5 Six Factory System Tricks for Extensibility and Library Reuse

Kevin Dill

5.1 Introduction
5.2 Background
5.3 Trick #1: Abstracting the Data Format
5.4 Trick #2: Encapsulating the Initialization Inputs
5.5 Trick #3: Consistent Object Construction and Initialization
5.6 Trick #4: Injecting External Code into the AI
5.7 Trick #5: Using Templates and Macros to Standardize Factory Definitions
5.8 Trick #6: Global Object Configurations
5.9 Conclusion
References

5.1 Introduction

These days, many games employ a data-driven AI. In other words, the decision-making logic for the AI is not hardcoded in C++ but instead is placed in a configuration file (generally using XML or some similar format) and loaded at runtime. This file generally specifies the configuration of a variety of different types of polymorphic objects.

The standard solution for instantiating polymorphic objects while maintaining loose coupling between the object's implementation and its owner is the *factory* pattern (Gamma et al. 1995). Not all factory implementations are equal, however. This chapter presents tricks and lessons learned from the factory system for the Game AI Architecture (GAIA), with a strong emphasis on providing actual implementation details that you can use. GAIA is discussed in detail in a later chapter of this book (Dill 2016); this chapter focuses specifically on its factories and on aspects of those factories that are worthy of reuse. Specific topics include:

- A brief overview of both the factory pattern and the design requirements that drove our implementation.
- Trick #1: Abstracting the data format.
- Trick #2: Encapsulating the information objects will need to initialize themselves.
- Trick #3: A consistent approach for object construction and initialization.
- Trick #4: The injection of code from outside of the AI library.
- Trick #5: The use of templates and macros to standardize factory definitions.
- Trick #6: Using global object configurations to remove duplication from the data.

After implementing the ideas contained here, you will have a factory system that:

- To the greatest extent possible, eliminates code duplication. This greatly simplifies debugging and maintenance, as well as reducing the chance that a bug will be introduced in the first place.
- Makes it easy to add new factories (one line of code) or to add new objects to an existing factory (two lines of code).
- Provides a powerful, consistent, easy-to-use mechanism for instantiating and initializing the objects being constructed.
- Is decoupled from the data format, so you can switch formats without rewriting anything else.
- Allows code from external libraries (such as the game engine) to be injected into the AI library without creating dependencies on the external library (critical for AI library reuse).
- Enables reuse within the configuration (i.e., the XML).

5.2 Background

GAIA is a modular AI architecture that is designed to be (a) extensible, meaning that as new capabilities are needed, they can easily be added and (b) reusable, meaning that the architecture can be quickly and easily ported from project to project, including across different game engines (and even to things that are not game engines, like high fidelity military simulations). Both of these requirements have implications for the factory implementation. In order to understand them, however, we first need to understand what factories are, what conceptual abstractions and modular objects are, and how they interact.

5.2.1 The Factory Pattern

The factory pattern gives us a way to instantiate polymorphic objects from data without exposing the type of the object outside of the factory itself. To break that down, imagine that your configuration contains regions that are used for triggers, spawn volumes, and other similar purposes. Your designers want more than one way to define a region, however, so they have you implement a "`circle`" region (takes a center point and radius), a "`rectangle`" region (takes two sides of an axis-aligned rectangle), and a "`polygon`" region (takes an arbitrary sequence of vertices and creates a closed polygon from them).

In data, each type of region needs to be defined differently, because they require different arguments. In code, however, we do not want to have special case logic for each type of region in each place they are used. Not only would this result in a huge amount of duplicate code, but it would be a complete nightmare if the designers requested a new type of region late in the project! Instead, we define a base class for all of our regions that provides the interface for all types of regions (with pure virtual functions like `GetRandomPoint()` and `IsInRegion()`). Code which uses regions can simply interact with that interface. Thus, for example, each nonplayer character (NPC) might have a spawner, defined in data, which specifies the region where the NPC should spawn. That spawner could use any type of region, but the spawner code works with the base class (i.e., the interface) so that it does not have to know or care which specific type of region a particular NPC is using.

This still leaves a problem, however. All of the places in code that use regions need to be able to create a region object of the appropriate subclass (rectangle, circle, etc.), as specified in the data, but we do not want them to have to know the particular type of region that is being created, nor do we want to have to duplicate the code for creating and loading in a region of the correct type. We solve this by creating a *region factory*, which is responsible for looking at the data, determining the type of region, creating an object of the appropriate subclass (a.k.a. *instantiating* it), initializing that object using the values stored in the data (the center and radius for the circle, the two sides for the rectangle, etc.), and then returning the object using the interface's type. Internally, factories are typically just giant `if-then-else` statements, so a naïve implementation of the region factory might look like Listing 5.1.

Listing 5.1. A naïve implementation of a region factory.

```
AIRegionBase*
AIRegionFactory::Create(const TiXmlElement* pElement) {
    // Get the type of region we want to create
    std::string nodeType = pElement->Attribute("Type");

    // Create a region of the specified type
    AIRegionBase* pRegion = NULL;
    if (nodeType == "Circle") {
       pRegion = new AIRegion_Circle(pElement);
    } else if (nodeType == "Rectangle") {
       pRegion = new AIRegion_Rect(pElement);
    } else if (nodeType == "Polygon") {
       pRegion = new AIRegion_Poly(pElement);
    }

    return pRegion;
}
```

5.2.2 Conceptual Abstractions and Modular Components

GAIA defines *conceptual abstractions*, which represent the different fundamental types of objects that make up our AI, and *modular components*, which are specific implementations of their respective abstractions. In other words, the conceptual abstraction is the interface, and the modular components are the subclasses which implement that interface. The region, discussed in the previous section, is one example of a conceptual abstraction, and the rectangle, circle, and polygon regions are different types of modular components that implement that abstraction. GAIA contains many other conceptual abstractions (targets and actions, to name just two), and each type of conceptual abstraction has many different types of modular components. Each conceptual abstraction has its own factory, which knows how to instantiate all of its modular components. Thus the region factory instantiates all of the different types of regions, the target factory instantiates all of the targets, and so on.

5.2.3 Extensibility

One of the primary design goals of GAIA (and a key advantage of Modular AI) is that it is *extensible*. In other words, we need to be able to rapidly and safely add new types of modular components to meet new requirements from design. Furthermore, we also need to be able to add new types of conceptual abstractions as we discover new reusable concepts that can improve the way in which we build our AI. Finally, because we expect to have quite a few different conceptual abstractions (currently, there are 12), and each conceptual abstraction can define the interface for a lot of different types of modular components (in some cases, dozens), we want to ensure that there is as much consistency and as little code duplication as possible either within a given factory (as it handles the different types of modular components) or between the factories for different conceptual abstractions.

Trick #3 discusses how we support the addition of new types of modular components and minimize code duplication in their instantiation, while Trick #5 discusses how we add new types of conceptual abstractions and minimize code duplication between factories.

5.2.4 Reuse

GAIA was designed to be used like middleware. In other words, it is an independent code library that can be plugged in and used with any project, on any game engine. In order to make it easy to decouple GAIA from a particular application (such as a game) and reuse it on another, GAIA cannot have any dependencies on the application.

Conceptual abstractions provide one way to do this. Continuing our spawning example, imagine that the selection of a spawn position is part of a decision that is made inside of GAIA. However, the designers want a custom type of region that is game-specific. Our custom region can be implemented in the game engine but inherit from the `AIRegionBase` interface (which is part of GAIA). The rest of GAIA can then access it via the interface without any dependencies on the game-specific code. This example may seem a bit contrived—how often will a region implementation be game-specific? The answer is, "more often than you might think." That aside, there are other conceptual abstractions that are much more often game-dependent. Actions, for example, are modular components that are executed as a result of the AI's decisions. The vast majority of them (move, shoot, play an animation, speak a line of dialog, etc.) need to call into the game engine. These actions are injected into GAIA in exactly the same way as our hypothetical game-specific region.

There is still a problem, however. We need some mechanism for the factories to be able to instantiate application-specific modular components, but the factories are part of GAIA, not the application! Thus we need to be able to add application-specific components to the factory without adding dependencies on the application or disturbing the code that instantiates all of the modular components that are built into GAIA. The solution to this problem is the subject of Trick #4, while Tricks #1 and #2 discuss other ways to make reuse of the AI library easier. Finally, Trick #6 discusses reuse of data, rather than code.

5.3 Trick #1: Abstracting the Data Format

The first thing to keep in mind as you design your load system is that the format of your configuration may change. For example, the factory in Listing 5.1 uses a `TiXmlElement`, which is an object from Tiny XML (a popular open-source XML parser [Thomason n.d.]). The problem with depending on a specific implementation like this, or even depending on a specific format like XML, is that over time you might discover that XML is too verbose, or that the publisher requires you to use a binary data format, or that the designers are more comfortable working in JSON or YAML, or that some project wants to use an application-specific format, or that your favorite XML parser does not compile using the archaic version of GCC that some application requires, or that its implementation does not meet your stringent memory management requirements, or... you get the idea. Consequently, the format of your data (and the code to parse it) should be wrapped so that it can be replaced in a single location rather than having to chase its tendrils through the entire application. Ideally, it should be treated like any other conceptual abstraction – that is, define the parser interface and provide one or more implementations that ship with the AI library, but also allow the application to define its own parser if needed. This allows the AI library to support project-specific data formats without adding dependencies on the project.

In GAIA, we parse all of the configurations when the application is loaded and store them internally as `AISpecificationNodes`. Specification nodes have a *name* (the label from the XML node), a *type* (the type attribute from the XML node), and *attributes* and *subnodes* (the remaining attributes and subnodes from the XML). This is a pretty thin wrapper, but it gives us enough leverage to support JSON or YAML, for example.

Wrapping the data in this way provides another major benefit as well. Because we control the representation, we can control the way it is accessed. Thus, instead of exposing the data in a raw format, we provide a host of functions that allow you to get data of different types. This includes functions that read in simple types (floats, integers, Booleans, [`x`, `y`, `z`] positions, etc.) as well as abstract types (i.e., conceptual abstractions such as regions). As a result, we can do things such as:

- Standardize the way configurations are formatted so that it will be consistent for all modular components (e.g., Booleans must be "`true`" or "`false`", not "`yes`" or "`no`", and positions are specified using separate `x`, `y`, and `z` attributes rather than a single attribute that provides all three values).
- Check for errors in the data format (e.g., ensure that a parameter that is supposed to be a float is actually a valid number and not, for example, "`12.2.4`").
- Standardize default values (e.g., a position that does not specify `z` will have it set to `0`).
- Provide conventions for specifying particular values (e.g., we allow both "`FLT_MAX`" and "`–FLT_MAX`" as floating point numbers, since both are important in the way our AI is specified).
- Provide a consistent set of conventions for handling the situation where values are missing or of the wrong type (e.g., the code requests a particular attribute as a float, but that attribute was not specified or is a nonnumeric string such as "`true`").

5.4 Trick #2: Encapsulating the Initialization Inputs

Initializing a modular component often requires more than just that component's specification node. For example, for a component in an AI for NPCs, the component may need access to the NPC it is a part of, or to a shared blackboard used by all of that NPC's AI components. What is more, even if a specific modular component does not need these things, other components that share the same conceptual abstraction might, or might contain components that do, so we want to pass all of the necessary metadata through every object's initialization regardless of whether we think it is going to be needed or not.

The simple solution is to wrap all of the initialization inputs (including the specification node) into a single object—in GAIA, it is the `AICreationData`. Each creation data is *required* to contain an `AISpecificationNode` and a pointer to the `AIActor` that represents the NPC that this component helps to control, as well as a variety of other required and optional metadata.

One nice thing about this approach (i.e., encapsulating all of the metadata in an `AICreationData`) is that it gives us a place to hang our hat as we add shared data and capabilities to the factory system. For example, there will often be cases where the application needs to pass application-specific data to its application-specific components. The application might have its own blackboard that handles things like line-of-sight queries (so that it can control the rate at which they happen), and it might implement a specialized line-of-sight AI component that makes use of these queries, for instance. This line-of-sight component needs access to the application's blackboard, but it is created by the AI library's factory, so the application blackboard needs to be added to the `AICreationData`. There are a number of possible ways to handle this. The one that GAIA uses is to have the `AICreationData` include a pointer to an AICreationData_App, which is just an empty base class that the application can subclass off to inject its custom data. Another (arguably better) solution would be to allow the application to create a subclass of the `AICreationData` and place application-specific values there.

Other examples of the usefulness of the `AICreationData` can be found in some of the tricks below.

5.5 Trick #3: Consistent Object Construction and Initialization

One mistake that we made when implementing GAIA is that we allowed every factory to use its own approach for initializing the objects. Our earliest factories worked like the one shown in Listing 5.1—that is, they simply passed the `TiXmlElement`, or the `AISpecificationNode`, or the `AICreationData` (depending on how early they were) into the constructor of the object being created.

As development progressed, however, we discovered that there were features we wanted to support in our factories that simply were not possible with this approach, so over time we started to have a variety of different approaches used by different factories, or even by different types of modular components in a single factory. This resulted in a mess that, while manageable if you understand the code, is highly confusing to engineers who are trying to integrate GAIA into a new application. Consequently, we have worked out a robust approach to object construction and initialization, and are slowly converting the entire library to use it. This approach has four basic steps:

1. Instantiate the object.
2. Optionally run a preload function on the object, which allows default values to be set by the object's owner before it is initialized.
3. Initialize the object from the creation data.
4. Check if the initialization succeeded. If so, return the object. Otherwise, delete it and return `NULL`.

In order to ensure that this four-step process is applied consistently, we implemented it as part of the `AICreationData`, as shown in Listing 5.2. This implementation

Listing 5.2. Using the AICreationData to instantiate and initialize objects.

```
class AICreationData {
public:
    // Not shown: Accessors and storage for the metadata from
    // trick #2.
    ...

    // The PreLoadCallback is called by ConstructObject()
    // after the object has been instantiated but before it
    // is initialized.
    typedef void (*PreLoadCallback) (AIBase* object);
    void SetPreLoadFunction(PreLoadCallback pFunction) {
        m_pPreLoadFunction = pFunction;
    }

    // Construct an object of the specified type, using this
    // creation data to initialize it.
    template<class T>
    T* ConstructObject() const {
        T* pObject = new T;

        if (m_pPreLoadFunction)
            m_pPreLoadFunction(pObject);

        bool bSuccess = pObject->Init(*this);
        if (!bSuccess) {
            // Print an error in the log!
            AI_ERROR ("Failed to initialize object of type "
                      "'%s'", GetNode().GetType().c_str());

            delete pObject;
            pObject = NULL;
        }

        return pObject;
    }
};
```

passes the preload function in as an argument, which has fairly arcane syntax. It also relies on the fact that all of our modular components inherit from `AIBase` (which is the base class of all of their interfaces). If we were starting over, we would probably address both of these issues by using a functor object for preload, but the details of that improvement are beyond the scope of this chapter (especially since it has not been implemented yet!).

Listing 5.3 shows the `Create()` method with all improvements. Notice that this function is still just as concise as Listing 5.1. Adding a new type of modular object just means adding another clause to the `if-then-else`, which amounts to two lines of code. Also note that while the creation data supports preload functions, the factory itself does not use them. If it is used, the preload function is set by the owner of the object the factory is creating and is generally used to specify default values.

Listing 5.3. The region factory's `Create()` function, with all improvements.

```
AIRegionBase*
AIRegionFactory::Create(const AICreationData& cd) {
    // Get the type of region we want to create
    const AIString& nodeType = cd.GetNode().GetType();

    // Create a region of the specified type
    AIRegionBase* pRegion = NULL;
    if (nodeType == "Circle") {
        pRegion = cd.ConstructObject<AIRegion_Circle>();
    } else if (nodeType == "Rectangle") {
        pRegion = cd.ConstructObject<AIRegion_Rect>();
    } else if (nodeType == "Polygon") {
        pRegion = cd.ConstructObject<AIRegion_Poly>();
    }

    return pRegion;
}
```

5.6 Trick #4: Injecting External Code into the AI

The tricks up to this point have focused on the process of instantiating objects. The remainder of the tricks will focus on other aspects of the factory system.

The next issue we address is the need to inject application-specific code into the AI library without creating any AI dependencies on the application. This is especially important for a library that is expected to be reused across many projects, but even within a single project it can be awfully nice to have the AI cleanly decoupled from the game.

We accomplish this by allowing the game to add custom object creators to the factories. We call these object creators *constructors*, although they should not be confused with a class's constructor function. A constructor is a sort of mini-factory that knows how to instantiate some subset of the modular components for a particular conceptual abstraction. The GAIA library includes a *default constructor* for each conceptual abstraction. That default constructor is responsible for instantiating the modular components built into the core library. Applications can also implement their own constructors and add them to the factory. Like most things in GAIA, the constructors all share a common base class that defines their interface, and the factory only interacts with that interface, so application-specific dependencies in the constructor will not place any requirements on GAIA.

To give a concrete example, let us return to our region factory. We built three types of regions into our AI library, as you will recall: the circle, rectangle, and polygon regions. Imagine that our game requires custom regions that are tied to specific navmesh nodes, or that our game has custom mechanics for underwater movement and has to have regions that reflect those mechanics, or that our game supports multi-story structures such as parking garages. We need to be able to add game-specific regions that know how to properly use the game-specific concepts that define the regions' location and extent.

In order to handle this, we first implement one or more game-specific region classes, all inheriting from `AIRegionBase` (as do the circle, rectangle, and polygon regions). Note that GAIA cannot depend on the game, but it is fine for the game to depend on GAIA, so there is no problem with including `AIRegionBase.h` in our game-specific code (if this were not true, the game would not be able to use GAIA at all). Next, we implement a game-specific region constructor that knows how to create our game-specific regions. The game-specific region constructor does *not* have to know how to create the circle, rectangle, or polygon regions—those are already handled by the default constructor that is built into the AI library. Finally, when the game is initialized, before anything is loaded, we add the game-specific region constructor to the region factory. Then, when a region is read in from the data the region factory goes down its list of constructors and calls the `Create()` function on each one until it finds a constructor that successfully returns a non-`NULL` region. If the region is game-specific, then it will be returned by the game-specific constructor. If it is one of the core types (like circle or rectangle) then it will be returned by the default constructor.

One other trick that only comes up very rarely, but is easy to support with constructors, is that occasionally a project will want to replace the built-in implementation of some modular component with a custom version. For example, imagine that your AI library is used by a project that specifies positions using something other than (`x, y, z`) triplets (this is not a completely contrived example—one of the projects that uses GAIA has this problem, although this is not how we solved it). You still want to have rectangle, circle, and polygon regions, but the built-in regions will not work for you. You can solve this by implementing project-specific circle, rectangle, and polygon regions that are instantiated by the project-specific constructor. When the factory goes through its constructors, it does so in reverse order of the order they were added, so the custom constructors will be checked first and the default constructor will be checked last. Thus when the custom constructor is checked and successfully returns a region with a type of "`Circle`", for example, the factory stops checking the remaining constructors, so the project-specific type will be used in place of the one built into the AI library.

In the interests of reducing duplication in this chapter (just as we do in our code), we will refrain from providing a code listing showing the region factory for this trick. The implementation of these ideas can be found along with the ideas from the next trick in Listings 5.4 and 5.5.

Listing 5.4. Using templates to encapsulate duplicate factory code in a base class.

```
template<class T>
class AIConstructorBase {
public:
    virtual ~AIConstructorBase() {}

    // Attempts to create an object from the creation data.
    // Pure virtual so that child classes will be forced to
    // implement it.
    virtual T* Create(const AICreationData& cd) = 0;
};

template<class T>
class AIFactoryBase {
public:
    virtual ~AIFactoryBase();

    // Add a custom constructor. Takes ownership.
    void AddConstructor(AIConstructorBase<T>* pCnstr) {
        m_Constructors.push_back(pCnstr);
    }

    // Looks through all the constructors for one that can
    // create a region. Any constructor which doesn't know
    // how to handle an object of the creation data's type
    // should simply return NULL.
    T* Create(AICreationData& cd);

private:
    std::vector<AIConstructorBase<T>*> m_Constructors;
};

template<class T>
T* AIFactoryBase<T>::Create(AICreationData& cd) {
    T* pRetVal = NULL;

    // NOTE: Pay attention to the stop condition - we break
    // out as soon as we find a constructor that can handle
    // this creation data. We want to try them in the
    // reverse order from which they were added, so loop
    // backwards.
    for (int i = (int)m_Constructors.size() - 1;
         !pRetVal && (i >= 0); --i)
    {
        pRetVal = m_Constructors[i]->Create(cd);
    }

    if (!pRetVal)
        AI_ERROR_CONFIG("Factory failed to create an object "
                        "of type '%s'.",
                        cd.GetNode().GetType());

    return pRetVal;
}
```

Listing 5.5. The region factory when a templatized base class is used.

```
class AIRegionBase;

class AIRegionConstructor_Default
    : public AIConstructorBase<AIRegionBase> {
public:
    virtual AIRegionBase* Create(const AICreationData& cd);
    };

class AIRegionFactory
    : public AIFactoryBase<AIRegionBase>
    , public AISingletonBase<AIRegionFactory> {
public:
    AIRegionFactory() {
        AddConstructor(new AIRegionConstructor_Default);
    }
};
```

5.7 Trick #5: Using Templates and Macros to Standardize Factory Definitions

The approach in Trick #4 gives us a strong basis for our factories, but we do not want to have to copy-and-paste all of the code to implement it for every different conceptual abstraction—if we do, it is virtually guaranteed that the factories will diverge over time (and indeed, in GAIA they did). This quickly becomes a maintenance nightmare, as each factory is mostly the same but ever-so-slightly different than all the others.

Looking more closely, the only major differences between the region factory and another factory (such as the action factory or the target factory) are:

1. The type of object they create (`AIRegion` vs. `AITarget` or `AIAction`).
2. The implementation of the default constructor's `Create()` function (which has to actually instantiate objects of the appropriate types).

As a first step, then, we can use C++ templates that take the type of the object being created and handle most of the duplication. The result is shown in Listing 5.4.

With this done, the declaration of the region factory becomes much shorter, as shown in Listing 5.5. Of note, as this listing shows, GAIA's factories are also singletons. The singleton pattern is beyond the scope of this chapter, but it is well known and GAIA is not doing anything horribly unusual with it.

Listing 5.5 is quite good, but there is still a lot of code there. We are going to add new conceptual abstractions from time to time, and we want this to be as simple as possible. With C++ macros we can. Macro programming is painful, but the macros we need are fairly simple, and we only need to implement them once (and you, lucky reader, can benefit from our example).

The first step is to write a macro that can construct the code in Listing 5.5, but can substitute other words in place of "Region." Thus we could pass "Action" or "Target" into the macro to get the action factory or the target factory. That macro is shown in Listing 5.6.

Listing 5.6. The factory declaration macro.

```
#define DECLARE_GAIA_FACTORY(_TypeName)                                     \
class AI##_TypeName##Base;                                                  \
                                                                            \
class AI##_TypeName##Constructor_Default                                    \
    : public AIConstructorBase<AI##_TypeName##Base> {                       \
public:                                                                     \
    virtual AI##_TypeName##Base*                                            \
        Create(const AICreationData& cd);                                   \
};                                                                          \
                                                                            \
class AI##_TypeName##Factory                                                \
    : public AIFactoryBase<AI##_TypeName##Base>                             \
    , public AISingletonBase<AI##_TypeName##Factory> {                      \
public:                                                                     \
    AI##_TypeName##Factory() {                                              \
        AI##_TypeName##Constructor_Default* pDefault =                      \
            new AI##_TypeName##Constructor_Default;                         \
                                                                            \
        AddConstructor(pDefault);                                           \
    }                                                                       \
};
```

Next, we define a macro that calls other macros, and passes in the name of each conceptual abstraction to each one. Listing 5.7 shows what this macro would look like if we had only region, action, and target conceptual abstractions, along with the call into it that actually creates the factories using the macro from Listing 5.6.

Listing 5.7. The `GAIA_EXECUTE_FACTORY_MACRO` macro.

```
#define GAIA_EXECUTE_FACTORY_MACRO(_FACTORY_MACRO)                          \
    _FACTORY_MACRO(Action)                                                  \
    _FACTORY_MACRO(Region)                                                  \
    _FACTORY_MACRO(Target)

GAIA_EXECUTE_FACTORY_MACRO(DECLARE_GAIA_FACTORY);
```

The `GAIA_EXECUTE_FACTORY_MACRO` is also used to define the singleton object for each factory, and to add the conceptual abstraction into the global object manager (global objects are the subject of Trick #6). Thus, when we want to add a new type of conceptual abstraction to GAIA, all that we need to do is add the name of the abstraction to the `GAIA_EXECUTE_FACTORY_MACRO`, and then implement the `Create()` function for the default constructor. Everything else—all of the infrastructure to create the factory, make it a singleton, and support its global storage—is created auto-magically by the macros. This is a huge boon, especially since we only add new conceptual abstractions very rarely (maybe once every year or two at this point), so it saves us from having to remember all the different places that changes would need to be made when we do.

5.8 Trick #6: Global Object Configurations

Duplication can be as much of a problem in the configuration as it is in the code. For example, imagine that we have a region which defines a spawn volume for a whole bunch of different NPCs. We do not want to have to copy-and-paste this region into every NPC configuration—if nothing else, it is likely to change as the designers tune the game, and we do not want to have to hunt down all the copies in order to change it!

Global object configurations allow us to define a configuration for a particular object once, and then use it elsewhere just as if we had copy-and-pasted it into place. We can do this for any conceptual abstraction—so we can have global regions, global targets, global actions, and so on. In the configuration file, we place the globals in a special node, named for the type of conceptual abstraction (`RegionDefinitions`, `TargetDefinitions`, `ActionDefinitions`, etc.). Each global configuration has to be given a unique name, which is used to look it up elsewhere. For example, we might have two Circle regions that represent spawn zones for friendly and enemy troops:

```
<RegionDefinitions>
  <Region Name="FriendlySpawnRegion"
          Type="Circle" Center="(0,0,0)" Radius="100"/>
  <Region Name="EnemySpawnRegion"
          Type="Circle" Center="(300,0,0)" Radius="100"/>
</RegionDefinitions>
```

We can then refer to these regions using the special type "Global" and the unique name, as follows:

```
<Region Type="Global" Name="EnemySpawnRegion"/>
```

In order to make this work, we need two things. First, we need a global object manager, which is a singleton. When the configurations are parsed, the global object manager is responsible for reading in all of the global definition nodes (such as the `RegionDefinitions` node) and storing away all of the global configurations. Second, the templatized `Create()` function in the `AIFactoryBase` class from Listing 5.4 needs to be extended, so that it resolves any globals before invoking the constructors. The modified function definition is shown in Listing 5.8.

Listing 5.8. The `AIFactoryBase`'s `Create()` function with support for globals.

```
template<class T>
T* AIFactoryBase<T>::Create(AICreationData& cd) {
    T* pRetVal = NULL;

    // Check if this is a global, and if so use the
    // specification node stored on the global manager.
    const AISpecificationNode& node = cd.GetNode();
    const AIString& nodeType = node.GetType();
    if (nodeType == "Global") {
```

(Continued)

```
        AIString globalName =
              node.GetAttributeString("Name");

        const AISpecificationNode* pActualNode =
              AIGlobalManager::Get().Find(globalName);

        if (!pActualNode) {
              AI_ERROR("Factory does not have a definition "
                       "for a global object named '%s'.",
                       globalName.c_str());
        } else {
              // Set the node on the creation data to the
              // actual node for this global, create the
              // object, then set the node on the creation
              // data back to its previous value.
              cd.SetNode(*pActualNode);
              pRetVal = Create(cd);
              cd.SetNode(node);

              return pRetVal;
        }
    }

    // The rest is the same as Listing 4.
    ...
}
```

5.9 Conclusion

In this chapter we covered a number of different tricks that have been learned through the hard work of fixing mistakes and refactoring of the factory system for our reusable AI library, GAIA. Together, these tricks provide a consistent approach for specifying and initializing objects across all of our factories, allow us to decouple the AI library from the rest of the game, unify the factory code so that new factories (along with all of their support structures) can be created with a single line of code, and greatly reduce the duplication both within the factory code and also within the configurations themselves. The result is a factory system that supports our goals of extensibility (new modular components can be added with two lines of code, and new conceptual abstractions can be added with only one) and reuse (the AI library is cleanly decoupled from the rest of the game, allowing us to reuse the AI library with impunity).

References

Dill, K. 2016. Modular AI. In *Game AI Pro 3*, ed. S. Rabin. Boca Raton, FL: CRC Press, pp. 87–114.

Gamma, E., R. Helm, R. Johnson, and J. Vlissides. 1995. *Design Patterns: Elements of Reusable Object-Oriented Software*. Boston, MA: Addison-Wesley, pp. 107–116.

Thomason, L. n.d. TinyXML Main page. http://www.grinninglizard.com/tinyxml/ (accessed June 10, 2016).