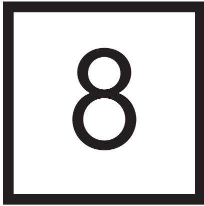
Copyright Material – Provided by Taylor & Francis



Modular AI

Kevin Dill and Christopher Dragert

8.1 Introduction
8.2 Theoretical Underpinnings
8.3 GAIA Overview
8.4 GAIA Infrastructure
8.5 Modular AI: Conceptual Abstractions and Modular Components
8.6 Combining Considerations
8.7 Pickers
8.8 Conclusion
References

8.1 Introduction

Repetition is everywhere in AI. The same patterns, the same fragments of code, the same chunks of data, the same subdecisions get used over and over again in decision after decision after decision. As a general rule, when we as software engineers see repetition we try to encapsulate it: put it in a procedure, put it in a class, or build some abstraction so that there is only a single instance of that repeated pattern. This encapsulation can now be reused rather than rewriting it for each new use-case. We see this approach throughout software engineering: in procedures, in classes, in design patterns, in C++ templates and macros, and in data-driven design—to name just a few examples.

Reducing repetition has numerous advantages. It decreases the executable size. It decreases the number of opportunities to introduce a bug, and increases the number of ways in which the code is tested. It avoids the situation where you fix a bug or make an improvement in one place but not others. It saves time during implementation, allowing you to write new code rather than rewriting something that already exists. It allows you to build robust, feature-rich abstractions that can perform complex operations which would take too much time to implement if you were only going to use them once. Beyond all of these incremental advantages, however, it also offers something fundamental. It allows you to take a chunk of code in all of its nitty-gritty, detail-oriented glory, and wrap it up into a human-level concept that can be reused and repurposed throughout your project.

Copyright Material – Provided by Taylor & Francis

It allows you to work closer to the level of granularity at which you naturally think, and at which your designers naturally think, rather than at the level which is natural to the machine. It changes, for example:

```
d = sqrt(pow((a.x - b.x), 2) + pow((a.y - b.y), 2));
```

into:

```
d = Distance(a, b);
```

The challenge with AI, however, is that while there are often aspects of a decision that are similar to other decisions, there are also invariably aspects that are quite different. The AI might measure the distance between two objects both to determine whether to shoot at something and to determine where to eat lunch, but the objects that are evaluated and the way that distance is used in the larger decision is certain to be different (unless you are building a nonplayer character (NPC) that likes to shoot at restaurants and eat its enemies). As a result, while the distance function itself is a standard part of most math libraries, there is a much larger body of code involved in distance-based decisions that is more difficult to encapsulate and reuse.

Modular AI is fundamentally about this transition. It is about enabling you to rapidly specify decision-making logic by plugging together modular components that represent human-level concepts. It is about building up a collection of these modular components, where each component is implemented once but used over and over, throughout the AI for your current game, and on into the AI for your next game and the game after that. It is about enabling you to spend most of your time thinking about human-sized concepts, to build up from individual concepts (e.g., distance, line of sight, moving, firing a weapon) to larger behaviors (taking cover, selecting and engaging a target) to entire performances (ranged weapon combat), and then to reuse those pieces, with appropriate customization, elsewhere. It is an approach that will allow you to create your decision-making logic more quickly, change it more easily, and reuse it more broadly, all while working more reliably and generating fewer bugs because the underlying code is being used more heavily and thus tested more robustly, and also because new capabilities added for one situation immediately become available for use elsewhere as part of the reusable component they improved.

This chapter will first discuss the theoretical underpinnings of modular AI and relate them to broadly accepted concepts from software engineering, and then describe in detail the Game AI Architecture (GAIA). GAIA is a modular architecture, developed at Lockheed Martin Rotary and Mission Systems, that has been used to drive behavior across a number of very different projects in a number of very different game and simulation engines, including (but not limited to) both educational games and training simulations. Its roots go back to work on animal AI at Blue Fang Games, on boss AI for an action game at Mad Doc Software, and on ambient human AI at Rockstar Games.

8.1.1 Working with this Chapter

This chapter goes into great depth, and different readers might be interested in different aspects of the discussion. If your primary interest is in the big ideas behind modular AI and how the modular pieces work together, your focus should be on Sections 8.2, 8.5, and 8.6. If you are interested in an approach that you can take away right now and use in an existing

Copyright Material – Provided by Taylor & Francis

architecture, without starting over from scratch, then you should consider implementing just considerations (Sections 8.5.1 and 8.6). Finally, if you are interested in a full architecture that can be reused across many projects, across many game engines, and which allows you to rapidly configure your AI in a modular way, then the full chapter is for you!

8.2 Theoretical Underpinnings

Modular AI, and modular approaches in general, seek to raise the level of abstraction of development. Rather than focus on algorithms and code, a good modular solution leads to a focus on AI behaviors and how they fit together, abstracting away the implementation details. The question is how this can be done safely and correctly, while still giving designers and developers the fine-grained control needed to elicit the intended behaviors.

Success in modular AI development is driven by the same principles found in good software development: encapsulation, polymorphism, loose coupling, clear operational semantics, and management of complexity. Each of these familiar concepts gains new meaning in a modular context.

Modules themselves encapsulate a unit of AI functionality. Good modules follow the “Goldilocks Rule”: not too big, not too small, but sized just right. Large modules that include multiple capabilities inhibit reuse—what if only part of the functionality is needed for a new AI? Modules that are too small do not do enough to raise the level of abstraction. The goal is to capture AI functionality at the same level that a designer uses to reason about NPCs in your game. Then, the development problem shifts to selecting and integrating the behaviors and capabilities needed for a new NPC, rather than implementing those capabilities from scratch, which is a highly appropriate level of abstraction.

Using modules in this fashion requires that module reuse to be safe. For this, module encapsulation must be strictly enforced. Preventing spaghetti interactions between modules ensures that each module can run correctly in isolation. This is essential for reuse—even subtle dependencies between modules quickly become problematic.

Encapsulation leads naturally to the creation of a module interface. Much like an API, a module interface describes exactly how to interact with that module. It shows the inputs that it can accept, the outputs that it provides, and details the parameters that are exposed for customization of module behavior when applied to a specific AI. With an explicit interface, handling dependencies between behaviors becomes a much simpler problem of connecting the inputs and outputs needed to properly express the behavior. Since each module is properly encapsulated, the results of adding and removing new modules become predictable.

Polymorphism arises as a result of this loose coupling. Imagine a module tasked with fleeing from an enemy. As a bite-sized module, it could perform the checks and tests needed to find an appropriate flee destination, and then send off a move output. The module that receives this output no longer matters. One AI can use a certain type of move module, while a different AI can use another. The exact type of move module should not matter much. Complicating factors, like “is my NPC on a bicycle,” or “is she on a horse,” and so on, can all be handled by the move module, or by other submodules. This keeps each module cleanly focused on a single functional purpose while ensuring that similar behaviors are not repeated across modules.

Copyright Material – Provided by Taylor & Francis

8.3 GAIA Overview

GAIA is a modular, extensible, reusable toolset for specifying procedural decision-making logic (i.e., AI behavior). GAIA emphasizes the role of the game designer in creating decision-making logic, while still allowing the resulting behavior to be flexible and responsive to the moment-to-moment situation in the application.

Taking those points more slowly, GAIA is:

- A library of tools that can be used to specify procedural decision-making logic (or "AI behavior").
- Focused on providing *authorial control*. In other words, the goal of GAIA is not to create a true artificial intelligence that can decide what to do on its own, but rather to provide a human author with the tools to specify decisions that will deliver the intended experience while still remaining flexible enough to handle varied and unexpected situations.
- *Modular*, meaning that behavior is typically constructed by plugging together predefined components. Experience has shown that this approach greatly improves the speed with which behavior can be specified and iterated on.
- *Extensible*, making it easy to add new components to the library, or to change the behavior of an existing component.
- *Reusable*, meaning that GAIA has been designed from the ground up with reuse in mind. This includes reuse of both code and data, and reuse within the current project, across future projects, and even across different game engines.

GAIA is data driven: behavior is specified in XML files and then loaded by the code at runtime. This chapter will generally refer to the XML as the *configuration* or the *data* and the C++ as the *implementation* or the *code*. For simplicity, this chapter will also use the term *NPC* to denote GAIA-controlled entities, *PC* to denote player-controlled entities, and *character* to denote entities that may be either NPCs or PCs.

8.3.1 GAIA Control Flow

GAIA makes decisions by working its way down a tree of decision makers (*reasoners*) that is in many ways similar to Damian Isla's original vision of a Behavior Tree (BT) (Isla 2005). As in a BT, different reasoners can use different approaches to decision-making, which gives the architecture a flexibility that is not possible in more homogenous hierarchical approaches (e.g., hierarchical finite-state machines, teleoreactive programming, hierarchical task network planners, etc.).

Each reasoner picks from among its *options*. The options contain *considerations*, which are used by the reasoner to decide which option to pick, and *actions*, which are executed if the option is selected. Actions can be *concrete*, meaning that they represent things that the controlled character should actually do (e.g., move, shoot, speak a line of dialog, cower in fear, etc.), or *abstract*, meaning that they contain more decision-making logic.

The most common abstract action is the `AIAction_Subreasoner`, which contains another reasoner (with its own options, considerations, and actions). Subreasoner actions are the mechanism GAIA uses to create its hierarchical structure. When an option that contains a subreasoner is selected, that subreasoner will start evaluating its own options

Copyright Material – Provided by Taylor & Francis

and select one to execute. That option may contain concrete actions or may, in turn, contain yet another subreasoner action.

Options can also contain more than one action, which allows them to have multiple concrete actions, subreasoners, or a combination of both, all operating in parallel.

8.3.2 GAIA Implementation Concepts

Reasoners, options, considerations, and actions are all examples of *conceptual abstractions*. Conceptual abstractions are the basic types of objects that make up a modular AI. Each conceptual abstraction has an *interface* that defines it, and a set of *modular components* (or just *components*) that implement that interface. As discussed above, there are multiple different types of reasoners, for example, but all of the reasoners—that is, all of the modular components that implement the reasoner conceptual abstraction—share the same interface. Thus the surrounding code does not need to know what types of components it has. The reasoner, for example, does not need to know what particular types of considerations are being used to evaluate an option, or how those considerations are configured—it only needs to know how to work with the consideration interface in order to get the evaluation that it needs. This is the key idea behind modular AI: identify the basic parts of the AI (*conceptual abstractions*), declare an interface for each abstraction, and then define reusable *modular components* that implement that interface.

Modular components form the core of modular AI reuse—each type of component is implemented once but used many times. To make this work, each type of component needs to know how to load itself from the configuration, so that all of the parameters that define the functionality of a particular instance of a modular component can be defined in data.

Continuing with the distance example from the introduction, GAIA makes distance evaluations reusable by providing the `Distance` consideration. The configuration of a particular `Distance` consideration specifies the positions to measure the distance between, as well as how that distance should be evaluated (Should it prefer closer? Farther? Does it have to be within a particular range?). For example, a sniper selecting a target to shoot at might use a `Distance` consideration to evaluate each potential target. This consideration might be configured to only allow the sniper to shoot at targets that are more than 50 m and less than 500 m away, with a preference for closer targets. This consideration could then be combined with other considerations that measure whether the prospective target is friend or enemy, how much cover the target has, whether it is a high-value target (such as an officer), and so on. What is more, the consideration does not work in isolation—it makes use of other conceptual abstractions in its configuration. For instance, the two positions are specified using *targets*, and the way the distance should be combined with other considerations is specified using a *weight function*. Targets and weight functions are two of the other conceptual abstractions in GAIA.

One advantage of this approach is that it is highly extensible. As development progresses and you discover new factors that should be weighed into a decision, you can create new types of considerations to evaluate those factors and simply drop them in. Because they share the same interface as all the other considerations, nothing else needs to change. Not only does this make iterating on the AI behavior much faster, it also decreases the chance that you will introduce a bug (because all changes are localized to the consideration being added, which can be tested in isolation). Consequently, it is safer to make more aggressive

Copyright Material – Provided by Taylor & Francis

changes later in the development cycle, allowing you to really polish the AI late in development once gameplay has been hammered out and QA is giving real feedback on what you have built.

This ability to rapidly specify and then easily reuse common functionality greatly reduces the amount of time it takes to specify behavior, generally paying back the cost of implementation within weeks. We have used modular AI with great success on several projects where there were only a few months to implement the entire AI—including one game whose AI was implemented in less than 4 months that went on to sell millions of copies.

8.3.3 An Example Character: The Sniper

Throughout this chapter, we will use as our example a sniper character that is based on, but not identical to, a character that was built for a military application. Broadly speaking, the sniper should wait until there are enemies in its kill zone (which happens to be an outdoor marketplace), and then take a shot every minute or two as long as there are still enemies in the marketplace to engage, but only if (a) it is not under attack and (b) it has a clear line of retreat. If it is under attack then it tries to retreat, but if its line of retreat has been blocked then it will start actively fighting back, engaging targets as rapidly as it can (whether they are in the kill zone or not). The overall structure for this configuration is shown in Figure 8.1.

At the top level of its decision hierarchy, our sniper has only four options to choose between: snipe at a target in the kill zone, retreat, fight back when engaged, or hide and wait until one of the other options is available. The decision between these options is fairly cut-and-dried, so a fairly simple reasoner should work. GAIA includes a `RuleBased` reasoner that works much like a selection node in a BT—that is, it simply evaluates its options in the order in which they were specified and takes the first one that is valid given the current situation. In this case, a `RuleBased` reasoner could be set up as follows:

- If the sniper is under fire and its line of retreat is clear, retreat.
- If the sniper is under fire, fight back.

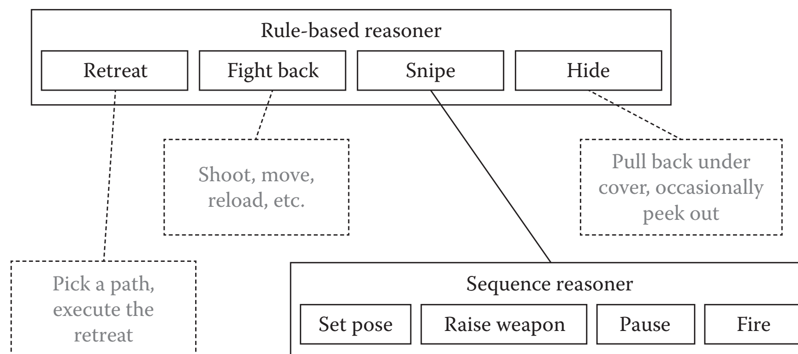


Figure 8.1

Overall structure for a sniper AI. Aspects of the AI that are not fully defined are shown with dotted lines.

Copyright Material – Provided by Taylor & Francis

- If the sniper's line of retreat is clear, there is a valid target in the kill zone, and a minute or two has elapsed since the sniper's last shot, snipe.
- Hide.

None of those options are likely to contain concrete actions. Retreat, for example, will require the AI to pick a route, move along that route, watch for enemies, react to enemies along the way, and so forth. Fight back requires the AI to pick targets, pick positions to fight from, aim and fire its weapon, reload, and so on. Hide requires it to pull back out of sight and then periodically peer out as if checking whether targets have become available. Thus, once the top-level reasoner has selected an option (such as Snipe), that option's subreasoner will evaluate its own options. In Figure 8.1, we see that the Snipe option, for example, uses a `Sequence` reasoner to step through the process of taking a shot by first getting into the appropriate pose (e.g., the "prone" pose), then raising its weapon, pausing (to simulate aiming), and finally firing.

8.4 GAIA Infrastructure

Before delving into the different conceptual abstractions and modular components, it is helpful to have an understanding of the surrounding infrastructure. This section provides an overview of the major singletons, data stores, and other objects which, while not themselves modular, support evaluation of and communication between the modular components.

8.4.1 The `AIString` Class

Strings are tremendously useful, but they also take up an unreasonable amount of space and are slow to compare. Many solutions to this problem exist; GAIA's is to use the djb2 hash function (http://www.cse.yorku.ca/~oz/hash.html) to generate a 64 bit hash for the strings and then to keep a global string table that contains all of the raw strings (as `std::strings`). This lets GAIA do constant time comparisons and store copies of strings in 64 bits. It also lets GAIA downcase the string when it is hashed, so that comparisons are case insensitive (which makes them much more designer friendly). On the other hand, it has an up-front performance cost and makes a permanent copy of every string used (whether the string itself is temporary or not), so GAIA still uses `char*` and `std::string` in places where the `AIString` does not make sense (such as debug output).

Of note, no hash function is a guarantee. If you do take this approach, it is a very good idea to have an assert that checks for hash collisions; simply look in the string table each time you hash a string and ensuring that the stored string is the same as the one you just hashed.

8.4.2 Factories

GAIA uses factories to instantiate all of the modular objects that make up the AI. In other words, the portion of a configuration that defines a consideration, for example, will be contained within a single XML node. GAIA creates the actual C++ consideration object by passing that XML node (along with some supporting information, like a pointer to the NPC that this configuration will control) to the `AIConsiderationFactory`, which instantiates and initializes an object of the appropriate type. GAIA's factory system was the topic of an earlier chapter in this book, so we will not repeat the details here (Dill 2016).

Copyright Material – Provided by Taylor & Francis

8.4.3 The `AIDataStore` Base Class

Data stores are `AIString`-indexed hash tables that can store data of any type. GAIA uses them as the basis for its blackboards and also all of its different types of entities. As a result, individual modular components can store, share, and retrieve information to and from the blackboards and/or entities without the rest of the AI having to know what is stored or even what type the information is. This allows data to be placed in the configuration that will be used by the game engine if an action is executed, for instance, or even for the game engine to pass data through to itself. Of course, it also allows AI components to share data with one another—we will see an example of this in the sniper configuration, below.

There are many ways to implement this sort of hash table, and GAIA's is not particularly special, so we will skip the implementation details. It is worth mentioning, however, that since the data stores are in essence of global memory, they run the risk of name collisions (that is, two different sets of components both trying to store data using the same name). With that said, experience has shown that as long as you have a reasonably descriptive naming convention, this is not normally a problem. Nevertheless, GAIA does have asserts in place to warn if the type of data stored is not the expected type. This will not catch every possible name collision, but it should catch a lot of them.

GAIA currently has two types of blackboards and three types of entities, described as follows.

8.4.3.1 The `AIBlackboard_Global` Data Store

The game has a single *global blackboard*, which can be used as a central repository for information that should be available to every AI component, regardless of what character that component belongs to or what side that character is on.

8.4.3.2 The `AIBlackboard_Brain` Data Store

Every AI-controlled character also has a blackboard built into its brain. The *brain blackboard* allows the components that make up that character's AI to communicate among themselves.

8.4.3.3 The `AIActor` Data Store

Every NPC is represented by an *actor*. The game stores all of the information that the AI will need about the character (e.g., its position, orientation, available weapons, etc.), and AI components can look that information up as needed. The actor also contains an `AIBrain`, which contains the top-level reasoner and all of the decision-making logic for that character.

On some projects, actors are used to represent every character, whether AI controlled or not. In these cases, actors that are not AI controlled may not have a brain or, if they switch back and forth between AI and player control, they will have a brain but it will be disabled when the AI is not in control.

8.4.3.4 The `AIContact` Data Store

As hinted above, there are two ways to keep track of what an NPC knows about the other characters in the game. The first is to use actors to represent every character and give the AI components in each NPC's brain direct access to the actors for other characters.

Copyright Material – Provided by Taylor & Francis

This works, but it either means that all NPCs will have perfect information, or that every AI component has to properly check whether they should know about a particular piece of information or not. Furthermore, even if the AI components make these checks, it still means that everything that they know is correct. This makes it much more difficult to, for example, allow the AI to know that an enemy exists but have an incorrect belief about its location.

The alternative is to have each NPC create a *contact* for every other character that it knows about, and store its knowledge of that NPC on the contact. Thus the contacts represent what the NPC knows about other characters in the game, whether that knowledge is correct or not. For example, imagine an RPG where the player steals a uniform in order to sneak past hostile guards. The guards would each have a contact that stores their knowledge of the player, and if the guards are fooled then that contact would list the player as being on their side even though he or she is actually an enemy. This allows each NPC to have its own beliefs about the characters that it is aware of, but it also means that the AI has to store a lot of redundant copies of the information for each character.

There is no single right answer here, which is why GAIA supports both approaches—the best one is the one that best supports the needs of the individual project. Using actors to represent all characters is simpler and more efficient when there is little uncertainty in the environment (or the behavior of the characters is not greatly affected by it), while using contacts is better when imperfect situational awareness plays an important role in the behavior of the NPCs.

8.4.3.5 The `AIThreat` Data Store

Threats represent things that an NPC knows about and should consider reacting to. They can include enemy characters (whether represented as actors or contacts), but may also include more abstract things such as recent explosions, or locations where bullets have impacted. This enables characters to react to the impact of a sniper's shot even if they do not actually know about the shooter, or to break out of cover based on where rounds are impacting rather than where they are being fired from, for example. Like contacts, threats are not used by every project, and are stored on the brain blackboard.

8.4.4 Singletons

Singletons provide managers that are globally accessible. You can access a singleton from anywhere in the codebase by simply calling the static `Get()` function for that class. You can also replace the default implementation of any singleton with a project-specific version (which must be a subclass) by calling `Set()`.

8.4.4.1 The `AIManager` Singleton

The *AI manager* is responsible for storing all of the actors. It also has an `Update()` function that the game should call every tick in order to tick the actors, and thus their brains.

8.4.4.2 The `AISpecificationManager` and `AIGlobalManager` Singletons

As we have said, GAIA is data driven. All of the decision-making for an NPC is stored in its configuration, in XML. The *specification manager* is responsible for loading, parsing, and storing all of the configurations. Then, when the game creates an NPC's brain, it specifies the name of the configuration that character should use.

Copyright Material – Provided by Taylor & Francis

Duplication can happen in data as well as in code. GAIA partially addresses this by allowing configurations to include *globals*, which are component specifications that can be reused within a configuration or across all configurations. The globals are stored by the *global manager*. Globals were discussed in the earlier chapter on factories (Dill 2016).

8.4.4.3 The `AIOutputManager` Singleton

Good debug output is critical to AI development. GAIA has the usual mix of log messages, warnings, errors, and asserts, along with "status text," which describes the current decision (and is suitable for, for example, displaying in-engine next to the NPC in question). The *output manager* is responsible for handling all of those messages, routing them to the right places, enabling/disabling them, and so forth.

8.4.4.4 The `AITimeManager` Singleton

Different game engines (and different games) handle in-game time in different ways. The *time manager* has a single function (`GetTime()`), and is used throughout the AI to implement things like cooldowns and maximum durations. The built-in implementation just gets system time from the CPU, but most projects implement their own time manager that provides in-game time instead.

8.4.4.5 The `AIBlackboard_Global` Singleton

As described above, the global blackboard is a shared memory space that can be used to pass information between the game and the AI, and/or between AI components. It is a singleton so that it will be globally accessible, and also so that projects can implement their own version which is more tightly coupled with the data being shared from the game engine if they wish.

8.4.4.6 The `AIRandomManager` Singleton

Random numbers are central to many games, both inside and outside of the AI. The *random manager* contains functions for getting random values. The default implementation uses the dual-LCG approach described by Jacopin elsewhere in this volume (Jacopin 2016), but as with other singletons individual projects can replace this with a custom implementation that uses some different RNG implementation if they so desire. For example, we have a unit test project whose RNG always returns 0, making it much easier to write deterministic tests.

8.5 Modular AI: Conceptual Abstractions and Modular Components

Conceptual abstractions define the base class types that the architecture supports. In other words, these abstractions define the interfaces that the rest of GAIA will use. A modular component is the counterpart, with each component providing a concrete implementation for a conceptual abstraction. This approach, where objects interact through well-defined interfaces, allows GAIA to provide an environment that supports loosely coupled modular composition untethered from specific implementations. The developer is free to think about types of abstractions that will produce desired behaviors, and then configure the implementation by reusing and customizing existing modular components, or by creating

Copyright Material – Provided by Taylor & Francis

new components as necessary. This section will describe the major conceptual abstractions used by GAIA, provide examples of their use, and give their interfaces.

8.5.1 Considerations

Considerations are the single most useful conceptual abstraction. If you are uncertain about building a full modular AI, or are just looking for a single trick that you can use to improve your existing architecture, they are the place to start.

Considerations are used to represent each of the different factors that might be weighed together to make a decision. At the core, each type of consideration provides a way to evaluate the suitability of an action with respect to the factor being considered. Listing 8.1 shows the consideration interface in full.

Listing 8.1. The consideration interface.

```
class AIConsiderationBase
{
public:
    // Load the configuration.
    virtual bool Init(const AICreationData& cd) = 0;

    // Called once per decision cycle, allows the
    // consideration to evaluate the situation and determine
    // what to return.
    virtual void Calculate() = 0;

    // These are GAIA's weight values. They return the
    // results computed by Calculate().
    virtual float GetAddend() const;
    virtual float GetMultiplier() const;
    virtual float GetRank() const;

    // Certain considerations need to know if/when they are
    // selected or deselected.
    virtual void Select()   {}
    virtual void Deselect() {}
};
```

To understand how these work in action, let's take a look at the sniper's decision to snipe at an enemy. It will only select this option if:

- The line of retreat is clear.
- There is a target in the kill zone.
- It has been "a minute or two" since the last time the sniper took a shot.

Building the configuration for this option requires three considerations: one for each of the bullet items above. Each one is a modular component that implements the consideration interface.

First, an `EntityExists` consideration is used to check whether there are any enemies in the area that the sniper will retreat through. The `EntityExists` consideration goes through all of the contacts (or all of the actors, or all of the threats, depending on how

Copyright Material – Provided by Taylor & Francis

it is configured) to see whether there is at least one which meets some set of constraints. In this case, the constraints are that the entity must be an enemy and that it must be inside the area that the sniper plans to escape through. That area is defined using a region, which is another conceptual abstraction (described below). This first consideration vetoes the option (i.e., does not allow it to be picked) if there is an enemy blocking the line of retreat, otherwise it allows the option to execute (but the other considerations may still veto it).

Next, the sniper needs to pick a contact to shoot at, and for this a second `EntityExists` consideration is used. The contact must be an enemy and must be in the kill zone. Other constraints can easily be added—for example, the sniper could be configured to prefer contacts that are closer, those that have less cover, and/or those that are high-value targets (such as officers). The consideration is configured to use a picker (discussed in a later section) to select the best target and store it on the brain's blackboard. If the option is selected then the Fire action will retrieve the selected target from the blackboard, rather than going through the process of selecting it all over again. As with the escape route, this consideration will veto the option if no target is found, otherwise it has no effect.

Finally, an `ExecutionHistory` consideration is used to check how long it has been since the sniper last fired a shot. This consideration picks a random delay between 60 and 120 seconds, and vetoes the option if the time since the last shot is less than that delay. Each time the option is selected (i.e., each time the reasoner picks this option and starts executing it) the consideration picks a new delay to use for the next shot.

Considerations are the single most powerful conceptual abstraction, and can be used with or without the other ideas described in this chapter. They are straightforward to implement (the only slightly tricky part is deciding how to combine them together—that topic is discussed in detail later in this chapter), but all by themselves allow you to greatly reduce duplication and increase code reuse. Once you have them, configuring a decision becomes a simple matter of enumerating the options and specifying the considerations for each option. Specifying a consideration is not much more complex than writing a single function call in code—it typically takes anywhere from a few seconds to a minute or two—but each consideration represents dozens, or often even hundreds of lines of code. From time to time you will need to add a new consideration, or add new capabilities to one that exists—but you only need to do that once for each consideration, and then you can use it again and again throughout your AI.

As considerations are heavily reused, they also allow you to take the time to add nuance to the decision-making that might be difficult to incorporate otherwise. `EntityExists` is a good example of how complex—and powerful—considerations can become, but even a very simple consideration like `ExecutionHistory` can make decisions based on how long an option has been executing, how long since it last ran, or whether it has ever run at all. This allows us to implement things like cooldowns, goal inertia (a.k.a. hysteresis), repeat penalties, and one-time bonuses with a single consideration (these concepts were discussed in detail in our previous work [Dill 2006]). It can also support a wide range of evaluation functions that drive decision-making based on that elapsed time—for example, by comparing to a random value (as we do in this example) or applying a response curve (as described in Lewis's chapter on utility function selection and in Mark's book on Behavioral Mathematics [Lewis 2016, Mark 2009]). Having a single consideration that does all of that means you can reuse it in seconds, rather than spending minutes or even hours reimplementing it. It also means that when you reuse it, you can be confident that it will work because it has already been heavily tested and thus is unlikely to contain a bug.

Copyright Material – Provided by Taylor & Francis

One issue not discussed above is how the reasoners actually go about combining considerations in order to evaluate each option. This is a big topic so we will bypass it for now (we devote an entire section to it below) and simply say that considerations return a set of *weight values* which are combined to guide the reasoner's decisions.

8.5.2 Weight Functions

While considerations do a lot to reduce duplication in your code base, there is still a lot of repetition between different types of considerations. Consequently, many of the remaining conceptual abstractions were created in order to allow us to encapsulate duplicate code within the considerations themselves. The first of these is the *weight function*.

Many different types of considerations calculate a floating point value, and then convert that single float into a set of weight values. The weight function abstraction is responsible for making that conversion. For example, the `Distance` consideration calculates the distance between two positions, and then uses a weight function to convert that floating point value into a set of weight values. Some games might use a `Health` consideration, which does the same thing with the NPC's health (or an enemy's health, for that matter). Other games might use an `Ammo` consideration. The `ExecutionHistory` consideration that we used on the sniper is another example. It actually has three weight functions: one to use when the option is selected, one to use if it has never been selected, and one to use if it was previously selected but is not selected right now.

Of course, not all considerations produce a floating point value. The `EntityExists` consideration, for example, produces a Boolean: `TRUE` if it found an entity, `FALSE` if it did not. Different instances of the `EntityExists` consideration might return different weight values for `TRUE` or `FALSE`, however. In the sniper example, one `EntityExists` consideration vetoes the option when an entity was found (the one that checks line of retreat) while the other vetoes the option if one is not found (the one that picks a target to shoot at). This is done by changing the configuration of the weight function that each one uses. Other considerations might also produce Boolean values—for instance, some games might have a `LineOfSight` consideration that is `TRUE` if there is line of sight between two characters, `FALSE` if there is not.

There are a number of different ways that we could convert from an input value to a set of weight values. For floating point numbers, we might apply a response curve (the `BasicCurve` weight function), or we might divide the possible input values into sections and return a different set of weight values for each section (e.g., veto the Snipe option if the range to the enemy is less than 50 m or more than 300 m, but not if it is in between—the `FloatSequence` weight function), or we might simply treat it as a Boolean (the `Boolean` weight function). We might even ignore the input values entirely and always return a fixed result (the `Constant` weight function)—this is often done with the `ExecutionHistory` consideration to ensure that a particular option is only ever selected once or that it gets a fixed bonus if it has never been selected, for example.

The consideration should not have to know which technique is used, so we use a conceptual abstraction in which the conversion is done using a consistent interface and the different approaches are implemented as modular components (the `BasicCurve`, `FloatSequence`, `Boolean`, or `Constant` weight functions, for example). The interface for this conceptual abstraction is given in Listing 8.2.

Copyright Material – Provided by Taylor & Francis

Listing 8.2. The weight function interface.

```
class AIWeightFunctionBase
{
public:
    // Load the configuration.
    virtual bool Init(const AICreationData& cd) = 0;

    // Weight functions can deliver a result based on the
    // input of a bool, int, float, or string. By default
    // int does whatever float does, while the others all
    // throw an assert if not defined in the subclass.
    virtual const AIWeightValues& CalcBool(bool b);
    virtual const AIWeightValues& CalcInt(int i);
    virtual const AIWeightValues& CalcFloat(float f);
    virtual const AIWeightValues& CalcString(AIString s);

    // Some functions need to know when the associated option
    // is selected/deselected (for example, to pick new
    // random values).
    virtual void Select()   {}
    virtual void Deselect() {}
};
```

Coming back to the sniper example, both of the `EntityExists` considerations would use a `Boolean` weight function. The `Boolean` weight function is configured with two sets of weight values: one to return if the input value is `TRUE`, the other if it is `FALSE`. In these two cases, one set of weight values would be configured to veto the option (the `TRUE` value for the escape route check, the `FALSE` value for the target selection), while the other set of weight values would be configured to have no effect on the final decision.

The `ExecutionHistory` consideration is a bit more interesting. It has three weight functions: one to use when the option is executing (which evaluates the amount of time since the option was selected), one to use if the option has never been selected (which evaluates the amount of time since the game was loaded), and one to use if the option has been selected in the past but currently is not selected (which evaluates the amount of time since it last stopped executing). In this instance, when the option is selected (i.e., when the sniper is in the process of taking a shot) we use a `Constant` weight function that is configured to have no effect. We also configure the weight function for when option has never been selected in the same way—the sniper is allowed to take its first shot as soon as it has a target. The third weight function (which is used if the option is not currently selected but has been executed in the past) uses a `FloatSequence` weight function to check whether the input value is greater than our cooldown or not, and returns the appropriate result. This weight function is also configured to randomize the cooldown each time the option is selected.

8.5.3 Reasoners

As discussed in previous sections, *reasoners* implement the conceptual abstraction that is responsible for making decisions. The configuration of each reasoner component will specify the type of reasoner, and also what the reasoner's options are. Each option can contain a set of considerations and a set of actions. The considerations are used by the reasoner to evaluate

Copyright Material – Provided by Taylor & Francis

each option and decide which one to select, and the actions specify what should happen when the associated option is selected. The interface for this abstraction is given in Listing 8.3.

Listing 8.3. The reasoner interface.

```
class AIReasonerBase
{
public:
    // Load the configuration.
    virtual bool Init(const AICreationData& cd);

    // Used by the picker to add/remove options
    void AddOption(AIOptionBase& option);
    void Clear();

    // Enable/Disable the reasoner. Called when containing
    // action is selected or deselected, or when brain is
    // enabled/disabled.
    void Enable();
    void Disable();
    bool IsEnabled() const;

    // Sense, Think, and Act.
    // NOTE: Subclasses should not overload this. Instead,
    // they should overload Think() (ideally they shouldn't
    // have to do anything to Sense() or Act()).
    void Update();

    // Get the current selected option, if any. Used by the
    // picker.
    AIOptionBase* GetSelectedOption();

    // Most reasoners are considered to be done if they don't
    // have a selected option, either because they failed to
    // pick one or because they have no options.
    virtual bool IsDone();

protected:
    void Sense();
    virtual void Think();
    void Act();
};
```

GAIA currently provides four different modular reasoner components:

- The `Sequence` reasoner, which performs its options in the order that they are listed in the configuration (much like a sequence node in a BT). Unlike the other types of reasoners, the sequence reasoner always executes each of its options, so it ignores any considerations that may have been placed on them.
- The `RuleBased` reasoner, which uses the considerations on each option to determine whether the option is valid (i.e., whether it should be executed, given the current situation). Each tick, this reasoner goes down its list of options in the order that they are specified in the configuration and selects the first one that is valid. This is essentially the same approach as that of a selector node in many BT implementations.

Copyright Material – Provided by Taylor & Francis

- The FSM reasoner, which allows us to implement a finite-state machine. For this reasoner, each option contains a list of *transitions*, rather than having considerations. Each transition specifies a set of considerations (which determines whether the transition should be taken), as well as the option (i.e., the state) which should be selected if the transition does fire. The reasoner uses a picker (described in Section 8.7, below) to pick from among the transitions.
- The DualUtility reasoner, which is GAIA's utility-based reasoner. The dual utility reasoner calculates two floating point values: the *rank* and the *weight*. It then uses these two values, along with the random number generator, to select an option. Dual utility reasoning is discussed in our previous work (Dill 2015, Dill et al. 2012) and also in Section 8.6.2.

Of course, a modular architecture does not need to be limited to only these approaches to decision-making. For example, we have often considered implementing a Goal-Oriented Action Planner (GOAP) reasoner (for those cases when we want to search for sequences of actions that meet some goal). Like the FSM reasoner, this would require a bit of clever thinking but should be quite possible to fit into GAIA by implementing a new type of reasoner component.

8.5.4 Actions

Actions are the output of the reasoner—they are responsible for sending commands back to the game, making changes to the blackboard, or whatever else it is that the reasoner has decided to do. Their interface is given in Listing 8.4.

Listing 8.4. The action interface.

```
class AIActionBase
{
public:
    // Load the configuration.
    virtual bool Init(const AICreationData& cd) = 0;

    // Called when the action starts/stops execution.
    virtual void Select()   {}
    virtual void Deselect() {}

    // Called every frame while the action is selected.
    virtual void Update()   {}

    // Check whether this action is finished executing. Some
    // actions (such as a looping animation) are never done,
    // but others (such as moving to a position) can be
    // completed.
    virtual bool IsDone()   { return true; }
};
```

As discussed above, actions can either be abstract or concrete. Abstract actions are actions which exist to guide the decision-making process, like the subreasoner action. Other abstract actions include the Pause and SetVariable actions. The Pause action delays a specified amount of time before marking itself as complete. It is commonly used in

Copyright Material – Provided by Taylor & Francis

the Sequence reasoner, to control the timing of the concrete actions. The SetVariable action is used to set a variable on a data store (most often the brain's blackboard).

Concrete actions, by their very nature, cannot be implemented as part of the GAIA library. They contain the game-specific code that is used to make NPCs do things. Common concrete actions include things like Move, PlayAnimation, PlaySound, FireWeapon, and so on. Our factory system handles the task of allowing the developer to inject game-specific code into the AI (Dill 2016).

8.5.5 Targets

Targets provide an abstract way for component configurations to specify positions and/or entities. For example, the Distance consideration measures the distance between two positions. In order to make this consideration reusable, GAIA needs some way to specify, in the configuration, what those two positions should be. Perhaps one is the position of the NPC and the other is the player. Perhaps one is an enemy and the other is an objective that the NPC has been assigned to protect (in a "capture the flag" style game, for instance). Ideally, the distance consideration should not have to know how the points it is measuring between are calculated—it should just have some mechanism to get the two positions, and then it can perform the calculation from there. Similarly, the LineOfSight consideration needs to know what positions or entities to check line of sight between, the Move action needs to know where to move to, the FireWeapon action needs to know what to shoot at, and so on.

GAIA's solution to this is the *target* conceptual abstraction, whose interface is shown in Listing 8.5. Targets provide a position and/or an entity for other components to use. For example, the Self target returns the actor and position for the NPC that the AI controls. The ByName target looks up a character by name (either from the actors or the contacts, depending on how it is configured).

Listing 8.5. The target interface.

```
class AITargetBase
{
public:
    // Load the configuration.
    virtual bool Init(const AICreationData& cd) = 0;

    // Get the target's position. If the target has an
    // entity, it should generally be that entity's
    // position.
    virtual const AIVectorBase* GetPosition() const = 0;

    // Not all types of targets have entities. If this one
    // does, get it. NOTE: It's possible for HasEntity() to
    // return true (i.e. this type of target has an entity)
    // but GetEntity() to return NULL (i.e. the entity that
    // this target represents doesn't currently exist). In
    // that case, HasEntity() should return true, but
    // IsValid() should return false.
    virtual AIEntityInfo* GetEntity() const { return NULL; }
    virtual bool HasEntity() const          { return false; }
```

(*Continued*)

Copyright Material – Provided by Taylor & Francis

```
    // Checks whether the target is valid. For instance, a
    // target that tracks a particular contact by name might
    // become invalid if we don't have contact with that
    // name. Most target types are always valid.
    virtual bool IsValid() const            { return true; }
};
```

All targets can provide a position, but some do not provide an entity. For example, the `Position` target returns a fixed (`x`, `y`, `z`) position (which is specified in the target's configuration), but does not return an entity. The person writing the configuration should be aware of this and make sure not to use a target that does not provide an entity in situations where an entity is needed, but GAIA also has checks in place to ensure that this is the case. In practice this is really never an issue—it simply would not make sense to use a `Position` target in a situation where an entity is needed, so why would a developer ever do that?

As with all conceptual abstractions, some types of targets can be implemented in GAIA, while others need to be implemented by the game. For example, some games might add a `Player` target which returns the contact (or actor) for the PC. For other games (such as multiplayer games, or games where the player is not embodied in the world) this type of target would make no sense.

8.5.6 Regions

Regions are similar to targets, except that instead of specifying a single (`x`, `y`, `z`) position in space, they specify a larger area. They are commonly used for things like triggers and spawn areas, although they have a myriad of other uses. The sniper configuration, for example, would use them to specify both the kill zone (the area it should fire into) and the line of retreat (the area it plans to move through in order to get away).

Regions are a conceptual abstraction because it is useful to supply the AI designer with a variety of ways to specify them. Implementations might include a circular region (specified as a center position—or a target—and a radius), a parallelogram region (specified as a base position and two vectors to give the length and angle of the sides), and a polygon region (specified as a sequence of vertices). Similarly, some games will be perfectly happy with simple 2D regions, while others will need to specify an area in all three dimensions. The interface for this abstraction is given in Listing 8.6.

Listing 8.6. The region interface.

```
class AIRegionBase
{
public:
    // Load the configuration.
    virtual bool Init(const AICreationData& cd) = 0;

    // Test if a specified position is within the region
    virtual bool IsInRegion(const AIVector& pos) const = 0;
```

(*Continued*)

Copyright Material – Provided by Taylor & Francis

```
    // Set the outVal parameter to a random position within
    // the region
    // NOTE: IT MAY BE POSSIBLE FOR THIS TO FAIL on some
    // types of regions. It returns success.
    virtual bool GetRandomPos(AIVector& outVal) const = 0;
};
```

8.5.7 Other Conceptual Abstractions

Conceptual abstractions provide a powerful mechanism that allows us to encapsulate and reuse code which otherwise would have to be duplicated. The abstractions discussed above are the most commonly used (and most interesting), but GAIA includes a few others, including:

- Sensors, which provide one mechanism to pass data into the AI (though most projects simply write to the data stores directly).
- Execution filters, which can control how often reasoners and/or sensors tick.
- Entity filters, which are an alternative to pickers for selecting an entity that meets some set of constraints.
- Data elements, which encapsulate the things stored in data stores.
- Vectors, which abstract away the implementation details of how a particular game or simulation represents positions (it turns out that not every project uses (`x`, `y`, `z`)).

Furthermore, as GAIA continues to improve, from time to time new conceptual abstractions are found and added (vectors are the most recent example of this). GAIA includes a system of macros and templatized classes that allow us to create most of the infrastructure for each conceptual abstraction, including both their factory and the storage for any global configurations, by calling a single macro and passing in the name of the abstraction (Dill 2016).

8.6 Combining Considerations

Considerations are the single most important type of modular component. They are, in many ways, the key decomposition around which GAIA revolves. In general terms, they are the bite-sized pieces out of which decision-making logic is built. They represent concepts like the distance between two targets, the amount of health a target has left, or how long it is been since a particular option was last selected. Reasoners use the considerations to evaluate each option and select the one that they will execute—but how should reasoners combine the outputs of their considerations?

Over the years we have tried a number of different solutions to this problem. Some were quite simple, others were more complex. This chapter will present one from each end of the spectrum: a very simple Boolean approach that was used for a trigger system in an experimental educational game (Dill and Graham 2016, Dill et al. 2015) and a more complex utility-based approach that combines three values to perform the option's evaluation (Dill 2015, Dill et al. 2012). While the latter approach might initially seem too hard to work with, experience has shown that it is both extremely flexible and, once the basic conventions are understood, straightforward to use for both simple and complex decisions.

Copyright Material – Provided by Taylor & Francis

8.6.1 Simple Boolean Considerations

The simplest way to combine considerations is to treat them as Booleans. Each option is given a single consideration, which either returns `TRUE` (the option can be selected) or `FALSE` (it cannot). Logical operations such as `AND`, `OR`, and `NOT` can be treated as regular considerations, except that they contain one or more child considerations and return the combined evaluation of their children. Thus an option's single consideration will often be an `AND` or an `OR` which contains a list of additional considerations (some of which may, themselves, be Boolean operations).

This approach was used for the trigger system in *The Mars Game*, which was an experimental educational game, set on Mars, that taught topics drawn from ninth and tenth grade math and programming. An example of a *Mars Game* trigger is shown in Listing 8.7 (specified in YAML). This particular trigger waits 15 seconds after the start of the level, and then plays a line of dialog that gives a hint about how to solve a particular challenge in the game, and also writes a value on the blackboard indicating that the hint has been played. However, it only plays the hint if:

- The hint has not already been played during a previous level (according to that value on the blackboard).
- The player has not already started executing a Blockly program on their rover.
- The player's rover is facing either south or west (i.e., 180° or 270°), since the hint describes how to handle the situation where you start out facing the wrong way.

Listing 8.7. Trigger a hint.

```
playHint_2_9_tricky:
  triggerCondition:
  - and:
    - delay:                    # Wait 15 seconds after the
      - 15                      #   start of the level.
    - not:
      - readBlackboard:         # Check the blackboard and
        - thisOneIsTrickyHint   #   only play it once.
    - not:                      # Don't play it if the player
      - isBlocklyExecuting:     #   has already started their
        - rover                 #   program.
    - or:
      - hasHeading:             # Only play it if the rover
        - rover                 #   is facing south or west.
        - 180
      - hasHeading:
        - rover
        - 270
  actions:
  - playSound:                  # Play the hint dialog.
    - ALVO37_Rover
  - writeToBlackboard:          # Update the blackboard so
      - thisOneIsTrickyHint     #   that it won't play again.
```

Copyright Material – Provided by Taylor & Francis

This approach has the obvious advantage of great simplicity. Most developers—even game designers—are comfortable with Boolean logic, so it is not only straightforward to implement but also straightforward to use. It works quite well for things like trigger systems and rule-based reasoners that make decisions about each option in isolation, without ever needing to compare two options together to decide which is best. It suffers greatly, however, if there is ever a case where you do want to make more nuanced decisions—and those cases often pop up late in a project, when the designer (or QA, or the publisher, or the company owner) comes to you to say "what it does is mostly great, but in this one situation I would like it to…"

With that in mind, most projects will be best served by an approach that allows Boolean decisions to be specified in a simple way, but also supports complex comparisons when and where they are needed—which brings us to dual utility considerations.

8.6.2 Dual Utility Considerations

Dual utility considerations are the approach used by GAIA. Each consideration returns three values: an *addend*, a *multiplier*, and a *rank*. These three values are then combined to create the overall *weight* and *rank* of the option, which are the two utility values that give this approach its name.

8.6.2.1 Calculating Weight and Rank

Taking those steps one at a time, the first thing that happens is that the addends and multipliers are combined into an overall weight for the option (W_O). This is done by first adding all of the addends together, and then multiplying the result by all of the multipliers.

$$W_O = \left(\sum_{i=1}^{n} A_i\right) \cdot \left(\prod_{i=1}^{n} M_i\right) \tag{8.1}$$

Next, the option's overall rank (R_O) is calculated. This is done by taking the max of the ranks of the considerations.

$$R_O = \text{Max}_{i=1}^{n}\left(R_i\right) \tag{8.2}$$

There are other formulas that could be used to calculate weight and rank, and GAIA does support some alternatives (more on this in a later section), but the vast majority of the time these two formulas are the ones that we use.

8.6.2.2 Selecting an Option

Once the weight and rank have been calculated, the reasoner needs to use them to select an option. Exactly how this is done depends on the type of the reasoner, but all are based on the dual utility reasoner.

The idea behind dual utility reasoning is that the AI will use the rank to divide the options into categories, and then use weight-based random to pick from among the options in the highest ranked category. In reality, there are actually four steps to accomplish this:

1. Eliminate any options that have $W_O \le 0$. They cannot be selected in step 4 and will make step 2 more complicated, so it is best to eliminate them up front.

Copyright Material – Provided by Taylor & Francis

2. Find the highest *rank* from among the options that remain, and eliminate any option with a rank lower than that. This step ensures that only options from the highest ranked category are considered.
3. Find the highest *weight* from among the options that remain, and eliminate options whose weight is "much less than" that weight. "Much less than" is defined as a percentage that is specified in the reasoner's configuration—and in many cases the reasoner is configured to skip this step entirely. This step makes it possible to ensure that the weight-based random will not pick a very low weight option when much better options exist, because doing so often looks stupid—the option was technically possible, but not very sensible given the other choices available.
4. Use weight-based random to select from among the options that remain.

A couple things are worth calling out. First, notice step 1. Any option can be eliminated simply by setting its weight to 0, no matter what the weights and ranks of the other options are. What is more, looking back at Equation 8.1, any consideration can force the weight of an option to 0 (i.e., *veto* it) by returning a multiplier of 0, no matter what the values on the other considerations. Anything times 0 is 0. This provides a straightforward way to treat dual utility options as if they had purely Boolean considerations when we want to. We say that an option is *valid* (which is to say that it is selectable) if it has $W_O > 0$ and *invalid* if it does not. The rule-based reasoner works by checking its options in order, and selecting the first valid one, regardless of rank.

8.6.2.3 Configuring Dual Utility Considerations

The key to implementing dual utility considerations is to provide default values that ensure that even though the system is capable of considerable complexity, the complexity is hidden when configuring a consideration unless and until it is needed. This section will discuss the default values, naming conventions, and other tricks that GAIA uses to accomplish this. Along the way, it will give examples that might be used by our sniper AI to pick a target.

In GAIA, the most basic way to specify weight values is to simply specify the `addend`, `multiplier`, and/or `rank` as attributes in the XML. Any of the three values that are not specified will be set to a default value that has no effect (i.e., an `addend` of 0, a `multiplier` of `1`, and a `rank` of `–FLT_MAX`, which is the smallest possible floating point value). Thus the developer who is configuring the AI only needs to specify the values that he or she wants to change.

As an example, a good sniper should prefer to shoot at officers. In order to implement this, the game can place a Boolean "IsOfficer" value on each contact (remember that contacts are data stores, so we can store any value that we want there). This value would be true if the NPC believes that contact to be an officer (whether or not the belief is true), false otherwise. Then, in the configuration, we use a `BooleanVariable` consideration to look up this value from the `PickerEntity` target (the picker entity is the entity that we are considering picking). The consideration uses a `Boolean` weight function to set the multiplier to 10 if the value is true, otherwise it does nothing (i.e., returns default values). Assuming that there are about 10 enlisted enemies (each with a weight of roughly 1) per officer (with a weight of roughly 10) this means that, all other things being equal, the sniper will shoot at an officer about half of the time. This consideration's configuration is shown in Listing 8.8.

Copyright Material – Provided by Taylor & Francis

Listing 8.8. A consideration that prefers to pick officers.

```
<Consideration Type="BooleanVariable"
               Variable="IsOfficer"
               DataStore="Target">
  <DataStoreTarget Type="PickerEntity"/>
  <WeightFunction Type="Boolean">
    <TrueWeights Multiplier="10"/>
  </WeightFunction>
</Consideration>
```

In some cases, a consideration wants to prevent its option from being selected no matter what the other considerations say. For example, when the sniper is picking its target we might want to ensure that it only shoots at entities that it thinks are enemies. This could be implemented by storing the "Side" of each contact as an `AIString`, with possible values of "Friendly," "Enemy," or "Civilian." If the "Side" is not "Enemy," then the sniper should not select this target no matter where it is or whether it is an officer or not. This could be configured by specifying a `multiplier` of 0, but configurations should be more explicit and easier to read. With this in mind, rather than specifying an `addend`, `multiplier`, and `rank`, weights can specify a Boolean `veto` attribute. If `veto` is true then, under the covers, the `multiplier` will be set to 0. If it is false, then the default values will be used for all three weight values.

The resulting consideration for the sniper would look like Listing 8.9. This consideration is much like the one in Listing 8.8, except that it looks up a string variable rather than a Boolean one, and passes the result into a `String` weight function. The string weight function tries to match the string against each of its entries. If the string does not match any of the entries, then it returns the default values. In this case, that means that if the string is "Enemy," then the consideration will have no effect (because when `veto` is false it returns the default values), otherwise it will set the `multiplier` to 0 (because `veto` is `TRUE`) and thus make the option invalid.

Listing 8.9. A consideration that vetoes everything other than enemies.

```
<Consideration Type="StringVariable"
               Variable="Side"
               DataStore="Target">
  <DataStoreTarget Type="PickerEntity"/>
  <WeightFunction Type="String">
    <Entries>
      <String Value="Enemy" Veto="False"/>
    </Entries>
    <Default Veto="True"/>
  </WeightFunction>
</Consideration>
```

As an aside, the considerations in Listings 8.8 and 8.9 do a nice job of showing exactly why modular AI is so powerful. These considerations evaluate the value from a variable on a data store. It could be any variable on any data store. In this particular case the data store is specified using a target (rather than being, say, the NPC's actor or the brain's blackboard),

Copyright Material – Provided by Taylor & Francis

which again could be any type of target that specifies an entity. Once the consideration has looked up the value for the variable, it passes that value to a weight function to be converted into weight values. Without the ideas of considerations, and data stores, and weight functions, we would have to write a specialized chunk of code for each of these checks that is only used inside of the sniper's target selection, and is duplicated anywhere else that a Boolean data store variable is used. Furthermore, that code would be dozens of lines of C++ code, not a handful of lines of XML. Most importantly, though, the values being specified in the XML are for the most part the sorts of human concepts that we would use when describing the logic to a coworker or friend. What should the AI evaluate? The target that it is considering shooting (the `PickerEntity` target). How should it evaluate that target? By checking whether it is an enemy, and whether it is an officer. What should it do with this evaluation? Only shoot at enemies, and pick out the enemy officers about half the time.

There is one other detail to configuring considerations that has not been discussed yet. In order to be selected, every option needs to have a weight that is greater than 0, but the default addend for all of the considerations is 0. If we do not have at least one consideration with an addend greater than 0 then the overall weight is guaranteed to be 0 for the same reason it is when we set the multiplier to 0—anything times 0 is 0. Furthermore, we would like the default weight for all options to be something reasonable, like 1.

We address this problem with the `Tuning` consideration, which is a consideration that simply returns a specified `addend`, `multiplier`, and `rank`, and which has a default `addend` of 1. The option's configuration can (and often does) specify a `Tuning` consideration, but if it does not then a default `Tuning` consideration with an addend of 1 will automatically be added.

8.6.2.4 Changing Combination Techniques at Runtime

Up until now, we have said that the option owns the considerations, and is responsible for combining them together for the reasoners. This is actually slightly inaccurate. The option has an `AIConsiderationSet`, which in turn contains the considerations. The consideration set is responsible for combining its considerations and returning the overall weight and rank, and it can also return the combined addend and multiplier for its considerations without multiplying them into an overall weight. Its interface is shown in Listing 8.10. This distinction is important, because it means that we can place flags on the consideration set to specify that the considerations in that particular set should be combined with different rules. What is more, there is a special type of consideration that contains another consideration set (called the `AIConsideration_ConsiderationSet`). This makes it possible to have different rules for some of the considerations on an option than for the others.

The most commonly used alternate approaches for combining considerations are ones that apply different Boolean operations to the weights. By default, if any consideration vetoes an option (i.e., returns a multiplier of 0) then that option will not be selected. This is in essence of a conjunction (i.e., a logical `AND`)—all of the considerations have to be "true" (i.e., have multiplier greater than 0) in order for the option to be "true" (i.e., valid). In some cases, rather than an `AND`, we want a logical `OR`—that is, we want the option to be valid as long as at least one consideration does not have a multiplier of 0. This is implemented by having the consideration set ignore any consideration with a multiplier less than or equal to 0, unless every consideration has a multiplier that is less than or equal to 0. Similarly, `NOT` is implemented by having the consideration replace any multiplier that is less than or equal to 0 with 1, and any multiplier that is greater than 0 with 0.

Copyright Material – Provided by Taylor & Francis

Listing 8.10. The `AIConsiderationSet` interface.

```
class AIConsiderationSet
{
public:
    bool Init(const AICreationData& cd);

    // Evaluate all of the considerations and calculate the
    // overall addend, multiplier, weight, and rank.
    void Calculate();

    // Sets the best rank and weight currently under
    // consideration. These don't change the calculated
    // values, but they will change the values returned by
    // GetRank() and GetWeight().
    void SetScreeningWeight(float bestWeight);
    void SetScreeningRank(float bestRank);

    // Get the rank and weight. GetWeight() returns 0 if
    // the screening rank or screening weight checks fail.
    float GetWeight() const;
    float GetRank() const;

    // Get the raw values, unscreened.
    float GetAddend() const;
    float GetMultiplier() const;
    float GetWeightUnscreened() const;
    float GetRankUnscreened() const;
};
```

GAIA also supports different approaches for combining the ranks: rather than taking the max, it can take the min or add all of the considerations' ranks together to get the overall rank. All of these changes are configured just like everything else—which is to say that there is an attribute that tells the consideration set which calculation method to use, and the defaults (when the attribute is not specified) are to use the standard approaches.

More techniques for configuring dual utility considerations and for working with utility in general can be found in Dill (2006), Dill et al. (2012), Lewis (2016), and Mark (2009).

8.7 Pickers

The last topic that we will cover in this chapter is *pickers*. Pickers use a reasoner to go through a list of things (typically either the list of contacts, actors, or threats, but it could also be the list of transitions on an FSM option) and select the best one for some purpose (e.g., the best one to talk to, the best on to shoot at, the best one to use for cover, etc.). There are slight differences between the picker used by the `EntityExists` consideration (which picks an entity) and the one used by an FSM reasoner (which picks a transition), but the core ideas are the same; in the interests of brevity, we will focus on picking entities.

While most reasoners have all of their options defined in their configuration, a picker's reasoner has to pick from among choices that are determined at runtime. For example,

Copyright Material – Provided by Taylor & Francis

we might use a picker to look through our contacts to pick something to shoot at, or to look through our threats to pick one to react to, or to look through nearby cover positions to pick one to use (although the game would have to extend GAIA with support for cover positions to do this last one). The `EntityExists` considerations in our sniper example use pickers to pick something to shoot at, and also to check for an entity that is blocking its line of retreat. The first picker should use a dual utility reasoner, because it wants to pick the best entity. The second one might use a rule-based reasoner, because it just needs to know whether such an entity exists or not.

Picker options are created on-the-fly by taking all of the entities in some category (for instance all of the actors, or all of the contacts, or all of the threats), and creating an option for each one. Each of these options is given the same considerations, which are specified in the configuration. The considerations can access the entity that they are responsible for evaluating by using the `PickerEntity` target. For example, the picker that is used to pick a sniper's target might have considerations that check things like the distance to the target, whether it is in the kill zone, how much cover it has, whether it is an enemy, whether it is an officer, and so forth. The picker that checks line of retreat would simply check whether each entity is an enemy, and whether it is in the region that defines the escape route.

Putting everything together, a simple option for the sniper that considers taking a shot might look like Listing 8.11. This option uses an `EntityExists` consideration to pick a target, and then stores the selected target in the `SnipeTarget` variable on the brain's blackboard. The picker has two considerations—one to check that the target is an enemy, and the other to check that it is between 50 m and 300 m away. It uses a `Boolean` weight function to veto the option if a target is not found. If a target is found, it uses the `Fire` action to take a shot. In reality, we would probably want to add more considerations to the picker (to make target selection more intelligent), but the key ideas are shown here.

Listing 8.11. An option for the sniper, which picks a target and shoots at it.

```
<Option Type="ConsiderationAndAction">
  <Considerations>
    <!-- Look through the contacts, pick a target, store it
         on the brain's blackboard as SnipeTarget. -->
    <Consideration Type="EntityExists"
                   Location="Contacts"
                   Variable="SnipeTarget">

      <Picker>
        <!-- This picker uses a dual utility reasoner because
             it wants to pick the *best* target. A picker
             that just wants to check whether any entity
             meets some set of constraints (like the one for
             checking line of retreat) would likely use a
             rule-based reasoner instead. -->
        <Reasoner Type="DualUtility"/>
```

(Continued)

Copyright Material – Provided by Taylor & Francis

```
        <Considerations>
          <!-- Only targets between 50m and 300m away -->
          <Consideration Type="Distance">
            <FromTarget Type="Self"/>
            <ToTarget Type="PickerEntity"/>
            <WeightFunction Type="FloatSequence">
              <Entries>
                <Entry Exact="50" Veto="true"/>
                <Entry Exact="300" Veto="false"/>
              </Entries>
              <Default Veto="true"/>
            </WeightFunction>
          </Consideration>

          <!-- Only enemies -->
          <Consideration Type="StringVariable"
                         Variable="Side"
                         DataStore="Target">
            <DataStoreTarget Type="PickerEntity"/>
            <WeightFunction Type="String">
              <Entries>
                <String Value="Enemy" Veto="False"/>
              </Entries>
              <Default Veto="True"/>
            </WeightFunction>
          </Consideration>

          <!-- Other considerations (like the one to prefer
               officers, or one to check that the target is
               in the kill zone) could be added here. -->

        </Considerations>
      </Picker>

      <!-- Use a default Boolean weight function – that is,
           veto if a target is not found -->
      <WeightFunction Type="Boolean"/>
    </Consideration>

    <!-- The considerations to check line of retreat and time
         since the last shot would go here. -->
    ...

  </Considerations>

  <Actions>
    <!-- Fire at the target the picker picked -->
    <Action Type="Fire">
      <Target Type="DataElement_EntityList"
              Variable="SnipeTarget"/>
    </Action>
  </Actions>
</Option>
```

Copyright Material – Provided by Taylor & Francis

8.8 Conclusion

Modular AI is an approach to AI specification that draws heavily on principles from software engineering to dramatically reduce code duplication and increase reuse. It allows the developer to rapidly specify decision-making logic by plugging together modular components that represent human-level concepts, rather than by implementing code in C++. Because these components are implemented once and then widely reused, they become both more robust (i.e., more heavily tested) and more feature laden (i.e., capable of more subtle nuance) than would be feasible if each component were only ever used once. What's more, because most of the work consists of invoking code that has already been written, AI specification can be done much, much faster than would otherwise be possible. Modular AI has been used with success on several projects that were only a couple months long, including one game that sold over 5,000,000 copies in which we implemented all of the boss AI, from scratch (including implementing the architecture), in less than 4 months.

This chapter presented a full modular architecture (GAIA), which uses a variety of different types of modular components. Of all of those conceptual abstractions, considerations are by far the most powerful. For those who are constrained to work within an existing architecture, it is very possible to get much of the benefit of modular AI even within an existing architecture, simply by implementing considerations and allowing them to drive your evaluation functions. We took this approach on another best-selling game with great success.

References

Dill, K. 2006. Prioritizing actions in a goal based RTS AI. In *AI Game Programming Wisdom 3*, ed. S. Rabin. Boston, MA: Charles River Media, pp. 321–330.

Dill, K. 2015. Dual utility reasoning. In *Game AI Pro 2*, ed. S. Rabin. Boca Raton, FL: CRC Press, pp. 23–26.

Dill, K. 2016. Six factory system tricks for extensibility and library reuse. In *Game AI Pro 3*, ed. S. Rabin. Boca Raton, FL: CRC Press, pp. 49–62.

Dill, K., B. Freeman, S. Frazier, and J. Benito. 2015. Mars game: Creating and evaluating an engaging educational game. *Proceedings of the 2015 Interservice/Industry Training, Simulation & Education Conference*, December 2015, Orlando, FL.

Dill, K. and R. Graham. 2016. Quick and dirty: 2 lightweight AI architectures. *Game Developer's Conference,* March 2016, San Francisco, CA.

Dill, K., E.R. Pursel, P. Garrity, and G. Fragomeni. 2012. Design patterns for the configuration of utility-based AI. *Proceedings of the 2012 Interservice/Industry Training, Simulation & Education Conference*, December 2012, Orlando, FL.

Isla, D. 2005. Handling complexity in Halo 2 AI. http://www.gamasutra.com/view/feature/130663/gdc_2005_proceeding_handling_.php (accessed June 26, 2016).

Jacopin, É. 2016. Vintage random number generators. In *Game AI Pro 3*, ed. S. Rabin. Boca Raton, FL: CRC Press, pp. 471–478.

Lewis, M. 2016. Choosing effective utility-based considerations. In *Game AI Pro 3*, ed. S. Rabin. Boca Raton, FL: CRC Press, pp. 167–178.

Mark, D. 2009. *Behavioral Mathematics for Game AI*. Boston, MA: Charles River Media.