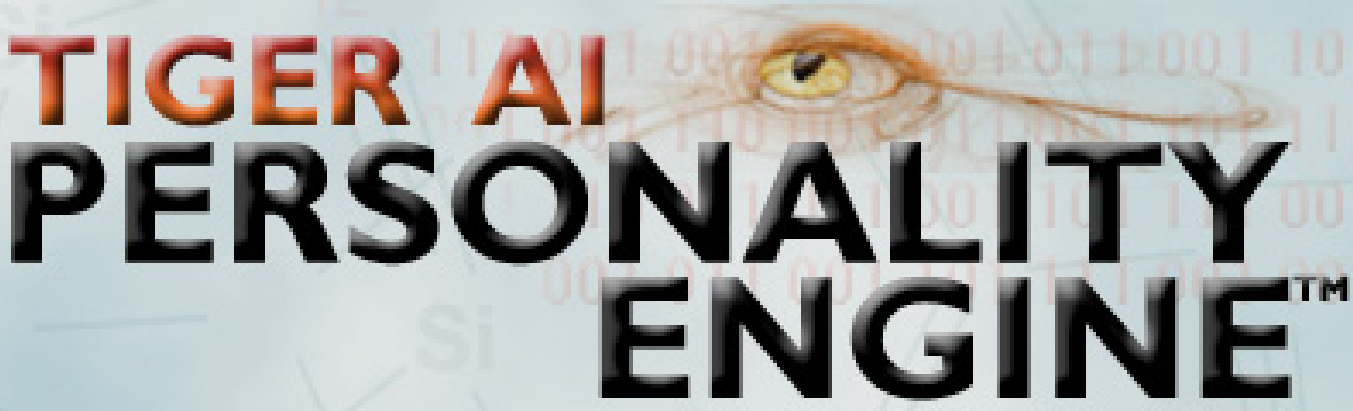

User Manual

Unity-specific version
rev 1.04, October 2012

Table of Contents

Introduction

Welcome to the Tiger AI Personality Engine (TaiPE), Unity edition! TaiPE provides a simple but complete add-on to create and change non-player personalities both before and during a game. With such power, your characters will react more realistically and will change their opinions of players depending on how they are treated. You can create different personalities for as many NPCs as you wish, and each will develop independently of all the others, with potentially different reactions depending on interactions with a player, with other players and NPCs, and even with world events.

Through a set of menus in the Unity editor, you can easily create and edit character personalities and assign them to NPCs. And through scripting, you can access these personalities in-game and have your NPCs react, grow, and change throughout the game's run, giving every play-through a different feel, giving your players that much more incentive to return to your game and play it again, wondering what would happen if they had treated those soldiers, or shopkeepers, or henchmen a little differently …

Also see the example Unity project using the Personality Engine (Unity_TaiPE_Example)—the README that came with your download has detailed information.

Please feel free to contact us at Quantum Tiger Games with questions, or even just to share your experiences using the Personality Engine, at *support@quantumtigergames.com*.

Thank you!
Jeff Georgeson
Quantum Tiger Games, LLC

1.0 Installation and Overview

1.1 System requirements

The Unity-specific version of the Tiger AI Personality Engine character setup works in Unity 3.x (there is also a Windows exe version available). The disk space, memory, etc. requirements are less than minimal.

The Personality Engine itself will work in any environment supporting C# dlls and SQLite databases.

1.2 Initial setup for Unity

When using the Unity-specific version of the Personality Engine, there are a few steps required to get up and running.

First, you will need to change the PlayerSettings (Edit->Project Settings->Player, or click Player Settings from the Build Settings dialogue box) setting for API Compatibility Level (at the bottom, under Optimization) to .NET 2.0 (NOT the default ".NET 2.0 Subset"). The subset lacks the functionality necessary to read some of the dlls used after compiling.

Second, place the following file in your Unity project's main folder:

dbPersonality.db This is the personality database; it is an encrypted SQLite 3 database. There's no need to import it as an asset.

Third, import the following as assets (go to Assets->Import New Asset, or copy the files into the correct folder):

TaiPE_Lib.dll This is the library containing functions dealing with the database and the AI. These are discussed in Section 2.2.2. Import this dll into the same folder as any scripts using it.

You also need some way for your game to recognize a SQLite database and its encryption. Import these dlls into every folder containing a script accessing the database (e.g., import copies into the same folder as the TaiPE_Lib dll, and into the Editor folder containing your TaiPE_Character_Setup dll [see below]). You will also need a copy in your 'Standard Assets' folder. We recommend:

System.Data.dll Some engines need this (e.g., Unity).

System.Data.SQLite.dll Phoenix Software's SQLite dll. Note that you must use the larger version for the TaiPE to work (it's the "Interop Library" version), as opposed to the smaller "Managed Only" version.

Note that the Mono.SQLite dll won't work, as it doesn't read the encrypted database properly.

In order for the TaiPE option to show up in the Unity menu bar, you need to create an "Editor" folder. You can do this by clicking Create in the Project window, then selecting Folder. Name the new folder Editor. (The Editor folder should be in the outermost set of Folders, as shown in Figure 1.) Then import the following asset into the Editor folder:

TaiPE_Character_Setup.dll This is the library creating the character setup menus that appear in Unity. Remember to place a copy of the System.Data dlls in the same folder.

You will also need a "Resources" folder, set up in the same way as the Editor folder. Into this folder import:

Contents of TaiPE_images.zip The images used by the editor menus.

After importing these, click on Unity's menu bar, and the TaiPE menu should appear, with four submenus: Character Editor, Choose Preset, Create Your Own Preset, and About Quantum Tiger Games. These will be discussed further in Section 2.1, *Setting Up and Editing Characters*.

One final note: After your build your game, you will need to manually copy the database (dbPersonality.db) and three of the dlls (TaiPE_Library.dll, System.Data.dll, and System.Data.SQLite.dll) into the same folder as your exe; for some reason Unity does not include these in its build. (Even though the dlls may be in the Managed folder of the _Data folder accompanying your exe, the database won't work properly unless you also copy the files and the database into your exe's directory.)

And that's it!

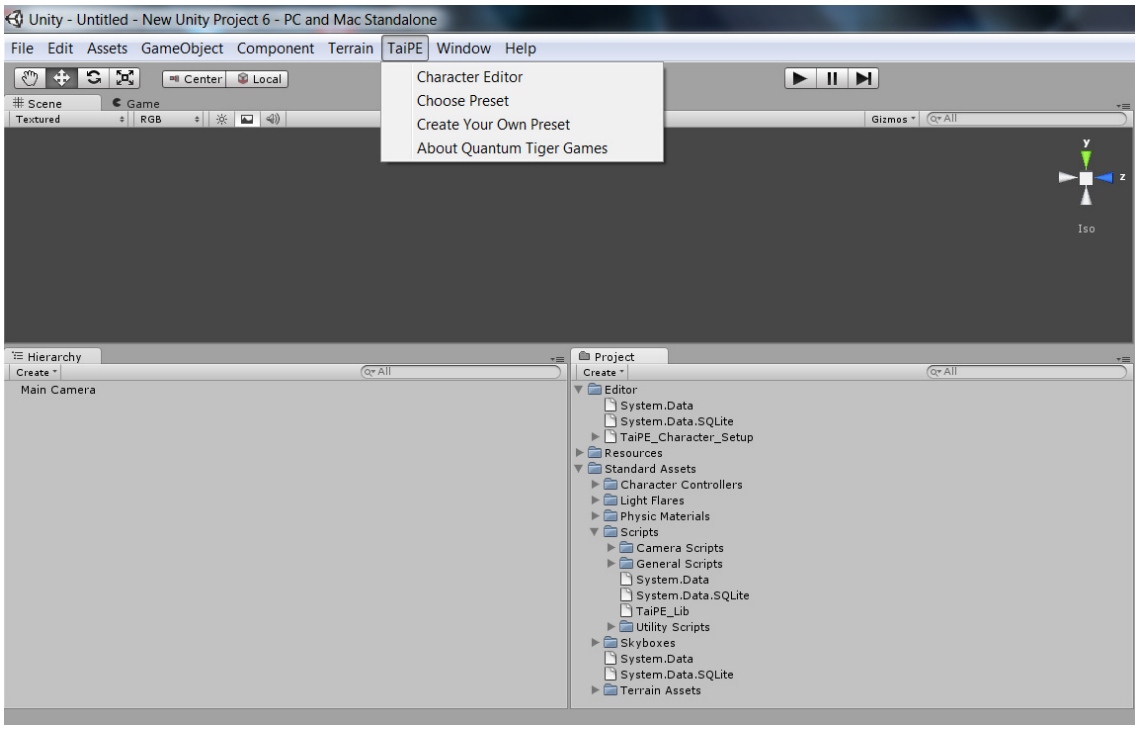

Figure 1 Unity editor screen, including TaiPE menu and Project window.

2.0 Using the Engine

2.1 Setting up and editing characters

The various submenus of the TaiPE menu allow you to create and edit the base personalities for your NPCs. This is all pregame stuff; the characters will begin your game in whatever configuration you choose here.

2.1.2 Background

You can create personalities for as few or as many NPCs as you wish; there is no requirement that every NPC in your game use the Personality Engine. We have endeavored to make the creation process as simple as possible, however, giving you the option of using preset base personalities or tweaking as many as 37 different aspects of your characters' personalities.

These aspects are the same stimuli/responses you will be able to choose from your game code. For instance, you can adjust the initial intensity of an NPC's "kindness"—that is, his likelihood of responding to kindness with kindness, or to perform a kind act—using the character editor. Note that you are not directly changing the underlying facets of the NPC's personality (there is no "kindness" facet, for instance), but are affecting the underlying facets that go into making us more prone to kindness. Because of this, when you change kindness, you affect the other stimuli/response types that have similar facets as components.

Also note that the numerical values you see are for your guidance only; the underlying personality facets are the core of the engine, and are ultimately used (along with our personality algorithms) to choose/change levels of response in the actual game situation.

2.1.3 What are these "stimulus/response types"?

Personality theory in general posits that there are certain base elements that go into creating a person's personality, and that these factor in various ways into the sorts of deci-

Openness	**Extraversion**	**Neuroticism**
Fantasy	Warmth	Anxiety
Aesthetics	Gregariousness	Angry Hostility
Feelings	Assertiveness	Depression
Actions	Activity	Self-Consciousness
Ideas	Excitement Seeking	Impulsiveness
Values	Positive Emotions	Vulnerability
Conscientiousness	**Agreeableness**	
Competence	Trust	
Order	Straightforwardness	
Dutifulness	Altruism	
Achievement Striving	Compliance	
Self-Discipline	Modesty	
Deliberation	Tender-Mindedness	

Table 1 The Five Categories and Their Facets (categories in bold)

sadness	deceitfulness	assertiveness
happiness	condescension	humour
anger	quarrelsome	intellectual
sternness	helpfulness	demanding
jealousy	selfishness	productive
anxiety	affectionate	ambitious
impulsive	dependability	orderly
stubborn	efficiency	kind
guilt	moodiness	annoying
standoffish	wittiness	intimidating
reluctance	excitability	gregarious
conformity	imaginative	
distrustful	talkative	

Table 2 Possible Stimuli/Response Types (to be queried by the developer)

sions and actions we make in our lives. In the Five Factor Model, a well-respected theory (see, for instance, Goldberg, 1993; Costa & McCrae, 1995; John et al, 2010; McCrae & Costa, 2010; DeYoung, 2010*), these elements are the 30 personality facets listed in Table 1, which are grouped into five overall categories (hence the 'Five' in the name of the model).

Rather than having the developer try to check the individual personality facets and figure out how these might apply to various everyday stimuli (such as another person being nice to you) or response types (like whether an NPC will go up to a stranger and be nice without provocation), the Personality Engine provides 37 possible stimuli/responses (see Table 2) that have already been keyed into the various facets (using our own extensive research and applying this to game-type situations). Thus the developer merely has to query whether a stimulus will elicit a strong reaction, and the Engine will take care of not only finding that answer but figuring out how much (if at all) having that interaction affects the personality of the NPC in future interactions, either with the same player or with other players (or NPCs or, even, world events). Thus, just as in real life, the very act of interacting with others has an effect on how a character deals with the future, and on how similar stimuli are dealt with.

Ultimately, of course, the extent to which you consult the Personality Engine is up to you; if you want a certain NPC to follow a very prescribed set of actions, that's fine; it does not interfere with the Engine's functioning to do this (although it won't be able to develop that NPC's personality, obviously). If an NPC has to give out a piece of important information, the Engine will not forbid it (how could it?); what you might do, however, is have the NPC's phrasing be different depending on how she feels about this player. Even subtle differences like this can make the gameplay even more interesting than it already is, can add a bit of mystery and intrigue and depth (especially depth) to the world and the characters and people in it, and ultimately lead to a more satisfying experience for the (presumably human) players playing your game.

* The full references can be found at the back of this manual, for those interested.

Also as in real life, having high tendencies toward a certain reaction does not mean the character will react in the same exact way each and every time; our algorithms take this into account and return a reaction intensity from which you can make your own judgment as to what the NPC says or does.

2.1.4 TaiPE menu

See again Figure 1. Pretty self-explanatory. From here you choose:

Character Editor to create or edit an existing character using the full set of possible stimuli/response types—gives you access to the 37 different stimuli/response types. Use this to fine-tune a character.

Choose Preset shows you a list of preset personality types—allowing you to quickly make a "type" of character without having to adjust anything. Note that you can, however, later edit these characters using the full set of stimuli/response types if you wish.

Create Your Own Preset—find yourself making a lot of similar characters, but none of our presets quite fits? This lets you create up to four of your own presets.

About Quantum Tiger Games—gives us the opportunity to talk about ourselves.

2.1.5 Creating/editing characters (the TaiPE submenus)

There are two ways to create a character: fine-tuning a character in detail, or using a preset to make a quick general type. For example, you could tailor a specific NPC to be more kind, intellectual, and imaginative (which will make that character more prone to wittiness and slightly less likely to anger as well, due to the underlying facets). You could also try the Professor preset, which will give you a basically intellectual character, but not one who is necessarily kind or imaginative. Characters created from a preset can be fine-tuned later using the character editor.

Following are detailed instructions for creating characters.

> **Who is that character to the side?** To the right of the various TaiPE screens is Raina (super-deformed version), QTG's spokescharacter (she hopes to be like Sonic or Mario one day—hey, aim high, right?). She offers advice about using the Personality Engine, for those times when you don't want to drag out the user manual.

Character Editor: Create a character in detail

If you click the "Character Editor" menu, a window opens containing a text field for name entry and a set of sliders (see Figure 2, top). Typing in a name will provide a sort-of drop-down list of existing characters; you can double-click on one of these to edit him or her, or you can continue typing in the name of a new character (the drop-down also helps prevent creating two characters with the same name).

Once you have typed/double-clicked on a name, press the Create/Edit Character button. This will activate the sliders, each set to a value between 0 (the character has an extreme-

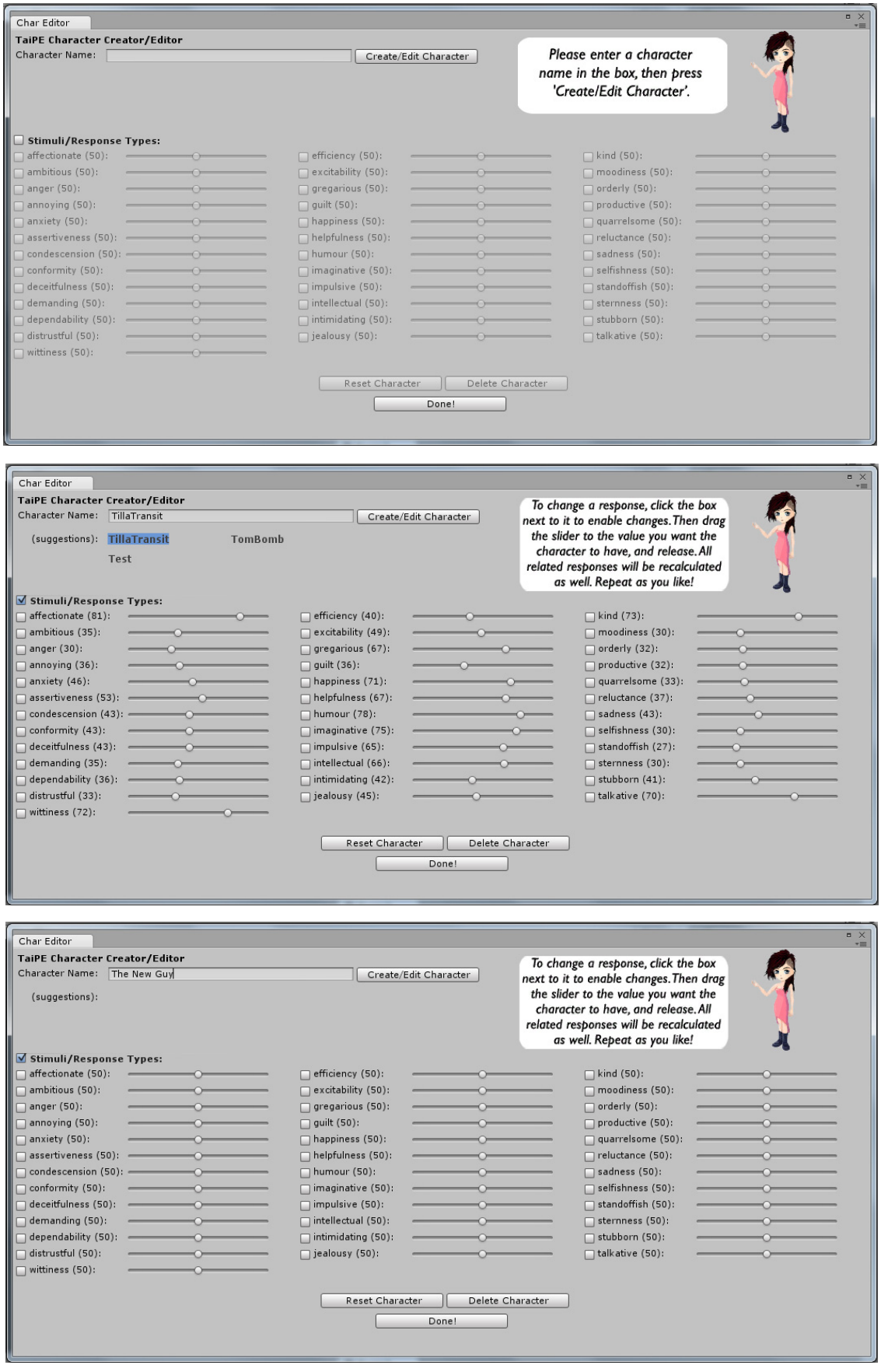

Figure 2 Create a detailed character. (top) Enter a character name, then (middle, character exists, or bottom, new character) edit her or his details.

ly low propensity for this) and 100 (the character reacts in this way very intensely) (see Figure 2, middle or bottom). Zero kindness would indicate a very unkind person; 100 would be someone who was almost saintly. If the character already exists, its current values from her/his facets will be calculated (Figure 2, middle); a new character will start out with all 50s (Figure 2, bottom).

> **NOTE:** These settings create a base personality, applied initially to every interaction the character has with anyone and anything in the game world. As the character interacts with other individuals, her reactions toward those individuals will vary depending on what those interactions have been; in other words, she may grow to despise a player who has been continually nasty to her, yet love someone who has been nothing but kind and giving. Additionally, an NPC's base responses (to strangers, say) will be affected by all of her interactions with others and the world; an NPC who has been treated badly by almost everyone will grow to be cynical and distrustful, even if he started out a kind and genial soul.

To adjust the values, click the toggle box next to a response type, then move the slider up or down. When you release the slider, the character's facets will be refigured to match the new response value, and then all his/her responses will be recalculated to match the new facets. For example, changing The New Guy's Affectionate score from 50 to 70 also increases his Kindness (to 57) but decreases his Anger (tendency to become angry) somewhat (to 46), along with changing several other values slightly. This is because the psychological factors underlying the Kindness response (the facets) were changed when you adjusted Kindness, and thus every other response based on any of the same factors changed as well (see Figure 3).

When you are finished adjusting the values, click the Done button. This closes the editor window. You can also click Done without ever entering a character name; this closes the window without creating or editing anything.

The character's values are changed in the database live as you change them, and a new

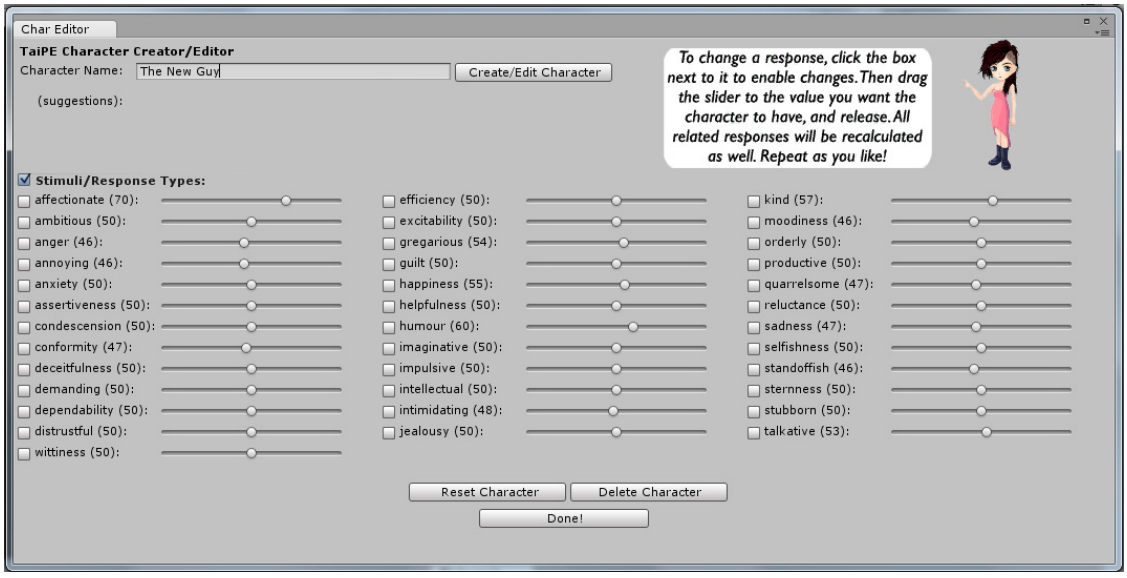

Figure 3 The New Guy with adjusted stimuli/response values

character with default “average” values is created the moment you click Create/Edit Character; you do not have to click Done to create a character or save your changes.

If you’re done with a character and wish to edit another, simply type another name in the Character Name box and click Create/Edit Character again. You will then be given a set of sliders to adjust for that character.

And what do the other buttons on this screen do?

Reset Character—The Reset button resets a character’s slider values to those he/she had when imported (or created, if new). New characters will revert to all average values.

Delete Character—Deletes the current character. Note that this is not reversible!

Choose Preset: Create a character using presets

Clicking TaiPE -> Choose Preset allows you to create a character quickly, without having to adjust slider values. There are 28 different presets, some of which are subtypes of the same general personality.

First, type a name for the character in the Character Name box (see Figure 4); this is the same as in the Create/Edit Character window: Typing in a name will provide a sort-of drop-down list of existing characters; you can double-click on one of these to edit him or her, or you can continue typing in the name of a new character. Click “Begin Preset Character” when ready.

After this the Preset Character options will become available (see Figure 5). Click the

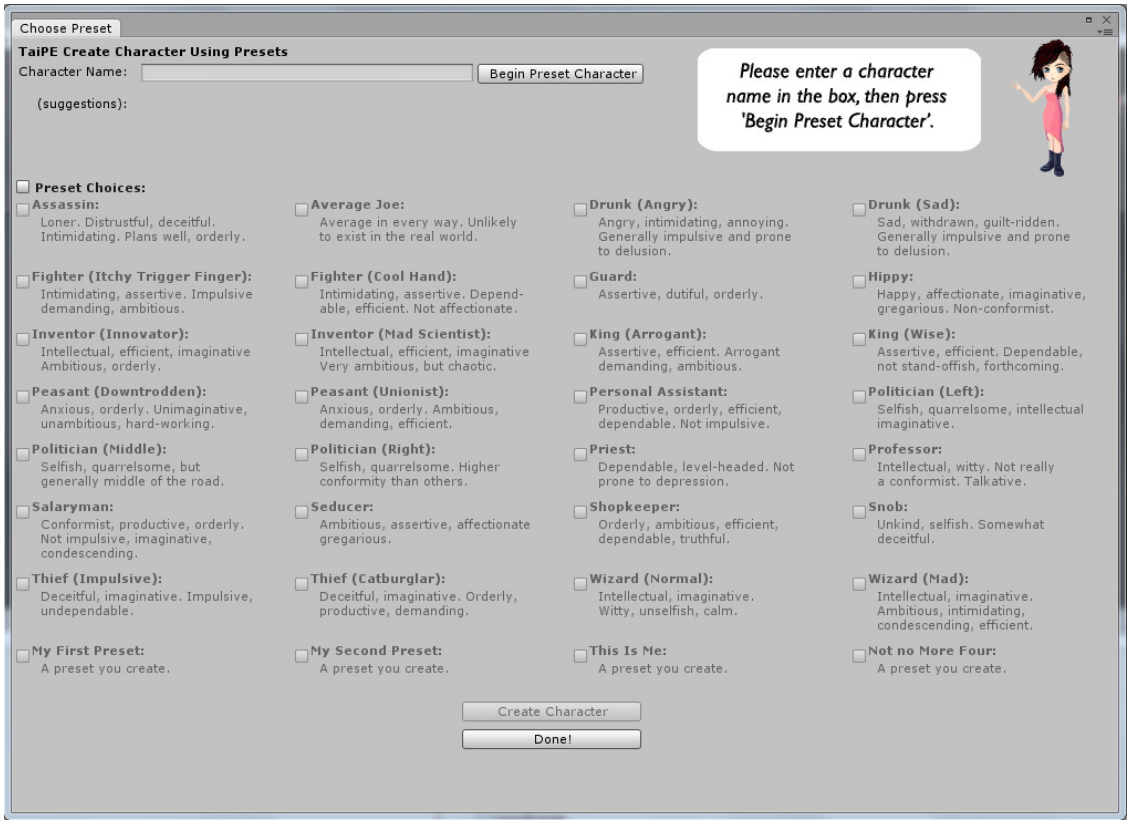

Figure 4 Create a character using presets.

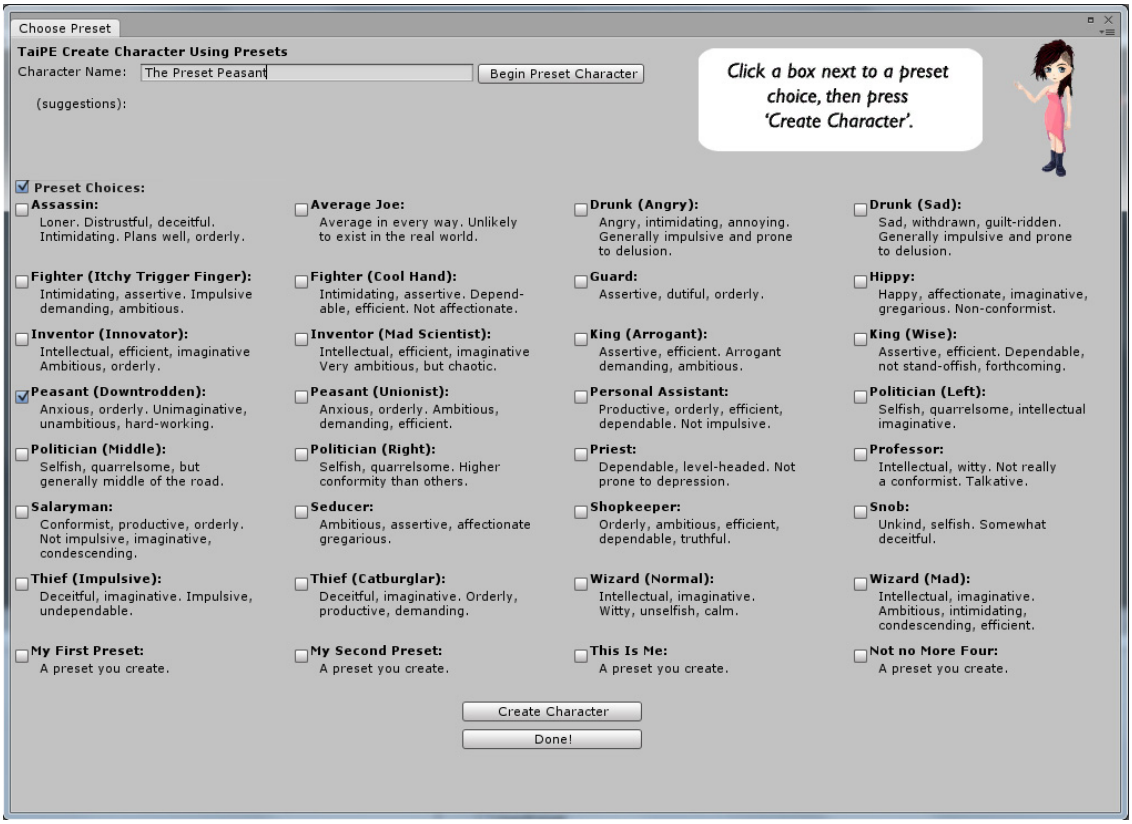

Figure 5 The Preset Choices

check box next to the desired preset, then click “Create Character”. This creates the character.

At this point you can type in another name to create another preset character, or you can press “Done” to close the window. Pressing “Done” before pressing “Create Character” closes the window and does NOT create the preset character.

The last row of preset choices is all presets you can create and edit yourself, as covered next.

Create Your Own Preset: Your personal army of preset personalities

If none of the existing presets works for you, you can create up to four of your own. You do this by selecting Create Your Own Preset from the TaiPE menu.

In the window that comes up, you will be given four options from which to choose (see Figure 6); these are the current user-created presets (or the defaults if you haven’t created any). Double-click on one of these to select it for editing.

Once selected, the name of the preset will appear in the “New Name (if desired)” box and the sliders will become available. To rename the preset, type a new name in the box and click “Change Preset Name”. To change the preset’s stimulus/response values, click the toggle box next to a response type, then move the slider left or right. When you release the slider, the preset’s facets will be refigured to match the new response value,

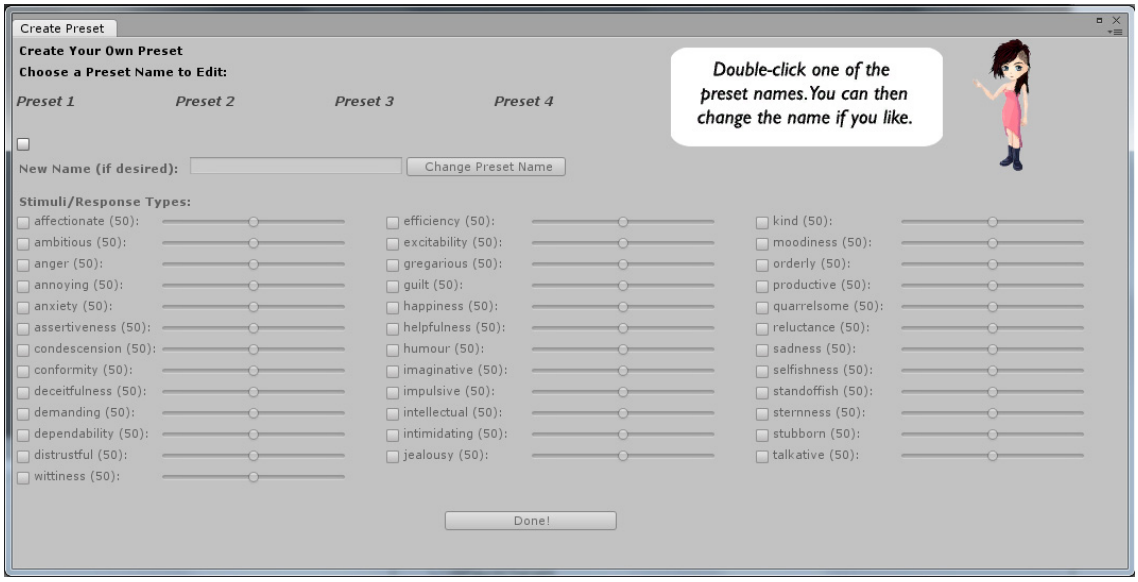

Figure 6 Double-click a preset name to edit

and then all its responses will be recalculated to match the new facets (just as when using the Character Create/Edit screen). The preset's values are changed in the database live as you change them. (Note, however, that the preset's name is changed only if you click the "Change Preset Name" button.) See Figure 7.

When you're done editing the sliders, press Done! to exit the screen, or double-click another preset to edit.

You can now use your new preset to create NPCs!

About Quantum Tiger Games

Self-explanatory, really. Clicking this on the TaiPE menu gives you more info about the Personality Engine and Quantum Tiger Games (and shows Raina in her non-SD glory).

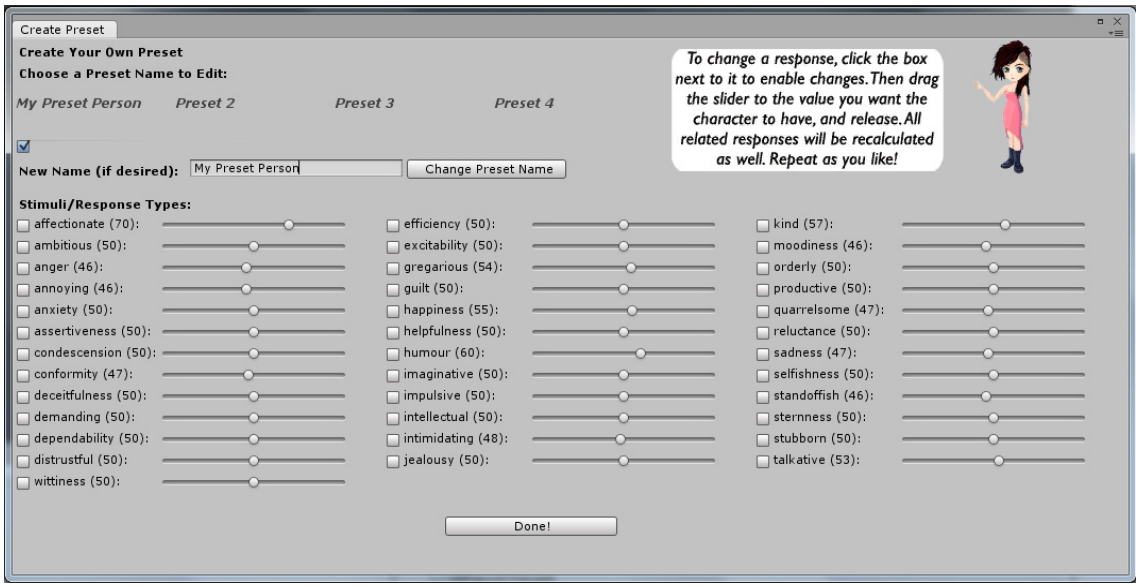

Figure 7 Change the name and adjust the sliders

2.2 Using the character personalities in a game

Now that you have the base personalities for your NPCs, it's time to make them game-accessible. Make sure the TaiPE_Lib dll has ben added to your Assets, as per the installation instructions (see Section 1.2).

2.2.1 For each NPC …

Each NPC should use its own instance of the AIClientInterface module. It also must know how to use the TaiPE_Lib.dll. For example, in C# you would:

1) place a "using TaiPE_Lib;" statement before the class definition in your code
2) declare a variable of type AIClientInterface (e.g., "AIClientInterface myAICI;")
3) instantiate a copy of the module (e.g., "myAICI = new AIClientInterface();")

Please see the example Unity code in part 3.

Note that all NPCs use the same database, so there are not multiple copies of this. In fact, you don't need to create any connections to the database at all in your code (this is done by the AI module).

2.2.2 Naming the NPC

Also, you should set a variable equal to the NPC's name (matching a character whose personality you created when creating characters using the TaiPE menus). The NPC's name will be needed every time you query the database on his/her behalf. Note that this connection is all you need to associate personalities with NPCs.

For example, if you've created a personality for Edgar the Excellent in the database, then in scripts accessing Edgar's personality you'll need a string variable set to "Edgar the Excellent".

2.2.3 What you can do

The TaiPE_lib dll contains the following user-accessible functions:

Init() [returns nothing]
ChangeBaseChange(float amount) [returns nothing]
ChangeBkgrndChange(float amount) [returns nothing]
SetPlayerName(string playerName) [returns nothing]
AIReturnResult(string characterName, string playerName, string stimulus, bool toChange = true) [returns int]
AINoResult(string characterName, string playerName, string stimulus, bool posChange = true) [returns nothing]
AIPlayerHistory(int lastNum, string characterName, string playerName, float percMatch, params string[] responses) [returns bool]
AIPlayerHistoryNums(int lastNum, string characterName, string playerName, params string[] responses) [returns int]

These are described in detail below (and again, see the examples in Part 3, and the Unity scene example that came with the Engine).

Init() [returns nothing]
This function initialises the AI module, and must be called before you try to use it. It doesn't return anything.

ChangeBaseChange(float amount) [returns nothing]
This function changes the base rate at which the NPC's personality changes in relation to an individual player (or event type). The default is 5.0. This is a bit more quickly than people's personalities change in real life, but to notice the changes in a game, wherein the player may only speak to an NPC a few times, this is the value chosen after testing various rates. Remember that changing this rate for one NPC does not change it for any other (so you can have characters whose personalities differ in terms of rate of change, just like humans).

ChangeBkgrndChange(float amount) [returns nothing]
This function changes the base rate at which an NPC's personality changes in relation to EVERYTHING; i.e., if one player is particularly nice to the NPC, this rate determines the amount the NPC is more kindly disposed to everyone in the world as a result. The default for this rate is 0.1, which again is higher than the rate for humans in the real world, but is desirable for noticing the changes in-game. Changing this rate by even a fraction has a large effect; raising it as high as 1 would create a wildly changing personality.

Again, changing this rate for one NPC does not change it for any other.

SetPlayerName(string playerName) [returns nothing]
In order to recalculate the NPC's personality, the module must know to which player the interaction refers; therefore, every player must be recognized by the database before any interactions occur. This is done by simply passing the player's name to the module through the SetPlayerName function. This is done once, before any interactions with this NPC can occur (ideally when the player chooses a name at the beginning of the game, or when the game initializes (if players don't choose names). Note that, because every instance of the AIClientInterface uses the same database, you only need to set the player's name once.

AIReturnResult(string characterName, string playerName, string stimulus, bool toChange = true) [returns int]
This is one of the primary functions for the Personlaity Engine; it takes a stimulus/response type (e.g., kind) and returns an integer from 0 to 4 indicating the intensity of the response:

0	Extremely low; could indicate a reaction opposite to that queried, or just an absolutely neutral response (e.g., a kindness intensity of 0 could mean the NPC just ignores the player, or perhaps reacts negatively—the actual reaction is that which suits the situation. Or, if another check indicates the NPC has a high anger intensity, perhaps the NPC reacts very negatively to the player's kindness … it's up to you how many checks you want to make for any given interaction)
1	Low; slightly negative response, or relatively neutral
2	Average; reacts as the "average" person would in this situation
3	High; reacts positively (note that having a high, positive reaction for one stimulus (e.g., kindness) may be very different from a high, positive response to another (e.g., anger)
4	Extremely high; intensely positive response

Why only five possible intensities? Because we had to choose between further shades of intensity and speedier response times from the engine. Five seemed the best compromise.

AIReturnResult can be used in different ways. The default takes the character name, player name, and stimulus/response type, and changes the personality of the NPC in regard to both this player and to everything else in the world (see also the discussion of the base change functions, above). Alternatively, you can call the function with an added boolean indicating that no changes are to be made to the personality (done by setting the toChange bool to false). You may want to do this when you are simply checking on an NPC's reaction intensity but aren't in a situation where the personality would be changed, such as checking whether an NPC would be the first to start a conversation (checking gregariousness); no real interaction with a player has yet occurred, so there isn't a reason to change the NPC's reactions. There also may be times when you don't know at the outset in what way the NPC's personality would be changed by an event; say you check her impulsiveness to see whether she'll follow the player off the edge of a cliff. It would depend on the outcome of this action whether she later thought of it as a negative or positive reinforcer of her impusiveness. Which brings us to …

AINoResult(string characterName, string playerName, string stimulus, bool posChange = true) [returns nothing]
So your NPC has jumped over a cliff and had an absolutely smashing time … in the best sense of the word. Her impulsiveness should be increased, but you don't need any sort of value returned for a new response—the response and event already happened.

That's when you use AINoResult. This function changes the NPC's personality without returning a result. The default, taking the character name, player name, and stimulus, changes the values in a positive way, indicating a step towards more intenisty the next time the stimulus is checked.

What if our NPC had had a smashing time in the worst possible sense of the word? You could then change her impulsiveness to be less likely to rule her thoughts by calling AINoResult with an additional boolean value (posChange) set to false.

AIPlayerHistory(int lastNum, string characterName, string playerName, float percMatch, params string[] responses) [returns bool]
Over time your NPCs will develop a history with a player, and while this is approximated by the player-specific personality changes for each NPC, sometimes it makes more sense to know more specifically whether a certain interaction has occurred several times. Does this player always cause the NPC to check how angry he is, but now is being playful? Has the player been acting like a jerk for the last several encounters, but this time is trying to be kind? Without knowing better, the NPC will just check his kindness response and may be far nicer than the situation warrants.

You can, however, check any number of previous interactions and adjust any responses accordingly by using the AIPlayerHistory function. By providing the number of previous interactions you wish to check, the character name and player name, the least percentage of these that should match to return true, and the type(s) of stimuli you are searching for, you can find out if, say, 75% of the last five interactions involved anger; or if 50% involved either anger or intimidation. You can check for as many different responses as you wish at the same time (although it will be checking whether ANY of the listed responses match).

Of course, there will be times when there haven't been as many interactions as you're checking for (say, if the player has only spoken to this NPC twice, and you're checking the previous 5 interactions). The function then checks whether the percentage of the interactions that have occurred is equal to or greater than the percentage you're looking for.

AIPlayerHistoryNums(int lastNum, string characterName, string playerName, params string[] responses) [returns int]
If you find it important to know whether a certain number of interactions of a certain type have taken place between a player and NPC, you can use this function to find out. It looks for up to the lastNum interactions and finds the number of these that match the character, player, and stimulus (or stimuli) requested. Thus, even if the previous function tells you that 100% of the previous 5 interactions have involved annoyance, you may want to hold back on throwing the player out of the shop if there has really been only one interaction, not five.

See Part 3 for examples of these functions in use, and take a look at the sample Unity scene for a demonstration. You can also visit ***quantumtigergames.com*** for another example (the voters demo).

3.0 Examples

Of course, just the cut-and-dry enumeration of function calls doesn't really demonstrate the possibilities, the range of character responses and development when using the Personality Engine. Following are a few examples of using the engine in a Unity environment. (Note that the examples assume you are using C#.) See also the Unity sample demo included with the TaiPE.

3.1 Remember to set the NPC and player names!

As mentioned before, it's a good idea to set a string variable to the name of the NPC whose personality you are accessing, as this name will be necessary every time you query the database. However, you can change an NPC's personality from any script, anywhere, as long as you use his/her name in the function call.

Also, the Engine must know which player (or thing) is interacting with the NPC in order to return values that make sense. Setting the player name can be done when the player registers his or her name with the game or at any other point before actually calling a function that requires it; the example in Figure 3.1 does it in a script that runs when the First Person Controller is started. Note that the player name is set in the database for access by all instances of the AIClientInterface, so you only need to set it once (unlike the Speed of Change functions listed in Section 3.3, which change values only for that instance and thus that NPC).

Note in the example that you can also set up non-player "players" with which to interact. Here we've created a way to have the NPC react to world events by creating a "world_events" player. You would do something similar for interactions with other NPCs (making them "players" and thus setting their names as well).

3.2 Getting it all started: using Init

To use the engine, you have to create an instance of it for a character and then initialize it. This is easy to do: in a script you attach to a character, do the following (and see the boldface items in Figure 3.2):

```
//set the player name here (in a script attached to the First Person Controller)
//can also be attached to anything that is accessed BEFORE any interactions with NPCs,
//such as the terrain, even

string playerName = "JJ Player";
myAICI.SetPlayerName(playerName);

//also, because we've decided world events (not having to do with a specific
//player) can affect an NPC personality, we'll set the world as a "player"
//(normally this would be done outside of an individual FPC,
//but for demonstration purposes it's here)

string worldEvents = "world_events";
myAICI.SetPlayerName(worldEvents);
```

Figure 3.1 Setting player names

```
using UnityEngine;
using System.Collections;
using TaiPE_Lib;

public class NPCClientSide : MonoBehaviour {

	/*
		Author: Jeffrey Georgeson, build 13 April 2012
		Copyright Quantum Tiger Games, LLC 2012
	*/

	//who is the NPC we're dealing with in this script?
	string NPCName = "Edgar the Excellent";

	//instantiates copy of AI Client Interface
	AIClientInterface myAICI;
	//alternatively, could use AIClientInterface myAICI = new AIClientInterface();
	//which would take care of instantiation all at once

	...

// Use this for initialization
	void Start () {

		//continue instantiation of AI interface
		myAICI = new AIClientInterface(); //unless already done through alternate
			//code above

		myAICI.Init();

		...
```

Figure 3.2 Initializing a copy of the AIClientInterface for a character

1) make sure to include a "using TaiPE_Lib;" statement above the class declaration
2) when declaring variables for use in the class, instantiate a copy of AIClientInterface
3) in the Start function provided by Unity (or some other function called when the script is first called), continue the instantiation (C# requires not only declaring it in the variables, but then setting it equal to a "new AIClientInterface()")
4) also in the Start function, call the Init() function of the AIClientInterface to initialize the Engine for this character

That's it! Now you're ready to access the character's personality, which you created using the character setup menus described previously (see description in Section 2.1).

3.3 Changing speed of change

If you want to change the speed at which this character's personality changes (not recommended until after you've used the PE for a while to get a sense of how it works), now would be the time to do it—just after initializing the Engine, still within the Start function. (Not that this is required. The speed at which different facets change in real people's lives is already factored into our Engine, and so continually tinkering with the overall speed of change will make the NPC more unrealistic and possibility quite wacky.)

```
//use the following function to change the rate of change per interaction with each player.
//this represents the attitude the NPC has toward specific individuals, rather than the overall
//change (i.e., towards Mike the First Player, but not towards everyone and everything
//generally).
//note that the default is 5f

float someAmount = 5f;
myAICI.ChangeBaseChange(someAmount);


//use the following function to change the rate of change for every interaction the NPC has.
//this is like changing the rate an overall personality changes. In the real world, such changes
//usually take years; the default rate for gameplay is 0.1f (which is still faster than real-world
//changes, but necessary for most gameplay situations).

float someOtherAmount = 0.1f;
myAICI.ChangeBkgrndChange (someOtherAmount);
```

Figure 3.3 Altering the speed at which a character's personality changes

To do this (see Figure 3.3):
1) call the **ChangeBaseChange** function of your instance of the AIClientInterface (in the example we've called it "myAICI"), with a float representing the new value. This will change per interaction the attitude the NPC has toward specific individuals, rather than change the overall personality. The default is 5. Note that whole number changes in this value will have a very significant effect.
2) call the **ChangeBkgrndChange** function of myAICI to change the rate of change for every interaction the NPC has. This is like changing the rate an overall personality changes. In the real world, such changes usually take years; the default rate for gameplay is 0.1 (which is still faster than real-world changes, but necessary for most gameplay situations).

3.4 Using AIReturnResult and AINoResult

Sometimes you will want to get a result from the NPC's personality, but not have the mere fact of getting a result alter that personality. In the example in Figure 3.4, we are checking whether an NPC will start a conversation with a player who has just entered the NPC's shop. We don't want to change the NPC's personality (maybe we'd change it after the encounter, though, if for instance things go well and reinforce the NPC's desire to start conversations), so we check using **AIReturnResult**, but with the final parameter (toChange) set to false.

To change the values later without needing another result, we could use **AINoResult**. For example, assume that after the conversation, we want to change the personality depending on whether we've evaluated things to have gone well (and thus reinforcing a stimulus, in this case the desire to start another conversation) or badly (thus making it less desirable to start conversations in the future, especially with this player). We would write something like the code in Figure 3.5, adjusting in a positive direction (the default) or a negative one.

The default action for any personality check requiring a result is to adjust the NPC's personality at the same time (as this seems the more likely pattern). Thus a player being

```
private bool StartConversation()
{

    //determine whether NPC wishes to start the conversation, or wait for the player to
    //start. Use the AI to determine by checking against NPC's gregariousness

    //pass NPC name, player name, tendency to check, and whether to change
    //personality based on this check (in this case, we don't; just getting a result)

    int startConversation = myAICI.AIReturnResult(NPCname, plName, "gregarious",
false);

    //this returns a value between 0 and 4, 0 being low in this tendency, 4 being very high

    //we don’t need a separate response for every return value, so ...
    if (startConversation > 2)
    {
        //will definitely start conversation
        return true;
    } else if (startConversation < 2) {
        //will definitely not
        return false;
    } else {
        //in between; in this case, decided randomly
        System.Random random = new System.Random();
        int randomNumber = random.Next(0, 100);
        if (randomNumber >= 50)
        {
            return true;
        } else {
            return false;
        }
    }
}
```

Figure 3.4 Getting a result without changing the NPC personality

```
private void AfterConversation(bool thingsWentWell)
{

    //after the conversation, we decide to adjust the NPC's gregariousness based on
    //whether there was a positive outcome or a negative one

    //pass NPC name, player name, tendency to check, and whether to change
    //personality in a positive direction or a negative one

    if(thingsWentWell)
    {
        AINoResult(NPCName, plName, “gregarious”);
        //note that the default is for positive change
    } else {
        AINoResult(NPCName, plName, “gregarious”, false);
    }

}
```

Figure 3.5 Changing the personality without returning a result

```
//player has been kind
        whichResponse = myAICI.AIReturnResult(NPCname, plName, "kind");
        switch (whichResponse)
        {
                case 0:
                        NPCResponse = "<replies something neutral>";
                        break;
                case 1:
                        NPCResponse = "<replies something neutral>";
                        break;
                case 2:
                        NPCResponse = "<replies something kind>";
                        break;
                case 3:
                        NPCResponse = "<replies something really kind>";
                        break;
                case 4:
                        NPCResponse = "<offers help to player>";
        break;
        }

        ...
```

Figure 3.6 Changing the personality and returning a result

intimidating to an NPC will immediately change the NPC's personality; there's no need to wait until later to evaluate things. This is even simpler than the previous AIReturnResult example: call AIReturnResult without the final parameter (see Figure 3.6).

In the above example, the player is kind to the NPC, and so she checks her response to kindness and changes slightly for the experience. There are more complex ways to check (and change) the NPC's personality, however. In Figure 3.7, we see an example of an NPC being intimidated by a player. In this case, we decide to check not just one but two response types—one for the NPC's intimidation response, and one for her anxiety response. Why? Because her desire to be intimidating right back might be outweighed by her anxiety over the situation. So in this case we check both, evaluate them against each other, and then figure out the NPC's response.

```
//so we're checking how the NPC responds to intimidation (is she more
//anxious or intimidating right back?) using a more complex combination
//of responses

        int isAnxious = myAICI.AIReturnResult(NPCname, plName, "anxiety");
        int isIntimidating = myAICI.AIReturnResult(NPCname, plName, "intimidating");

        whichResponse = isIntimidating - isAnxious;

        switch (whichResponse)
        {
                case -4:
                        NPCResponse = "<calls for help!>";
                        break;
                case -3:
                        NPCResponse = "<replies something frightened>";
                        break;
                case -2:
                        NPCResponse = "<replies something nervous>";
                        break;
                case -1:
                        NPCResponse = "<replies something neutral>";
                        break;
                case 0:
                        NPCResponse = "<replies something neutral>";
                        break;
                case 1:
                        NPCResponse = "<replies something neutral>";
                        break;
                case 2:
                        NPCResponse = "<replies something annoyed>";
                        break;
                case 3:
                        NPCResponse = "<tells player to get lost>";
                        break;
                case 4:
                        NPCResponse = "<ATTACKS!>";
                        break;
        }
```

Figure 3.7 More complex use of AIReturnResult

3.5 Using AIPlayerHistory and AIPlayerHistoryNums

So now the player and your NPC have met several times. The player has been nothing but nasty the first five times he's been in the shop, but now, he tries being nice as a new tactic. You can't just check using the "kind" response, because even if the NPC's kindness levels have dropped quite a bit, she may still react inappropriately.

Or, say the situation is more complicated: the player was nasty twice, nice once, nasty again, and now tries being the kindest soul in the world. What's an NPC to do?

She should remember what's happened before, not just in the way her personality has been affected, but specifically remembering what kinds of stimuli/responses were called for and how many times.

```
//check AICI for certain patterns to influence further checks
//checking last 5 response checks for Tilla and this player; returns true if 60% or more
//were "kind" or "intimidating" + "anxiety" checks

bool mostKind = myAICI.AIPlayerHistory(5,"TillaTransit",plName,60f,"kind");

//also checking the actual number of matches returned out of the last 5 response checks
//although this isn't used in the response check; just here to show how to do it
int mostKindNum = myAICI.AIPlayerHistoryNums(5, "TillaTransit",plName,"kind");

bool mostIntimidated =myAICI.AIPlayerHistory (5,"TillaTransit",plName,60f,"intimidat-
ing","anxiety");
int mostIntimidatedNum = myAICI.AIPlayerHistoryNums (5,"TillaTransit",plName,"intimi-
dating","anxiety");

if(mostKind) //if the player has been mostly kind, then this response will check kindness
{
        whichResponse = myAICI.AIReturnResult(NPCname, plName, "kind");

        switch (whichResponse)
        {
                case 0:
                        NPCResponse = "<replies something neutral>";
                        break;
                case 1:
                        NPCResponse = "<replies something neutral>";
                        break;
                case 2:
                        NPCResponse = "<replies something kind>";
                        break;
                case 3:
                        NPCResponse = "<replies something really kind>";
                        break;
                case 4:
                        NPCResponse = "<offers help to player>";
                        break;
        }
} else if(mostIntimidated && mostIntimidatedNum > 3) {
        //if not only checked for intimidation or anxiety, but did this more than 3 times ...

        //set NPC response similarly to that in Figure 3.7
} ...
```

Figure 3.8 Using AIPlayerHistory and AIPlayerHistoryNums

That's where AIPlayerHistory and AI PlayerHistoryNums come in. In Figure 3.8, we see an example of AIPlayerHistory being used to find out whether 60% of the last five interactions requested a "kind" or "intimidating" check; if so, we then check the exact number of these using AIPlayerHistoryNums (in case there haven't been five interactions; if there's only been one, maybe we want the NPC to give the player the benefit of the doubt and be nice just one more time).

Note that because our checks for intimidation were more complex (using both anxiety and intimidation response types; see section 3.4), we want to see if the last five responses were either of these; AIPlayerHistory allows you to check any number of response types at the same time, finding instances of any of the responses (so, in this case, either intimidation or anxiety checks).

In the example, we can then use this information to make a more commonsense choice as to the NPC's reactions.

Persistence of Memory: In the near future, Quantum Tiger Games will release an even more specific memory package for NPCs, in which they will be able to remember actual events and the people attached to them, rather than just reaction types. We'll notify you when it's ready! (And you'll get a special upgrade deal if you already own the Personality Engine!)

4.0 Support and Credits

4.1 Support

We're here to help! Contact us at support@quantumtigergames.com with any issues. We'll also post a FAQ on our website (once we have enough questions to have a FAQ!).

4.2 Credits

Design, coding: Jeffrey Georgeson
Raina character created by Mark Foley.

5.0 Citations

What!? References in a user manual? Yes, indeed; because our game AI is based on research dealing with real humans and personality development. The following are a few of the many works dealing with the Five Factor Model of personality:

Costa, PT, Jr and McCrae, RR 1995, 'Domains and Facets: Hierarchical Personality Assessment Using the Revised NEO Personality Inventory', *Journal of Personality Assessment*, vol. 64, no. 1, pp. 21-50.

DeYoung, CG 2010, 'Toward a Theory of the Big Five', *Psychological Inquiry*, vol. 21, no. 1, pp. 26-33.

Goldberg, L 1993, 'The structure of phenotypic personality traits', *American Psychologist*, vol. 48, no. 1, pp. 26-34.

John, OP et al 2010, 'Paradigm Shift to the Integrative Big Five Trait Taxonomy: History, Measurement, and Conceptual Issues', in John et al (ed.), Handbook of personality: Theory and research, 3d ed, Guilford, New York. (US edition.)

McCrae, RR & Costa, Jr PT 2010, 'The Five-Factor Theory of Personality', in John et al (ed.), *Handbook of personality: Theory and research*, 3d ed, Guilford, New York. (US edition.)