Gamma World Character Library

By Anthony Langsworth

12/30/2012

Contents

[Introduction 2](#_Toc344590249)

[Legal Notice 2](#_Toc344590250)

[Vision 2](#_Toc344590251)

[Using the Library 3](#_Toc344590252)

[Extending the Library 4](#_Toc344590253)

[Structure 5](#_Toc344590254)

[Concepts 5](#_Toc344590255)

[Scores and Modifier Sources 5](#_Toc344590256)

[Constructing the Modifier Graph 6](#_Toc344590257)

[Example: Create an Origin 7](#_Toc344590258)

[Example: Create a Power 8](#_Toc344590259)

# Introduction

This Gamma World Character Library aims to model a character in Gamma World, a role-playing game from Wizards of the Coast (WotC, <http://www.wizards.com/dnd/gammaworld.aspx>). Specifically, this library provides easy programmatic rendering of character sheets and “what-if” simulations.

This document assumes the reader is familiar with the Gamma World rules. Readers are strongly encouraged to purchase the rules and reference them when using the library. Specific references are in the form of “GWXX” where XX is a page number of the Gamma World rulebook. Familiarity with similar systems, such as the fourth edition (4e) of Dungeons and Dragons (D&D), is helpful but the Gamma World rules are self-contained.

The library is developed on Visual Studio 2012 in C# on .Net Framework 4.0. It has not been tested in other environments, including Mono. This document assumes the reader is familiar with C#.

# Legal Notice

The author is unaware of any specific legal guidance from Wizards of the Coast (WotC) surrounding implementing all or part of the Gamma World rules in software. If nothing else, the term “Gamma World” and the text of the rules is covered by copyright owned by Wizards of the Coast.

Gamma World is very similar to 4e D&D and products that use 4e D&D rules are governed under WotC Gaming System License (GSL) (<http://www.wizards.com/d20/files/4E_GSL.pdf>). Unfortunately, this agreement specifically precludes "character creators" and "interactive products" and the FAQ (<http://www.wizards.com/d20/files/4E_GSL_FAQ.pdf>) reinforces this.

That said, the intention of the author is to create a library that may assist existing players and GMs when playing Gamma World. Therefore, the included library includes the base rules and a limited selection of origins (Android, Cockroach, Doppelganger, Electrokinetic, Empath, Felinoid, Giant, Gravity Controller, Hawkoid and Hypercognitive), their novice powers (only) and characters up to level 2.

# Vision

The vision was for a library to “model” a character, specifically:

1. Enforce all character creation rules and not permit invalid characters. For example, that a character cannot take three origins or wield multiple two-handed weapons. It also means many aspects of the character are immutable once created, such as origins and ability scores.
2. Follow the rules as closely as possible, making it easy to understand for those familiar with Gamma World.
3. Not be data driven, that is load its data from XML or a similar external database. Gamma World, like 4e D&D it is based on, is designed around a relatively simple core set of rules then powers, origins and the like breaking those rules. Creating a system to cope with all the potential combinations and permutations is prohibitively complex so origins, powers and the like are implemented in code.
4. Adding new origins, powers and so on should require minimal effort and should be easy to test.
5. Models character creation, not combat. For example, the library stores the characters maximum hit points and equipment but not current hit points, temporary hit points, action points and so on. Descriptions of power affects need only be provided in text, not an object model. The library does not track Omega Technology or Alpha Mutations.
6. Not provide a UI to create or serialize characters (at least not initially). Characters are created in code and output as something presentable for a gaming session, such as HTML.
7. It is not intended to replace the rules or other reference material. For example, the library not need provide a list of origins, powers and so on to calling code suitable for selection or browsing in a UI.

# Using the Library

Take creating a level 2 Cockroach Android as an example:

using GammaWorldCharacter;

using GammaWorldCharacter.Origins;

using GammaWorldCharacter.Gear;

using GammaWorldCharacter.Gear.Weapons;

using GammaWorldCharacter.Gear.Armor;

...

Character character;

character = new Character(

// Ability scores rolled randomly on 3d6

new int[] { 12, 11, 10, 15 },

// Primary origin (from GammaWorldCharacter.Origins)

new Cockroach(),

// Secondary origin

new Android(),

// Randomly determined bonus skill

ScoreType.Science)

{

// Name the character

Name = "Character name",

// Name the player

PlayerName = "John Doe",

};

// Equip a heavy one-handled weapon in the main hand. Returns the previously

// equipped item in that hand (null in this case).

character.SetHeldItem(Hand.Main,

new MeleeWeapon(WeaponHandedness.OneHanded, WeaponWeight.Heavy));

// Equip a shield in the off hand

character.SetHeldItem(Hand.Off, new Shield());

// Also carry a one-handed ranged weapon

character.Gear.Add(

new RangedWeapon(RangedType.Weapon, WeaponHandedness.OneHanded,

WeaponWeight.Heavy));

// Don light armor. This determines the slot (only the body slot exists in GW) and

// returns the previously equipped item.

character.SetEquippedItem(new LightArmor());

// Add a standard explorer’s kit (See GW76)

character.Gear.AddRange(ExplorersKit.Contents);

// Add the second level, choosing the critical hit benefit from the

// primary origin (Cockroach).

character.AddLevels(new Level02(OriginChoice.Primary));

// Update the character’s scores after changes.

character.Update();

Commonly used scores like ability scores, skills, defenses, hit points and initiative can be accessed using properties, such as **character.Strength** or an indexer, such as **character[ScoreType.Strength]**. Not all scores are available as properties but all are available through the indexer.

The character’s powers are accessed using **character.GetPowers()**. This returns an IEnumerable<Power> that can be used as a basis for LINQ query. The **GetUsablePowers()** method returns the powers that the character can use with the current gear. For example, a weapon based power may not be usable if the character does not have a weapon equipped.

The example shows how to manipulate equipment. Note that the library will prevent common issues. For example, an exception will be thrown if the character tries to equip a two handed weapon in each hand or wear a weapon as armor.

Namespaces within the GammaWorldCharacter assembly likely to be needed are:

|  |  |
| --- | --- |
| Namespace | Description |
| CharacterGenerator | The top level namespace that contains important classes like Character, Dice, ModifierSource, Modifier and Score. |
| CharacterGenerator.Origins | Each origin in the Gamma World rules has a corresponding class here. |
| CharacterGenerator.Gear | Item and ExplorersKit classes. |
| CharacterGenerator.Gear.Weapons | MeleeWeapon and RangedWeapon classes. |
| CharacterGenerator.Gear.Armor | Shield, LightArmor and HeavyArmor classes. |
| CharacterGenerator.Levels | Classes that represent each level beyond one. |
| CharacterGenerator.Powers | Contains base classes WeaponAttackPower, AttackPower and UtilityPower along with a number of helper classes. |
| CharacterGenerator.Powers.Origins | Powers supplied by origins. |

# Extending the Library

From a development perspective, the goals of the library are:

1. Create a library capable of representing characters, not the entire rules. Indeed, although the intention is to be able to model as much of the rules as possible, only the rules needed to model specific characters is implemented.
2. To reduce complexity, the library may not support every possible character, at least not without further modification.
3. The library should be testable. Code that cannot be tested in an automated fashion indicates a design flaw.
4. The library should avoid code repetition, also a design flaw.

## Structure

The library is composed of:

* GammaWorldCharacter: The main portion of the library. This contains code for modeling characters, origins, skills, powers, gear and other important aspects of characters.
* GammaWorldCharacter.Samples: Sample characters.
* GammaWorldCharacter.Test.Unit: Unit tests.
* GammaWorldCharacter.Test.Integration: Integration tests that testing whether sample characters are constructed correctly.
* GammaWorldCharacterViewer: A Windows Presentation Foundation (WPF) based program that shows and prints the sample characters. It can also toggle showing all the modifiers summed to give the scores, e.g. "Initiative +1" versus "Initiative +1 (+1 Dex)".
* CreateCharacterTest: A project that uses Visual Studio's T4 library to create an NUnit test class to test a character (used to turn sample characters in GammaWorldCharater.Samples into tests in GammaWorldCharacter.Test.Integration).

## Concepts

### Scores and Modifier Sources

One central tenant of a Gamma World character (like 4e D&D) is that certain “scores”, such as ability scores, influence the values of other scores by adding “modifiers” to them. For example, a character with Dexterity score of 12 would add the modifier “+1” (using the table on GW59) to their Initiative score (GW31).

These modifiers and scores are transitive. For example, a character’s Constitution score is added to 12 to determine their hit points at first level (GW31). The character’s Bloodied score is half their hit points. Therefore, the Bloodied score should not be calculated before Constitution has been added to the hit points. In other words, there is a dependency relationship between “modifier sources” and “scores”, effectively a directed graph.

This is modeled by three classes (in the GammaWorldCharacter namespace in the GammaWorldCharacter project):

* **Score**: A number that can receive modifiers from modifier sources, such as ability scores, skills, defenses, speed, saving throws and power attack and damage bonuses. The value of a score is usually the sum of a base value (for example, defenses start at 10) and all modifiers that apply to it.
* **ModifierSource**: Something that can add modifiers to Scores, such as origins and equipment. Scores are also modifier sources (in other words, **Score** inherits from **ModifierSource**). For example, each ability score adds a modifier to one or more skills.
* **Modifier**: A modifier is composed of three elements: (1) the **ModifierSource** that is the source of the modifier, (2) the **Score** being modified and (3) an integral value that may be positive (a bonus) or negative (a penalty). Modifier sources add modifiers to scores.

These are linked in a directed graph where each vertex is a ModifierSource, implemented in the **GraphNode<T>** class in GammaWorldCharacter.Collections namespace (GammaWorldCharacter project).

However, this graph is not static. For example, the greater of the character’s Intelligence and Dexterity modifiers are added to Armor Class only if the character is not wearing heavy armor. This means each modifier source needs to examine the character first to determine what modifiers are applied to what scores.

### Constructing the Modifier Graph

When a character is first created or modified, the character’s scores must be recalculated by calling **Character.Update()**. This constructs the relationships between **ModifierSource**s and the **Score**s in two stages: dependency mapping and updating scores.

The first or dependency mapping stage is implemented by overriding **ModifierSource.AddDependencies(Action<ModifierSource, ModifierSource> addDependency, Character character).** During this stage, the modifier source examines the character, identifying (1) the modifier sources it may add modifiers to and (2) any modifier sources this depends on, creating a dependency graph.

The character must not be modified during this stage, since it may lead to inconsistencies. If uncertain, a modifier source should add dependencies on all potential modifier sources.

Also, this is the first opportunity for a ModifierSource to ensure it is applied to a suitable character, so it should ensure any prerequisites are met and either throw an exception (if it is an error constructing the character) or add no modifiers(otherwise).

The second or updating scores stage is implemented by overriding **ModifierSource.AddModifiers(CharacterUpdateStage stage, Action<Modifier> addModifier, Character character)**. During this stage, the dependency graph created in the first stage is walked such that the modifiers to a score are only added after all dependent modifier sources have been visited.

If a modifier source always depends on and modifies the same modifier sources, **AddDependencies** need not be implemented – the default implementation of **AddDependencies** calls **AddModifiers** (with stage set to CharacterUpdateStage.DependencyMapping rather than CharacterUpdateStage.UpdatingScores) and creates dependencies from the result.

Although very powerful, this approach produces two potential problems. First, the graph cannot be cyclic. Thankfully, the Gamma World rules do not permit cycles because, otherwise, characters could not be created. If a character’s hit points modified the Constitution score and the Constitution score modified the hit points, the both the hit points and Constitution would be indeterminable.

The second problem is whether scores “pull” modifiers from modifier sources or whether modifier sources “push” the modifiers they provide. The answer is dictated by extensibility. Scores, such as initiative, hit points and skills, and powers “pull” modifiers from modifier sources, allowing new scores and powers to be added without modifying existing ones. By comparison, modifier sources that are not scores, such as origins, “push” their modifiers onto existing scores.

Also note the dependency and modifier graphs use references, not equality. This means the same scores and modifier sources must be used throughout the update process and not recreated at each step. Scores and modifier sources must not be shared between characters.

## Example: Create an Origin

To create a new origin:

1. Create a new class that inherits from **GammaWorldCharacter.Origins.Origin**.
2. In the new origin class constructor (usually parameterless):
   1. Call the base **Origin** constructor, supplying (1) the new origin name, (2) the primary ability score, (3) the power source and (4) the critical hit benefit gained at either level 2 or level 6.
   2. Add any traits by passing a new **GammaWorldCharacter.Traits.Trait** to **Origin.AddTrait(Trait trait)**. Most traits are descriptive rather than mechanical (in other words, they do not affect the character’s scores) so a new class for Traits are rarely required.
   3. Add the novice power by passing it to **AddPower(Power power)**. More detail on powers is described below.
3. Override **AddModifiers** to add the standard skill and defense bonuses, along with anything origin specific:
   1. Call the base class implementation of AddModifiers() first.
   2. Call **addModifier** (a delegate supplied as an argument to AddModifiers) for each modifier.

An example is the Android origin (GW36):

using System;

using GammaWorldCharacter.Traits;

using GammaWorldCharacter.Powers.Origins;

namespace GammaWorldCharacter.Origins

{

/// <summary>

/// The Android origin (see GW36).

/// </summary>

public class Android: Origin

{

/// <summary>

/// Create a new <see cref="Android"/>.

/// </summary>

public Android()

: base("Android", ScoreType.Intelligence, PowerSource.Dark,

"Deal 1d10 extra damage and the target grants combat advantage to "

+ "you until the end of the encounter.")

{

AddTrait(new Trait("Machine Powered",

"You do not need to eat, drink or breathe"));

AddPower(new MachineGrip());

}

/// <summary>

/// Add modifiers.

/// </summary>

/// <param name="stage">

/// The stage during character update this is called.

/// </param>

/// <param name="addModifier">

/// Add modifiers by calling this method.

/// </param>

/// <param name="character">

/// The character to add modifiers for.

/// </param>

protected override void AddModifiers(CharacterUpdateStage stage,

Action<Modifier> addModifier, Character character)

{

base.AddModifiers(stage, addModifier, character);

addModifier(new Modifier(this, character[ScoreType.Fortitude], 2));

addModifier(new Modifier(this, character[ScoreType.Science], 4));

}

}

}

Note that this need not implement **AddDependencies** because the scores modified are fixed.

## Example: Create a Power

To create a new power:

1. Create a new class that inherits from either **GammaWorldCharacter.Powers.WeaponAttackPower** (for powers that inflict damage based on the weapon used), **AttackPower** (for powers that inflict damage or affect enemies irrespective of the weapon used) or **UtilityPower** (for powers that do not inflict damage like the DoppelGanger novice power “Double Trouble” or origin utility powers). The following focuses on **AttackPower**, since it is the most complicated.
2. In the new attack power class constructor:
   1. Call the base **AttackPower** constructor, supplying (1) the power name, (2) the origin that supplies the power and (3) the required level (1 for a novice power, 3 for utility or 5 for expert). The origin and minimum level are additional checks the library performs to ensure the character is valid.
   2. (Optional) Set the power’s description or flavor text by calling **SetDescription**.
   3. Set the power’s details by calling **SetPowerDetails** with: (1) how often the power can be used, (2) the power source, (3) the damage types , (4) the effect types (if any), (5) the action required to use the power and (5) a trigger (used for immediate actions only).
   4. Set the power’s attack type and range by calling **SetAttackTypeAndRange**.
   5. Add one or more attacks by creating an **AttackDetails** object and passing it to **AddAttack**. The **AttackDetails** constructor’s parameters are:
      1. A description of the target.
      2. The bonus to the attack roll, usually an **AbilityPlusLevelBonus** object whose parameters are (1) a name (used for debugging only), (2) a list of ability score modifiers to add and (3) a multiplier for the character’s level.
      3. The damage, usually a **PowerDamage** (where damage is specified using the **Dice** object) or **WeaponDamage** object.
      4. The targeted defense.
      5. Additional **ModifierSource** objects used in the description or elsewhere by the power.
      6. The description of the power following its damage dice.
      7. The description of what happens when the power misses (or null if there is none).

This seems complicated (and it is). However, it is basically data entry from the power description. **WeaponAttackPower**s have overloads to simplify adding attacks.

Continuing the example above, the Android novice power machine grip is:

using System;

using GammaWorldCharacter.Powers;

using GammaWorldCharacter.Origins;

namespace GammaWorldCharacter.Powers.Origins

{

/// <summary>

/// The Machine Grip novice <see cref="Android"/> power (GW36).

/// </summary>

public class MachineGrip: AttackPower

{

/// <summary>

/// Create a new <see cref="MachineGrip"/>.

/// </summary>

public MachineGrip()

: base("Machine Grip", typeof(Android), 1)

{

SetDescription(

"When you get a hand on an enemy, your grip tightens "

+ "like a steel-jawed vice.");

SetPowerDetails(PowerFrequency.AtWill, PowerSource.Dark,

DamageTypes.Physical, EffectTypes.None, ActionType.Standard, null);

SetAttackTypeAndRange(AttackType.Melee, "1");

AddAttack(new AttackDetails(

"One creature",

new AbilityPlusLevelBonus("Machine Grip attack bonus",

new ScoreType[] { ScoreType.Intelligence }, 1),

new PowerDamage("Machine Grip damage", new Dice(1, DiceType.d10)),

new AbilityPlusLevelBonus("Machine Grip damage bonus",

new ScoreType[] { ScoreType.Intelligence }, 2),

ScoreType.Reflex,

new ModifierSource[] { },

"physical damage and the target is immobilized until the start "

+ "of your next turn. If you move to a square that isn't adjacent "

+ "to the target, the immobilization ends.",

null));

}

}

}