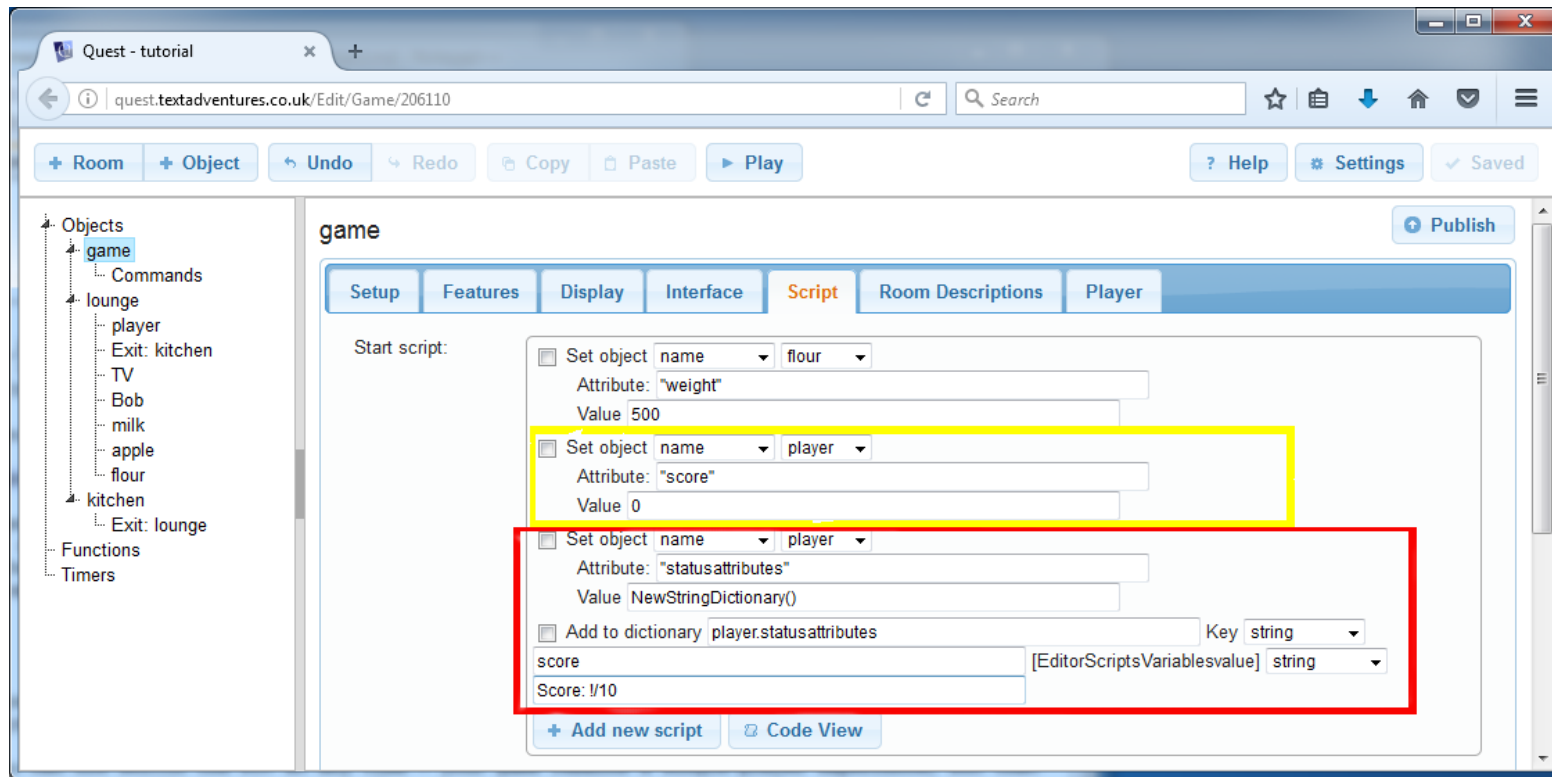
We can going to give Quest two bits of information. The first is the name of the attribute, and the second is how to display it, so again type “score” for the first bit (this must be exactly as you did it before, because Quest will need to match this to the attribute). You can leave the second bit blank, and Quest will decide how to display it, but we try to do it a bit more fancy. Paste in this:

```
Score: !/10
```

The exclamation mark is a stand-in for the actual number, so when the score is zero, the player will see “Score: 0/10”.

Status Attributes - Web Version

On the web version, we have no *Attributes* tab, so we have to do this in the start script, as we did with the weight attributes earlier. Go to the *Scripts* tab of the game object; you should see the script commands setting the weight attributes (only one is done in the image below). You need to add three new script commands like this:



The first line, highlighted in yellow, is obviously setting up the “score” attribute, just as before.

The two lines to set up the status attribute are marked in red.

Quest stores information about status attributes in a dictionary attribute called “statusattributes”, and the first line creates one for the player object (Quest does this automatically in the desktop editor). The other line adds one entry to that dictionary. It has two parts, the name of the attribute, “score”, and the display format, “Score: !/10”.

Okay, it is not as easy as with the desktop version, but it is doable!

Status Attributes

Whatever version you are using, you should be able to go in game, and find that a new panel has appeared on the right, with the score displayed!

You can use status attributes with any type of attribute (on the game or player), but it works best with numbers and strings.

Next: Using timers and turn scripts

Using timers and turn scripts

You can use timers to make something happens every so many seconds, whilst with a turn script you can make it happen every turn.

Timers

In a previous section, we made a bee fly into the kitchen after the player opened a window. We'll now make that bee a bit more annoying as it flies around the kitchen – every 20 seconds, it will buzz past the player.

To do this, we will use a timer. This timer will only be activated when the player opens the window in the kitchen. When the timer is activated, every 20 seconds it will print the message “The bee buzzes past you. Pesky bee”. This message will only be printed if the player is in the kitchen.

First, let's set up the timer. After we have done this, we will add the script command to activate it at the right time.

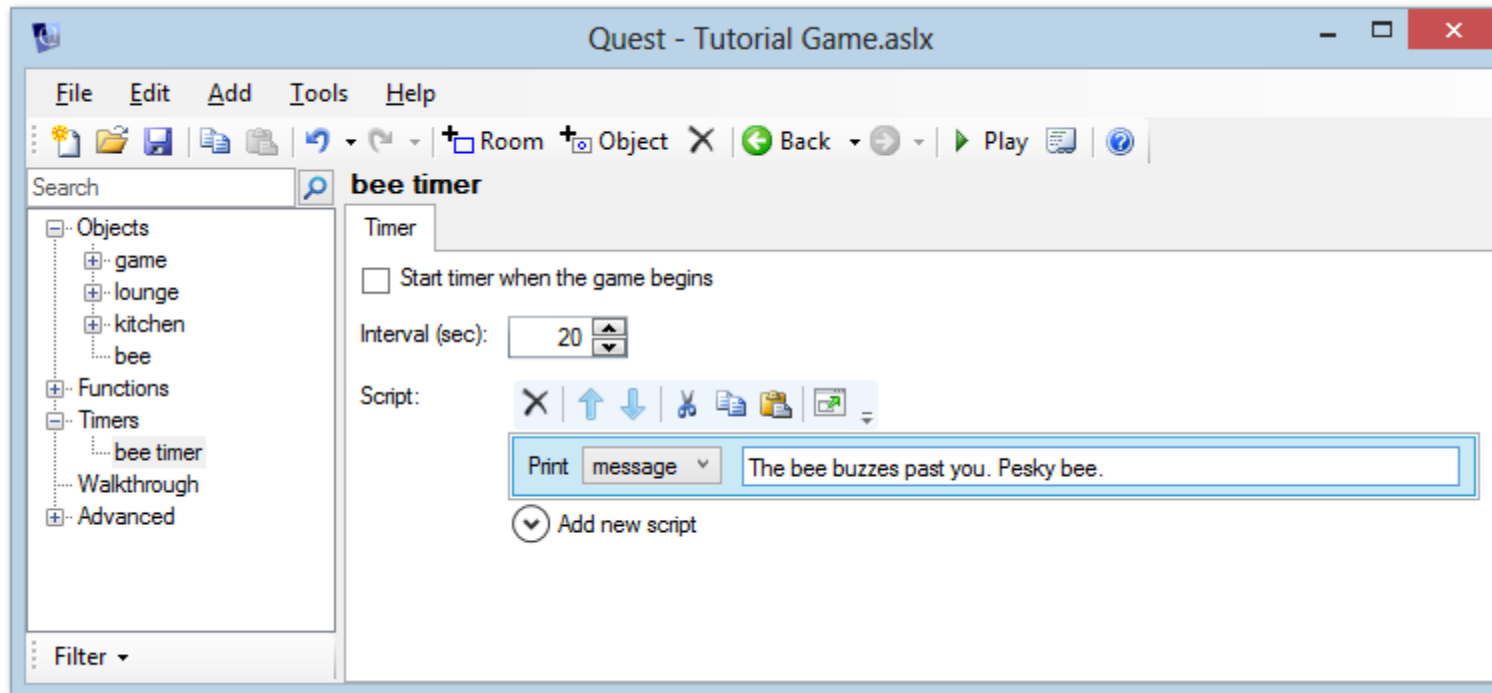
Setting up the Timer

In the Windows desktop version, add the timer by right-clicking the tree, or using the Add menu. In the web version, select “Timers” from the tree and click Add.

Enter the name “bee timer”.

The timer editor will now be displayed. The Interval specifies how often the timer fires – in this case, we want it to fire every 20 seconds, so enter “20”. Leave the box “Start timer when the game begins” unticked.

For the timer script, add a “print a message” command to display “The bee buzzes past you. Pesky bee.”



Activating the Timer

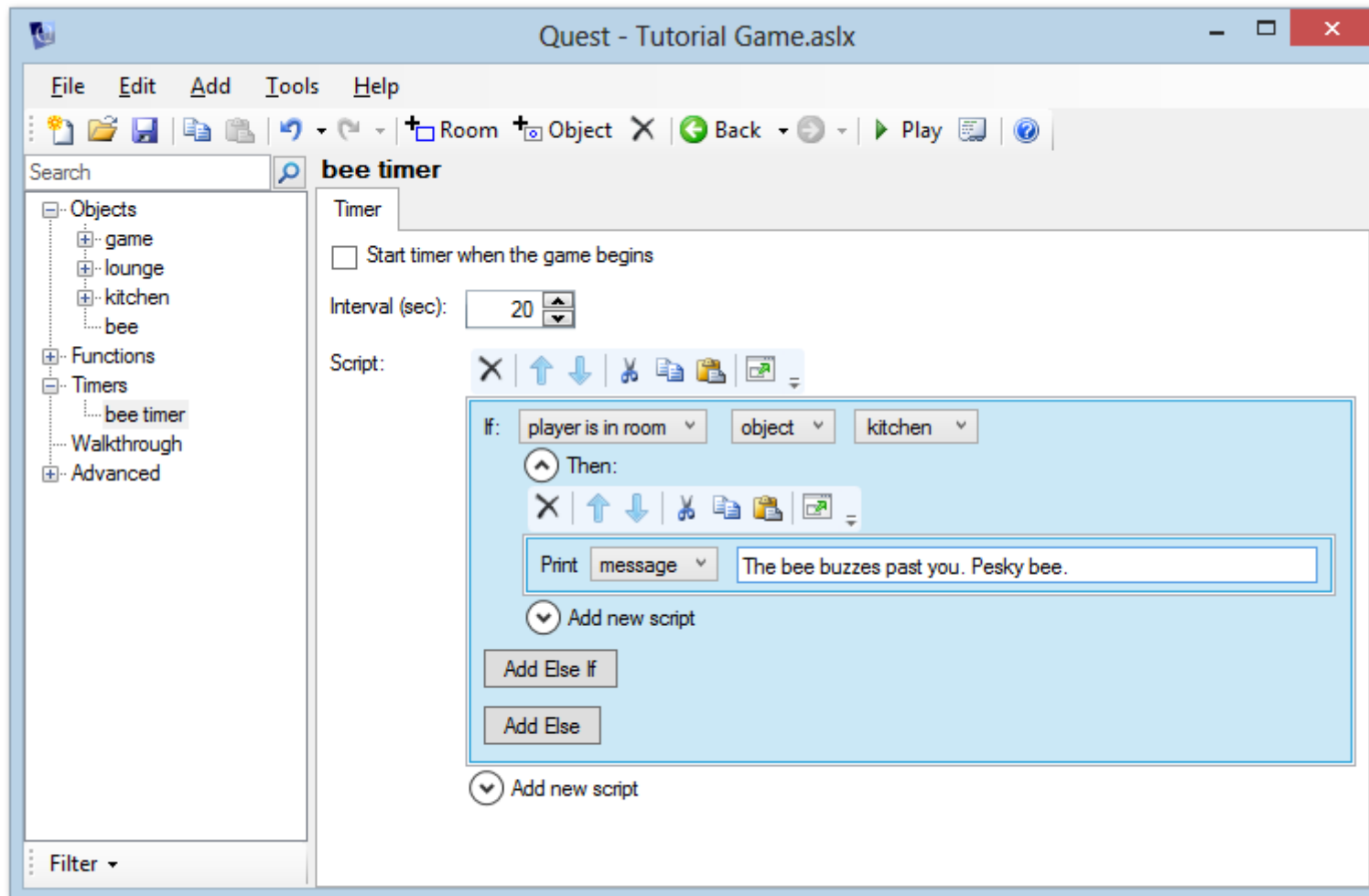
Go back to the “window” object and edit the script which runs when the bee enters the kitchen - this will be the “Else” script if you’ve followed the tutorial so far. Add a command after the “move object” command - from the Timers category, choose “Enable timer”. Select “bee timer” from the list.

Launch the game, go to the kitchen and open the window. Wait for a while and verify that the message is printed every 20 seconds.

Now go north to the lounge. You’ll see that we still get the message about the bee flying around. Woops! That bee is only in the kitchen. We’ll need to update the timer script so that it only prints the message if the player is in the kitchen.

[home](#)

You've already seen how to do this - an "if" command can check "player is in room". So add a check to the "bee timer" script - if the player is in the kitchen, print the message. If not, then do nothing.



Turn scripts

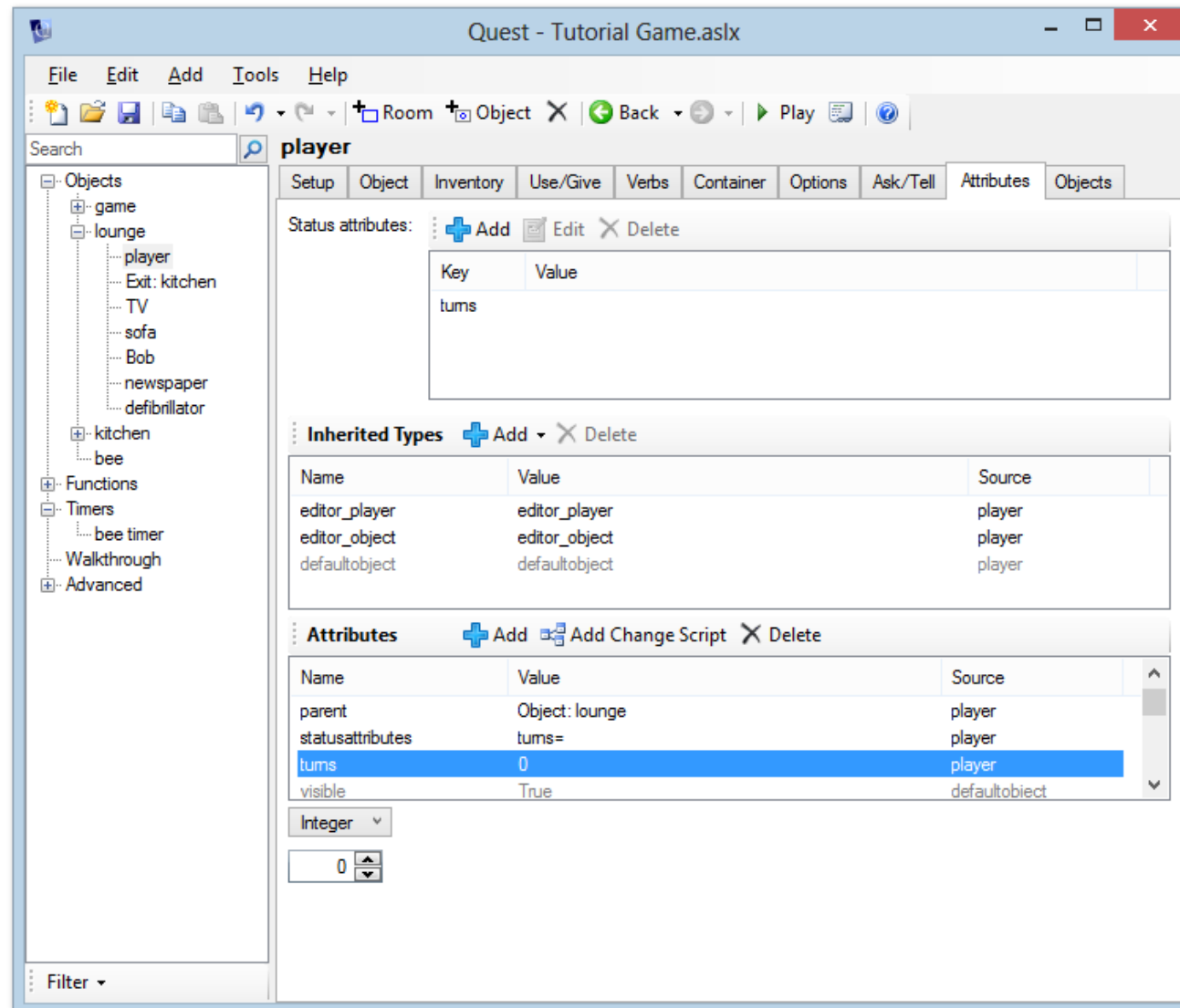
In this section, we'll look at running a script after each turn in the game - a **turn script**. We'll store the number of turns a player has taken in an attribute called "turns" on the player object.

Setting up the turn counter as a status attribute

We created a status attribute on the last page, this is just the same.

For the desktop version, to set up our "turns" attribute, select the "player" object and go to the Attributes tab. Click the Add button next to the Attributes list at the bottom of the screen, enter the name "turns" and make this an integer. To make this into a status attribute, we need to add it to the Status Attributes list at the top of the screen, so click Add there and add "turns" to the list.

[home](#)



[home](#)

For the web version, go to the *Scripts* tab of the “game” object, where we set the player score as a status attribute. Now we need to add a new attribute to the player called “turns”, and a new entry in the dictionary for that attribute. It’ll look like this (except it should be “turns” not “turn”!):

[home](#)

CopyPastePlay

game

ien

SetupFeaturesDisplayInterfaceScriptRoom DescriptionsPlayer

Start script:

☐

Set object

name

flour

Attribute:

weight

Value

500

☐

Set object

name

player

Attribute:

score

Value

0

☐

Set object

name

player

Attribute:

turn

Value

0

☐

Set object

name

player

Attribute:

statusattributes

Value

NewStringDictionary()

☐

Add to dictionary

player.statusattributes

Key

string

score

[EditorScriptsVariablesvalue]

string

Score: !/10

☐

Add to dictionary

player.statusattributes

Key

string

turn

[EditorScriptsVariablesvalue]

string

+ Add new script

Code View

Script when entering a room:

+ Add new script

Code View

Turn scripts - run after every turn the player takes in this game:

Id

Edit

Delete

Move up

Move down

If you launch the game now, you should see the turns variable displayed on the right-hand side of the Quest window. We've not yet added the script to increase the value of this though, so it will always say "Turns: 0" no matter how many turns we take. Let's add this script now.

Increasing the turn counter after each turn

A turn script can apply to a specific room, or it can apply to the entire game. To make a turn script apply for just one room, you simply need to create it in that room. If you create a turn script outside of a room, it will apply to the entire game. So, right-click the tree menu or use the Add button to create a turn script.

On the desktop version, you can drag it to the "Objects" label at the top of the tree to move it outside of all rooms. On the web version, click the move button towards the top right.

You can optionally specify a name for your turn script. You can use this if you want to be able to switch your turn script on and off using script commands, in a similar way to how we switched a timer on and off in the previous section. You can leave the name blank for this turn script, as this will always be running.

Make sure the "Enabled when the game begins" box is ticked.

We're going to add a script command which will increase the value of the player's "turns" attribute by 1 each time it is called.

To do this, add the "Set a variable or attribute" command.

In the left box, type:

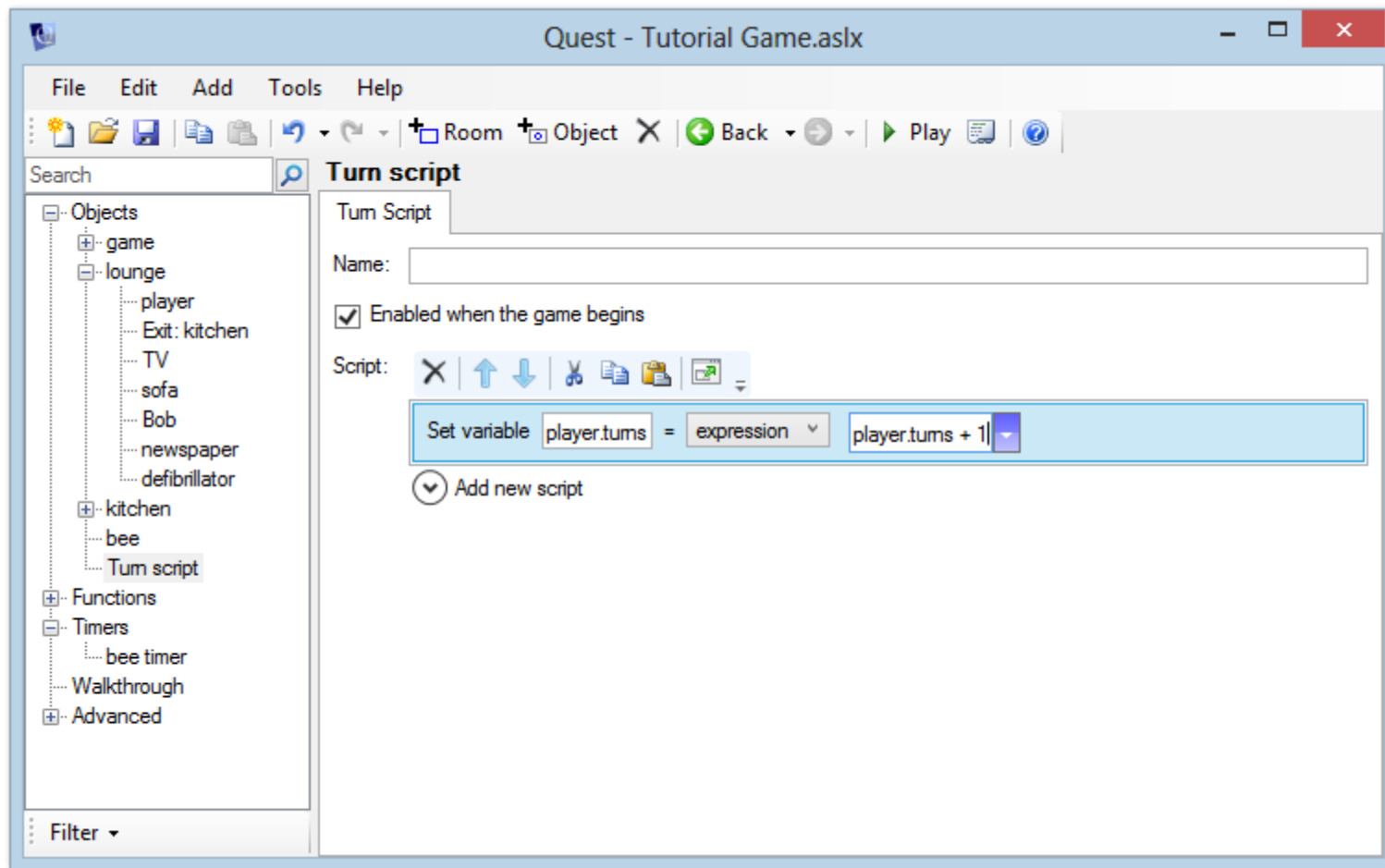
```
player.turns
```

Then in the box on the right, type

```
player.turns + 1
```

This will add 1 each time the script is called.

[home](#)



[home](#)

Launch the game now and verify that whenever you type a command, the “Turns” value is automatically updated.

Congratulations, you now know the basics of using Quest. There is much more to it, but you are probably best learning that as you need it. Now go make that great game!

2. Creating a gamebook

NOTE: *Rather than using the Game Book feature of Quest, we would suggest you use either the full product (and turn off the game panes and command bar so the player just uses hyperlinks), or use [Squiffy](#). Quest Text Adventures have a full world model, where objects and rooms relate to each other in a meaningful way, and have numerous features not supported by Game Books. Squiffy has no world model, but is great for creating multiple choice games that focus on text and story, and will produce a game that can be run in any browser, without the need for a dedicated server. Quest Game Books represent the worst of both worlds.*

Creating a blank game

A gamebook creates a file that opens inside Quest, and is playable inside Quest.

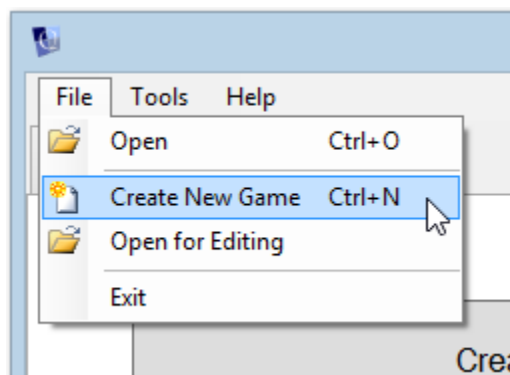
You cannot convert a gamebook into a webpage.

Most of the tutorial is applicable to both the Windows desktop version of Quest, and the web version. Any differences in the two versions will be mentioned as we go along.

This tutorial guides you through creating your first gamebook game. If you want to create a text adventure instead, see the main Quest tutorial.

Windows desktop version

To create a new game, open Quest and click the File menu, then Create New Game.



Alternatively, you can switch to the Create tab and click the “Create a new game” button.

You’ll see a screen like this:

[home](#)

Create New Game

Game type: ☐ Text adventure ☒ Gamebook

You can see an [object](#).
> look at object

You can see an object.
[Pick up the object](#)
[Leave the room](#)

Language:
English

Name:

File name:

Ensure that “Gamebook” is selected, and enter a name like “Tutorial Game”. Quest will create a folder and a game file for you. You can change where it puts the file by clicking the “Browse” button - it is recommended that you put your game file in its own folder.

Click OK and you’ll see the main Editor screen:

[home](#)

Quest - Gamebook Tutorial.aslx

File Edit Add Tools Help

Search

game

Page1
player
Page2
Page3

Setup Display

Game name: Gamebook Tutorial

Subtitle:

Author:

Version: 1.0

The Game ID below is used to uniquely identify your game. You should only generate a new ID if you have copied a game to create a new one.

Game ID: 8b2d2869-31f7-4df4-9d2a-699f39f3bda9 Generate...

Category: ▾

Year of release: 2013

Cover art: None ▾ Browse...

Description: **B** *I* U

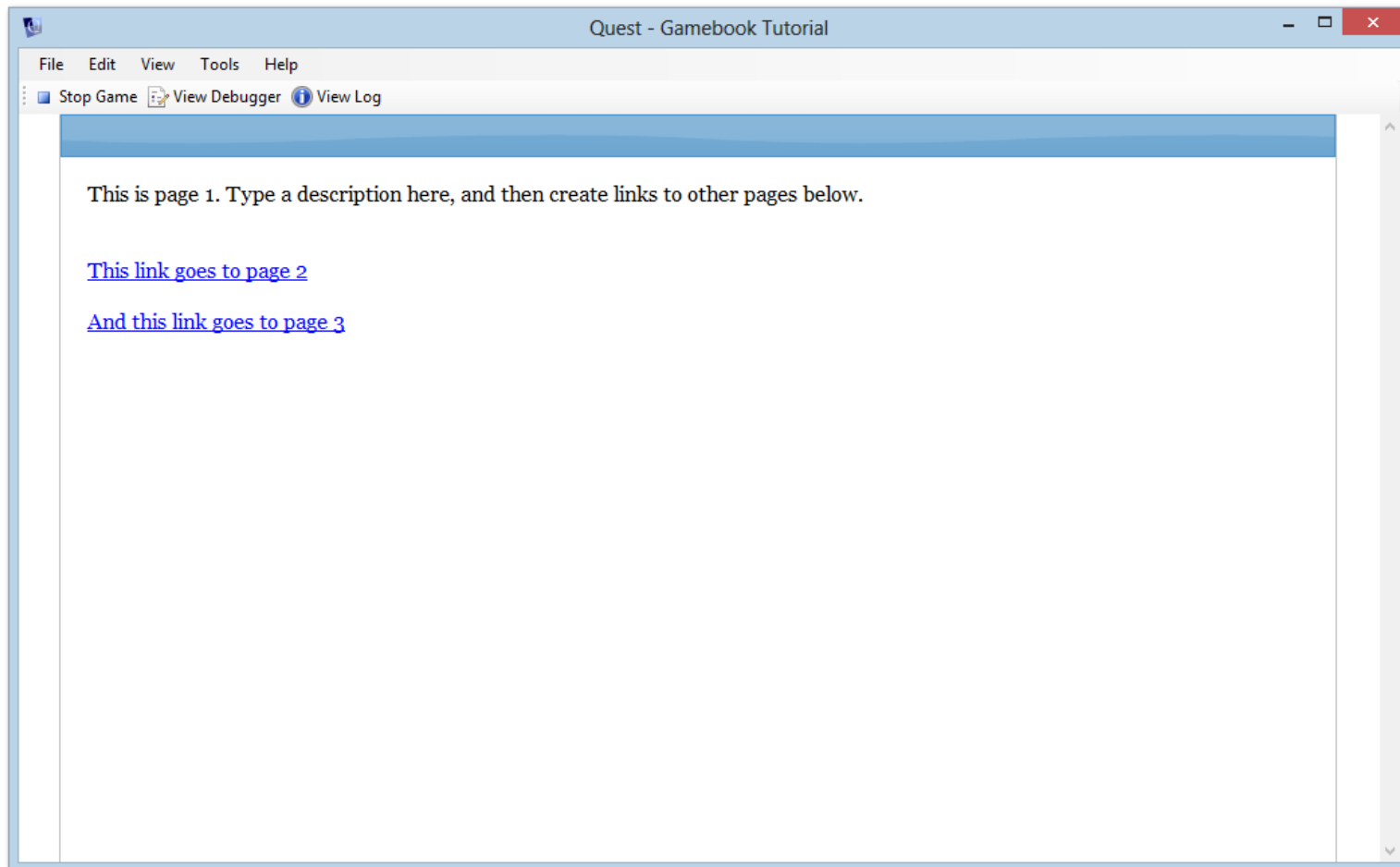
[home](#)

On the left is a tree showing you the pages in the gamebook, and a place to set options about the game itself. “Game” is currently selected, so that’s what we can see in the pane on the right.

Quest has created three example pages for us, and inside Page1 is the “player” object, which is where the game begins. You can test the game by clicking the “Play” button on the toolbar, or “Play Game” from the File menu. You can also press the F5 key.

As you’ll see, it’s a pretty empty game at the moment. We can navigate to pages 2 and 3, but that’s it.

[home](#)



You can go back to the Editor by clicking “Stop Game” in the top left of the screen. (You can hit the Escape key)

[home](#)

Web version

To create a new game, log in to Quest. You'll see the "New game" form.

Create a new game

Game name:

Game type:

 ▼

Create

Ensure that "Gamebook" is selected. Enter a name like "Tutorial Game" and click the "Create" button. Click the link which appears, and you'll see the main Editor screen.

[home](#)

The screenshot shows a web browser window with two tabs: 'Create a text adventure game' and 'Quest - Gamebook Tutorial'. The address bar shows the URL 'make.textadventures.co.uk/Edit/Game/5622'. The browser's toolbar includes navigation buttons (back, forward, refresh, home), a star icon for bookmarks, and icons for extensions. Below the browser window, the application interface has a top toolbar with buttons for '+ Page', 'Undo', 'Redo', 'Copy', 'Paste', 'Play', 'Help', 'Settings', and 'Saved'. On the left is a sidebar with a tree view containing 'game', 'Page1', 'player', 'Page2', and 'Page3'. The main area is titled 'game' and has a 'Publish' button. It features two tabs: 'Setup' (active) and 'Display'. The 'Setup' tab contains the following fields:

- Game name:
- Subtitle:
- Author:
- Version:
- Category:
- Year of release:
- Cover art:
- Description:

At the bottom right of the interface is a button with a speech bubble icon and the text 'Submit a suggestion or bug'.

[home](#)

On the left is a tree showing you the pages in the gamebook, and a place to set options about the game itself. “Game” is currently selected, so that’s what we can see in the pane on the right.

Quest has created three example pages for us, and inside Page1 is the “player” object, which is where the game begins. You can test the game by clicking the “Play” button at the top of the screen. The game will open in a new browser tab or window.

As you’ll see, it’s a pretty empty game at the moment. We can navigate to pages 2 and 3, but that’s it.

Simple Mode

When starting out with Quest, you may find it easier to run in “Simple Mode”. This hides much of Quest’s more advanced functionality, but still gives you access to the core features.

You can toggle Simple Mode on or off at any time:

- on the Windows desktop version, you can toggle Simple Mode from the Tools menu.
- on the web version, click the Settings button at the top right of the screen.

Editing pages

To create your game, edit the text for Page1. Underneath the text, the “Options” list shows which pages a player can get to from here. You can add new pages directly from here, or create links to other pages which already exist.

Page types

Text

This is the standard page type. It simply shows a paragraph of text, followed by the list of options.

Picture

This is the same as the Text type, but you can also choose a picture to display at the top of the screen.

YouTube

This is the same as the Text type, but you can also choose a YouTube video to display at the top of the screen. You will need the YouTube id of the video - an easy way to get this for a YouTube video is to find the video you want and click Share. The id will be displayed at the end of a URL like <http://youtu.be/qDlakzXcnro> where “qDlakzXcnro” is the id you want.

External Link

This is a special page type which takes the player directly to another website. It doesn't display any text of its own.

Playing sounds

You can play a sound when a player reaches a page. Go to the Action tab and browse for a sound file.

Quest Compiler

Is the program used to convert a Quest Game into an HTML webpage supported by a js folder and images folder, complete with 8 CSS files. (When you open Quest Compiler, there is a message about files supported, just click OK.)

Converting:

1. In Quest, go to the Tools menu and select Publish. B. In the dialogue box, select the Save Path and click Save. This will create a .Quest file in a window that appears. (Note: Quest allows the *creator* to make a Walkthrough, which is just storing each command used during gameplay in a column. The webpage does not support a walkthrough due to the way the webpage loads data.)
2. Open Quest Compiler. Load the Path to .Quest file you just created. B. Select the Destination Path, and click Compile. The finished HTML file will open in your default browser.