Design Patterns for the Configuration of Utility-Based AI

Kevin Dill
Lockheed Martin
Global Training and Logistics
Burlington, Massachusetts
Kevin.Dill@lmco.com

Eugene Ray Pursel
Marine Corps
Warfighting Laboratory
Quantico, Virginia
Eugene.Pursel@usmc.mil

Pat Garrity, Gino Fragomeni
U.S. Army Research Laboratory
Simulation and Training Technology Center
Orlando, Florida
Pat.Garrity@us.army.mil
Gino.Fragomeni@us.army.mil

ABSTRACT

There is an ongoing need for improved autonomous virtual characters for military training, particularly in areas such as squad-level scenarios for the Army and Marines. In the past, simulations have often used techniques such as scripting or Finite State Machines for Artificial Intelligence (AI) control of non-player characters. These approaches allow the scenario creator to have precise control over the actions of the characters, but the cost of configuration and the quality of the result scale poorly as the complexity of the AI grows. As a result, they tend to lead to AI behaviors that are rigid and predictable, and thus are insufficiently reactive to unexpected situations and not suitable for replay or repeated use.

In previous papers we have endorsed utility-based AI as our preferred alternative. This approach enables the developer to think in terms of heuristic equations rather than simple black-and-white decisions, and thus to create an AI which can examine the subtle nuance of the situation and select actions accordingly. The resulting characters retain the strong authorial control of previous approaches, but they can be far more believable, adaptable, and reactive to the situation around themselves.

Utility-based AI is flexible and powerful, but newcomers may find guidance useful in the face of such flexibility. In this paper we propose several design patterns that can be applied to the configuration of utility-based AI. Much like design patterns for software engineering, the intent is to share "simple and succinct solutions to commonly occurring design problems" (Gamma et. al., 1994). These patterns can provide a complete solution for simple AI problems, but more importantly they provide a solid foundation on which more complicated logic can be built.

ABOUT THE AUTHORS

Kevin Dill is a member of the Group Technical Staff at Lockheed Martin Global Training and Logistics, and the Chief Architect of the Game AI Architecture. He is a recognized expert on Game AI and a veteran of the game industry, with seven published titles. He was the technical editor for Introduction to Game AI and Behavioral Mathematics for Game AI, and a section editor for AI Game Programming Wisdom 4. He has taught classes on game development and game AI at Harvard University, Boston University, and Worcester Polytechnic Institute.

Ray Pursel served over 23 years as a Marine in both enlisted and officer roles. His billets ranged from Aviations Operations Clerk to Helicopter Section Leader to Modeling and Simulations Officer. He earned a B.S. in Computer Science and Mathematics Minor from the Pennsylvania State University in 1995 and an M.S. in Modeling, Virtual Environments and Simulation from the Naval Postgraduate School in 2004. Now retired from active duty, he is serving as a Modeling and Simulations Analyst with the Marine Corps Warfighting Laboratory.

Pat Garrity is the Chief Engineer for Dismounted Soldier Training Technologies at the Army Research Laboratory's Simulation and Training Technology Center. He currently works in the Ground Simulation Environments Branch conducting research and development in the area of dismounted Soldier training and simulation where he was the Army's Science and Technology Objective Manager for the Embedded Training for Dismounted Soldiers Science and Technology Objective. His current interests include Human-In-The-Loop

networked simulators, virtual and augmented reality, and immersive dismounted training applications. Garrity earned his B.S. in Computer Engineering from the University of South Florida in 1985 and his M.S. in Simulation Systems from the University of Central Florida in 1994.

Gino Fragomeni serves as a Science and Technology Manager for Dismounted Soldier Technologies at the U.S. Army Research Laboratory, Simulation and Training Technology Center. He currently works in Ground Simulation Environments Branch conducting research and development in the area of dismounted soldier training and simulation. His current interests include artificial intelligence and immersive environments centric to dismounted training applications. Gino is a highly qualified science and technology manager as well as being a reservist with the United States Army Special Operations Command-Central with over 28 years of military experience. He earned a Master of Science from the University of Central Florida (2002) and has specialized training in Systems Engineering.

Design Patterns for the Configuration of Utility-Based AI

Kevin Dill
Lockheed Martin
Global Training and Logistics
Burlington, Massachusetts
Kevin.Dill@lmco.com

Eugene Ray Pursel
Marine Corps
Warfighting Laboratory
Quantico, Virginia
Eugene.Pursel@usmc.mil

Pat Garrity, Gino Fragomeni
U.S. Army Research Laboratory
Simulation and Training Technology Center
Orlando, Florida
Pat.Garrity@us.army.mil
Gino.Fragomeni@us.army.mil

There is a strong need for high quality autonomous virtual characters for military training, particularly in areas such as squad-level training for the Army and Marines. Such characters require Artificial Intelligence (AI) to control their behavior. Utility-based AI is one commonly used approach (Mark 2009, Dill 2012, 2011, 2008, and 2006, and Davis 1999, among others). It provides a combination of authorial control, reactivity, and believability that can be difficult to match using other architectures (Dill 2012).

The term *utility-based AI* is used to describe a class of techniques in which decisions are made on the basis of heuristic functions that represent the relative value (or appropriateness) of each option under consideration in terms of a floating-point value. Thus, utility-based approaches typically have three general steps:

1. Build a list of ***options***, which are the choices from which we will choose.
2. Evaluate each option and calculate one or more floating point values that describe how attractive the option is given the current situation. These values can have a variety of names, such as utility, priority, weight, rank, urgency, importance, etc.
3. Select an option (or set of options) for execution on the basis of the values calculated in step 2.

A key point is that the evaluation in step 2 **must occur at run time**. In other words, the utility values are not selected when the scenario is designed and fixed thereafter, but rather calculated at run time based on the details of the situation in the simulation at that particular moment. Thus utility-based AI is constantly reevaluating the situation and selecting the most appropriate option or options at each moment in time.

The use of a floating point evaluation, rather than a series of Boolean checks, allows us to have a much higher level of granularity. Instead of only having a few values to choose from (a series of "yes or no" checks), we can express things in ways that are fully continuous.

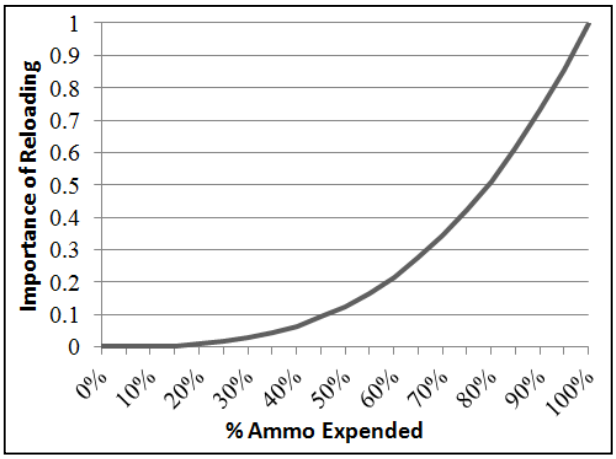

Figure 1: A hypothetical utility function calculating the importance of reloading based on percent ammo expended.

For example, we might decide that the importance of reloading should vary as the cube of the percent of ammo expended (**Error! Reference source not found.**). While that might seem a bit strange and awkward, it is actually quite expressive. It tells the AI that the importance of reloading increases as ammo is expended. Furthermore, this urgency increases gradually at first but then more and more rapidly as the current magazine empties. Thus if we have a typical infantryman's magazine with 28 rounds then we will only assign an importance of 0.125 to reloading when 14 rounds remain, but that importance will increase to roughly 0.42 when seven shots remain and 0.8 when we are down to our last two shots. Finally, we can further modify the formula at will – so we might prevent early reloads by setting the utility to 0 when the magazine is less than ¾ empty, for instance, and then use the above formula thereafter.

Utility values are meaningless in isolation. In other words, the above formula doesn't tell us anything by

itself, because we have nothing to compare it to. Thus the next step is to calculate the importance for every other option – that is, the importance of getting behind cover, of firing at the enemy, of applying first aid to a wounded buddy, and so forth. We might scale each of these from 0 to 10, so reloading is relatively unimportant as long as we have rounds left, but we still might consider it if we didn't have any other good options. Of course we can't fire a weapon with an empty magazine, but that can be reflected by setting the utility of the firing option to 0 when the magazine is empty. Thus we would be prevented from trying to fire in that situation, but we still might choose to get behind cover or to apply first aid if appropriate.

Again, and this can't be over-emphasized, each of these values is calculated in simulation, at run time, based on the moment-to-moment situation facing the character, so that the utility values represent an up-to-date appraisal of current priorities. The result is an AI that is capable of examining the subtle nuance of the situation in as much detail as we care to encode and selecting a course of action accordingly. It is able to think in terms of shades of gray, rather than the stark black and white of purely Boolean approaches.

That expressiveness does come at a cost. Configuring a utility-based AI requires the programmer to express every decision in terms of numbers, which is somewhat unnatural (especially at first). Of course, the same could be said of any programming task. Humans don't think the way that computers do, and as a result we have to learn to express ourselves in ways that the computer can more easily understand.

The field of software engineering has sprung up to provide programmers with a collection of shared techniques and conventions for writing computer code. From object oriented design to polymorphism to design patterns to newer concepts such as test-driven development and pair programming, these techniques can be used to help manage the complexity of expressing human ideas in terms of machine code.

In Design Patterns: Elements of Reusable Object-Oriented Software (Gamma et. al., 1994), one of the classic books on software engineering, the authors describe the situation in the early 1990s as follows:

> None of the design patterns in this book describe new or unproven designs… [but] most of [them] have never been documented before. They are either part of the folklore of the object-oriented community, or are elements of some successful object-oriented system – neither of which is easy for novice programmers to learn from.

Much the same could be said of the situation for utility-based AI today. There are a variety of techniques which are known to some, but engineers either have to discover these on their own or learn them by word of mouth. Although there is a book dedicated to the topic (Mark 2009), and some shorter articles exist, there hasn't been a coherent effort to lay out shared tips, tricks, conventions, and techniques for the process of configuring a utility-based AI.

The goal of this paper, then, is to begin to discuss what we might call *Utility Engineering*, and in particular to present design patterns that we have found in our work with utility-based AI. These patterns serve several purposes. First, many AI problems really are quite simple. Advocates of more Boolean approaches often argue that utility-based AI is needlessly complex, complicating the configuration process when in many cases simpler approaches can work just as well. Utility patterns address this by giving us consistent, reusable solutions for the simple problems, while retaining the flexibility to use more complex evaluations as needed.

When we do face one of those more complex decision-making problems, utility patterns can provide us with a starting point from which our solution can grow. While not all decisions can be expressed in terms of patterns, it is often possible to combine these patterns together to create behavior that is much more complex than any of the patterns alone, or to start with a pattern and then extend or modify it to fit our needs.

Finally, utility patterns are *shared conventions*. Thus they can provide us with a common vocabulary to use when discussing the details of an AI configuration with our teammates or other professionals, and can also allow us to more quickly recognize the intent of unfamiliar code.

The remainder of this paper will first briefly discuss the particular utility-based architecture that we use, not because that architecture is necessary for the use of utility patterns but rather because the examples we give will rely on an understanding of it. Next, we will talk about some of the other conventions (aside from utility patterns) that are important to agree upon within your team. Third, we will present several of the most common patterns we've found in our own work. Finally, we will give an example that combines several utility patterns together into a cohesive whole.

GAME AI ARCHITECTURE OVERVIEW

It is difficult to talk about utility engineering without having a specific utility-based architecture in mind. While the techniques we discuss are adaptable to other architectures (just as software engineering techniques designed for Java can often be adapted to C++ or vice versa), it's difficult to describe them without the context of the architecture that they will run on. With that in mind, we first present an overview of the relevant aspects of the Game AI Architecture (GAIA).

GAIA provides a modular, hierarchical decision making framework which, similar to the popular behavior tree architecture (Isla 2005), allows the user to select the most appropriate approach to decision making for each decision to be made. The architecture was described in detail in a recent paper (Dill 2012), so we focus here on modular decision making and the Dual Utility Reasoner.

The Dual Utility Reasoner

There are two common approaches to utility-based selection. The first, *absolute utility*, is to evaluate every option and take the one with the highest utility. The second, *relative utility*, is to select an option at random, using the utility of each option to define the probability that it will be selected. The probability (P) for selecting an option (O) is determined by dividing the utility (U) of that option by the total utility of all options:

$$P_O = \frac{U_O}{\sum_{i=1}^{n} U_i}$$

This approach is commonly referred to as *weight-based random* or *weighted random*.

The Dual Utility Reasoner combines both of these approaches. It assigns two utility values to each option: a *rank* (absolute utility) and a *weight* (relative utility). Conceptually, rank is used to divide the options into categories, where we only select options that are in the best category. Weight is used to evaluate options within the context of their category. Thus the weight of an option is only meaningful relative to the weights of other options within the same rank category – and only the weights of the options in the best category truly matter.

When making a decision, we begin by calculating the rank and weight for each option and eliminating any options with a weight of 0. As the above formula indicates, these options cannot be selected by the weighted random step, so eliminating them at this early stage simplifies the remaining logic. It also gives the AI designer a convenient way to eliminate a particular option in a given circumstance – simply set the weight to 0 and an option will be rejected, even if it otherwise would have had the highest rank.

Next, we find the highest rank from among the options that remain, and eliminate any options with lower rank. Again, conceptually what we are doing is finding the most appropriate category of options, and eliminating options that don't belong.

Third, we eliminate options whose weight is significantly less than that of the best remaining option. The intent here is to eliminate options that would look stupid if selected.

The exact cutoff ratio to use in this step is data-driven, and varies depending on the decision being made. For example, if we're choosing between two weapons, one with a weight of 5 and the other with a weight of 1, then we should probably not select the second weapon (doing so will look stupid). On the other hand, sometimes we have a large number of options that are very similar, whose collective weight is what's important. For example, if we're configuring a target selection AI for a sniper shooting at a platoon of Marines, we might want the probability of shooting the platoon leader to be roughly twice that of shooting one of the other Marines. Since there are roughly 40 enlisted Marines in a platoon, the probability of shooting each of them would only be about 1.25% that of shooting the platoon leader – far less than the 20% cutoff ratio given above. If we allow this step to eliminate those low weight options then the probability of shooting the platoon leader becomes 100%, which was not the intent.

Finally, we use weight-based random to select from among the options that remain.

Again, those four steps are as follows:

1. Eliminate all options with a weight of 0.
2. Determine the highest rank category, and eliminate options with lower rank.
3. Eliminate options whose weight is significantly less than that of the best remaining option.
4. Use weighted random on the options that remain.

We will work through numerous examples of this process as we present the design patterns.

Modular Decision Making

The modular approach to decision making used in the Dual Utility Reasoner was first discussed in Game Programming Gems 8 (Dill 2010) and further refined in more recent papers (Dill 2011, Dill 2012). The key idea is that the logic for a decision can be broken into one or more discrete *considerations*. A consideration is a piece of code which examines a single aspect of the situation in isolation and then returns an evaluation that can be combined with those of the other considerations to guide the overall decision.

There are a near-infinite number of possible decisions that might need to be made in some AI somewhere. For example, we might need to select a target to shoot at or to look at, we might need to select a weapon to use, we might need to decide whether to lay down suppressive fire or to advance toward the enemy, or we might need to decide whether to eat a hamburger or a hot dog. There are far fewer types of considerations that might be used to make those decisions, however. We currently support only 16 types of considerations, and that has been sufficient for several different scenarios.

Using this approach, the process of configuring an option becomes one of simply specifying the considerations to apply and the control parameters for each consideration. Because there are relatively few types of considerations, this turns out to be much more compact than fully implementing that logic each place it is used. Furthermore, the considerations map much more closely to human concepts than do C++ commands. Thus AI designers are able to think in terms of larger, higher-level concepts that more naturally fit the way that they themselves make decisions. Finally, each consideration is reused over and over again. Thus rather than duplicating our AI logic each place that it is needed, we implement it once, test it carefully, and then rely on a single, well-maintained implementation. This results in code that is far more robust and maintainable than would otherwise be possible.

In our current implementation, each consideration returns a rank (R_i), a bonus (B_i), and a multiplier (M_i). The overall rank and weight of an option are then determined as follows:

The rank of an option (R_O) is typically calculated by taking the maximum of the ranks returned by its considerations:

$$R_O = Max_{i=1}^{n}\left(R_i\right)$$

Recently we have experimented with allowing the AI designer to specify that a particular decision should take the minimum or sum instead, but those are rarely used.

The weight of an option (W_O) is calculated by multiplying the sum of its bonuses by the product of its multipliers.

$$W_O = \left(\sum_{i=1}^{n} B_i\right) \cdot \left(\prod_{i=1}^{n} M_i\right)$$

As with the ranks, this approach for combining the considerations' results is quite simple, but experience has shown that it is extremely expressive. Each consideration can do any of the following:

- Give the option a higher rank
- Eliminate the option by returning a multiplier of 0
- Adjust the option's weight through the use of non-zero bonuses and multipliers

When we need a more complex formula – for example, the ammo formula given in Figure 1 – we can simply encapsulate that formula within the consideration itself and return its value in the form of a bonus or multiplier.

CONVENTIONS

Configuration of a utility-based AI is easier if you have a set of conventions which describe consistent values to be used in common situations. The following are examples of conventions from our work:

- The default weight for an option is 1 (that is, an option with no considerations will be given a weight of 1). We tune the weights of other options accordingly. Thus if we want to make the AI twice as likely to select an option, then that option's weight should be 2.
- The default rank for an option is 0. Higher or lower ranks can be used.
- Most normal, unexceptional options should have a rank between -5 and +5, with 0 as the baseline. For example, the ambient (noncombat) behavior of civilians would typically have a rank of 0, but specific options might be slightly higher or lower.
- Urgent, high priority options should have a rank between 5 and 15, with 10 as the baseline. For example, that same character would typically use a weight of 10 for combat options (fleeing,

seeking cover, returning fire, etc). Again, specific options might be slightly higher or lower.
- A rank of 20 or higher indicates an option that is extremely important and should always be executed immediately.
- A rank of 1,000,000 is reserved for autonomic reactions that the AI has no control over, such as stumbling backward or dying when hit by a bullet.

These conventions serve three purposes. First, when we are debugging the AI and we see a particular value, we know at a glance roughly what that value should indicate – and we can judge whether it is appropriate, and adjust it if it is not. Second, when we are adding a new option to the AI, we have guidelines as to how we should set its utility values so that they will function smoothly with the values of other options. Finally, much like a coding standard, they define a shared standard that all of the AI engineers on a project can agree to, so that work done by one engineer will function smoothly when used in a portion of the AI configured by somebody else.

UTILITY PATTERNS

We are ready to turn our attention to the utility patterns. Each pattern is designed to be as simple as possible while still capturing the concept that it is represents.

We will use two hypothetical characters in our examples. Neither exactly matches one that we have built, but both are based on genuine characters from our recent work.

Our first example character is an insurgent who has climbed onto a rooftop with the intention of sniping at a Marine patrol that is passing through the marketplace below. This character is very similar to a character we've created for a recent GAIA demonstration. We will examine the decision making process he goes through as he decides when to take a shot and when to withdraw in hopes of fighting again another day.

The second character is a Muslim woman, very similar to our Angry Grandmother character (Dill 2011) who is home alone when the Marines enter and search her house. This is a nonkinetic scenario in which the woman largely just rants at the Marines, exhorting them to leave, but she can respond to Marine actions (e.g. if they aim their weapons at her, fire their weapons, enter or leave the room, and so forth). Thus her decisions largely involve picking her next line of dialog and selecting moments to play specialized sequences (such as when the Marines enter the room or aim at her).

OPTION VALIDATION PATTERNS

While the great strength of utility-based AI is that it allows us to evaluate the situation in terms of shades of gray, there are often cases where decisions are clear cut. Thus we need to be able to validate our options, ensuring that only those that are reasonable (given the current situation in the simulation) will execute.

Opt Out

The Opt Out pattern is the simplest pattern that we will discuss. At times, there is a consideration which knows absolutely that an option should not be executed – even if the option would otherwise have a high rank. We do this by having the consideration return a multiplier of 0. Doing so will result in an overall weight of 0, which will cause the option to be eliminated in the first step of the decision-making process (as described above).

For example, imagine an experienced sniper deciding whether or not to take a shot. He will not fire if any one of the following is true:

- He does not have a clear line of sight to his target
- He does not have a clear line of retreat (i.e. his planned escape route is under observation)
- He has fired a shot too recently
- He has already fired enough times that he fears further shots will expose his position

Each of these four factors would be represented by a single consideration, which will return a multiplier of 0 if that consideration feels that the shot should not be taken. Since any multiplier of 0 will set the option's overall weight to 0, and we eliminate options with a weight of 0 in the first step of the decision-making process, this will prevent the Shoot option from being selected when firing is inappropriate.

Opt In

The Opt In pattern is similar to Opt Out, except that instead of ruling the option out, the considerations rule it in. Conceptually, the Opt In pattern is appropriate when there is more than one reason that you might select a particular option, and **only one** of those reasons needs to be true in order for the option to be valid. In contrast, the Opt Out pattern says that **all** reasons must

be valid if we are to select the option. In other words, Opt Out provides a logical *and*, while Opt In provides a logical *or*.

In order to make the Opt In pattern work, we first need to have a default option which has a relatively low rank, but is always valid. For example, it might always return a rank of 0 and a weight of 1 (the defaults from our shared conventions). We would then configure the option which can be opted in so that its base rank is -1 (i.e. lower than that of our default option). Each of its considerations will return a rank greater than 0 when the option is a reasonable choice. Since the option's rank is the maximum of the ranks of its considerations, the result is that if none of the Opt In considerations apply then the rank will be -1, which will prevent the option from being selected (the default option will be selected instead). If at least one consideration "opts in," however, than the option is certain to be selected over the default option (although there may be other options with still higher tanks as well).

As an example, consider our angry woman. Through much of her performance, she will simply scream at the Marines, nearly incoherent, trying to get them out of her house. This screaming could be handled by the Rant option. We might also want to have several specialized performances which can be delivered in particular situations. For example, the Marines Threaten option might be selected if the Marines aim their weapons at her, if shots are fired, or if the exercise operator presses the corresponding button at the Instructor/Operator Station (IOS). This option would have our character take a few steps backwards, raise her hands in supplication, and say something like "Please! Don't shoot! I'm just an innocent woman!"

In order to implement this, we would use the Rant option as our default option. Thus it will always have a rank of 0 and a weight of 1. Next we would place four considerations on the Marines Threaten option:

- A Tuning consideration (which sets the default rank to -1).
- An Aimed At consideration, which returns a rank of 10 if the Marines point their weapons at her.
- A Shots Fired consideration, which returns a rank of 10 when shots are fired nearby.
- A Button consideration, which returns a rank of 10 when the corresponding IOS button is pressed.

The result is that the character will normally just rant, but if she is threatened by the Marines (or if the exercise operator tells her to) then she will play her specialized performance instead.

Combining Opt In and Opt Out

It is often useful to combine the Opt In and Opt Out patterns. For example, we might not want our sniper to open fire the moment the first Marine enters the marketplace. Instead, we might want him to wait until one of the following is true:

- At least five Marines are in the market
- The last Marine is leaving the market
- He has identified a high priority target (such as an officer, medic, or radio operator) in the market
- The corresponding IOS button is pressed

Of course, he should still not fire if any of the four considerations discussed in the Opt Out section are true.

In order to make Opt In work, we need a default option. Thus we would create a Wait option with a fixed rank of 0 and weight of 1 This option would simply have the character remain in hiding, waiting for the opportunity to take a shot. Next, we would add Opt Out considerations to the Fire option. Thus he would have considerations for checking line of sight, line of retreat, and so forth that will return a multiplier of 0 when a shot is inappropriate. Finally, we would add the Opt In considerations: a tuning consideration that sets his default rank to -1, and a collection of situational considerations that return a higher rank when there are at least 5 Marines in the market, when the last Marine is leaving the market, and so forth. The end result would be that he will only fire if (a) none of the opt-out considerations apply and (b) at least one of the Opt In considerations is valid.

We could further extend this character by adding a Flee option, which uses the Opt In pattern to make the character run away if he has reached the maximum allowable number of shots, or if he's taking aimed fire from the enemy. Fleeing is more important than firing, so the considerations might set a rank of 11 or 12, rather than the 10 used by the Fire option.

More complex combinations of Opt In and Opt Out are possible, but they require us to be able to group considerations together (much as the parentheses group together logical and and or clauses). This is a fairly obvious extension that is supported in our code, but is beyond the level of complexity appropriate to a utility pattern. Remember that the purpose of utility patterns is

to provide simple examples which can illustrate the basic concept and serve as the foundation for your implementation, not to cover every possible use case.

EXECUTION HISTORY PATTERNS

There are a number of patterns which depend on knowledge of the option's execution history. We might want to adjust our decision if the option has never been selected, for example, or if it is (or is not) currently executing. Consequently we have an Execution History consideration which can be configured to return different values in each of these states. In each case, the values returned can vary as a function of the amount of time since the last state change (i.e. the time since we last started or stopped execution, or the time since the scenario was loaded if we have never been selected). There are several patterns which capitalize on this consideration.

Commit

There are often cases where we want to ensure that we carry an action through to completion, even if the considerations that originally caused us to select it are no longer being triggered. For example, consider the Flee option for our sniper. This option will be selected if the sniper takes aimed fire from the Marines. If the Marines stop firing – perhaps because the sniper ran around a corner, or got behind cover – he should still keep fleeing. Once this option is selected, we should commit to it until it has been successfully completed.

The Commit pattern uses an execution history consideration to return a high rank value whenever the option is executing. Thus the option can only start executing as a result of some other consideration, but once it is executing the rank will remain high until execution stops. Because we want to commit to this option, we set the rank to be even higher than the rank used to select the option in the first place (i.e. now that this option has been selected, it is more important than other options which would otherwise be in its rank category). Thus if the Flee option normally used a rank of 12, the Commit pattern might set a rank of 14.

The option can still be deselected normally. For example we might have a consideration which opts out (setting a multiplier of 0), which forces the option to be deselected, or we might have another option with even higher rank that replaces it.

Inertia

The Inertia pattern is similar to the Commit pattern, except that in this case we configure the execution history consideration so that it will only maintain a portion of the original rank.

For example, imagine that the Marines aim their weapons at our angry woman, triggering her "Marines Threaten" performance, and then aim away again. We don't want her to go back to ranting in the middle of this performance – that would look odd. On the other hand, we do want her to remain responsive if something else important happens (if the operator presses a different button, for example, or if the Marines find contraband in her room). Thus if the special performance options are selected with a rank of 10, then we will configure our execution history consideration to set the rank to something like 7 whenever the option is executing. Again, these values should be standardized where possible, and specified in your shared conventions.

Is Done

The Is Done pattern is a special case of the Opt Out pattern, in which we return a multiplier of 0 when (a) the option is currently selected, and (b) the option's execution has been completed. We have a specialized consideration, the Is Done consideration, whose only responsibility is to make this check.

For example, our sniper would have an Is Done consideration on his Shoot action. This forces him to stop executing that option once the shot has been taken, so that he doesn't get stuck in a completed option forever. Similarly, our angry woman would have an Is Done consideration on her Marines Threaten option, which would return her to ranting once the option's performance is complete.

Cooldown

We often need to prevent the AI from selecting the same thing twice in close succession. For example, if our sniper does take more than one shot, he should delay several minutes between them so as to reduce the probability that he will be spotted on successive shots.

The Cooldown pattern enforces this by opting out (that is, returning a multiplier of 0) for a predetermined period of time once the option stops executing. The duration of the cooldown can either be fixed (for example, the sniper might always wait three minutes between shots) or it can be randomly selected (so in the

sniper's case, we might vary the duration of the cooldown randomly between 90 seconds and five minutes). In the latter case, we should select a new random duration each time the option stops executing.

Cooldowns are not only used for purely behavioral reasons. Often, we use a cooldown to disguise the reuse of assets. For example, the angry woman's Rant option would use a subreasoner to select specific lines of dialog, and specific gesture animations to play with each line. We want to ensure that we don't ever repeat the same line of dialog or gesture twice in a row, or even twice within a few seconds of each other, because such repetition would be obvious to the viewer, and would look distinctly odd. As a result, we might place a 20 second cooldown on every gesture and every line of dialog, to prevent immediate reuse. This technique is one of several that was described in our 2011 I/ITSEC paper for building characters that are more believable and immersive (Dill 2011).

Do Once

One of the reasons why animated movie characters can be so much more compelling than video game characters is that a movie is a single two-hour-long animation, with every moment carefully choreographed and hand-animated. At any given moment in the movie the animator knows exactly what happened a moment before – down to the jiggle in the earlobe – and exactly what will happen a moment in the future. In contrast, characters for games and simulations are built out of hundreds or often thousands of animations, each extremely short (often less than a second), procedurally stitched together in real time to create an interactive experience. These animations are used over and over, but each is so short, and they are combined in so many ways, that the repetition is typically not obvious (or at least, it's not obvious if we've done our job well).

We can give our characters longer, deeper, more compelling animations, but there are two challenges with this approach. First, these animations are hard to interrupt – they often take the character far away from their base pose, resulting in at best an awkward blend (and often foot sliding or worse) if you interrupt them. The second problem is that the whole point of these animations is for them to be big and memorable – so if we reuse them, that repetition is bound to be noticed. Real humans rarely do exactly the same thing exactly the same way, so this sort of repetition will break the viewer's suspension of disbelief.

As a result, we tend to reserve these animations for big, dramatic moments in our simulation – the ones where we really want to make an impact on the viewer. For example, when the Marines first enter, our angry woman might lurch to her feet from a sofa, dropping her knitting and frantically pulling her veil over her face, while she cries out "What! Who are you! Why are you here? This is my house – my husband isn't home! Get out, get out, get out!!" Similarly, our sniper might have a rant which he delivers from his rooftop while waving his gun in the air, just before he runs away.

We want to commit to these options once they have begun, so that they won't be interrupted, but we already know how to do that using the pattern described above. Additionally, we want to ensure that we only ever do them once. If the Marines leave and then come back, for instance, the angry woman should not play that particular sequence – the repetition would be obvious. If this is likely to happen then we might have more than one performance to pick from (see the Sequence pattern, below), or she might go straight into ranting.

The Do Once pattern is the pattern that we use to ensure that an action is only done a single time, and it is one of the simplest patterns. In order to implement it, we use an execution history consideration that is configured to opt out if the option has ever been selected in the past.

One-Time Bonus

Similar to the Do Once pattern, there are sometimes situations where we want to give an option high priority to be selected the first time, but then lower priority thereafter. For example, while there are situations in which our sniper should not take any shots at all, he should also be significantly less aggressive when deciding to take a second, third, or fourth shot, because each additional shot significantly increases the likelihood that he will be spotted. As a result, we might apply a One-Time Bonus to the rank of the Shoot option, which is simply an execution history consideration which applies a bonus to the weight or proposes a high rank. In the case of the sniper, we might configure it to propose a rank of 15. This marks it as something that is quite important, while still allowing other considerations to opt out when shooting is inappropriate (for instance if there are no targets, or if the sniper's escape route is compromised).

Note that this pattern is incompatible with the Opt In pattern as it's described above. Again, there are possible solutions (such as a Reasoner Selection Consideration, which uses a reasoner to select from

among two or more sets of child considerations), but they are beyond the complexity appropriate for a design pattern.

Repeat Penalty

Applying a one-time bonus to the first shot fired is one way to discourage the sniper from taking multiple shots. It would be preferable, however, to decrease the rank of the Fire option with each shot fired. We might have an initial rank of 10, for example, and then decrease that rank by 2 for every shot fired (so the first shot would be rank 10, the second rank 8, the third rank 6, and so forth). We could then balance the rank of the Flee option accordingly – perhaps giving a higher rank if the AI is taking aimed fire, or a lower rank if the Marines are running away.

The Repeat Penalty pattern uses a special consideration (called the Repeat Penalty consideration) to keep track of how many times an option has been selected, and adjust its rank accordingly. It will return a rank (*R*) equal to the initial rank (*I*) minus the repeat penalty (*P*) times the number of past executions (*E*)

$$R = I - (P \cdot E)$$

In the case of the Fire option, we would use an initial rank of 10 and a repeat penalty of 2.

COMBINING PATTERNS

While each of these patterns defines a conceptual building block that can be used in the creation of our character configuration, the most interesting thing is not that they can be applied in isolation, but rather the ways in which they can be used together to shape behavior. We have already hinted at that in many of the examples above, but it's worth pulling together all of the pieces of our sniper's AI so that we can examine the ways in which they work together.

Our sniper has two options: Shoot and Flee. The Shoot option has him select an actual target, and then take a single shot at that target. The Flee option has him select an appropriate exit strategy. This could include anything from leaving his weapon and calmly walking away (if he hasn't been spotted) to frantically jumping from rooftop to rooftop as he flees under fire.

For the Fire option, the first thing we will do is apply the Opt Out considerations, as follows:

- Check line of sight and opt out if there aren't any valid targets in the kill zone.
- Check whether any Marines are able to observe our escape route, and opt out if this is the case.
- Use an Execution History consideration to apply the cooldown.

Next we will apply the One-Time Bonus and the Repeat Penalty:

- Use an Execution History consideration to suggest a rank of 15 if the option has not yet executed
- Use a Repeat Penalty consideration to suggest a rank of 10 with a repeat penalty of 2.

As a result the Fire option will be invalid if there are no targets, when the escape route is compromised, or when on cooldown. If none of those applies, the option's rank will be set based on the One-Time Bonus and the Repeat Penalty.

The final option we need to configure is the Withdraw option. We could apply considerations as follows:

- Use a Shots Fired consideration to check if the Marines are firing at or near the sniper (i.e. is he taking aimed fire). If so, propose a rank of 20.
- Use another Shots Fired consideration to check if shots are being fired, but they aren't shooting near him. In this case we could set a lower rank, such as 7 or 5 (allowing him to fire two or three shots, respectively)
- Check if the Marines are moving to encircle the building. If so, propose a rank of 10 (allowing him to fire only a single shot, and then only if his line of retreat is still clear).
- Check to see if the Marines are moving away from the marketplace. If **not**, propose a rank of 1 (so if they stay in the marketplace, he's limited to five shots).
- Finally, use a tuning consideration to set a minimum rank of -10.

The result will be that the character will withdraw immediately if he is taking fire. He will take only a single shot if the Marines are surrounding his building. Otherwise, he will normally take five shots, but may take as many as ten if the Marines appear to be running away.

As discussed above, the one thing we weren't able to include here is the Opt In strategy that would allow the sniper to wait until there are multiple targets, until there

is a high priority target, or until the last Marine is leaving the kill zone. That level of complexity is beyond the scope of these design patterns, but is the sort of thing that utility-based AI can do very well.

The design patterns are intended to provide a place to begin, a place where novice users can look as they learn to work with utility-based AI and a place where experienced users can find consistent implementations for common, simple use cases. More complex cases can often build upon them, such as in the case of the consideration sets and Reasoner Selection Consideration which we alluded to above. For the advanced user, *Behavioral Mathematics for Game AI* (Mark 2009) provides a good starting point for learning how to build formulas that represent complex decisions.

CONCLUSION

In this paper we first outlined the benefits of utility-based AI in general, and described our Dual Utility Reasoner and its modular approach to decision making. Next, we talked briefly about the importance of using shared conventions when configuring utility-based AI, even when working alone but especially when more than one AI designer is working on the same project. Finally, we outlined a number of the recurring design patterns that we have found in our work, discussed how they could be configured in the Dual Utility Reasoner, and gave an example showing how multiple patterns can be combined to create a more complex configuration.

As utility-based AI continues to become better known and more widely used, we hope that we will see an increase in the number of papers and books that discuss utility engineering, which is to say, best practices for configuring utility-based AI.

ACKNOWLEDGEMENTS

The authors would like to thank the US Marine Corps Warfighting Laboratory and the U.S. Army Research Laboratory's Simulation and Training Technology Center for their generous support of this work.

REFERENCES

Davis, I. (1999). Strategies for Strategy Game AI. *AAAI 1999 Spring Symposium on AI and Computer Games Proceedings*.

Dill, K. (2006). Prioritizing Actions in a Goal-Based RTS AI. *AI Game Programming Wisdom 3*. Boston, Massachusetts: Cengage Learning.

Dill, K. (2008). Embracing Declarative AI with a Goal-Based Approach. *AI Game Programming Wisdom 4*. Boston, Massachusetts: Cengage Learning.

Dill, K. (2010). A Pattern-Based Approach to Modular AI for Games. *Game Programming Gems 8*. Boston, Massachusetts: Cengage Learning.

Dill, K. (2011). A Game AI Approach to Autonomous Control of Virtual Characters. *Proceedings of the 2011 Interservice/Industry Training, Simulation and Education Conference*.

Dill, K. (2012). Introducing GAIA: A Reusable, Extensible Architecture for AI Behavior. *Proceedings of the 2012 Spring Simulation Interoperability Workshop*.

Gamma, E., Helm, R., Johnson, R., and Vlissides, J. (1994). *Design Patterns: Elements of Reusable Object-Oriented Software*, Boston, Massachusetts: Addison-Wesley.

Isla, D. (2005). Handling Complexity in Halo 2 AI. http://www.gamasutra.com/gdc2005/features/20050311/isla_01.shtml.

Mark, D. (2009). *Behavioral Mathematics for Game AI*, Boston, Massachusetts: Cengage Learning.