Hexkit Strategy Game System

User's Guide

Christoph Nahr
christoph.nahr@kynosarges.de

Abstract

This document describes the fundamental concepts of Hexkit, a construction kit for turn-based strategy games played on maps that are divided into squares or hexagons. We cover the default game rules and the creation of custom rules; the internal and external storage formats for scenarios and saved games; and the available computer player algorithms.

Hexkit comprises two applications: Hexkit Game which allows users to play games based on a scenario; and Hexkit Editor which facilitates the design of such scenarios. Both applications provide a standard Windows user interface which is documented in the online help.

The current versions of this *User's Guide* and of the Hexkit software packages are available at the Hexkit home page, http://www.kynosarges.de/Hexkit.html. Please consult the ReadMe file included with the binary package for system requirements and other information.

Online Reading. When viewing this *User's Guide* in Adobe Reader, you should see a document navigation tree to the left. Click on the "Bookmarks" tab if the navigation tree is hidden. Click on any tree node to jump to the corresponding section.

Moreover, all entries in the following table of contents, and all phrases shown in blue color, are clickable hyperlinks that will take you to the section or address they describe.

Revision History

Revision 2.8.2, published on 31 May 2011
Based on Hexkit 4.2.9 and Tektosyne 5.5.1, revised to reflect the split of the Tektosyne library in two assemblies, and to move graph algorithms to the new *Tektosyne User's Guide*.

Revision 2.8, published on 24 October 2010
Based on Hexkit 4.2.7 and Tektosyne 5.1.5, revised to move UML diagram creation from Sparx Enterprise Architect to my new Class Diagrammer application.
Several intermediate revisions reflected changes in Tektosyne 4.2 through 5.1.

Revision 2.7.5, published on 27 September 2009
Based on Hexkit 4.2, revised for new HCL instructions and Poitiers scenario.
Several intermediate revisions reflected changes in Hexkit 4.1.1 through 4.1.6.

Revision 2.7, published on 17 May 2009
Based on Hexkit 4.1, revised for new entity semantics and variable modifier system.

Revision 2.6.2, published on 07 March 2009
Based on Hexkit 4.0.5, revised for increased variable ranges and location storage.

Revision 2.6.1, published on 16 February 2009
Based on Hexkit 4.0.4, revised for first WPF release and various other changes.

Revision 2.6, published on 02 June 2008
Based on Hexkit 3.7.1, thoroughly revised for new features and partly rearranged.

Revision 2.5, published on 16 May 2007
Based on Hexkit 3.6.1, revised for the updated graphics engine.

Revision 2.4, published on 26 October 2006
Based on Hexkit 3.5, revised to cover the newly added variable map geometry.

Revision 2.3, published on 08 December 2005
Based on Hexkit 3.4.1, thoroughly revised for new features and source code changes.

Revision 2.2, published on 28 February 2005
Based on Hexkit 3.2, revised to cover the Hexkit Command Language (HCL).

Revision 2.0, published on 30 November 2004
Based on Hexkit 3.0, thoroughly revised and with several new chapters.

Revision 1.0, published on 31 March 2004
Initial release with Hexkit 2.8.8, containing all "Info" pages from the online help.

Colophon

This guide was written using Adobe FrameMaker 10 in "structured mode." The document structure is based on a custom subset of DocBook XML 4.3. The hyperlinked PDF file was created with FrameMaker's built-in PDF generation facility, using Adobe Acrobat Distiller 9.4.

The UML diagrams were reverse-engineered from the compiled .NET assemblies using my free Class Diagrammer application, available at http://www.kynosarges.de/Diagrammer.html. The diagrams were embedded in the FrameMaker document as PDF files.

Typesetting. The main text is set in Minion 12 pt. Chapter titles and footnotes are set in various sizes and weights of the same font, using typefaces from Adobe's Minion Pro Opticals collection. Minion was designed by Robert Slimbach of Adobe.

Intermediate titles are set in various sizes of Myriad Semibold, using typefaces from Adobe's Myriad Pro collection. Program code, file names, and GUI labels are set in Myriad Regular 11 pt, and the text of UML diagrams is set in other sizes and variants of the same font family. Myriad was designed by Robert Slimbach and Carol Twombly.

Table of Contents

List of Figures

List of Tables

Chapter 1: Introduction

Hexkit is a complete framework to design and play turn-based strategy games on hexagon maps. Starting with version 3.5 the program also supports maps composed of squares.

While many computer games come with scenario editors, Hexkit offers greater flexibility than most because scenario designers may customize not only the map layout, the participating factions and units, and the (admittedly modest) graphics, but even the actual game rules.

The rest of this guide details how to operate and customize Hexkit. But first let's talk about the ideas behind the whole project and highlight some noteworthy features.

1.1 Target Audience

So who would want to use Hexkit, anyway? What are its benefits?

— Wargaming for the masses! Thanks to its extensive customization features, Hexkit might finally allow you to create the scenarios you always wanted to make, but for which other games weren't flexible enough.

— Building blocks for other games. Hexkit's thoroughly documented source code is published under the MIT license which allows for commercial and non-commercial reuse. Base your own project on Hexkit, or take whatever components you need.

If you plan to do this, also check out the Tektosyne library which provides the mathematical structures and pathfinding algorithms used by Hexkit. The Tektosyne home page is located at http://www.kynosarges.de/Tektosyne.html .

— Rapid prototyping of game ideas. If you're working on your own game that won't have anything to do with Hexkit, you might still appreciate Hexkit as a ready-made testing framework for new game mechanics.

While I'd also like to provide scenarios that are somewhat fun to play, I have few illusions about my own abilities when it comes to designing games or (worse) drawing pictures. I did create some incredibly boring scenarios based on historical battles, though…

1.2 Feature Overview

Hexkit consumes user-created scenarios, rule scripts, and graphical tilesets which define the specifics of a game. These include all available units, terrains, and so on; the competing factions; the game rules; and the game map and its visual appearance. Factions can be played by the computer or by other human players. Multiple human players can take turns via "hotseat" play or play-by-email. An included GUI editor can be used to create and alter Hexkit scenarios.

The current Hexkit implementation offers the following features, where "unlimited" means limited only by available system resources:

— Scenarios and saved games are described by schema-validated XML files, so you can use any XML editor to examine or modify them. Scenarios are divided into five sections which may be stored in separate XML files for easy reuse. Saved games retain the

complete history of all issued commands, allowing you to inspect and replay an entire game at will – particularly useful for PBEM and scenario debugging.

— Scenario-specific game rules are defined by script files written in C#, JScript .NET, or Visual Basic .NET. The script file is compiled before use, ensuring that script methods execute at the same speed as the game engine's built-in C# methods. Alternatively, the rule script may be supplied as a precompiled .NET assembly written in an arbitrary language. Some rules may be customized by simply setting values in the GUI editor.

— Graphics tiles are stored in collections of bitmap files. Transparency may be specified via alpha channel or by a user-defined "magic" color value. Tiles may be scaled to the size and shape of a map polygon. One image may contain multiple tiles, allowing idle animation in the style of early *Ultima* games. The GUI editor provides some limited facilities for image composition without having to alter the bitmap files.

— Unlimited map size (as defined by the scenario). Although the display is buffered to prevent flickering, rendering speed and required display memory are not affected by map size – unlike simple display buffering schemes such as MFC's document/view architecture or the Windows Forms ScrollableControl that require a buffer bitmap large enough to hold the entire "document."

— Unlimited number of independent scrollable and zoomable map views that may be attached to any internal game state and any output window as desired. This allows the use of separate map views in dialog boxes to plan attacks and movements.

— Unlimited zoom levels for map views, currently divided into five steps from 50% to 200% for ease of use. Zoom levels other than 100% use WPF anti-aliasing for improved visual quality. However, bitmap caching ensures that rendering speed per square or hexagon is virtually unaffected by zoom level.

1.2.1 Missing Features

On the other hand, before anyone gets too excited, I'd also like to point out some popular features that are missing from Hexkit.

— Simultaneous play over a LAN or the Internet. Hexkit is not designed to communicate with other instances of the executable running on remote systems.

— Diplomacy. The original Hexkit design included a fairly sophisticated diplomacy model but that was dropped early on in order to speed development. In the current version, all factions are at war with each other all the time.

— Fog of war. Currently the entire map is always visible to all players. Limited visibility is obviously a requirement for any serious wargame, and it should be possible to add this feature based on the visibility algorithm introduced in Hexkit 3.7.

— Complex unit rotation as in the old *Steel Panthers* games where a tank's chassis and turret could face into arbitrary, independent directions.

This feature, which is only relevant to tactical scenarios with modern tanks, would require significant internal changes to Hexkit. I see little point in this effort, given that there are already several free re-editions of *Steel Panthers* available.

1.3 A History of Hexkit

Hexkit has come a long way since its inception. This section notes some milestones.

1.3.1 The Early Days: C and C++

Hexkit came about rather by accident. I've always enjoyed playing wargames and other turn-based strategy games, so in early 2000 I wrote a small C library for drawing hexagon maps with a vague idea to make my own wargame. Once I had a working map view I wrapped it in a basic GUI application which I put up for download.

At that point, development stalled for a while. I was unsure about the direction I should take, I discovered that I'm *really* bad at drawing (i.e. creating unit and terrain graphics), and the folks who had originally expressed interest in a hexagon map library were never heard of again once the implementation was available.[1]

During the next year, I reorganized Hexkit as a C++ project that was robust enough to sustain a larger feature set. Then I found DeBray Bailey's excellent free tileset which provided the outdoors tiles I needed.[2] DeBray graciously consented to my use of his work, so at the beginning of 2002 I set out again to make a better Hexkit.

With the switch to C++ came the first definitions of the XML-based scenario and save game formats. The unique "command history" format for saved games was inspired by my earlier freeware game, *Star Chess*. I had grown rather fond of the ability to examine and replay any and all turns in a game at will – it's a tremendous debugging aid, particularly when occasional hiccups in computer player algorithms are involved.

1.3.2 Reimplementation in C#

The new design and direction held up well, and in May 2002 I could move Hexkit to Microsoft's .NET Framework and the C# language with little difficulty. The switch *halved* the line count of the Hexkit project at the time, and roughly quadrupled my productivity. Nevertheless, the rest of 2002 and 2003 went by at a plodding pace as one small feature after another was put into place. If I had continued using C++, I would likely have dropped development at this point.

In January 2004, the rule script mechanism finally worked well enough to support a version of the "Troll Chess" demo scenario with customized rules. In March 2004, computer players roamed the map for the first time. I started writing proper documentation, namely this *User's Guide*, to supplement the *Hexkit Class Reference* that was auto-generated from C# source code.

I also indulged my love for antiquity, and showed off what Hexkit could handle, with a huge "Roman Empire" scenario that had been under development since Summer 2004 and which was finally released with Hexkit 3.0.

---

1. While lots of game aficionados dabble in programming and want to make games of their own, those plans rarely survive their first encounter with actual game code… Hopefully, the gradual replacement of the admittedly very complex and difficult C++ with easier-to-use languages such as C# will serve to lower the entry threshold for casual game programmers.

2. There are very few freely available graphics that are suitable for games, and most of those are for Roguelikes. Unfortunately, most Roguelikes take place in underground caverns. DeBray's tileset was the only graphics collection I found that provided the varied outdoors environments I wanted.

1.3.3 Hexkit Command Language

Hexkit 3.2, released in early 2005, introduced the Hexkit Command Language (HCL), a kind of "microcode" whose instructions each have a small, deterministic effect on the current world state. This is effectively a new game-specific programming language, except that it is implemented as a set of C# classes and methods.[3]

All game commands, which used to directly manipulate the data of the current world state, were reimplemented to emit HCL instructions instead. The implementation and advantages of this two-stage execution mechanism are detailed in "Command Execution" on page 90.

1.3.4 Upgrading to .NET 2.0

Version 2.0 of the .NET Framework was finalized during 2005, and before the end of that year Hexkit 3.4 was released for the new Framework. This update was a priority since Hexkit includes numerous strongly-typed collection classes which used to be auto-generated from CodeSmith templates. Replacing these classes with generic collections roughly halved the line count of the Hexkit solution, at no loss of functionality.

There were a number of new features as well. In addition to the command history, Hexkit now records data points for entities and factions for the creation of history graphs – one of my favorite features in strategy games. Every game should have a history graph!

Moreover, Hexkit can now handle building, placing, and destroying non-unit entities. The one available computer player algorithm still only knows about units, however, and there is no scenario yet to exploit these new features, either.

1.3.5 Geometry and Graphics

Hexkit 3.5 was finally capable of handling variable map geometries – a truly ancient entry in the feature wish list. Maps can now contain either squares or hexagons, and either polygon may be standing on a corner or lying on a side.

Wargame maps are quite often composed of hexagons lying on a side whereas older Hexkit versions had only supported hexagons standing on a corner, so as to better accommodate DeBray Bailey's tileset. The new geometry variants allow an easy translation of existing wargames to Hexkit, and also the creation of maps based on squares in the style of *Civilization*.

Hexkit 3.6 featured an enhanced graphics engine with several new options for animation, scaling, and image composition. DeBray's tileset is now entirely available to scenario authors, and we can build composite images such as riders on horses and soldiers with crossbows.

Sadly, our tile-based engine probably has been pushed to its limits with this release. Visual quality drops tremendously when a tile is stretched to fit a polygon shape it wasn't designed for, and the run-time construction of composite images from multiple tiles, e.g. a shore line crossing an arbitrary polygon in any possible direction, seems too complicated for practical use.

1.3.6 Right-Clicks and Visibility

Hexkit 3.7 offered a laundry list of improvements, starting with *Civilization*-style hotkeys and right-click commands that bypass the planner dialogs. This greatly speeds up movement and combat in scenarios with lots of units, such as the "Battle of Crécy" demo scenario added in this

3. This might remind some people of Philip Greenspun's Tenth Rule of Programming: "Any sufficiently complicated C or Fortran program contains an ad hoc informally-specified bug-ridden slow implementation of half of Common Lisp."

release. The new tactical scale of this scenario required some improvements to the game engine as well, the most important of which is the concept of lines of sight for ranged attacks.

A new algorithm for geometric visibility calculations determines which theoretical targets are actually obscured by trees, buildings, or most frequently by other units. This algorithm should also form the basis for a full "fog of war" implementation in the near future. The main challenge that remains is restricting human and computer players to incomplete information about the current world state – I anticipate many complex code changes to achieve that goal.

Another change related to the map geometry is the possibility to have units move across map sites that they may not occupy. This was a long-standing wish list item since such movement rules are popular among both traditional board games and historical wargames.

The last addition worth pointing is the concept of "standard variables". Most scenarios use a similar set of entity statistics that includes a unit's strength (health or hit points), its movement speed vs. the difficulty of traversed terrain, etc. Since the naming of entity variables is arbitrary we must establish a mapping between these conceptual statistics and actual entity variables. This used to require method overrides in custom rule scripts, but as of Hexkit 3.7 you can use the GUI editor to define the entity variables to use. Hexkit's default game rules now operate on these "standard variables", freeing rule scripts of a lot of repetitive plumbing.

1.3.7 Windows Presentation Foundation

Hexkit 4.0 was the first release for .NET 3.5 and the new Windows Presentation Foundation, the successor to Windows Forms which had been essentially a wrapper around Win32 functions. Theoretically, this switch is hardly worth mentioning – it resolved a few minor issues and added some visual effects such as drop shadows and animated arrows, but Hexkit's overall appearance and inner workings remained largely unchanged.

In practice, I found WPF so dramatically different from Windows Forms that I had to rewrite almost the entire existing GUI code and graphics engine. The complexity of the new API was much greater than I expected, and there were some nasty surprises such as appalling drawing performance and a total lack of synchronous display updates.

Still, these issues are largely solved now, and there is no question that WPF is a much more elegant and powerful framework than Windows Forms. Just make sure you have at least a dual-core CPU so unit movements won't turn into a slideshow…

1.3.8 Goodbye to Items

Hexkit 4.1 is notable for a change in the available game entities. Aside from the obligatory units and terrains, previous versions provided an all-purpose "item" category whose semantics varied depending on the context and certain flags. Items might represent physical objects that affect only local units, intangible bonuses for a faction and all its units, or physical objects that turned into such bonuses when picked up by a unit with the special "item recovery" flag.

All of this was exactly as confusing as it sounds, and this guide's attempt to explain items was never quite satisfying. The model I had in mind were magical items like those found in the classic *Heroes of Might and Magic* games, with an eye on extending the use of items to fully fledged character inventories similar to role-playing games.

However, Hexkit is fundamentally best suited for historical wargames and 4x strategy games which have little use for such RPG elements. Since making Hexkit work well for those kinds of games is already quite a task, I decided to cut the old item system entirely.

In their place, Hexkit 4.1 introduced "upgrades" – entities that always represent intangible faction bonuses and never appear on the map. Upgrades can implement predefined faction traits in a historical scenario or researchable technologies in a 4x game.

While thus simplifying the implicit game semantics, this release also expanded the range of options exposed in Hexkit Editor, namely in the area of variable modifiers. Any entity may now define multiple modifiers for the same variable, each with an explicit target: the entity itself, its owner, any units in the same location or within a given range, either with the same or a different owner. Terrains with appropriate modifier targets now take on the old role of non-recoverable placed items. Faction resource modifiers only affect the faction's own resources while upgrades provide any desired bonuses to the faction's units – or penalties to enemy units.

Simultaneously, the entire variable modification system was "locked down" and is no longer customizable by rule script code. This was a technical necessity due to the complex caching mechanism for ranged variable modifiers, but it's also a general trend I try to follow. User feedback gave me the impression that writing rule script code is not very popular, so trading off more options in Hexkit Editor for less flexibility in rule scripts seems like a good idea.

1.4 The End of Hexkit

Active development of Hexkit ceased with version 4.2. This release added the "Battle of Poitiers" demo scenario and a number of small improvements required for that scenario.

The current version is reasonably solid and complete, but I feel that putting further effort into this project is no longer worthwhile. First, consider the three broad genres of strategy games for which Hexkit might be suitable:

Grand Strategy (4x) Games, exemplified by the "Roman Empire" demo scenario —
: Two words: *Civilization IV* (and now its sequel, *Civilization V*). An almost perfect grand strategy game, with brilliant game design, reasonably strong AI, tremendous production values – and moddable to boot. Attempting to compete with this juggernaut is hopeless, not to mention many other popular commercial products in this genre (*Europa Universalis, Total War, Galactic Civilizations, Rise of Nations,* etc.).

Historical Tactical Warfare, exemplified by the Crécy and Poitiers demo scenarios —
: There is little competition in the field of pre-modern combat simulations, but sadly also little use for maps divided into hexagons. Ideally, we should precisely simulate the physical interaction of individual soldiers and formations, which is just not possible on a coarse grid. That leaves World War 2 and later, but those eras are so extensively covered by commercial wargames that I would rather avoid them.

Original Strategy Games, very poorly exemplified by the "Troll Chess" demo scenario —
: Please download Hexkit and make one! Sadly, I'm not the creative game designer type of guy, so I must leave this option to someone else…

There is another reason why Hexkit became a dead end: You must be a C# programmer to create anything worthwhile with Hexkit. As it turns out, most people who are into scenario design are not programmers, and you really cannot do very much with Hexkit unless you can write C# code on a fairly professional level.

I had originally intended to move all game rules from C# script files to the GUI editor, but now I doubt this is even possible for a truly universal game engine. At least, it would require far too much additional effort to stake on a vague hope. Hexkit also lacks certain desirable features such as mini-maps or unit actions other than attack/move, and while the implementation should be straightforward it still requires fluency in C# and WPF.

Aside from being a programmer, you also need to be an artist if your scenario requires any units or terrains that aren't covered by the provided tile set – another big obstacle for hobbyists.

Throughout the development of Hexkit, I was unable to find any tile sets that covered settings other than medieval fantasy dungeons (i.e. Roguelikes) with sufficient quality and without copyright restrictions. In retrospect, using bitmap art rather than abstract symbols was a mistake. The tile set allowed reasonably pretty demo scenarios but drastically limited the scope of possible scenario settings – a poor trade-off for a flexible game engine.

Meanwhile, I intend to pursue the study of historical tactical warfare with an entirely different game engine that allows an exact simulation of individual soldiers and entire formations, with real-world positioning and continuous movement. This is going to require lots of groundwork in computational geometry which will appear in ongoing Tektosyne releases.

Chapter 2: Hexkit Game

This chapter covers the main program, Hexkit Game, also simply called Hexkit.

— The first two sections describe managing games – starting, loading, saving, replaying – and how to set up the human or computer players that participate in a game.

— The next three sections deal with the elements of a Hexkit scenario: the competing factions, the game map and its contents, and the numerical variables (also called "statistics") associated with factions and other game entities.

— The last section lists the commands that a faction (or rather, its controlling player) may issue to manipulate the current game situation.

Throughout the discussion, user interface and technical issues are noted as appropriate, but please consult the Hexkit online help and the ReadMe file for more details on these subjects.

2.1 Managing Games

Hexkit offers the usual file management commands familiar from other games and applications, though with some unique quirks:

— All games are based on *scenarios*, which are immutable as far as Hexkit Game is concerned – see "Hexkit Editor" on page 32 for information on how to change them.

— Saved games are stored as a *history* of executed commands, and therefore require the underlying scenario to load.

— Thanks to this save game format, Hexkit offers an unlimited *replay* facility that is particularly useful in e-mail games and for analyzing a computer player's turn.

2.1.1 Starting a New Game

Hexkit is a strategy game *system*, not a self-contained game. While Hexkit does provide a set of default rules, it does not define fundamental parameters such as the game map, the quality and quantity of available units, or the number of competing factions.

All this data is supplied by a *scenario* which may also change or extend the default rules to a large degree. To start a new game with Hexkit, you load a scenario file. There are several ways to do this, as described in the online help:

— Click the Start New Scenario button in the express menu.

— Choose the File → New menu item from the main menu.

— Drag & drop the scenario file on the Hexkit Game application or its icon.

— Specify the name of the scenario file as a command line parameter.

Once Hexkit has digested the scenario file, the About Scenario dialog will appear and show some introductory text and copyright information. Next, the Player Setup dialog allows you to inspect and change the default player assignment.

You can later use the Info → About Scenario and Game → Player Setup menu items, respectively, to display these dialogs again. Please refer to "Managing Players" on page 12 for more information on player setup.

Note. Any errors that occur while attempting to start a new game should be reported directly to the scenario author. I don't mind receiving your error e-mails as well, but I'll only fix errors that were caused by Hexkit itself, rather than by the scenario – unless I created the scenario!

Scenario File Locations

Scenarios are stored as XML files (see "Scenario Structure" on page 32). Their default location is the Scenario folder in your Hexkit installation directory. They can be viewed and changed with the companion program, Hexkit Editor (see "Hexkit Editor" on page 32).

When starting a scenario that is distributed over multiple subsection files, be sure to open the file containing the Scenario section. This should be an XML file located in the Scenario folder itself. Hexkit Game cannot directly read subsection files, which are usually located in subdirectories of the Scenario folder.

Debugging Scenario Files

If the File → New or File → Open command fails to start a scenario, a temporary XML file named Scenario.Start.xml will remain in the Hexkit subdirectory of your (My) Documents folder.

Scenario designers will need this file for debugging because Hexkit reports scenario error locations relative to the temporary file, not to the original section files. The temporary file will be overwritten by the next New or Open command, and deleted automatically when a scenario was started successfully.

While playing a game, you can use two Debug menu items to recreate scenario files from memory data. Debug → Save Scenario File generates a monolithic scenario file from the original scenario data, and Debug → Save Map Contents creates an Areas subsection file from the current map contents that includes all changes since the game was started.

2.1.2 Opening a Saved Game

Once a game has been saved, as described in "Saving Your Game" on page 10, you can resume it at a later time by opening the save game file. Again, there are several options to do this:

— Click the Open Saved Game button in the express menu.

— Choose the File → Open menu item from the main menu.

— Drag & drop the save game file on the Hexkit Game application or its icon.

— Specify the name of the save game file as a command line parameter.

Please refer to the online help for details on these actions. Note that restoring the saved game situation may take a while, especially for big scenarios and long games.

Opening a saved game will abort the current game, if any. You will be prompted to save the current game first if any commands were issued since the last automatic or manual save.

Finding the Scenario File

Saved games do not include any scenario data – instead, they specify the scenario file they are based on. By default, scenario file paths are relative to the Scenario folder of the Hexkit installation directory, allowing saved games to be exchanged between different machines.

However, it is possible play a scenario that does not reside in the standard Scenario folder or one of its subdirectories (if any). In that case, the Save command will store the absolute path to the scenario file, and the Open command will first try to open the specified absolute path, and then search the Scenario directory for the file name itself, ignoring any path specification.

In either case, Hexkit automatically searches the standard Scenario folder and all its subdirectories (if any) for the file name itself, ignoring any path specification, if the scenario file could not otherwise be found. If this search fails as well, the saved game cannot be opened.

Verifying the Scenario File

Saved games also include a check sum for the scenario file they are based on. Hexkit will display a warning message if the check sum stored in the saved game differs from the scenario file found by the mechanism outlined above, and will ask you to continue or abort the operation.

Note. You may attempt to load a saved game despite a check sum mismatch, but you might experience unpredictable program errors if you choose to do so. This option is mainly intended for scenario developers who might wish to resume a test game after fixing some minor issue.

2.1.3 Saving Your Game

You can *manually* save the current game to disk using the File → Save menu item. The default file name consists of the current scenario name and the current turn and faction index, but you can enter a different file name if you wish.

Automatic Saves

The game is also saved *automatically* on various occasions. There are four types of these so-called "autosave" files, each of which is always created with the same file name.

Autosave.xml.gz — Created whenever a human player ends his turn, which is either you or another "hotseat" player on the local machine. This file records all commands up to, but not including, the final End Turn command (see "End Turn" on page 26). Reload this file to undo your last End Turn command.

ComputerSave.xml.gz — Created whenever a computer player ends its turn. This file records all commands issued by the computer player, *including* the final End Turn command. This file is useful mainly when analyzing the AI in games without human players.

EmailSave.xml.gz — Created whenever the game is about to be dispatched to a human player on a remote machine during a PBEM game. This is the same file that is attached to the PBEM message, so you can send it manually if automatic dispatching fails.

Session.Debug.xml.gz — Created when the program shuts down due to an unhandled error, along with FatalError.txt (containing the error message) and Scenario.Debug.xml (containing the entire current scenario).

This feature is mainly intended for developers, as the game is likely to crash again if you attempt to load this file. You will be asked to e-mail these files to me if you're running an unmodified distribution build of Hexkit.

Save File Locations

Saved games are stored as XML files. By default, all saved games are compressed with the GZip algorithm to save disk space and e-mail transmission time, hence their .xml.gz file extension.

However, you can also save games as uncompressed XML files with the usual .xml extension. This allows you to examine a saved game with an external XML editor, for instance.

The default location for saved games is the Hexkit\Games subdirectory of the current user's (My) Documents folder. Error files are created one level higher, in the Hexkit subdirectory.

You can specify a different folder when saving manually. Hexkit will create all autosave files in the default location, however.

Note. Because the current Windows user does not change while other players are taking their turns on the same machine (in "hotseat" mode), autosaves for *all* local players are written to the same folder as long as you are logged in. New autosaves will overwrite all earlier ones of the same type, including those of preceding players.

2.1.4 Replaying Your Game

You can use the Replay menu to start an interactive replay of previous game turns. An ongoing replay can be paused or aborted at any time. You may choose between four speed settings, and quickly skip forward to the next faction at any time.

There are several ways to start an interactive replay:

— Replay All Turns and Replay From Turn are intended for analyzing all or part of an ongoing game. This is particularly helpful for scenario developers.

— Replay Last Turn provides recipients of a PBEM game with a convenient way to study the recent actions of their remote opponents before entering their own commands.

— Computer players automatically replay any commands they have issued before yielding control to the next player. All controls and settings of the Replay menu are available in an automatic replay, just as in a manually initiated replay.

Most user interface elements remain fully usable during a replay, but some menu options automatically stop an ongoing replay. The online help notes where this is the case.

The Command History

Hexkit's replay function relies on the fact that games are saved as a complete "history" of game commands, not just a snapshot of the current game situation. The save file stores the path to the scenario on which the game is based, followed by a list of all commands that were executed since the start of the game. (Various game-specific settings are saved as well.)

When a saved game is opened, Hexkit internally "replays" all stored commands. This recreates the original game configuration that is finally presented to the user. An interactive replay differs from this process only in that the execution of each individual command is shown.

The history of all executed commands is available through the Info → Command History menu item (also see "Commands" on page 23).

Recording saved games as a sequence of commands is the norm for chess programs and other computerized board games. I also used this method in my first freeware game, *Star Chess*, which was inspired by computer chess. Many commercial games now provide an optional replay mode, too, but this usually requires creating a separate replay file in addition to any saved games.

2.2 Managing Players

Players and factions are fundamentally different concepts in Hexkit. Factions are the "sides" of a game, like black and white in chess. Only factions own units, conquer the map, and win or lose the game. We'll take a closer look at them in "Factions" on page 14.

Players, on the other hand, *control* factions. Any faction can be controlled by a human player (see "Human Players" on page 12) or by a computer player (see "Computer Players" on page 13). Use the Game → Player Setup menu item to view and change the current player assignment.

It is important to realize that only factions – not players! – are agents in terms of gameplay mechanics. The game engine does not know which player controls a faction, only whether it is a human or computer player, so as to allow for a scripted computer opponent.

Otherwise, Hexkit does not care which player controls a faction, or even whether multiple factions are controlled by the same player. In particular, factions are *not* automatically allied just because they are controlled by the same player, nor do they share any information or resources, except as defined by the game rules.

Since the current Hexkit version lacks any sort of diplomacy, all factions are always at war with each other, and that applies even if they are controlled by the same player. You can assign all factions to the same human player to play a Hexkit scenario in traditional boardgame solitaire mode, or you might assign all factions to the same computer player to watch a self-playing game between opponents using the same algorithm and options.

2.2.1 Human Players

A particular human player represents an individual, commonly of the species *homo sapiens*, which may control one or more factions in the current game.

Hexkit offers two different modes for human players: "hotseat" and play-by-email (PBEM). Hotseat mode means that all human players are crowded around the same computer system and change seats as their factions are activated. To use this mode, simply enter no e-mail address for any human player in the Player Setup dialog.

PBEM mode means that all or some human players are playing on different machines, and saved games must be sent back and forth between them via e-mail. When a faction is activated that is controlled by a remote human player, Hexkit will create a PBEM message to that player. The message contains the name of the current scenario, the current turn and faction indices, the player and faction that are supposed to become active, and a current saved game named Email-Save.xml as an attachment.

You can mix hotseat and PBEM mode by assigning several players to the same e-mail address. Hexkit will show a hotseat message, rather than generate a PBEM message, if two consecutive human players share the same e-mail address. Note that you cannot simply leave some address fields blank; if any of the human players participating in a game have a valid e-mail address, all must have one.

2.2.2 Address Search

When a PBEM game is started, Hexkit searches the names of all human players that control one or more factions for the current Windows user name. The e-mail address associated with that name is interpreted as the address of the local human player, or group of players.

If no such name is found, Hexkit does not make any assumptions about local players but simply dispatches a PBEM message as soon as the first human-controlled faction is activated.

PBEM mode does not currently incorporate any kind of identity check. Whoever opens the saved game attached to a PBEM message is assumed to be the player controlling the next active faction, and the e-mail address associated with this player is interpreted as the local address until the next PBEM message is generated.

Note. Future Hexkit versions may include encrypted passwords to ensure that only the intended recipient can open the saved game attached to a PBEM message.

2.2.3 Computer Players

Hexkit (potentially) supports a variety of different computer player algorithms, each with its own set of options. Each computer player corresponds to a particular combination of algorithm and optional settings. There is one predefined computer player for every faction in a scenario, allowing you to assign a unique combination of algorithm and optional settings to each.

You can transfer your own faction to a computer player at any time with the Game → Player Setup dialog. You can also interrupt an active computer player and take control of its faction in the same fashion, or more conveniently with the Game → Stop Computer command.

Computer players perform their calculations in the background. The user interface of Hexkit Game remains mostly functional during this time, although you will not be able to issue commands. Once a computer player has finished its calculations, the resulting commands will be replayed on the map view, as described in "Replaying Your Game" on page 11.

A dedicated field on the status bar shows the name of the computer player algorithm that is currently executing. If the algorithm runs for more than a few seconds, this status bar field may be updated periodically to show other information, such as the currently considered unit.

This about covers computer players from a user perspective. Please refer to "Computer Players" on page 139 for details on how the various algorithms are implemented.

Note. Computer players are still under development. There is currently only one basic computer player algorithm, called "Seeker". This algorithm does offer a few optional settings, but they do not significantly alter its playing strength.

2.3 The Game Map

Every game is played on single *map*, which is the equivalent of a traditional game board. The map consists of a rectangular grid of squares or hexagons, as with many turn-based wargames. We usually refer to these squares or hexagons by the generic term *map sites*.[4]

The size of the map and the shape of its elements are defined by the scenario on which a game is based. The map size and element shape never change during the course of a game. There is a hardcoded maximum of 10,000 sites along each map dimension, although the width and height of real maps should hardly reach one tenth of that limit.

Each site may be owned by a faction which we'll discuss subsequently (see "Factions" on page 14). A faction gains ownership of a site by capturing it with a unit, or due to special rules defined by the scenario.

4. Unfortunately, the English language lacks a generic term for "place on a game board", such as "Feld" in German or "case" in French. So I settled for "map site" as a shorter alternative to "map location".

   From a user's point of view, both terms are interchangeable and are treated as such in this chapter. They are distinct in Hexkit's programming model, however: a Site holds all the contents of a square or hexagon, and has an immutable two-dimensional Location as one of its properties.

Default victory and defeat conditions are associated with owning all or no sites on the map, respectively. The View → Show Owner menu item shows territory ownership for all factions with a color overlay on the map view.

Sites are not merely abstract locations. Each site contains at least one background terrain, plus any combination of additional terrains, units, and special effects. These game objects are collectively called "entities" and will be described in "Entities" on page 17.

When you select a map site containing one of your units that can attack or move, you'll see a number of highlighted locations around the selected site. These are the implicit targets for right-click commands which we'll describe in "Right-Click Targeting" on page 29.

2.4 Factions

Each of the sides contending for victory is represented by a *faction*. Every (surviving) faction is controlled by a human or computer player (see "Managing Players" on page 12), but the identity of a faction's controlling player is not relevant to the game rules.

Use the Info → Faction Status menu item to show information on all surviving factions in the game, and the Info → Faction Ranking menu item to compare their various possessions, both for the current turn and as they developed over the course of the game.

2.4.1 Possessions

Hexkit is a materialistic game, so the status of a faction is defined by its current possessions, which may include any or all of the following:

Territory — A collection of map sites. Under the default victory conditions, a faction must own at least one map site to stay in the game. See "The Game Map" on page 13.

Units — The "game pieces" or "counters" in board game terminology that constitute a faction's army. They are usually placed on the map but newly created units may be held in an off-map inventory, awaiting placement. See "Units" on page 17.

Terrains — Grass, towns, ocean, etc. Most terrain is a fixed part of some map site, but factions may be able to build certain terrain types, such as ditches, and place them on the map. Placed terrain belongs to the site's owner. See "Terrains" on page 18.

Effects — Certain permanent local effects such as fog may appear on a map site, like terrains. Factions cannot build such effects or store them off-map, but effects may modify a faction's resources when placed on its territory. See "Effects" on page 18.

Upgrades — Intangible faction bonuses, such as technological advances. Upgrades are always kept in a faction's off-map inventory. They may affect the faction's resource stockpile or the condition of some or all its units. See "Upgrades" on page 19.

Resources — Gold, fuel, ammunition, and so on. Factions start out with a certain amount of resources and may receive another amount for free each turn. Owned terrains and upgrades usually generate additional income, whereas units typically cost resources to build and maintain. See "Resources" on page 20.

2.4.2 Home Site and Alternatives

The scenario may also define a *home site* for each faction which could represent its capital, its headquarters, or simply a point near its initial deployment area. Losing control of the home site disables certain upgrade benefits and may prevent a faction from placing units or terrains on the map, depending on the placement rules defined by the scenario.

Moreover, the map view selects and centers on the home site whenever the faction becomes active. Since home sites are optional, Hexkit defines a list of alternative initial locations, which are considered in the following order:

— The faction's home site, if one is defined.

— The faction's first unit that is capable of attacking and/or moving, if one exists.

— The faction's first unit that is placed on the map, if one exists.

— The faction's first owned map site, if one exists.

If none of these alternatives exist, the currently selected map site will simply remain unchanged when the faction becomes active at the start of its turn.

2.4.3 Turn Sequence

The game proceeds in full turns in which each faction is activated once, in a scenario-defined order. A faction may issue commands only while it is active.

When a new game is started, and whenever a faction ends its turn during a game (see "End Turn" on page 26), the following turn sequence begins:

1. Check all factions for resignation and defeat conditions, and remove any resigned or defeated factions from the game. The game ends if less than two factions remain.
2. Activate the next surviving faction, as determined by the faction order shown in the Player Setup dialog. The turn counter is increased whenever the first faction in this list again becomes active.
3. If the active faction is controlled by a remote human player (PBEM mode), Hexkit sends a PBEM message to that player and closes the game on the local machine. The game will continue with the next step when opened on the remote machine.
4. If the active faction is controlled by a human player on the local machine, Hexkit shows a message announcing the active faction and its controlling player, unless there is only a single human player in the game.
5. If the active faction is controlled by a computer player, Hexkit starts a background thread for the computer player algorithm and shows its name in the status bar.
6. Hexkit automatically issues the obligatory Begin Turn command for the active faction (see "Begin Turn" on page 25).
7. Check the active faction for victory conditions.
   The game ends if the faction is victorious. Otherwise, the game now accepts input by a human player, or performs AI calculations for a computer player.

The turn sequence starts again when the active faction ends its turn.

Holding on to Victory

The last step in the turn sequence is the *only* time when Hexkit checks for a faction victory, excepting only the special case that all factions but one were eliminated in the first step.

Since this check occurs at the *beginning* of a faction's turn, it follows that a faction that has just acquired the last victory location or the like must hold on to its life and assets for another full game turn. Needless to say, all other factions will attempt to prevent the impending victory.

This sequence prevents "cheating" victories known from some wargames where an unarmed transport on its last drop of fuel captures the last victory location and instantly wins the game. Always make sure you can hold on to your vital possessions for at least one full turn!

Waiting for Defeat

Hexkit only registers the defeat of a faction at the beginning of the turn sequence, so you won't get an immediate reaction from the game after conquering the last bit of an enemy's territory. Hexkit will report the enemy faction's defeat once you've ended your turn.

2.4.4 Victory and Defeat

A game proceeds along the turn sequence until any faction meets a *victory condition*. Meanwhile, any faction is removed from the game as soon as it meets a *defeat condition*.

Hexkit offers built-in support for conditions based on the following game parameters:

— The number of map sites owned by a faction.

— The number of units owned by a faction (see "Units" on page 17).

— The total strength (manpower or health) of all units owned by a faction.

— The amount of a specific resource owned by a faction (see "Resources" on page 20).

— The number of elapsed game turns.

A scenario may define multiple victory and/or defeat conditions, each with its own numerical *threshold* that triggers the condition. The first faction to reach the threshold of any one of its victory conditions wins the game. Any faction that reaches the threshold of any one of its defeat conditions is eliminated from the game.

For all victory conditions, rising above the threshold also triggers the condition. Conversely, for most defeat conditions, falling below the threshold also triggers the condition. This does not include defeat conditions that are based on a turn count, for obvious reasons.

Conditions and thresholds vary between scenarios and individual factions. Use the Info → Faction Status → Conditions dialog page to view the conditions that apply to a given faction.

Resource Conditions

Resource-based conditions differ from other types in that they are tied to a resource rather than a faction, and thus apply to all factions equally. However, not all resources are accessible to all factions. Factions that cannot acquire a certain "victory resource" will be unable to meet the corresponding victory condition, and must try to win by some other means. Conversely, factions that cannot acquire some "defeat resource" will never lose due to that condition.

Default Rules

If a scenario does not specify any custom conditions, the following default rules apply:

— A faction is defeated if it loses ownership of its last map site.

— A faction wins by default if all other factions are defeated, for whatever reason.

— Consequently, a faction wins the game if it defeats all other factions by capturing all of their map sites with its units (see "Move Units" on page 27).

— A faction can deliberately quit the game in a hopeless situation, using the Resign Game command (see "Resign Game" on page 28).

All remaining units, upgrades, and unplaced terrains of a defeated faction will be removed from the game, and all of its map sites will revert to unowned status.

2.5 Entities

Entity is a common game design term that denotes an individual object in the game. In our case, this includes units, terrains, effects, and upgrades; also known as the four *entity categories*. Each category comprises an unlimited number of *entity classes* that are defined by the scenario author. An entity may be located on the map or in a faction inventory.

Use the Info → Entity Classes menu item to show all entity classes defined by the current scenario. Table 1 shows the four categories and some sample classes for each category.

Category	Sample Classes	Section Reference
Units	Knight, wizard, space marine	"Units" on page 17
Terrains	Grass, ocean, mountain, town	"Terrains" on page 18
Effects	Flying arrows, fire, smoke	"Effects" on page 18
Upgrades	Gunpowder, medicine, mining	"Upgrades" on page 19

Table 1: Entity Categories

Entities are usually associated with one or more numerical variables or "statistics", including attributes and resources, which we'll discuss in "Variables" on page 19.

All entities of the same class all start out as identical clones by default, although the scenario designer may assign different names, images, or statistics to specific entities. Moreover, the statistics of some entities (notably units) are likely to be affected by changing game situations, such as a warrior losing health or gaining experience in battle.

The Command → Manage Entities menu item shows lists of all entities owned by the active faction. Use the Info → Faction Status → Assets dialog page to show the equivalent lists for other factions. The Info → Faction Status → Classes dialog page shows the entity classes that are available to each faction, which includes all classes that the faction can build, as well as the classes of all entities that the faction already owns.

2.5.1 Units

Every map site may contain an arbitrary number of units. Every unit must belong to a faction, and all units in the same location must belong to the same faction. On the map view, units appear on top of all terrains but below any effects.

The map view may show a colored flag next to each unit stack, indicating its owner, size, and status. Please see the online help for the View → Show Flags menu item for more details.

The map view may also show a resource gauge (a.k.a. "health bar") below each unit stack, indicating the depletion status of some resource – typically a unit's combat strength. Please see the online help for the View → Show Gauges menu item for more details.

Factions may also hold unplaced units in their off-map inventories. They may be placed on the map at any time, but once placed may not be returned to the off-map inventory.

Use the Game → Cycle Unit menu items to cycle through all active units of your faction. These are all the units that are placed on the map and which can still take an action during the current turn, i.e. attack, move, or both.

The actions that a unit may perform are determined by its abilities (see "Abilities" on page 22). Units typically also have a set of attributes that further define their possible actions and combat performance, and a set of resources that determine their current strength or health, and how expensive they are to create and maintain. Other entities that are owned by the same faction or placed on the same map site may modify these variables.

Unit Names

Units show an individual name by default that is constructed by appending an index count to their class name (e.g. "Pikeman #34"). Units that factions cannot build merely show their class name by default (e.g. "Pikeman"). In either case, the scenario designer may assign another unspecific or individual name instead (e.g. "George the Pikeman"), as with all entities.

2.5.2 Terrains

Every map site contains exactly one background terrain, such as ocean or grassland. On top of this background terrain, an arbitrary number of foreground terrains may be placed, such as reefs in the ocean or flowers on grassland. Other physical objects such as ammunition containers or fuel tanks are also represented by foreground terrains.

On the map view, terrains appear below any other entity in the same location. Newly created terrains may be held in a faction's off-map inventory. Terrains have no effect while unplaced, and cannot be moved back into faction inventory once placed.

Terrains follow the ownership of their map site. All terrains in a site that belongs to a faction contribute to that faction's resources and may modify the statistics of its units. Regardless of their owner, terrains may also modify the variables of any units in and around their location.

A site can be captured by an occupying unit if both the unit and at least one local terrain have the Capture ability (see "Abilities" on page 22). Sites without such terrain change their owner only through direct manipulation by the rule script.

2.5.3 Effects

Every map site may contain an arbitrary number of effects which are shown on top of any other entity in the same location. Effects are always placed on the map and never appear in faction inventories, although they do change ownership along with their site.

Permanent effects such as fog or smoke behave exactly like terrains and can likewise modify the variables of their owning faction and of any nearby units. However, effects are often used for transient decorations such as flying arrows or explosions that only briefly appear on the map without actually influencing gameplay in any way.

2.5.4 Upgrades

Upgrades represent intangible faction bonuses that are never placed on the map but instead kept in a faction's off-map inventory. Some scenarios may not feature any upgrades at all – they are not required by the default rules.

Upgrades may modify the resources of their owner and the statistics of its units, either in and around the home location or anywhere on the map. If an upgrade's effect has a limited range, it will cease to exert that effect when the faction loses control of its home location.

2.6 Variables

Factions and entities may be associated with certain numerical variables, the equivalent of the so-called "statistics" in role-playing games or wargames. There are two fixed *variable categories*, each of which comprises an unlimited number of variables defined by the scenario author.

Table 2 shows the two categories and some sample variables for each category. All units and some other entities also possess ability flags, as described in "Abilities" on page 22. Taken together, an entity's variables and abilities are known as its *properties*, which is why variable values are usually shown under a "Property" list header.

Category	Sample Variables	Section Reference
Attributes	Attack, defense, range, speed	"Attributes" on page 20
Resources	Gold, iron, wood, health	"Resources" on page 20

Table 2: Variable Categories

Use the Info → Variables menu item to show all variables defined by the current scenario. The variables associated with a particular faction, entity class, or entity are shown in the corresponding information displays: the Info → Faction Status → Resources dialog page for factions, the Info → Entity Classes dialog for entity classes, and the data view to the right of the main map view for individual entities. You may also use the View → Show Variable dialog to show aggregate variable values right on the map view, either as numbers or lighter and darker shading.

A variable may appear as two different values: a *basic value* which is simply called "attribute" or "resource", and a *modifier value* which is called "attribute modifier" or "resource modifier". Any faction or entity may be associated with either or both variants of a given variable.

The initial basic and modifier values for a particular faction or entity are set by the scenario author. Both values may change throughout the game, although only basic values do so under the default rules. Once an entity has been created, its variable values may change independently of other entities based on the same class.

The legal range for the values for each variable is also defined by the scenario. The default is 0 through 1000. Note that these limits apply only to basic values, and only to the final results of variable calculations. The legal range for modifier values is ±100,000,000 which is also the maximum range for basic values. No restriction whatsoever is imposed on intermediate results.

Hexkit automatically manages variable modifiers and defines a few fundamental gameplay mechanisms that use basic values, including unit construction and upkeep, unit movement, and some aspects of combat. However, the scenario designer chooses the names of the variables used by these mechanisms, and usually also provides additional or refined custom rules. The scenario documentation should inform you of the actual meaning and use of its variables.

2.6.1 Attributes

Entities – typically unit classes – may be assigned *basic values* for one or more attributes such as attack, defense, or speed. The scenario author decides the gameplay effect of each attribute, and possibly creates new custom rules for some of the attributes. Hexkit supports several standard attributes that you'll find in many scenarios, namely the attack and movement range of units and the difficulty and elevation of terrains.

Entities may also be assigned *modifier values* for one or more attributes, each with a variety of possible targets. These modifiers may affect unit attributes as follows:

— An entity's *self-modifier* affects the entity's own attributes. This is the only attribute modifier that applies to entities other than units.

— An entity's *local unit modifiers* affect the attributes of all units in the same map site. For units, a local modifier is added to any self-modifier for the same attribute.

— An entity's *ranged unit modifiers* affect the attributes of all units within a given range around the entity's map site. The range may be infinite to affect all units on the map.

— Local and ranged unit modifiers may distinguish between units with the same owner as the entity that defines the modifier, and units with a different owner.

— Upgrades apply their local and ranged unit modifiers to their owner's home site and the surrounding map locations, respectively. Such modifiers have no effect if there is no home site, or if the upgrade's owner has lost control of its home site.

— However, an upgrade's ranged unit modifiers always apply if they are defined with an infinite range, regardless of the status of the owner's home site.

— The attribute modifiers of units and terrains that are held in faction inventories have no effect whatsoever until those entities are placed on the map.

All attribute modifiers take effect immediately when a unit is placed or moved, when a terrain or effect is added to or removed from a site, or when an upgrade is obtained.

Matching modifiers from different sources are cumulative. Positive and negative modifiers to the same attribute may cancel each other out.

2.6.2 Resources

Factions and entities may be assigned *basic values* of one or more resources such as gold, wood, oil, and iron. Hexkit recognizes two standard unit resources, combat morale and strength (i.e. manpower or health), although their exact role must be defined by custom rules.

— A faction's resources usually represent the amount of various goods it has stockpiled, but may also specify *victory points* that are accumulated towards a victory or defeat condition (see "Victory and Defeat" on page 16).

— The scenario may require that some faction resources are *reset* to their initial scenario values before adding modifiers at the beginning of each turn. Such resources might represent perishable goods or per-turn victory conditions. They are called *resetting* resources, as opposed to the normal case of *accumulating* resources.

— An entity's resources represent either the amount of various goods in its possession, such as the number of arrows in an archer's quiver, or else some aspect of the entity's status, such as the manpower of a larger military unit.

— The scenario may require that some entity resources are *limited* to their initial values, i.e. those defined for the corresponding entity class. This is useful for resources such as health that represent a maximum capacity. Adding modifiers to such resources will not increase them beyond their initial scenario values.

Factions and entities may also be assigned *modifier values* for one or more resources. These modifiers may affect faction and entity resources as follows:

— A faction's resource modifier affects the faction's own resources (and nothing else).

— An entity's self-modifier likewise affects the entity's own resources. This is the only resource modifier that applies to entities other than units.

— An entity's *owner modifier* affects the resources of its owning faction. For example, units often define negative owner modifiers that represent their maintenance costs.

— An entity's local and ranged unit modifiers affect the resources of local and nearby units, in the same way as described for attribute modifiers.

— The resource modifiers of units and terrains that are held in faction inventories have no effect whatsoever until those entities are placed on the map.

Entity resources and accumulating faction resources are updated by the Begin Turn command (see "Begin Turn" on page 25). Resetting faction resources are updated by the End Turn command (see "End Turn" on page 26), and also once at the start of a new game.

Units are disbanded automatically if their maintenance costs cannot be covered, i.e. if adding their resource modifiers to their owner's resources would cause one or more of the latter to fall below their legal minimum values (see "Destroy Entities" on page 26).

Resource Transfer

Resource modifiers generate *unlimited* production or spending for their recipients. For example, if a gold mine has an owner modifier of +5 gold, it will add 5 gold units to its owner's resources every single turn, for as long as the game lasts (unless the rule script changes the modifier).

Another way to change the basic value of a faction or entity resource is a *resource transfer* from another entity. Here, the same amount of a given resource that is added to the recipient is also subtracted from the provider, and therefore limited by the latter's resource stockpile.

On the receiving side, the transfer is capped by the legal minimum or maximum value for that resource. Hexkit transfers as much of a resource as possible, but no more than the recipient can accept without exceeding the resource's capacity.

— Factions may receive resource transfers from their upgrades.

— Units may receive resource transfers from terrains and effects in the same map site.

— Entities that have provided an automatic resource transfer are *depleted* if the transfer leaves them without any resources. Depleted entities that do not define any modifier values may be automatically removed from the game.

— The scenario determines which entities may transfer resources, and whether they are removed from the game when they are depleted.

— The default rules never transfer resources from units, or to entities other than units.

Resource transfers are performed during regular resource updates (see above), after all modifiers have been applied to the recipient's resources. In the case of factions, transfers are performed while updating accumulating resources only.

Note. The transferred amount may be negative as well as positive – an upgrade with a negative amount of a given resource will *drain* its owner's resource by that amount! In this case, too, the draining will continue until the upgrade's amount of that resource reaches zero.

Build Resources

In addition to the basic and modifier values described above, an entity class may specify *build resources* which represent its construction costs.

When a faction builds an entity, the build resources defined by its class are subtracted from the faction's matching resources. A faction cannot build an entity if that would cause one or more of its resources to fall below their minimum values. Moreover, a faction cannot build an entity if any of the required build resources do not appear in its resource stockpile.[5]

Build resources are completely unrelated to regular entity resources. An entity might track basic and/or modifier values for all, some, or none of the build resources defined by its class. In any case, the expended build resources are not transferred to the entity; they simply vanish.

2.6.3 Abilities

Abilities are non-numerical indicators for what a specific entity can do, or what can be done with it, at a given time. Some abilities are associated with entity classes, others with individual entities.

Each ability is either present or absent for any given entity or entity class, indicating that some kind of action is legal or illegal. Even if an ability is present, there may be additional requirements that prevent or restrict the actual execution of such action. For example, the Move ability is useless if there are no reachable sites within the unit's movement range.

The Info → Entity Classes dialog shows the abilities of an entity class following its variables, and the data view next to the main map view shows an entity's abilities preceding its variables.

Common Abilities

Unit, terrain, and upgrade classes may all have the Build ability. This is a "pseudo-ability" since it is really associated with factions, not with entities or entity classes. It indicates whether a faction can build new entities of a specific class, provided the faction has the required resources.

The Build abilities of all entity classes available to a faction are shown on the Info → Faction Status → Classes dialog page. The data view also shows a Build ability for each unit, indicating whether the unit's *current owner* can build new units of that class.

Unit Abilities

There are three additional unit abilities, two of which apply to individual units while the third applies to entire unit classes.

Attack — An individual unit may still attack during the current turn. This ability is restored at the beginning of the next turn.

Move — An individual unit may still move during the current turn. This ability is restored at the beginning of the next turn.

Capture — Units of this class capture any map site they occupy (but see below), and add them to their owner's territory. Units without this ability cannot capture sites.

5. This is different from victory and defeat conditions where the complete absence of a resource means that the corresponding resource condition does not apply.

The Capture ability only applies to the map location on which a unit ends its move. Any other locations that the unit may have traversed in the same movement are not captured.

Some defensive unit classes may restrict the Attack ability so that units automatically lose their Attack ability at the start of their turn. Such units can never initiate an attack, only defend against attacks during other factions' turns.

Terrain Abilities

Terrain classes may have the Capture ability which mirrors the eponymous unit ability. This is a passive ability that can be described as "the ability to be captured by units."

Map sites can only be captured by a unit with the Capture ability if they contain at least one terrain of a class that also has the Capture ability. Typically, terrain such as ocean that cannot realistically have an owner will lack this ability; and so will any abundant and worthless terrain that a computer player should not attempt to capture.

Terrain classes may also have the Destroy ability. Generally, a faction may destroy owned units at will, but no other entities. Terrains with the Destroy ability can be destroyed like units; terrains without this ability are indestructible like upgrades.

Note. Background terrains never have the Destroy ability. Deleting a placed background terrain would violate the fundamental requirement that all map sites have a background terrain, and unplaced background terrains in a faction inventory don't make a lot of sense.

2.7 Commands

When a faction under your control is active, you can issue commands using the Command menu. Most commands affect your units or other entities, but some perform global actions.

Table 3 shows the available game commands with a short description of each command. The rest of this section briefly describes each command and its effects, while section "Command Details" on page 28 talks about a few related game mechanics in greater depth.

Command	Description	Section Reference
Attack	Attacks a specific map site with one or more units.	"Attack Site" on page 25, "Combat Mechanics" on page 28
Automate	Container for arbitrary actions that are executed automatically.	"Automated Commands" on page 30
Begin Turn	Begins the turn for the active faction.	"Begin Turn" on page 25
Build	Builds new entities, and optionally places them on the map.	"Build Entities" on page 26
Destroy	Destroys owned entities.	"Destroy Entities" on page 26
End Turn	Ends the turn for the active faction.	"End Turn" on page 26
Move	Moves the selected units to another map site.	"Move Units" on page 27

Table 3: Game Commands

Command	Description	Section Reference
Place	Places unplaced entities on the map.	"Place Entities" on page 27
Rename	Renames an owned entity.	"Rename Entities" on page 28
Resign	Resigns the game for the active faction.	"Resign Game" on page 28

Table 3: Game Commands

The execution of any command may trigger one or more notification events (see "Command Events" on page 30) in addition to its normal effects. A scenario's custom rule script may change or enhance the effects of any command – check the scenario documentation for details.

A history of all executed commands, including any message events they may have triggered, is accessible through the Info → Command History menu item.

2.7.1 Menu Items vs. Commands

The game commands internally recorded by Hexkit do not correspond exactly to the Command menu items of the same name. Some commands are issued from an intermediate dialog with a different name, and some GUI buttons issue multiple commands. Conversely, both the event view and the Command History show issued commands, not menu actions.

To avoid confusion, Table 4 shows the relationship between menu items and internal game commands. The Selected Entities menu item is identical to Manage Entities, but initially shows the entities in the selected site. Note that the Begin Turn command is always issued automatically and does not have a corresponding menu item.

Menu Item	Issued Commands
Attack Site	One Attack command, unless cancelled.
Build Entities → Build and Place	One Build command, followed by one Place command, unless cancelled.
Build Entities → Build Only	One Build command, unless cancelled.
Manage Entities → Destroy	One Destroy command, unless cancelled.
End Turn	One End Turn and one Begin Turn command.
Move Units	One Move command, unless cancelled.
Manage Entities → Place on Map	One Place command, unless cancelled.
Manage Entities → Rename	One Rename command, unless cancelled.
Resign Game	One Resign command unless cancelled, followed by one End Turn and one Begin Turn command.

Table 4: Menu Items vs. Commands

2.7.2 Attack Site

When the active faction owns any units that are placed on the map, you can use the Command → Attack Site menu item to have one or more units attack the selected map site. Hexkit will show an attack planner dialog that lets you choose the units to participate in the attack.

Alternatively, you may first select a site that contains one or more of your own units, and then choose Command → Attack Site to attack enemy units in another location. In this case, the attack planner dialog will also let you choose the target site of the attack.

Either way, the dialog shows a rough estimate of the expected losses on both sides when combat is joined with the currently selected units. Loss estimates are expressed as a percentage of the total current strength of both sides. The event view shows an after-action report after the attack has been executed, including the actual losses using the same percentage scale.

You may issue multiple Attack commands per turn, as long as any placed units remain that are allowed to attack, i. e. possess the Attack ability (see "Abilities" on page 22). Under the default rules, each unit can only perform one attack per turn. The following rules also apply:

- All units may attack any enemy units within their attack range.
- Each attacker immediately destroys one (randomly selected) defender.
- Each surviving defender immediately destroys one (randomly selected) attacker.
- All surviving units on either side remain completely unaffected.
- If the scenario defines a morale resource, units require a positive morale to attack.
- Ranged attacks may require a clear line of sight to their target. The presence of certain entities, such as buildings or other units, may obstruct such a line of sight.

Most scenario authors will want to enhance this default implementation by taking the relative quantity, quality, and positions of attackers and defenders into account. Usually, the attributes of the involved units and of the contested terrain will affect the outcome of a battle, and inflicting damage will lower a unit's strength and/or morale rather than outright destroying it.

Some custom combat mechanics are easier to implement than others, and therefore more likely to appear in rule scripts. "Combat Mechanics" on page 28 describes the internal combat resolution and its likely modifications in more detail.

2.7.3 Begin Turn

The Begin Turn command is unique in that it lacks a corresponding GUI button. Instead, Hexkit *automatically* issues this command whenever a faction becomes active, as described in "Turn Sequence" on page 15.

On any turn but the first, the Begin Turn command updates all resources of the active faction (see "Resources" on page 20) and disbands unsupported units, as follows:

1. Issue a Destroy command for any units whose maintenance costs are not covered by available resources plus expected income (see "Destroy Entities" on page 26). Only accumulating resources are considered for this test; resetting resources are ignored.
2. Determine the faction's resource modifiers, representing income and expenses.
3. Update the faction's *accumulating* resources. The faction's resetting resources and the stockpiles of other factions remain unchanged.
4. Update the resources of the faction's units.

The first Begin Turn command of each turn also updates the resources of *all* entities in the game, regardless of ownership, *except* for units. This allows terrains to accumulate resources from some production source, for example.

The initial Begin Turn command of a new game updates the *resetting* resources of *all* factions, but does nothing else. All other Begin Turn commands issued during the first turn have no effect whatsoever. All the actions described above won't be executed until the second game turn.

2.7.4 Build Entities

Use the Command → Build Entities menu item to build one or more additional entities of any classes that the active faction is allowed to build, provided that it owns sufficient resources for the construction.

Click the Build and Place button to immediately place a newly built unit or terrain on the map. If you cancel the command before selecting a placement site, both the Build command and the Place command will be cancelled.

Click the Build Only button to keep a newly built entity in the faction's off-map inventory for now. You may place the entity later if possible (see "Place Entities" on page 27).

Factions may only build entity classes that possess the Build ability with respect to the active faction (see "Abilities" on page 22). Factions can use pre-created entities of classes without this ability but will not be able to replace them if they are destroyed.

2.7.5 Destroy Entities

Use the of the Command → Manage Entities or Command → Selected Entities menu items and click the Destroy button to destroy an entity owned by the active faction. Units may be destroyed at will, terrains only if they have the Destroy ability (see "Abilities" on page 22).

Destroyed entities are immediately removed from the game. Under the default rules, each destroyed entity also returns 20% of its construction costs to the active faction's stockpile, even if the entity was pre-created or if the faction cannot build new entities of the same class.

Destroying a unit may become necessary if you can no longer afford its upkeep costs. In fact, any units that you cannot pay for will be disbanded *automatically* at the start of your faction's next turn (see "Begin Turn" on page 25 and "Resources" on page 20).

When you see your resources running low, it might be wise to issue a pre-emptive Destroy command so that your most important front line units won't suddenly disappear! Hexkit attempts to disband the least valuable units first but its estimate may not agree with yours, and the default valuation does not take unit placement into consideration at all.

2.7.6 End Turn

When you are done issuing commands to the active faction, use the Command → End Turn menu item to end your faction's turn. Alternatively, you may click the End Turn button in the data view panel. Hexkit will take over and perform the following actions:

1. Restore the *resetting* resources of *all* surviving factions to their initial scenario values. The accumulating resources of all factions remain unchanged.
2. Restore the Attack and Move abilities to all units that belong to the active faction (see "Abilities" on page 22).
3. Start the turn sequence, as described in "Turn Sequence" on page 15.

A custom rule script may define additional or different actions for the first two steps. Hexkit will always start the turn sequence once all other actions have been completed, however.

2.7.7 Move Units

When the active faction owns any units that are placed on the map, you can use the Command → Move Units menu item to move one or more units from the selected map site to a different one. Hexkit will show a movement planner dialog that lets you choose which units to move.

As soon as you select a target location, Hexkit will show the planned movement path on the map view. This display is purely informational – you cannot change the path itself.

You may issue multiple Move commands per turn, as long as any placed units remain that are allowed to move, i.e. possess the Move ability (see "Abilities" on page 22). Under the default rules, each unit can only perform one movement per turn.

Hexkit's default movement rules allow each placed unit to move so far that the total terrain difficulty of all traversed map sites does not exceed its movement allowance. Units always try to find the path with the easiest terrain, so as to maximize their effective movement range.

The following default rules also apply to unit movement:

— If the target site contains no units or only friendly units, all moving units are added to whatever friendly units are already present there (i.e. no stacking limit).

— If the target site could be entered and is unowned or owned by another faction, *and* at least one terrain in the target site has the Capture ability, *and* at least one moving unit also has the Capture ability, the site's ownership changes to the active faction.

— You cannot move units onto sites that contain enemy units. Try clearing the target site with the "Attack Site" command first.

A scenario might customize movement by introducing stacking limits or restricting certain units to particular terrain types. Some terrains might be more difficult to traverse for some units, and there might be a special "pathfinder" unit that lowers the terrain difficulty.

2.7.8 Place Entities

Use the of the Command → Manage Entities or Command → Selected Entities menu items and click the Place on Map button to remove entities from the active faction's off-map inventory and place them on the map. Newly placed units do not possess the Attack and Move abilities until the faction's next turn (see "Abilities" on page 22).

You can only place entities on specific map sites. Under the default rules, valid placement sites differ depending on the game turn and on the entities involved:

— In the first turn, entities may be placed anywhere on the faction's territory.

— In subsequent turns, entities may be placed only on the faction's home site, if any.

— Some entity classes may define specific placement sites for entities of that class. These locations are available regardless of the game turn, and either in addition to the default locations or instead of them, depending on the entity class.

An overview of all placement locations available to the active faction is accessible through the Info → Placement Sites menu item. Use the Info → Faction Status → Classes dialog page to show the equivalent information for other factions.

Restrictions on Entity Placement

There are four basic restrictions that apply to entity placement under the default rules:

— An entity cannot be placed on a site that the active faction does not own.

— An entity cannot be placed on a site that contains enemy units.

— The Place command works only on unplaced entities. You cannot use it to change the location of entities that are already placed on the map.

— Once placed, entities remain on the map and cannot be moved back into a faction's off-map inventory.

Custom rules can lift the first restriction (perhaps for a paratrooper class), and programmatically circumvent the last restriction. However, the other two restrictions always apply.

2.7.9 Rename Entities

Use the of the Command → Manage Entities or Command → Selected Entities menu items and click the Rename button to change an entity's name. You can enter a new name, or reset the entity to its default name which is simply the name of its entity class.

There are no restrictions on this command, except that the active faction must own an entity in order to change its name. Computer players never use this command because they are quite happy with the default names!

2.7.10 Resign Game

In a hopeless situation, you may retire the active faction from the game with the Command → Resign Game menu item. This has the same effect as if the faction had been defeated. All units, upgrades, and unplaced terrains of the resigned faction will be removed from the game, and all of its map sites will revert to unowned status.

2.8 Command Details

This section provides more details on several game mechanics that are related to commands.

2.8.1 Combat Mechanics

The default rules outlined in "Attack Site" on page 25 internally rely on a combat resolution mechanism that a custom rule script can modify to provide fairly complex combat mechanics, simply by redefining a few parameters.

The following list describes the predefined concepts intended for reuse by rule scripts:

Attack Ability — By default, *all* attackers are marked ineligible for any further attacks during the same turn, even if they did not actually fight because the defenders were destroyed too fast. However, the ability to counter-attack is not so restricted. Custom rules might allow multiple attacks, or restrict the number of counter-attacks.

Combat Effects — By default, all units have a fixed strength and morale, and are immediately killed by any attack. Custom rules typically include a comparison of attack and defense values, gradual strength and morale damage, expending ammunition, etc.

Combat Sequence — By default, each attacker makes an attack on one defending unit, and then each surviving defender makes a counter-attack on one attacking unit, assuming there are any attackers within range. This sequence should serve most scenarios well, although custom rules might omit the counter-attack or merge the two phases.

Target Selection — By default, attacking units select their targets randomly among all surviving defenders, and counter-attacking defenders randomly select their targets among all surviving attacker within their range. Custom rules might include selection by a combat strength criterion, so as to hit the strongest or weakest defender first.

Note that the default combat sequence rewards massing units for an attack, and punishes attacking with an undersized army. The stronger the attack during the first phase, the fewer defending units will remain alive for the counter-attack phase! On the other hand, *all* attacking units are marked as ineligible for further attacks during the same turn, so attacking with an oversized army might be equally unwise…

2.8.2 Right-Click Targeting

Since the "Attack Site" and "Move Units" commands frequently operate only on single units or unit stacks, Hexkit provides a convenient shortcut that bypasses the planner dialogs. *Right-click targeting* lets you select attack and movement targets directly on the main map view.

When you select a map site containing friendly units which can attack and/or move, some locations around the selected site will be highlighted, indicating potential attack or movement targets for the currently selected unit.

— Hover over a highlighted map site that contains enemy units to show the estimated combat losses for both sides in the status bar. Right-click to execute the attack.

— Right-click on a highlighted map site that does not contain enemy units to execute a movement towards that location.

If the selected map site contains multiple friendly units, you can also issue right-click commands to a different unit or to all present units that are eligible for the command.

— Left-click repeatedly on the selected site to cycle through all units in the stack. You can also directly select a different unit in the data view.

— Hold down the Shift key to highlight the potential attack targets for all present units that can attack, and the potential movement targets for all present units that can move. The estimated combat losses in the status bar and the command executed by right-clicking on a target will likewise include all eligible units in the selected site.

When you have executed a right-click command with one or more units, the map view selection automatically jumps to your next unit that can still attack and/or move. Please refer to the Map View and Game Menu pages in the Hexkit online help for details on this mechanism.

Limitations

While right-click targeting is very convenient, it does not expose the full power of the Attack and Move commands. Here are some tasks for which you need the planner dialogs, available through the Command → Attack Site and Command → Move Units menu items:

— Choose a specific subset of eligible units to participate in an attack. Depending on the scenario, attacking with insufficient forces might increase your losses, while attacking

with superfluous forces might reduce available units and supplies for no good reason. The attack planner dialog lets you select an optimal force.

— Choose units from multiple map sites to participate in an attack. This is obviously not possible with right-click targeting since it only works on friendly units in the one map site that is currently selected.

— Choose a specific subset of eligible units to participate in a movement. Different units may have different movement speeds or terrain capabilities, including "pathfinder" abilities that confer bonuses on other moving units. The movement planner dialog lets you select the best unit mix for each moving group.

2.8.3 Automated Commands

"Menu Items vs. Commands" on page 24 listed several automated command sequences that are triggered by Command menu items. On top of that, a rule script may automatically issue one or more commands as a response to scenario-defined events. These commands will be executed immediately after the current command has finished execution.

If the active faction is controlled by a human player, automated commands will be replayed slowly, just as if they had been issued by a computer player. Watch the event view for details on what's happening, or check the Command History when replay has finished.

Automated commands may appear somewhat differently in the event view and Command History, depending on their exact implementation. You might see the effects of the automated command appended directly to the current command, without a separate history entry; or you might see another regular command (Attack, Move, etc.) inserted after the current one; or you might see a pseudo-command named Automate that performs the desired actions.

2.8.4 Command Events

The execution of a command may trigger one or more *command events* to report and visualize the command's effects to the local human player. Events fall in two categories:

Map View Events — Manipulate the map view to add heart-stopping visual excitement to the game! For instance, flames might appear briefly to indicate an explosion.

Message Events — Show text in the event view, usually to provide information on the current command. Important messages may appear in a separate pop-up dialog.

Several events in both categories are defined by the default rules; more can be added in the rule script by the scenario designer.

All events are faithfully reproduced during interactive replay. Moreover, the full text of past message events is accessible through the Info → Command History menu item.

Unit Movement

The smooth movement of units across the map in response to the "Move Units" command is the result of the only predefined map view event. Internally, Hexkit simply "teleports" units to their target location.

Custom rule scripts will usually restrict movement to target locations that can be reached by certain paths, determined by various unit and terrain attributes. The Move command will then move units along such a path on the map view, square by square or hexagon by hexagon.

Internal Events

Aside from command events, Hexkit also records a large number of internal "events" occurring to factions and entities, such as their creation and destruction. A list of these events is available through the Debug → Show Event History menu item.

As the name of the menu implies, these events are intended as a debugging aid. A published scenario should create message events for all unexpected changes to the world state so that players won't have to dig through the internal event log to figure out what's going on.

Chapter 3: Hexkit Editor

This chapter covers the companion program, Hexkit Editor. This application allows you to conveniently edit the XML files that constitute a Hexkit scenario.

Technically, you can edit these files with any text editor that supports the Unicode UTF-8 character format and (optionally) the XML syntax, and this is actually the most efficient way to handle extensive search & replace or copy & paste operations. However, normal data entry and certain "visual" tasks, such as defining bitmap tiles and editing map contents, are probably better done within Hexkit Editor.

3.1 Scenario Structure

We begin with an overview of the files that comprise a Hexkit scenario and their internal representation. Most of the operations that Hexkit Editor can perform on scenario data are adequately covered in the online help, so we'll focus our discussion on more fundamental issues.

3.1.1 External Structure

Externally, a Hexkit scenario is stored as a collection of three or more disk files. Any individual file may be reused between scenarios. The component files can be divided into three categories:

— One to six XML files, using the character format Unicode UTF-8. These are collectively called the *scenario section* files (see "Scenario Sections" on page 33).

— One or more *image files* supplying bitmap tiles (see "Bitmap Tiles" on page 40).

— One *rule script* defining the game rules (see "Custom Rules" on page 108).

Hexkit Editor allows you to create, examine, or change section files. Hexkit provides no facilities for editing image files or rule scripts – just use your favorite paint program or text editor.

3.1.2 Internal Structure

Internally, a Hexkit *scenario* is a hierarchical data structure whose elements correspond exactly to the XML elements of the scenario section files. Hexkit Editor and Hexkit Game both construct the same internal structure when loading a scenario description. This structure is defined in the Hexkit.Scenario assembly which is shared by both applications.

Hexkit Game then constructs a second hierarchical data structure, the *world state*, which represents a specific game situation. This structure is defined in the Hexkit.World assembly which is only used by Hexkit Game.[6] The rule script overrides some of the classes in this assembly to change the way world state data is manipulated during the course of a game.

6. Actually, this isn't true. Hexkit Editor does load Hexkit.World.dll because the map view on the Areas page requires a world state. However, Hexkit Editor skips a number of initialization steps performed by Hexkit Game, and completely ignores the rule script.

The scenario structure itself is immutable as far as Hexkit Game is concerned, except for some helper data. While it may be technically possible to change parts of the scenario data from within Hexkit Game, such actions are unsupported and strongly discouraged.

3.2 Scenario Sections

When all XML files of a Hexkit scenario are combined into a single monolithic file, that file conforms to the XML schema with the namespace http://www.kynosarges.de/Hexkit.Scenario[7] which is stored as the file Hexkit.Scenario.xsd in your Hexkit installation directory.

However, to allow for easy reuse of scenario components by the same or different authors, a Hexkit scenario is usually stored in up to six separate XML files. Each file contains one or more scenario sections. The following sections are present in all Hexkit scenarios:

Master — Identifies the scenario, imports all other section files, and defines the rule script.

Images — Identifies all required bitmap graphics files and defines all bitmap tiles.

Variables — Defines all attributes, resources, and counters. This section may be empty.

Entities — Defines all entity classes. At least one class must be defined.

Factions — Defines all factions and their possessions. At least one faction must be defined.

Areas — Defines the map size and geometry and all initial map contents. To reduce the size of this section, this section doesn't store individual sites but rectangular "areas" of sites with identical content. These areas are overlaid to create the actual map contents.

All scenario sections other than the Master section are referred to as *subsections*. Please refer to "Section Dependencies" on page 35 for an overview of subsection relationships. Chapter "Class Structure" on page 51 contains detailed UML diagrams for all sections.

3.2.1 Importing Subsections

If you know XML you may have frowned on the claim that the Master section "imports" the other section files. As a matter of fact, XML documents cannot import other files – except when they are defined as "external entities" by a DTD schema.[8]

That would be a great solution, but unfortunately the XML parser provided by the .NET Framework cannot validate the same document against both DTDs (for the file import) and XSDs (for checking the scenario structure).

Our workaround is, quite frankly, a hack – though one that works well enough. When a Master section file needs to import a subsection, the location where that subsection should appear is replaced with a non-schema element called include that looks like this:

```
<include element="images" href="Images/DeBray Bailey.xml" />
```

When Hexkit processes this line, the include element is replaced with an element whose name is specified by the element attribute. The contents for the new element are the entire contents of the XML file indicated by the href attribute.

7. As usual with XML namespaces, this is *not* a real URL, just a globally unique identifier.

8. The XInclude standard would be another option, but .NET does not yet support this mechanism.

The top-level element of the imported file must match the name specified by the element attribute. The imported file should not have any processing instruction since its entire contents are copied literally into the Master section file, bypassing the XML parser.

When Hexkit Game starts a scenario, its combined contents are written to a temporary monolithic file called Scenario.Start.xml which is then read by the validating XML parser. Any validation errors you might encounter therefore refer to this temporary file and not to the original subsection files, as described in "Starting a New Game" on page 8.

3.2.2 Sharing Subsections

The scenario itself is identified exclusively by its Master section. All other sections may be shared arbitrarily between scenarios, although you will have to ensure that all identifier references are properly resolved when the complete scenario description is assembled (see "Identifiers and Names" on page 36).

The prime candidate for sharing between scenarios is the Images section. Creating a set of decent image tiles is a lot of work, and not one that most programmers excel at! Because the artist will often differ from the scenario designer, the Master and Images sections contain each their own data fields for author, version, and copyright statement.

Another likely case is a set of scenarios that share all subsections *except* for the Areas section. Such scenarios would offer identical gameplay on different maps. This is the Hexkit equivalent of the classic "map editor" functionality offered by many commercial strategy games.

Note. You should always put your Master section files at the top level of the Scenario folder, or at least not in the same folder as a subsection file, to avoid confusing the player.

Example: The Battles of Crécy and Poitiers

All demo scenarios that ship with Hexkit use the same Images section, but some share additional subsections. Table 5 shows the files referenced by the four demo scenarios covering the medieval battles of Crécy and Poitiers, each in one historical and one hypothetical variant. Folder names and file extensions (.xml for all XML files and .cs for the rule script) were omitted for brevity.

<table>
<tr><th></th><th>Crécy</th><th>Crécy Variant</th><th>Poitiers</th><th>Poitiers Variant</th></tr>
<tr><td>Master</td><td>Crécy</td><td>Crécy Variant</td><td>Poitiers</td><td>Poitiers Variant</td></tr>
<tr><td>Images</td><td colspan="4">DeBray Bailey Images</td></tr>
<tr><td>Variables</td><td colspan="4">Crécy Variables</td></tr>
<tr><td>Entities</td><td>Crécy Entities</td><td>Crécy Entities Variant</td><td>Poitiers Entities</td><td>Poitiers Entities Variant</td></tr>
<tr><td>Factions</td><td colspan="2">Crécy Factions</td><td colspan="2">Poitiers Factions</td></tr>
<tr><td>Areas</td><td colspan="2">Crécy Areas</td><td colspan="2">Poitiers Areas</td></tr>
<tr><td>Rules</td><td colspan="4">Crécy Rules</td></tr>
</table>

Table 5: Sharing Subsections

Naturally, every distinct scenario has its own Master section, and as stated above they all share the same Images section. The Factions section might be shared as well, but dedicated files for each battle make the faction descriptions easier to read.

The interesting stuff happens in the remaining rows. Since the four scenarios all use the same basic game mechanics, they share the same Variables section and rule script. This required some trickery with both visible and hidden game variables (see "Counter Variables" on page 49) that are used in different ways by different scenarios.

Two different battlefields obviously require two different Areas sections (i.e. game maps), and these also include hidden variables that identify the English faction's initial movement area so that the same rule script can operate on both maps. The remaining differences are covered by the Entities sections which are specific to each battle and variant.

The entity definitions not only address the necessary variations in unit names and facing, but also change certain variable values to differentiate the gameplay between basic and variant scenarios: a higher defense for crossbowmen in the Crécy variant scenario, and a different morale for the second French division in the Poitiers variant scenario.

3.2.3 Section Dependencies

Figure 1 shows an overview of the most important data dependencies between subsections, as they appear on the corresponding tab pages of Hexkit Editor. Please note the following:

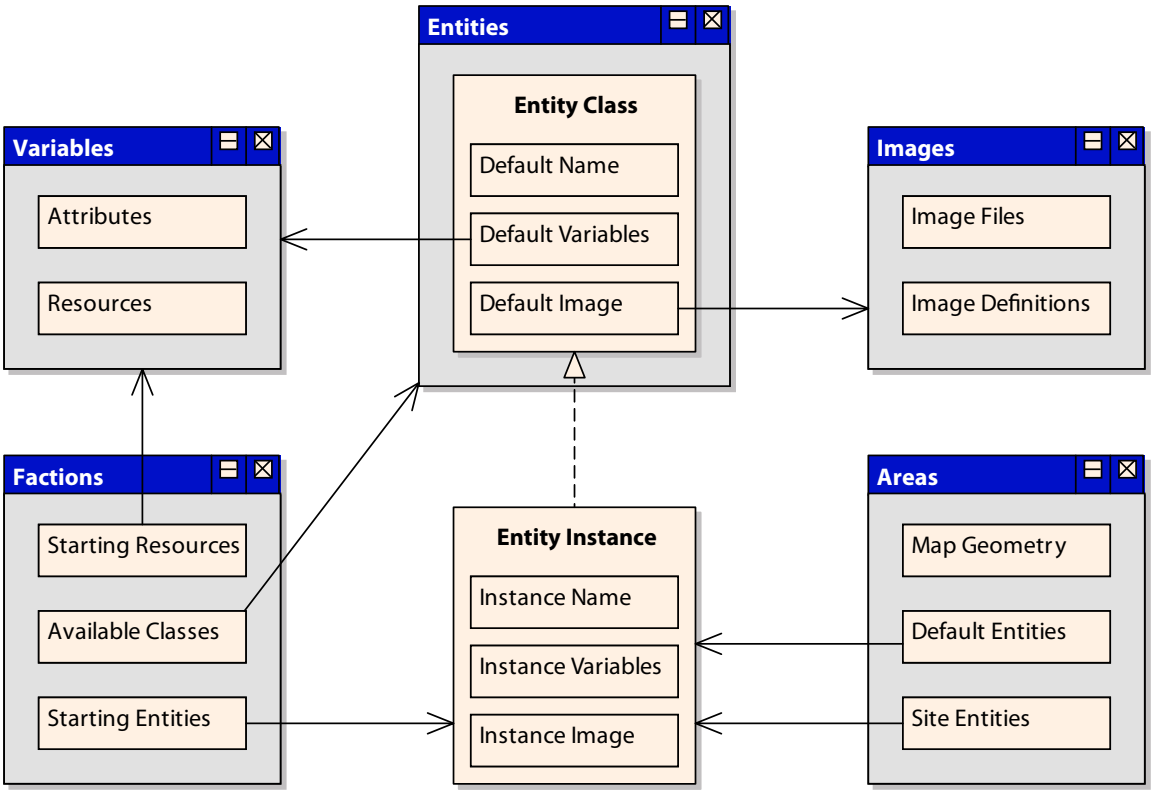


Figure 1: Subsection Dependencies

— The Variables and Images sections are entirely self-contained. All other sections depend on these two, directly or indirectly. The Variables section is optional, but you must have a valid Images section before you can design entities.

— The Images section is used only by the Entities section. Whatever images you wish to display in a scenario must be associated with an entity class.

— The Entities section defines *entity classes* which provide default values for all statistics of a concrete entity. Only the Factions section uses entity classes directly, to indicate which kinds of entities a faction may build during a game. The Change Faction → Classes dialog tab page indicates all classes that are available for building.

— A scenario may also define *entity instances* which are the concrete entities that are present at the start of a scenario. They may use the default statistics of their class, or customized statistics for each instance. Entity instances are stored with the Factions and Areas sections that define them, not with the Entities section.

A faction's unplaced starting entities are defined on the Change Faction → Entities dialog tab page. The Change Default Contents and Change Site Contents dialogs show all default and site-specific entities, respectively, that are placed on the map.

Note. Internally, most scenario elements have a class-instance relationship with the objects that constitute a world state of the game. The case of entity classes and instances is the only one to surface in Hexkit Game and Hexkit Editor, but scenario designers will encounter FactionClass and VariableClass objects in rule script code.

3.2.4 Identifiers and Names

Section files express the hierarchical relationships between scenario elements – for example, the units initially owned by a faction – as ID, IDREF, and IDREFS attributes. The text stored in an ID attribute remains associated with the corresponding internal objects as a string property named Id, and is displayed by Hexkit Editor as the object's *unique internal identifier*.

These identifiers never appear in Hexkit Game (except with debugging commands). Instead, every object that might require a textual representation during a game also has a Name property that does neither have to be unique nor conform to the XML NMTOKEN format.

Generally, unspecified entity names default to their class's name (e.g. an unnamed terrain entity based on a terrain class named "Grass" will display the name "Grass"), and unspecified class names default to the class's identifier. Naturally, a finished scenario should never contain any names that default to identifiers.

Editing Identifiers

Hexkit Editor provides limited support for changing multiple identifiers at once. If you change or delete an identifier that is referenced elsewhere in the scenario, Hexkit Editor will offer to extend the change to all other occurrences of that identifier. However, you may want to use a text or XML editor for complex search & replace operations.

Hexkit Editor does not currently check that all identifier references are properly resolved. Try loading a scenario into Hexkit Game to see if there are any unresolved references.

Note. When you decline the offer to change other occurrences of an identifier, you may end up with "orphaned" identifier references. For example, when you delete an attribute definition on the Variables tab page, any entity classes and instances that use this attribute will internally retain their variable values, but these values will no longer appear in the GUI. They will reappear as soon as you add a variable definition with the same identifier back to the Variables tab page.

3.3 Map Geometry

A Hexkit game map is always a rectangular grid of regular polygons, namely squares or hexagons which are either standing on a corner (a *vertex* of the polygon) or lying on a side (an *edge* of the polygon). The Areas section of a scenario defines the shape, size, and orientation of one polygon, as well as the total size of the game map in terms of polygon rows and columns.

Each polygonal element corresponds to a pair of grid coordinates, starting at $(0, 0)$ in the upper left corner and continuing to the right and downward. A map is always a "full" rectangle. That is, if the size of the map is $x \times y$ then every single coordinate in $(0 \ldots x - 1, 0 \ldots y - 1)$ will be associated with a polygon. Therefore, all rows must have the same width, and all columns must have the same height.

This causes a problem for element shapes other than lying squares: maps cannot be visually symmetric in both dimensions, as that would require a varying number of polygons in adjacent rows or columns. To alleviate the situation, Hexkit Editor provides a "grid shift" option that lets you decide which rows or columns should visually protrude beyond the map.

We'll illustrate the available choices with a series of figures showing a 5×5 grid in various configurations. The arrows show increasing x- and y-coordinates from the origin in the upper left corner. To illustrate the mapping of coordinates to grid elements, each figure also shows the grid element at location $(2, 2)$ and the coordinates of all its neighbors.

3.3.1 Maps of Squares

Squares may share either an edge or a vertex with a neighboring square. The four squares with shared edges are always considered immediate neighbors, but you may choose whether the four squares with shared vertices should be considered neighbors as well. If this option is enabled, units can enter a *vertex neighbor* in a single step; otherwise, they must move across an edge neighbor first, requiring two steps for the same movement.

Figure 2 shows a grid composed of squares resting on their edges. There is only one possible alignment for this element type, and the coordinate mapping is trivial. This is the layout used in many traditional board games, such as chess and checkers.

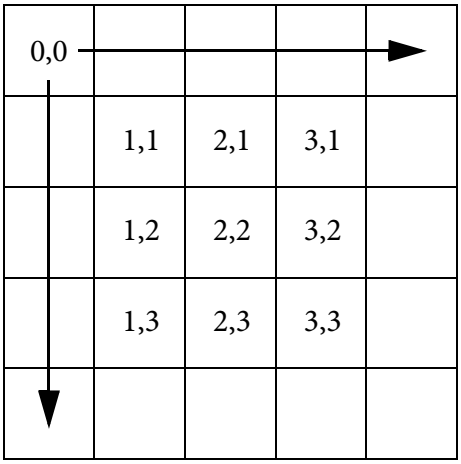
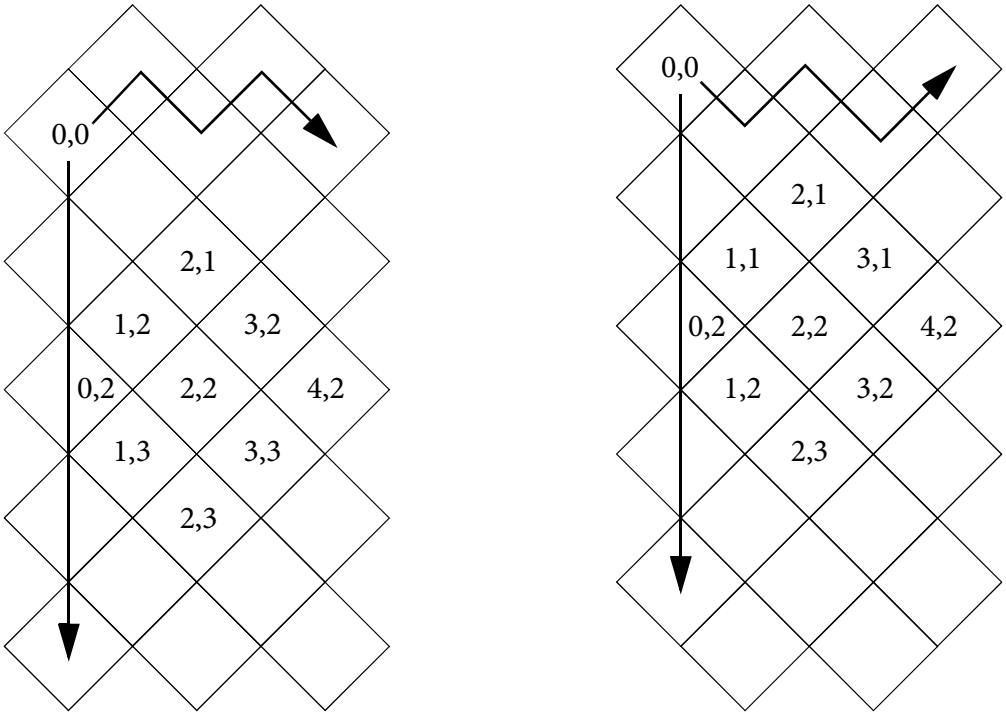


Figure 2: Squares on Edge

Figure 3 and Figure 4 show the four possible alignments for grids of squares standing on their vertices. Either even-numbered rows or even-numbered columns may be shifted compared to their odd-numbered neighbors, and the shift may occur in either of two directions.



Column up Column down

Figure 3: Squares on Vertex (Columns Shifted)

This is the layout used in the classic computer strategy game *Civilization*, although the view is tilted in this game to create a three-dimensional view of the game world. Hexkit may offer a similar display option in a future version, but this would require drawing a bitmap tile beyond the boundaries of its polygon which is not yet supported.

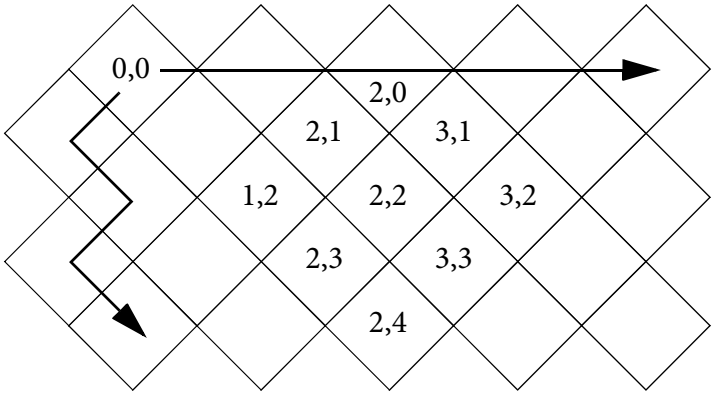
Standing squares cause a significant visual distortion relative to the original grid dimensions. A grid with equal width and height will look as if it was about twice as high as it is wide, or vice versa. You should keep this fact in mind when deciding on the size of your maps.

3.3.2 Maps of Hexagons

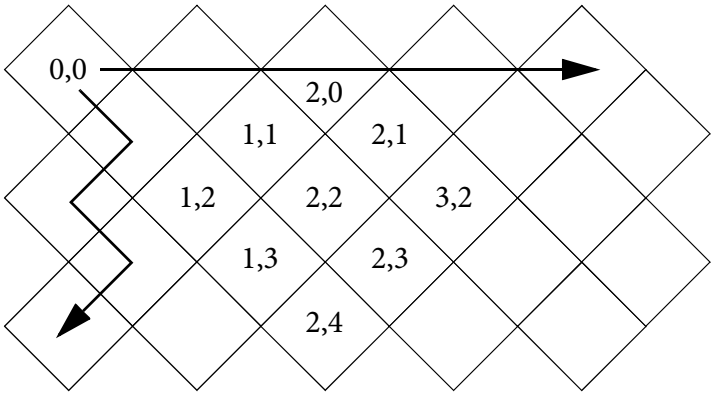
Hexagons only share edges with neighboring hexagons, so the vertex neighbor option discussed in the previous section is unavailable for hexagon maps.

Figure 5 shows the two possible alignments for hexagons resting on their edges. Even-numbered columns may be shifted upward or downward compared to odd-numbered columns. This is the most popular layout for board wargames, and many computer wargames as well.

Figure 6 shows the two possible alignments for hexagons standing on their vertices. Even-numbered rows may be shifted to the left or right compared to odd-numbered rows. This layout is best suited for the tall images in the default tileset. For this reason, it was the only available layout in Hexkit versions prior to 3.5, and remains the default when designing a new scenario.

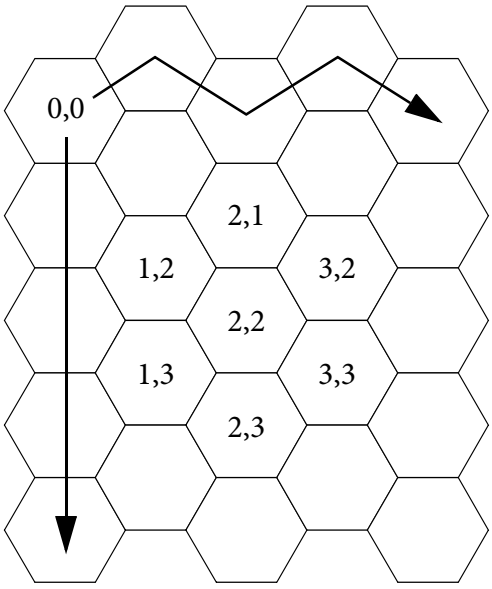


Row left

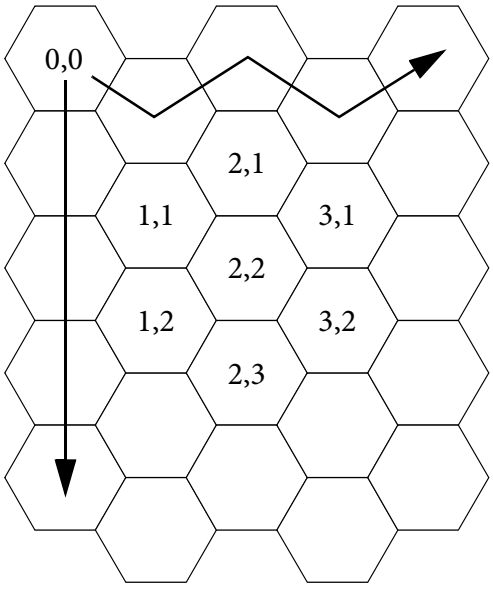


Row right

Figure 4: Squares on Vertex (Rows Shifted)



Column up



Column down

Figure 5: Hexagons on Edge

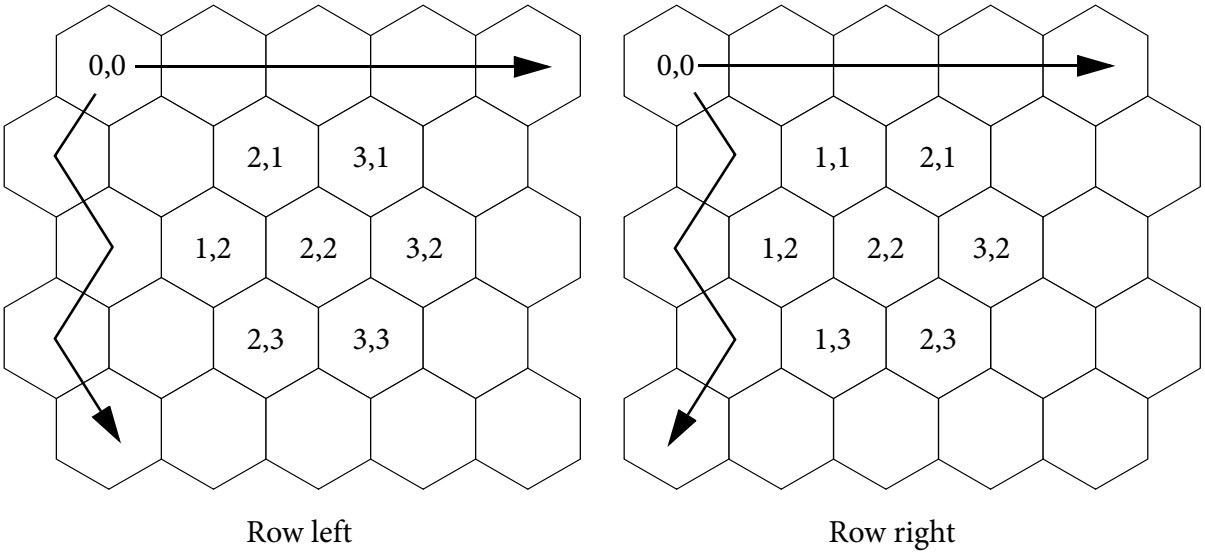


Figure 6: Hexagons on Vertex

3.4 Bitmap Tiles

Hexkit is an old-fashioned two-dimensional tile-based game, so all it needs for its graphical display is a set of rectangular *bitmap tiles*.

The tile collection for a scenario is supplied by one or more *image files* which contain entire sets of bitmap tiles. The Images section specifies the exact boundaries of each tile within its image file, associates bitmap tiles with images, and images with entity classes.

The format of the image files must be supported by the BitmapSource class provided by the Windows Presentation Foundation. This currently includes the following formats:

— Windows Bitmap (BMP)

— Graphics Interchange Format (GIF)

— Joint Photographic Experts Group (JPEG)

— Portable Network Graphics (PNG)

— Tagged Image File Format (TIFF)

All of these formats should work well enough, although I personally prefer PNG. This format provides efficient lossless compression and supports the 32-bit RGBA color format which is also used internally by Hexkit for all bitmap data. DeBray Bailey's bitmap tiles that ship with the Hexkit binary package are stored as PNG files.

Note. BitmapSource supports color palettes with GIF and TIFF but not with PNG encoding.

3.4.1 Transparency

Any entity other than background terrain must be superimposed on that terrain and possibly other entities. To achieve that, all bitmap tiles must be drawn with a transparent color that fills the space between their actual outlines and their rectangular bounding boxes.

Hexkit must somehow know about this transparent color. If the image file format supports transparency, and this feature is also supported by BitmapSource, the transparency data stored in the image file is automatically used by Hexkit. This is the case with some of the PNG files containing DeBray Bailey's tileset, for example.

However, some formats may not directly support transparency, or Hexkit may not recognize this feature. To accommodate such cases, Hexkit Editor allows you to set a specific "magic" RGB color value that should be treated as the transparent color in all image files.

Note. It is not currently possible to set a different transparent color for each image file. Also, an image file's built-in transparency data that Hexkit does recognize will always be used, even if a transparent color is defined in Hexkit Editor. In that case, *both* colors will be transparent.

Alpha Blending

The transparency associated with a bitmap pixel is also known as its *alpha channel*. For image files that support transparency, each pixel's alpha channel is interpreted as follows:

— Pixels with an alpha channel of zero are fully transparent. These pixels are discarded and have no visible effect.

— Pixels with an alpha channel of 255 are fully opaque. These pixels overwrite any existing pixel at the same map view location, obscuring previously drawn images.

— Pixels with an alpha channel between 1 and 254 are partly transparent. These pixels are *blended* with any existing pixel at the same map view location. The smaller the new pixel's alpha channel, the more of the old pixel's color will shine through.

Note. Pixels whose color was defined as transparent in Hexkit Editor always have an alpha channel of zero. Alpha blending is not possible with such pixels.

3.4.2 Scaling

The use of two-dimensional bitmap tiles with a variable map geometry creates a few problems for our graphics engine. Every tileset is designed for map polygons (squares or hexagons) of a certain size and shape. If our game map has a different geometry, we cannot simply copy each tile to its map polygon and expect satisfactory results.

Some computer games with hexagon maps simply require the artist to supply matching hexagonal bitmap tiles. While this approach maximizes visual quality, it has a serious disadvantage from the viewpoint of a freeware game: there aren't any free tilesets with hexagonal tiles!

On the other hand, there *are* a couple of free tilesets of good quality that use rectangular tiles, such as DeBray Bailey's, because this format is popular with "roguelike" games. Clearly we'd like to use any available tileset with every possible map geometry.

Hexkit therefore provides three *image scaling* options. Every image specifies one of these options to define if and how its bitmap tiles are scaled to the size and shape of a map polygon.

None — The tile is drawn once at its original size, centered on the polygon. This option is the default and typically used with foreground entities such as units.

You may reposition and mirror an image when defining the visual appearance of an entity class (see "Composing Image Stacks" on page 44).

Repeat — The tile is drawn repeatedly at its original size, with the first copy centered on the polygon and the others surrounding it seamlessly, until the entire polygon is filled. This option is best suited to background terrain that should "carpet" the map.

Stretch — The tile is drawn once, but shrunk or grown to match the rectangular bounding box of the polygon. This option degrades visual quality, as always when a bitmap is resized. However, it is necessary for tiles that should cover the entire polygon but are not suited to repeated copying, e.g. road and river tiles.

The copied tiles are always clipped to the outline of the map polygon, as you would expect.

You may apply different scaling options in the horizontal and vertical direction. For example, a tile showing a north-south road would use vertical repetition but no horizontal scaling. Scaling can be disabled entirely for specific entity classes (see "Composing Image Stacks" on page 44).

Limitations

These options allow a simple adaptation of tilesets designed for rectangular map elements to the various square and hexagon geometries used by Hexkit, but there are some limitations.

Repetition is only useful for homogeneous background terrain or certain graphical effects such as smoke. Stretching is only useful for tiles that are close to the current polygon shape and size. For example, stretching a small oblong tile to fill a large hexagon causes visual distortion and clipping that effectively makes the tile unusable.

A number of images in the default tileset use "Stretch" scaling anyway, but this is a stopgap solution to speed up the initial design of scenarios with square-on-edge geometry. Published scenarios should either use custom tiles, or compose entity images from multiple unscaled tiles with manual positioning.

Let's say you have a tile showing a road that comes in from the south and bends to the east. To scale this tile a larger map square, you would add north-south road tiles to the southern edge and east-west road tiles to the eastern edge, until you reach the edges of the square.

Conceivably, this process could be simplified by further enhancements to the scaling mechanism. However, there is a question of diminishing returns when adding ever more complexity to an archaic graphics engine. At some point it's less work to just draw some custom tiles…

3.4.3 Frames and Animation

Hexkit can associate multiple bitmap tiles with a single image. Since one purpose of this feature is animation, any tiles that are associated with an image are called *frames*.

By default, only an image's first frame is drawn. However, there are two exceptions:

— Animated images cycle through their frames while the game awaits user input.

— A specific frame can be selected to represent a particular entity or entity class.
This frame can be manually selected by the scenario designer, or Hexkit can randomly select a frame whenever an instance of the entity is created.

Manual frame selection is particularly useful for realistic roads and rivers. Any required directions, bends, forks, etc. can be stored as individual frames of a single image which is associated with a single entity class.

Random frame selection is a convenient way to provide some visual diversity to large areas with otherwise identical terrain, such as differently shaped trees in a forest.

Animation Modes

Hexkit supports a variety of "idle animations" which are played back while the game is otherwise inactive and awaits user input. You can specify an *animation mode* and an *animation sequence* for any image that contains multiple frames.

Animation Mode — *None* disables animation entirely. *Random* starts the specified sequence at random intervals. *Continuous* restarts the sequence as soon as it is finished.

Animation Sequence — *Random* picks a single random frame and terminates. *Forward* and *Backward* show all frames going in the specified direction, then return to the first frame. *Cycle* shows all frames going forward, then again going backward.

The combination Random/Random yields entirely random frame changes in the style of early *Ultima* games. Although not terribly sophisticated, this animation style has the great advantage that it doesn't require a full set of consecutive animation phases – all you need is two different postures, and you have an animated character!

Frame Connections

Each image frame may define a set of visual *connections* to adjacent polygons. For example, you would set the eastern and western connection flags for a frame that shows an east-west road.

The Hexkit default rules do not use this connection data, but custom rule scripts can call the method Site.IsConnected to determine if adjacent polygons contain entities of the same class whose connection flags match across the shared edge or vertex, as the case may be.

Site.IsConnected compares entity class identifiers with optional prefix matching. If a specified identifier ends with an asterisk, only the substring up to the asterisk is compared. This allows you to perform connection matching even among different entity classes, as long as the identifiers of all connection variants differ only in a suffix. Thus, when you design a set of entity classes that can form connections you should use identifiers that share the same prefix.

For simplicity, connections are always mapped to the eight major compass directions, even if the correspondence is not quite exact. Figure 7 on page 43 shows the available connection points for all possible polygon shapes and orientations. Parentheses indicate those connections that are only available when vertex neighbors are enabled.

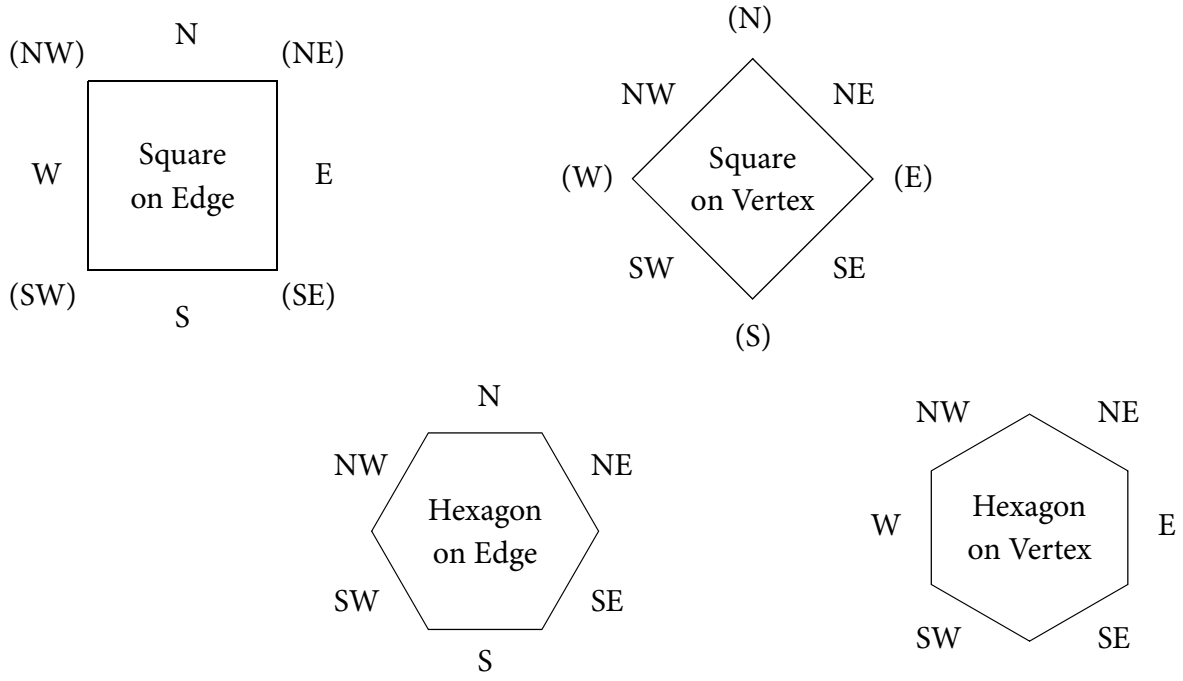


Figure 7: Polygon Connections

3.4.4 Composing Image Stacks

Once you have defined your images, you can associate them with entity classes. Every entity class requires at least one image as its visual representation on the map view.

However, you can also associate an entity class with multiple images. They are stored in an ordered sequence which is called the *image stack* of the entity class. The corresponding frames of all images are visually overlaid ("stacked"), with the last image's frames appearing on top.

Image stacks are an easy way to create composite images from multiple existing bitmap tiles, without having to manually combine the tiles in a paint program. Depending on your tileset, you might be able to create a large variety of terrain classes by overlaying different objects, such as flowers, on a basic terrain tile, such as grass.

Ultimately, all combined frames used by any entity class in the game are stored in the *bitmap catalog*. Technically, there are several bitmap catalogs: one master catalog that is created at the original scale of the source files, and a scaled catalog for each individual map view that reflects the map view's current zoom level. You can use the Debug → Show Bitmap Catalogs menu item in Hexkit Game to display the catalogs created for the current scenario.

Figure 8 shows how the Hexkit graphics engine creates the final bitmap catalog from the original bitmap files. In this example, image-two consumes two tiles and is in turn consumed by unit-one. Thus, the catalog contains the two unchanged source tiles. The image stack of terrain-one contains two images, image-one and image-three. Their frames are overlaid, and the combined tiles copied to the catalog. The other tiles in the original files are ignored.

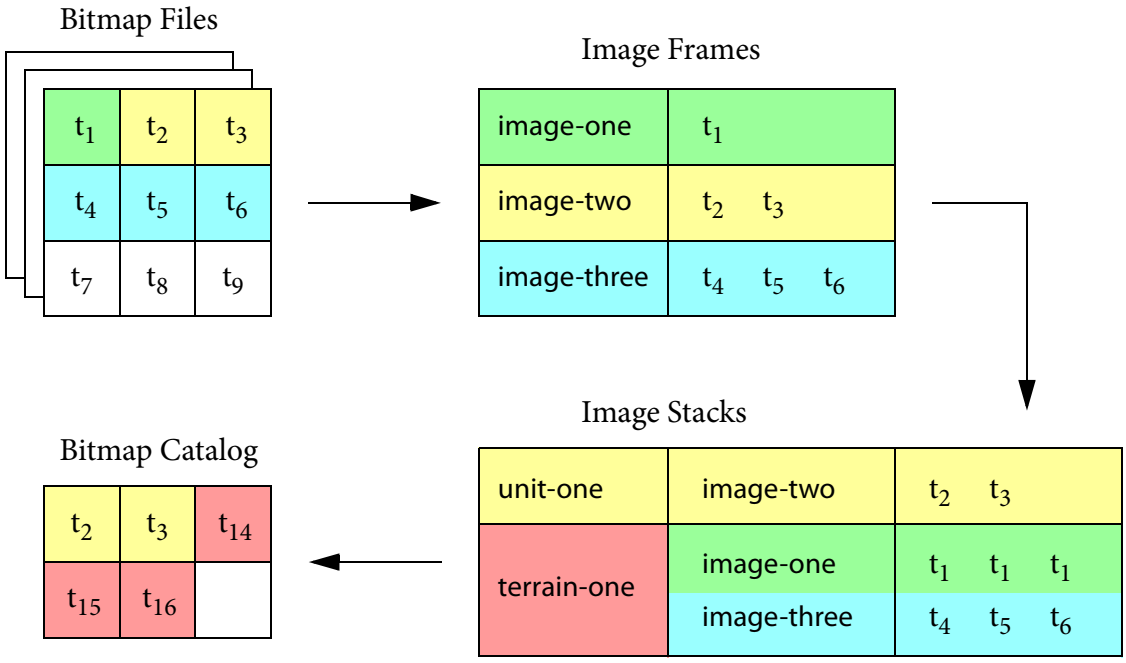


Figure 8: Catalog Creation

Combining Images

You'll note that the single frame defined by image-one is copied repeatedly, to match the three frames defined by image-three. The number of *catalog frames* (i.e. entries in the bitmap catalog) reserved for an entity class equals the highest number of frames defined by any stack image. The frames of other stack images are copied repeatedly, starting over with the first frame as often as necessary, so that every catalog frame gets one frame from each stack entry.

What happens to animated images? An entity class adopts the "highest" animation modes and sequences defined by any stack image, which is interpreted as follows:

Animation Mode — *Continuous* is higher than *Random* which is higher than *None*.

Animation Sequence — *Cycle* is higher than *Backward* which is higher than *Forward* which is higher than *Random*.

Combining images with different animation settings (other than *None*) may have surprising effects. Check the results to make sure they are what you intended.

Frame connections are likewise combined across all image stack entries. When one image frame specifies an eastern connection and the corresponding image frame of another stack entry specifies a western connection, the resulting catalog frame will have an east-west connection.

Customizing Images

Hexkit Editor offers several options to customize the standard image composition algorithm. All of the following options apply to individual image stack entries.

Unconnected — Ignore any frame connections defined by the image when determining the combined connections of the catalog frames. This allows the use of connected frames for their visual effect only.

Unscaled — Ignore any scaling options specified by the image. This allows the use of scaled images without covering the entire map polygon.

Single Frame — Use a single selected frame and ignore all other frames defined by the image. This also ignores any animation options specified by the image. When determining the total number of catalog frames for the entity class, the image is assumed to have only a single frame, and that frame is copied to all catalog frames.

Pixel Offset — Shift the image from its default center position. This lets you precisely align two images even if they were not designed to be used with each other.

Mirror — Mirror the image across the horizontal and/or vertical axis. This is also useful for precise image composition, or simply for providing some variety.

Color Shift — Shift the image's RGB channels by any value from −255 to +255 while leaving the alpha channel unchanged. This lets you brighten or darken an image, or change its overall hue by shifting different color channels in different directions.

Since these options can change the appearance of an image quite drastically, an image stack may contain the same image more than once, each time with an individual set of options.

Putting Riders on Horses

Let's have a look at the construction of an image stack with manual positioning. Figure 9 shows how to put a rider on a horse, using the three indicated images from the default tileset.

The horse and knight images are both centered by default but a rider sits atop of a horse, so we need to apply a positive vertical offset to the horse and a negative vertical offset to the knight. That results in the second image of the "Combined Stack" column which doesn't look quite right: one of the knight's legs should be hidden by the horse's body. For this purpose, the default tileset includes special "front" variants of all horses that include only their heads and forelegs.

There are no dedicated bitmap tiles for these variants; we simply use a smaller portion of the regular horse tiles. Therefore, these images require the same vertical offset as the complete horse image, but also a positive horizontal offset to correct the location of the horse's head.

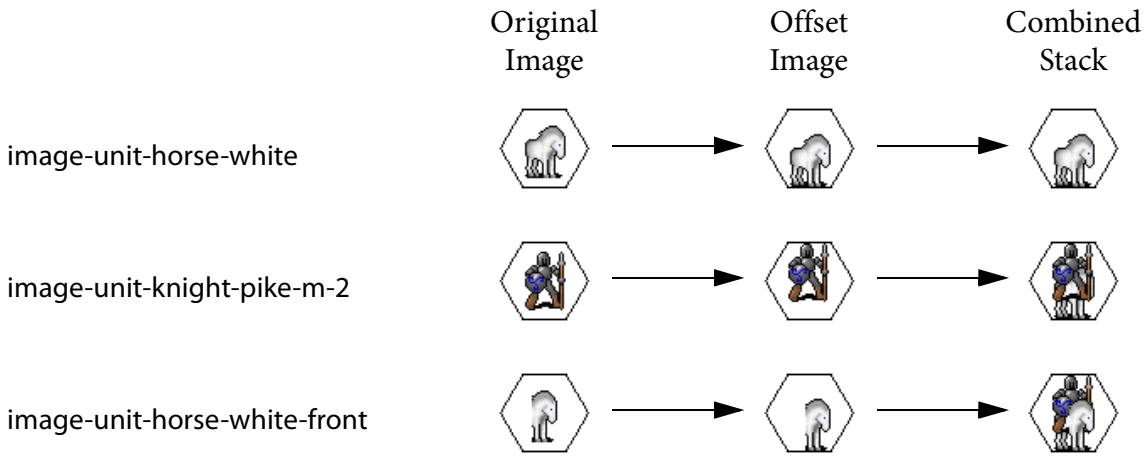


Figure 9: Image Composition

And with that we get the final combined stack: almost as good as hand-drawn! Note that all stack images contain two frames for random animation. Only the first frame of each image is shown here, but the combined stack is in fact animated, just like the individual images.

3.4.5 The Default Tileset

The Images section for the default tileset, DeBray Bailey Images.xml, defines hundreds of image identifiers. For ease of use, they all follow the same naming pattern: image-group-object.

image — All image identifiers start with the literal string "image".

group — One of the eight image groups listed below. I like to call them "namespaces" but they are not literally XML namespaces.

object — An arbitrary name denoting the specific object represented by the image. This name may contain further dashes, ordinal numbers to enumerate variants, and the suffix -x or -y to indicate images that are scaled only in the indicated dimension.

The names of the eight image groups were chosen to reflect the intended use of an image.

arms — Items that are used for attack or defense, including weapons and pieces of armor.

house — Indoors environments, both background "terrain" and related foreground objects such as furniture, sign posts, and entire buildings.

item — General items, including any non-moving objects that are not covered elsewhere.

river — Any terrain that includes a river touching at least one edge of the image.

road — Any terrain that includes a road touching at least one edge of the image.

split — Transitional images that combine part of one terrain with part of another.

terrain — Outdoors environments, both background terrain and any objects whose image includes such background terrain, except for those covered elsewhere.

unit — Unit images, including human and non-human characters, but also animals, vehicles and any other moving objects.

All river and road images define connection flags to indicate the direction(s) in which the river or road exits each tile. All split images and most river and road images use "Stretch" scaling. All background terrains and the -x/-y variants of all images use "Repeat" scaling.

Multi-Frame Images

Individual images in the default tileset may contain multiple frames in the following cases:

— Each frame shows a different animation phase of the same object.

— Each frame shows a different part or aspect of the same object. This includes different bends of a road on the same terrain, or different orientations of a piece of furniture.

— Each frame shows a variation that is slight enough to qualify as the "same" object. This excludes significant color variations which are placed in separate images.

The last case includes several basic terrains that use random animation to create the impression of wind and weather. Scenario authors may wish to use only a single frame from these terrains, or manually select different frames for each site to create some diversity without animation.

Unit images typically have two alternate frames and use random animation. A few items and terrains have a full set of consecutive animation phases for continuous animation.

Multi-Frame Image Sets

As of Hexkit version 3.6.2, the default tileset also contains a number of *image sets* that combine the frames of multiple existing images into a single new multi-frame image.

Sets are always composed of non-animated images with identical scaling options. An image set called xyz-set contains the frames of some or all images with the same identifier prefix but different numerical suffixes, i.e. xyz-1, xyz-2 and so on.

Some sets contain image frames that naturally fall into a few thematic subgroups. In this case you may find *subsets* whose naming scheme is xyz-set-a, xyz-set-b, and so on.

Image sets differ from regular multi-frame images in that they *duplicate* existing image frames. You never need to use image sets to get at some specific bitmap tile, but they are very convenient when you wish to provide a single entity class with a wide variety of visual representations. A good example are the different shapes of trees that might appear in a forest. Image sets and subsets are particularly well-suited to random frame selection.

3.4.6 Static Overlay Images

Starting with Hexkit version 4.1.2, map views can also show fixed *overlay images* such as hand-drawn maps. These come in two variants: as an editing aid and as gameplay decoration.

Overlay Images During Editing

The first variant is controlled by the Editor Overlay button on the Areas tab page of Hexkit Editor. The selected image appears *above* the map view, its opacity controlled by a slider control.

This kind of overlay image is intended solely as an editing aid to simplify the creation of Hexkit scenarios based on digital map images. The image will not appear during gameplay, and its data is saved only to the Hexkit Editor options for the current user.

Overlay Images During Gameplay

The second variant is controlled by the Overlay button on the Areas tab page of Hexkit Editor. The selected image appears *below* the map view and is always fully opaque.

Unless you want to use the image merely as border decoration, you must also suppress map view drawing for one or more entities – usually background terrains – by unchecking the Draw Image option in their Change Entity Class or Change Entity Instance dialogs. The overlay image will then provide your scenario's map graphics wherever you place invisible terrain entities.

This kind of overlay image appears both during editing and during gameplay, and its data is saved to the Areas section of your scenario file, like all other map-related data. Naturally, you must also distribute the physical image file along with your scenario.

Note. Using an overlay image to provide background terrain does *not* obviate the need to define background terrain entities! You still need to have one background terrain entity per map site. Overlay images are purely visual decoration with no gameplay effects whatsoever.

Moreover, all scenario entities still need a valid image stack, even if their Draw Image option is disabled, because the image stack visually represents its entity in dialogs and other contexts where the overlay image is not available. Visuals aside, image stacks also define site connections, e.g. for road movement (see "Frame Connections" on page 43).

3.5 Other Data

Most of the remaining scenario data available in Hexkit Editor correlates in a fairly obvious manner to the gameplay concepts presented in "Hexkit Game" on page 8ff. However, there are some obscure bits and pieces that may require additional explanation.

3.5.1 Entity Class Settings

Certain entity class settings are not directly visible in Hexkit Game. They are accessible on the Change Entity → Abilities and Change Entity → Other dialog pages in Hexkit Editor.

Blocks Attack — Indicates an entity that obstructs a ranged attacker's line of sight. See "Lines of Sight" on page 138.

Can Defend Only — Automatically disables the unit's Attack ability when its owning faction becomes active. See "Begin Turn Command" on page 123.

Can Heal Damage — Informs computer players that the unit can recover lost strength during the game. See "Comparing Losses" on page 146.

Modifier Range — The range of variable modifiers with unit targets, as described below.

Ranged Attack Mode — Indicates what targets are valid for a ranged attack by the unit. See "Attack Command" on page 120.

Resource Transfer — Specifies if and how an entity automatically transfers resources to other entities or factions. See "Attributes and Resources" on page 116.

Valuation — Informs computer players of an entity's desirability. The entire valuation system is described in "Evaluating Possessions" on page 140.

Moreover, unit and terrain classes define several "standard variables" which may be mapped to any of their attributes or resources. See "Standard Variables" on page 119 for details.

3.5.2 Modifier Targets

Entity classes may define several modifiers for each attribute and resource, namely one for each modifier target as described in "Variables" on page 19. Here we'll take a closer look at how those modifier targets work internally.

Self — Affects the entity's own variables. This is equivalent to a faction's resource modifiers; in other words, a faction only defines self-modifiers.

Owner — Affects the resources of the entity's owner. Has no effect for attributes.

Units — Affects the variables of any units in the same map site which are owned by *a different* faction than the defining entity. If the latter is unowned this includes units owned by *any* faction. Upgrades affect alien units on their owner's home site, if any.

OwnerUnits — Affects the variables of any units in the same map site which are owned by *the same* faction as the defining entity. Has no effect if the defining entity is unowned. Upgrades affect friendly units on their owner's home site, if any.

UnitsRanged, OwnerUnitsRanged — Same as before, but affects units *around* the entity's map site or its owner's home site, within the entity's modifier range.

Each entity class defines a single modifier range that applies to all ranged unit modifiers defined by the class. The default value of zero indicates an *infinite* range, i.e. all ranged unit modifiers affect all eligible units anywhere on the map.

An upgrade's unit modifiers usually have no effect if the upgrade's owner does not define or own a home site. However, if the upgrade's modifier range is zero, its ranged unit modifiers are always applied, regardless of their owner's home site or lack thereof.

Note. If an entity should have the same effect on (local or distant) units regardless of their owner, you must define both the Units(Ranged) and OwnerUnits(Ranged) targets for the same variable with the same value. The Hexkit Game GUI only displays both values if they are different, but in Hexkit Editor you must in fact define both values if you wish them to be identical!

3.5.3 Counter Variables

Remember when I said (in "Variables" on page 19) there were two categories of variables? I lied. There are actually *three* categories. Variables of the third category are known as *counters*, and they never show up in Hexkit Game. They are strictly for internal use by rule scripts, providing a simple way to store numerical data between game sessions. Section "Data Persistence" on page 111 explains this mechanism, and why counters are necessary.

You can define counters and basic values for both factions and entities in Hexkit Editor. Since they are never shown to the player you can specify whatever name and informational text you want, although I recommend using these fields to clarify their intended use.

Counters are also useful for controlling scripted computer players. The "Battle of Crécy" and "Battle of Poitiers" demo scenarios use counters to determine the areas defended by the English army, and to indicate which French units are allowed to move freely.

One special use for counters is the storage of map coordinates. As the maximum variable range is the square of the maximum coordinate value, a single counter may encode both the x- and y-coordinate of a map site. The class World.Variable provides helper methods that facilitate this encoding; see the rule script of the "Roman Empire" scenario for an example.

3.5.4 Unit Supply Resources

Factions may be associated with a list of *supply resources*, i.e. resources for which the following propositions are assumed to be true:

— One or more unit classes that are available to the faction maintain basic values of one or more of these resources.

— These unit resources can be depleted (their values can decrease) while a unit is placed on the map, e.g. by taking damage or expending fuel.

— These unit resources can also be restored (their values can increase) while a unit is placed on the map, e.g. by visiting a supply depot. This is called *resupplying*.

— Resupplying depleted unit resources is generally a desirable action.

Any depleted unit resources are resupplied automatically once per turn while the unit is placed on a supply location, via the usual resource accumulation mechanism.

The implementation of supply resources is described in "Calculating Supplies" on page 149. The relevant data is currently only used by computer players and not accessible through the Hexkit Game user interface.

Future Hexkit revisions may provide a more detailed supply model with a dedicated Supply command. In that case, Hexkit Game will also display supply information to human players.

3.5.5 Variable Display Scale

All variable values are internally computed and stored as integers, but sometimes it may be desirable to show fractional values instead.

For instance, the classic computer strategy game *Civilization* defines a movement cost of one point per square for easy terrain such as grassland, and a cost of 0.5 points along roads. This creates the impression that one point per square is the "standard" cost while roads permit movement at a "reduced" cost. Another example would be variable values that correspond to some real-world measure such as tons or liters but are not depleted in whole units.

By defining a *variable display scale* that is greater than one, you can force Hexkit Game to show fractional values for a given variable. All values for that variable are divided by the display scale before Hexkit Game shows them to the user. Note, however, that you must compensate by multiplying all actual variable values by a corresponding factor.

In the *Civilization* example, you would define a display scale of two, and accordingly set the movement cost of grassland to two points (displayed as 1) and that of roads to one point (displayed as 0.5). Any unit movement speed attributes should be likewise scaled.

Chapter 4: Class Structure

This chapter and the following chapters describe the internal structure of the .NET assemblies that implement Hexkit's scenario structure and state of the game world. You'll need a general idea of this structure to understand how to customize Hexkit game rules.

Before continuing, please visit http://www.kynosarges.de/Hexkit.html and download the *Hexkit Class Reference* if you haven't done so already. All Hexkit types and methods are thoroughly documented via Sandcastle, and this guide won't reiterate parameter information or the like that is already covered in the *Class Reference*.

However, the *Class Reference* is a big pile of help pages that is rather impenetrable by itself. So we'll take the game concepts described in the previous chapters, and try to show how they correlate to the implementation details documented in the *Class Reference*.

4.1 Assembly Hierarchy

Figure 10 shows a component diagram of all assemblies deployed by the Hexkit binary package, along with their public contents (with exceptions) and mutual dependencies. Each assembly contains one namespace of the same name as the assembly, and possibly sub-namespaces which are prefixed by the assembly's main namespace.

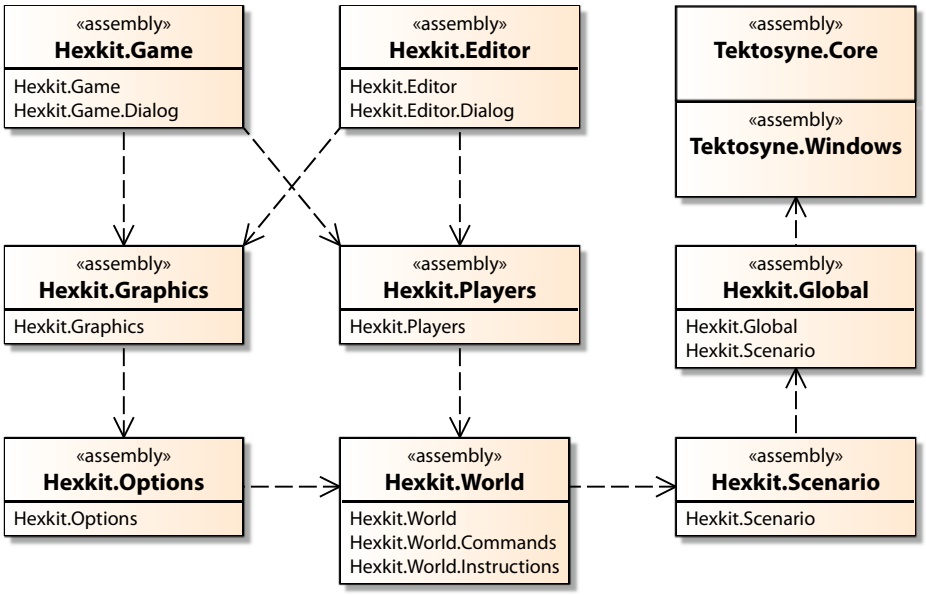


Figure 10: Assembly Hierarchy

The component diagram omits the following items:

— The contents of the three most important assemblies – Tektosyne, Hexkit.Scenario, and Hexkit.World. They would take up too much space, and we'll take a closer look at their relevant classes in the following sections.

— All required BCL assemblies. Hexkit needs a generous helping of those. Please refer to the C# project files in the source package if you're interested.

— Redundant assembly references. For example, all Hexkit assemblies require both the Tektosyne library and the Hexkit.Global assembly, but the only Tektosyne reference shown in the diagram is the one by Hexkit.Global.

Regarding the following sections, you should also note that class diagrams generally show only public members. Protected and internal members may appear as needed. Moreover, the prefix "Hexkit." that precedes all assemblies and namespaces except Tektosyne is usually omitted from both diagrams and text, except when needed for clarity.

4.1.1 Assembly Overview

Starting with the Tektosyne library and moving up the dependency hierarchy, the following list provides a brief description of all assemblies in the preceding diagram.

Tektosyne.Core & Tektosyne.Windows — These are the two assemblies of my general-purpose utility library. Besides various helper classes, this library provides the mathematical representations of polygons and polygon grids, and the pathfinding algorithms used by Hexkit (see "Pathfinding" on page 132).

More information and the latest version are available at the Tektosyne home page, http://www.kynosarges.de/Tektosyne.html.

Hexkit.Global — Provides global constants and helper methods used throughout Hexkit. This includes default file names and paths, XML serialization of standard library types, parsing of command line arguments, global exception handling, and so on.

Hexkit.Scenario — Provides the internal representation of a complete Hexkit scenario, including data validation and XML serialization. "Scenario Assembly" on page 53 gives an overview of the types in this assembly.

Hexkit.World — Provides the state of a Hexkit game world, including command management and XML serialization to a history file. The types in assembly are described in chapter "World State" on page 69 and chapter "Commands Namespace" on page 93.

This is the highest assembly in the reference hierarchy that is visible to rule script code. In particular, the graphics engine is *not* directly accessible to rule scripts.

Hexkit.Options — Manages global user settings, i.e. those that are not specific to a particular game, for all remaining Hexkit assemblies. The principal task of this assembly is the XML serialization of user settings to and from Hexkit options files.

Hexkit.Graphics — Provides the Hexkit graphics engine which can display entire map views as well as individual images for dialog labels or list boxes. The graphics engine is based on the Windows Presentation Foundation, with some direct bitmap manipulation in the map view code to achieve a decent drawing speed.

Hexkit.Players — Manages human and computer players for Hexkit Game, including any settings for individual players. This assembly also provides all computer player algorithms, which are described in chapter "Computer Players" on page 139.

Hexkit.Editor — Provides the Hexkit Editor application. This is essentially a GUI assembly that defines one tab page per scenario section, and a multitude of dialogs for editing all kinds of data in the Hexkit.Scenario namespace.

Hexkit Editor requires the Hexkit.World assembly for the map view display, but cannot edit or save world states. The Hexkit.Players assembly is required to determine the available computer player algorithms and related options.

Hexkit.Game — Provides the Hexkit Game application. This assembly contains all required GUI elements, but also manages interactive replays and game sessions in general (starting, loading, saving, and dispatching to the next player).

4.2 Scenario Assembly

The Hexkit.Scenario assembly and namespace provide the internal representation of a Hexkit game scenario. Most types in this assembly mirror XML elements found in the Hexkit scenario schema, Hexkit.Scenario.xsd. Figure 11 shows the principal classes and their associations.

To simplify this rather complex diagram, a number of auxiliary types were omitted. Some are described in the following sections. Here is a brief overview of the rest:

CategorizedValue — A simple data container that associates integer values with entity categories. Used by Hexkit Game to conveniently manage faction resources.

ScenarioElement — The abstract base class for all Scenario classes that represent Hexkit scenario elements, i.e. XML elements defined in the scenario schema.

All classes that inherit from ScenarioElement are noted in the diagram.

SectionPaths and SectionUtility — Two helper classes for managing scenario sections and their XML representations on disk.

XmlSerializable — The abstract base class for ScenarioElement. This class handles XML serialization while ScenarioElement validates and manipulates identifiers.

Now let's take a closer look at the remaining classes, grouped by scenario section according to the list in "Scenario Sections" on page 33. Each section is implemented by an eponymous class.

4.2.1 About the Section Diagrams

The UML diagrams for each scenario section usually show only public methods and properties. The hidden internal and protected members handle XML serialization and identifier validation. Most of them override functionality defined in the common base class ScenarioElement.

All classes that represent scenario elements support XML serialization and, except for abstract base classes, contain a constant field named ConstXmlName. This field holds the name for the XML element that corresponds to the class, as defined in Hexkit.Scenario.xsd.

4.3 MasterSection Class

The Master section represents an entire Hexkit scenario. As you can see in Figure 12, an instance of the MasterSection class contains one instance of each subsection class and directly stores the following scenario data:

Information — Contains the scenario's author, version information, and descriptive text.

RuleScript — Manages the scenario's rule script. This includes storing the file path, compiling the rule script if necessary, and loading the compiled assembly.

Please refer to "Rule Script Files" on page 108 for details on this procedure.

MasterSection.Title — A string that contains the scenario's title.

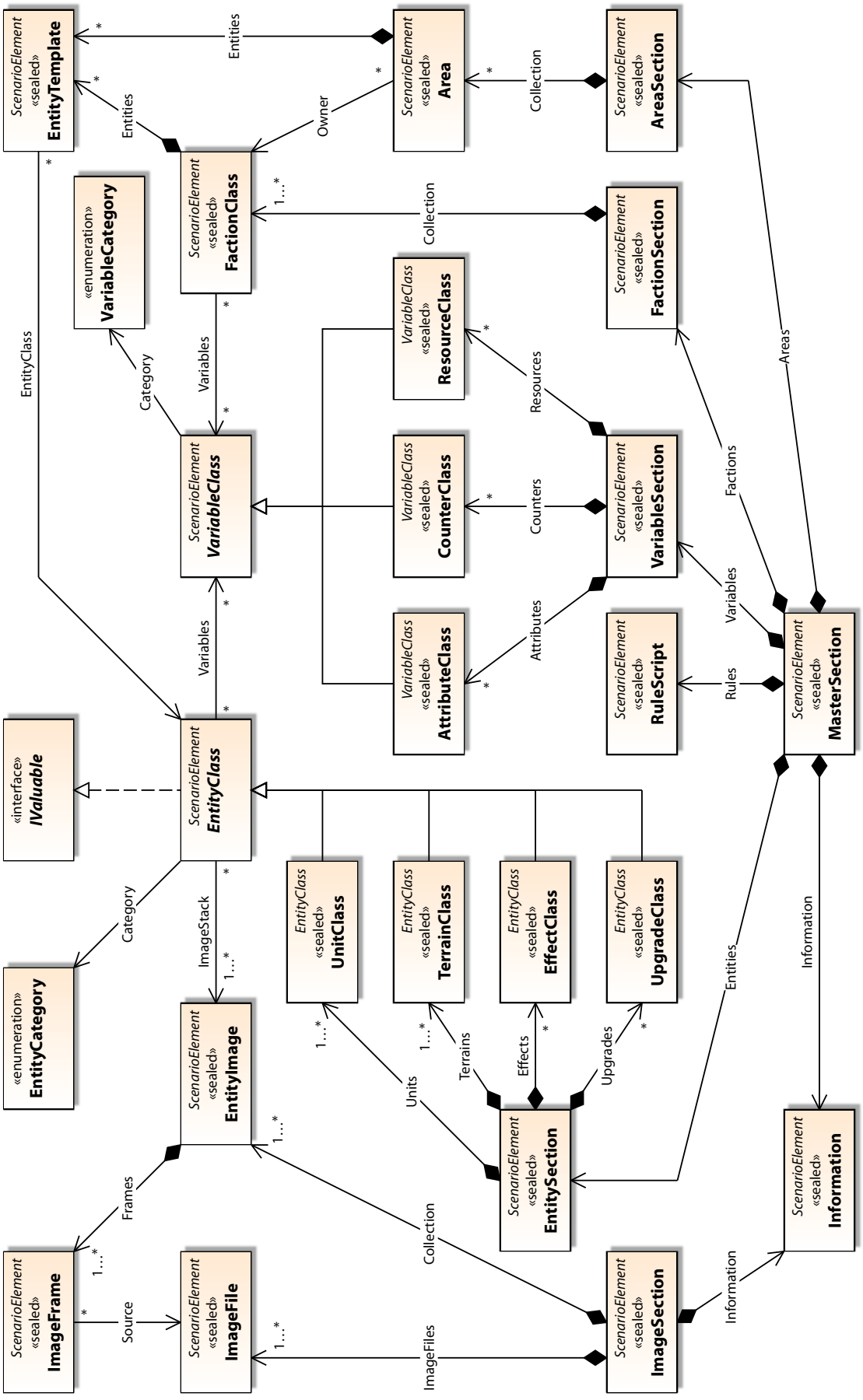


Figure 11: Scenario Namespace

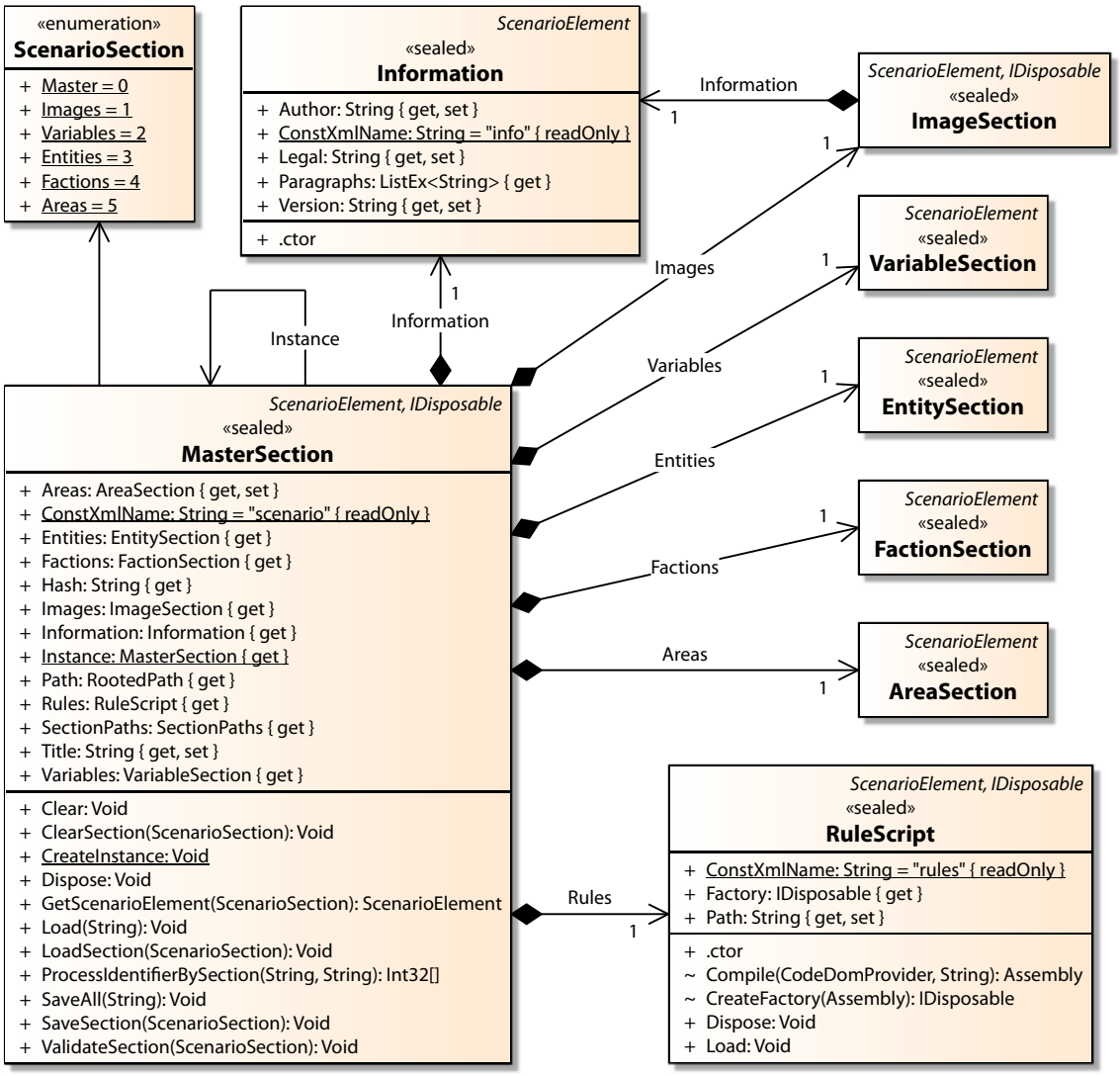


Figure 12: MasterSection Class

The remaining MasterSection members are used to save, load, and validate scenario data.

MasterSection is disposable because it contains instances of the disposable ImageSection and RuleScript classes. ImageSection is disposable to facilitate early unloading of unused bitmaps, whereas RuleScript implements IDisposable as a service to rule script authors who might want to perform clean-up operations when a game session ends.

The ScenarioSection enumeration holds symbolic names for all scenario sections. This enumeration is part of the Hexkit.Scenario namespace but stored in the Hexkit.Global assembly where it is required by various methods that handle section files.

4.3.1 The Scenario Instance

The MasterSection class implements the singleton pattern. All code anywhere in Hexkit, including the rule script, should access scenario data through the static property Hexkit.Scenario.MasterSection.Instance which returns the current global instance of the MasterSection class, also known as the current *scenario instance*.

Hexkit Editor maintains a single valid scenario instance throughout the lifetime of the application. Loading a new scenario replaces the data of the existing instance.

Hexkit Game creates a new scenario instance whenever a new game is started, and deletes the instance when the game is closed. The Instance property is a null reference when no game is running. However, this is not relevant to rule scripts or Hexkit.World methods since their code is never executed unless a valid scenario exists.

4.3.2 Identifiers and Objects

Subsections often contain pairs or dictionaries that map strings to another data type. Such strings are unique internal identifiers that represent XML IDREF or IDREFS attributes, as described in section "Identifiers and Names" on page 36. The referenced XML ID attribute is stored in the (obligatory) Id property of the target data type, and the second component of the pair is the actual object that contains the referenced Id value.

4.4 ImageSection Class

The Images section manages the bitmap data described in "Bitmap Tiles" on page 40. Figure 13 shows the central ImageSection class and the following associated classes:

EntityImage — Defines a single image for an entity class, i. e. a collection of one or more image frames with optional animation that is accessed by a unique identifier.

ImageFile — Contains the file path and, once loaded, the bitmap data of an image file.

ImageFrame — Defines a single image frame. That is a rectangular bitmap tile, measured by the Bounds property, within the image file referenced by the Source property.

Some frames also define visual connections to neighboring polygons at one or more edges of the bitmap tile, stored in the Connections property. This is a list of Compass values which are described in "Geometry Classes" on page 134.

AnimationMode and AnimationSequence — Specify whether multiple frames defined by an image constitute an animation sequence, and what kind of sequence. See "Frames and Animation" on page 42 for details.

ImageScaling — Specifies if and how the frames defined by an image are scaled to the size and shape of a map polygon. See "Scaling" on page 41 for details.

Information — Contains the name(s) of the artist(s) who created the bitmap tiles, along with version information and descriptive text.

The ImageFile class is disposable since it references a potentially large bitmap that is no longer required once the master bitmap catalog has been created. Thus, the ImageSection class which contains a collection of image files is disposable as well.

4.4.1 Displaying Images

The ImageSection class does not actually display screen images. This task is handled by the Hexkit graphics engine which is located in the Hexkit.Graphics assembly.

All ImageSection data is transferred into another internal format for this purpose, namely the so-called "catalog" bitmaps that contain all image tiles required by the scenario. This process is detailed in "Composing Image Stacks" on page 44.

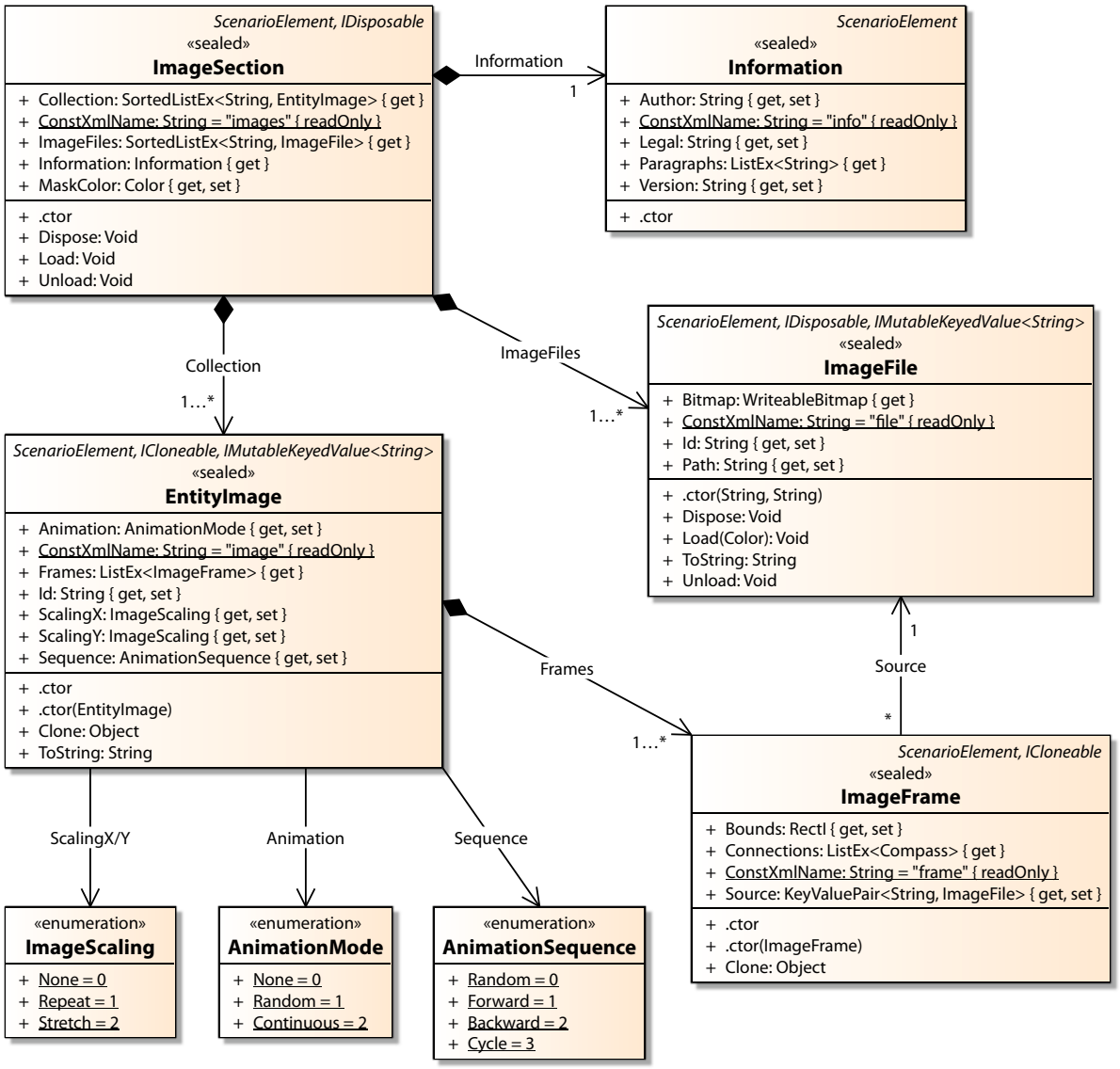


Figure 13: ImageSection Class

The types of the Hexkit.Graphics assembly are *not* available to rule script code. If you wish to manipulate the display from a rule script method you must emit HCL instructions that represent command events, as described in "Instructions Namespace" on page 99.

4.5 VariableSection Class

The Variables section implements the numerical game variables, a.k.a. "statistics", defined in "Variables" on page 19. The entity abilities also mentioned there are implemented by the Entities section (see "EntitySection Class" on page 59).

4.5.1 VariableSection Members

The VariableSection class defines three categorized lists of available variable classes – Attributes, Counters, and Resources – as well as several helper methods for its VariableClass collections.

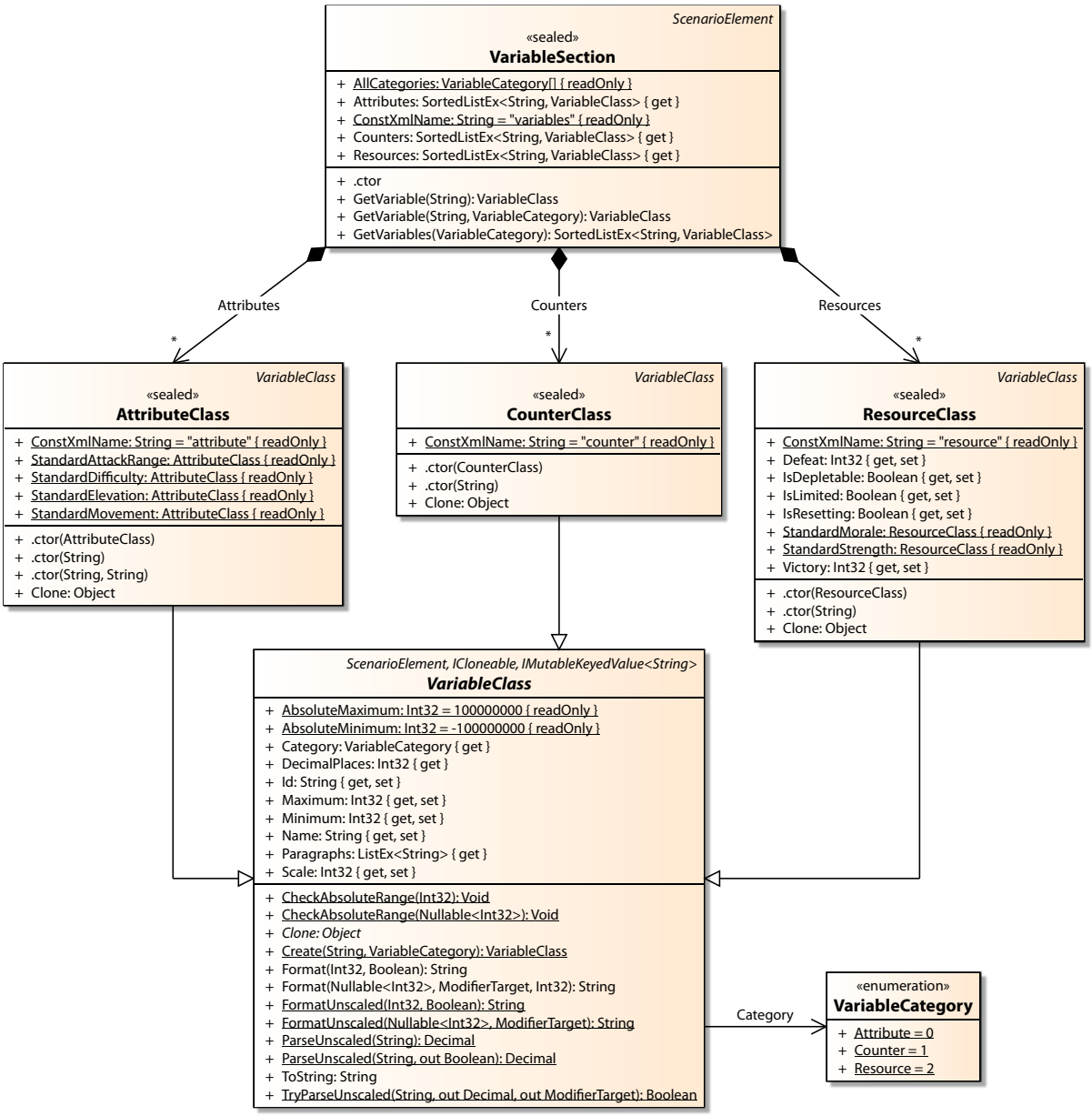


Figure 14: VariableSection Class

GetVariable — Finds the VariableClass with a given identifier, in any or a specific collection.

GetVariables — Gets the VariableClass collection for the specified category.

4.5.2 VariableClass Members

The abstract base class VariableClass defines a number of numerical values and helper methods that may be of interest. Some are accessible through Variable instances as well.

The handling of variable modifiers involves ModifierTarget values because entities may define target-specific modifiers, as described in "VariableModifier Members" on page 62.

AbsoluteMinimum and AbsoluteMaximum — The greatest range for any basic and modifier values of any variable, excepting intermediate values during variable calculations.

Minimum and Maximum — The legal range for the basic values of a specific variable. In the *Class Reference*, this is also called the variable's *result range*.

Scale — The variable's display scale; see "Variable Display Scale" on page 50. Setting the Scale also sets the DecimalPlaces shown by Format.

CheckAbsoluteRange — Checks a value against AbsoluteMinimum and AbsoluteMaximum.

Format — Formats a value with the correct number of DecimalPlaces for the variable's Scale, either as a basic value or as a modifier value for a given target.

This method is intended for displaying variable values in Hexkit Game.

FormatUnscaled, ParseUnscaled, TryParseUnscaled — Formats a value or parses a string without any DecimalPlaces, either as a basic value or as a modifier value for a given target.

These methods are intended for round-tripping variable values in Hexkit Editor.

4.5.3 Standard Variables

The derived classes AttributeClass and ResourceClass define several constant fields of the same type, prefixed with Standard. They represent the standard variables recognized by the default rules – see "Standard Variables" on page 83 and "Standard Variables" on page 119 for details.

These objects are "pseudo-variables" that do not appear in the corresponding VariableSection collections, and that have no valid properties except for a "magic" identifier and a display name. They are used internally to mix standard variables with concrete variables in GUI code, but rule script authors might also wish to use them when referring to standard variables.

4.5.4 ResourceClass Members

The derived class ResourceClass defines a few additional properties which affect basic values of the corresponding resources.

Defeat and Victory — Applies to faction resources. A faction loses if one of its resources drops to or below the resource's Defeat threshold, and a faction wins if one of its resources rises to or above the resource's Victory threshold.

IsDepletable — Applies to entity resources. The resource is expected to oscillate between its minimum and maximum values if its IsDepletable property is true, and Hexkit Game will visualize its current depletion level.

IsLimited — Applies to entity resources. An entity cannot raise a resource above its initial value, regardless of modifiers, if its IsLimited property is true.

IsResetting — Applies to faction resources. A faction's resource is reset to its initial value at the start of each turn, before adding modifiers, if its IsResetting property is true.

4.6 EntitySection Class

The Entities section implements the concepts defined in "Entities" on page 17. Figure 15 shows an overview diagram, and Figure 16 on page 61 shows EntityClass and several related types.

The IValuable interface that appears in Figure 16 is only relevant to computer players, so we defer its description to the corresponding chapter, "Context-Free Valuation" on page 140.

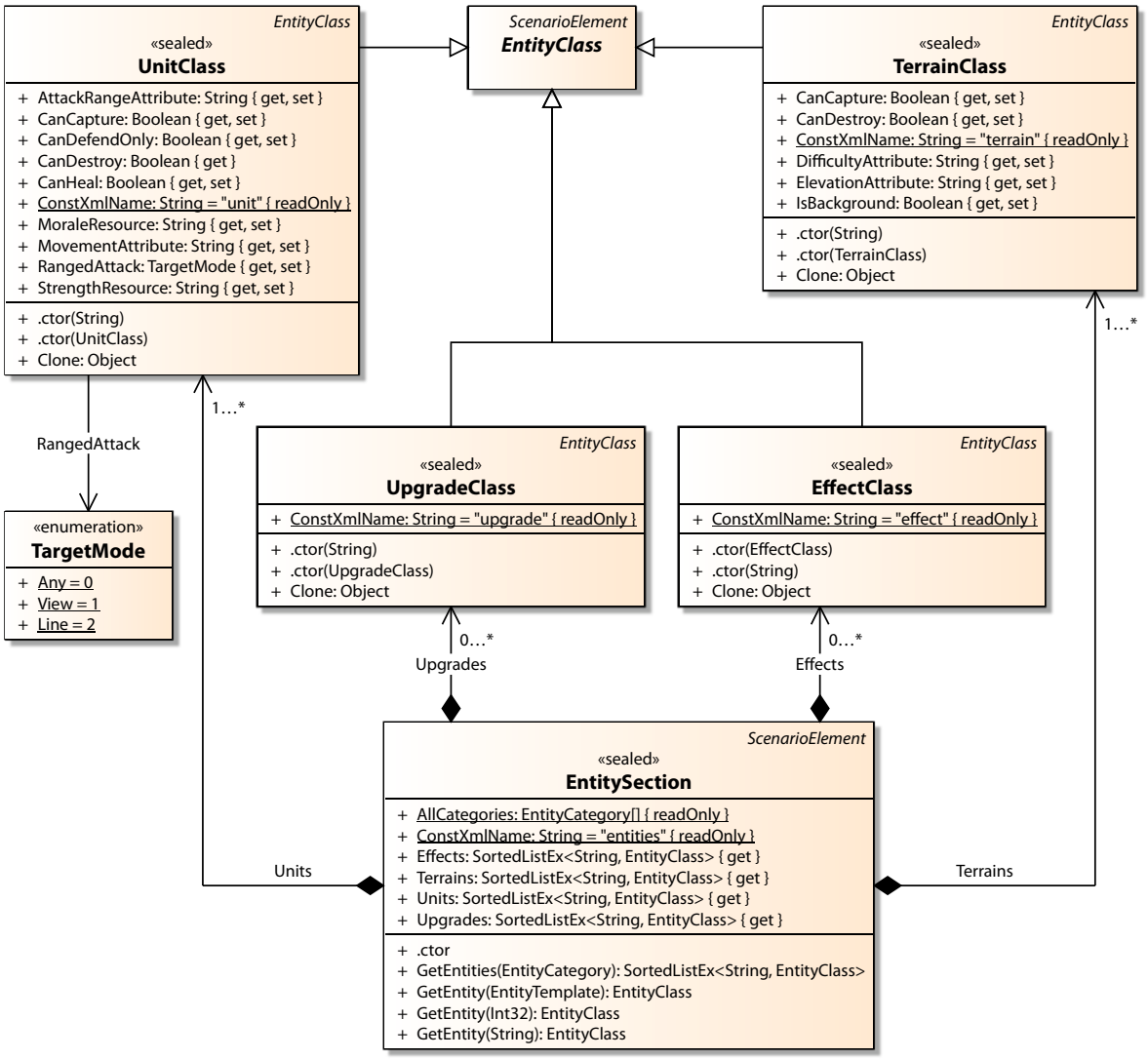


Figure 15: EntitySection Class

4.6.1 EntitySection Members

The EntitySection class defines four categorized lists of available entity classes – Effects, Terrains, Units, and Upgrades – as well as several helper methods for these EntityClass collections.

GetEntity — Finds the EntityClass with a given identifier (optionally stored in an EntityTemplate) or bitmap catalog index, in any collection.

GetEntities — Gets the EntityClass collection of the specified category.

4.6.2 EntityClass and Children

Many properties of the abstract base class EntityClass and its derived classes are duplicated by the corresponding Hexkit.World classes, insofar as they are relevant to gameplay code. Therefore, we defer their description to "Entity Class" on page 78.

BuildResources — Specifies the amount of each resource that a faction must expend to build a new instance of the entity class. See "Build Command" on page 125.

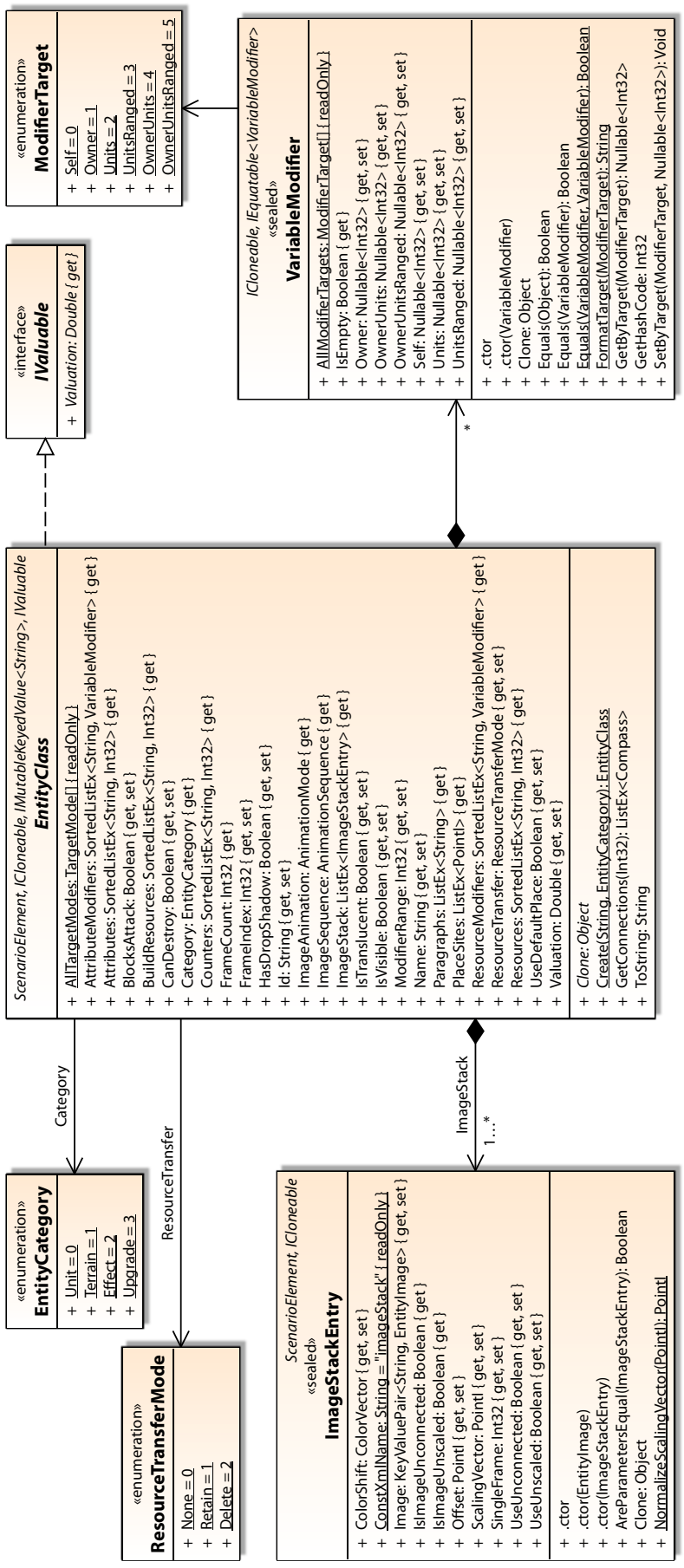


Figure 16: EntityClass Class

ModifierRange — Specifies the range of any variable modifiers with a unit target, or zero to modify units anywhere on the map. See "Attributes and Resources" on page 116.

PlaceSites and UseDefaultPlace — Determine the legal placement locations for instances of the entity class. See "Place Command" on page 129.

ResourceTransfer — Indicates whether instances of the entity class provide automatic resource transfer to other entities, and whether they are automatically removed from the game upon depletion. See "Attributes and Resources" on page 116.

4.6.3 VariableModifier Members

Whereas basic variable values and faction modifiers are stored as simple integers associated with variable identifiers, entities may define multiple modifier values for the same variable. The two entity modifier collections, AttributeModifiers and ResourceModifiers, associate each variable identifier with a VariableModifier instance that categorizes modifier values by target.

IsEmpty — Indicates whether all target-specific modifier values are either zero or undefined.

Self, Owner, Units, etc. — The modifier value for each possible ModifierTarget, or a null reference if the EntityClass does not define a modifier for the associated VariableClass.

GetByTarget and SetByTarget — Gets or sets the modifier value for a given ModifierTarget.

The Hexkit.World assembly transforms all variable values into a different format with built-in caching, as described in "Variable Class" on page 84. Rule script code still needs to use the Hexkit.Scenario format when retrieving a variable's original scenario value, however.

4.6.4 Other EntityClass Members

The remaining properties of EntityClass and its children are mostly intended for the Hexkit graphics engine and listed here for completeness.

ImageStack — The image stack of an entity class, composed of ImageStackEntry instances.

ImageAnimation and ImageSequence — Contain the highest Animation and Sequence values of any image stack entry that uses at least two frames, as described in "Composing Image Stacks" on page 44.

FrameIndex and FrameCount — The index of the first bitmap catalog tile and the total number of catalog tiles holding the combined image stack frames. FrameIndex is set during initialization of the Hexkit graphics engine after starting a new scenario.

The bitmap catalogs contain only frames that are actually used by any entity class, but the catalogs will contain duplicate tiles if different entity classes use identical image stacks. Optimizing away such rare duplicates is probably not worthwhile.

TerrainClass.IsBackground — Indicates whether the class represents background terrain. Once the game map has been created, every site contains exactly one background terrain, located at the bottom of the site's terrain stack.

HasDropShadow — Indicates whether the image stack is drawn with a drop shadow, so as to better distinguish the entity against background and other terrains.

IsTranslucent — Indicates whether the image stack contains semi-transparent pixels. Applies only to the Use Opaque Images performance option – see online help for details.

IsVisible — Indicates whether the image stack is drawn on map views. May be disabled for certain entity classes (usually background terrains) when using an overlay image.

4.6.5 ImageStackEntry Members

The image stack of an entity class does not contain raw EntityImage references but rather ImageStackEntry objects that associate each image with optional display parameters. These parameters control how the individual stack entries are merged into bitmap catalog tiles, as described in "Composing Image Stacks" on page 44.

Image — The actual image providing one or more frames to the stack entry.

ColorShift — An optional RGB vector that shifts the color channels of all image frames.

Offset — An optional two-dimensional pixel offset that shifts the position of all image frames, relative to the default position that is centered within a map polygon.

ScalingVector — An optional two-dimensional scaling vector that is currently only used to mirror all image frames. Negative values indicate that each image frame is mirrored in the corresponding dimension. Non-negative values have no effect.

SingleFrame — An optional frame selector that controls which image frames are used by the stack entry. The negative default value uses all image frames. A non-negative value is the index of the single frame to use.

UseUnconnected — Indicates whether the stack entry ignores any neighbor connections defined by image frames.

UseUnscaled — Indicates whether the stack entry ignores any scaling options specified by the image. This flag does not affect the associated ScalingVector.

The various public methods are used by Hexkit Editor to compare and change the values of these properties when editing an image stack.

4.7 FactionSection Class

The Factions section provides initial and constant data for all player factions, as described in "Factions" on page 14. The FactionSection class itself contains merely a list of all factions in the game, represented by FactionClass objects. The element order in FactionSection.Collection defines the factions' activation order.

The EntityTemplate class defines a pre-created entity instance that belongs to a faction and/or resides on the map. As with FactionClass objects, each EntityTemplate object corresponds to exactly one entity in the game. This class is described in "AreaSection Class" on page 65.

Most of the types in this section are not directly used by gameplay code. At the start of a game, all data is converted into equivalent Hexkit.World or Hexkit.Players types which provide copies or proxies for all relevant scenario properties.

4.7.1 Condition Members

The Condition structure defines a victory or defeat condition that applies only to the associated faction, as described in "Victory and Defeat" on page 16.

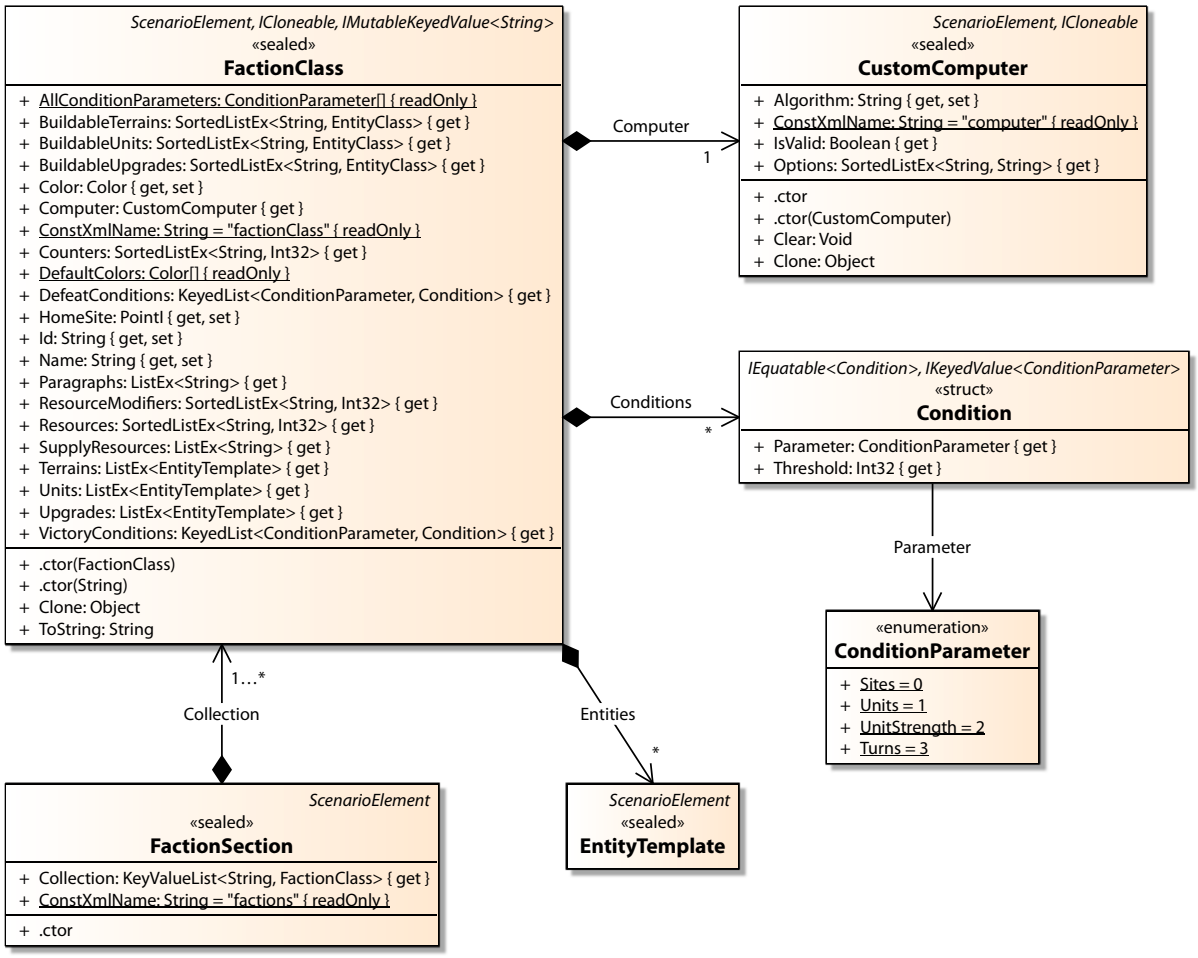


Figure 17: FactionSection Class

Parameter — A ConditionParameter value indicating the game parameter on which the condition is based: owned sites, number or strength of owned units, or elapsed turns.

Threshold — A numerical threshold that triggers the condition when reached or exceeded by the associated Parameter, depending on the type of the condition.

4.7.2 CustomComputer Members

The CustomComputer class can be used to specify a customized computer player algorithm that should control the associated faction at the start of a new game.

Algorithm — The identifier of the computer player algorithm that controls the faction.
If this property is a null reference or an empty string, the faction is controlled by its default human or computer player.

IsValid — Indicates whether Algorithm contains a non-empty string.

Options — A dictionary with the names and desired values of zero or more options offered by the specified Algorithm.

4.8 AreaSection Class

The Areas section manages the game map and all related data. Most of this data represents the initial map contents and is transformed into a different format before a game starts, namely a two-dimensional array of World.Site objects (see "WorldState Class" on page 70).

4.8.1 AreaSection Members

AreaSection.MapGrid returns the Tektosyne.Geometry.PolygonGrid object that defines the fixed geometrical structure of the game map. Use this object to determine the size of the game map, the step distance between two locations, all neighbors of a given location, etc.

The size and shape of each polygonal map element is defined by the Element property of the MapGrid which returns a Tektosyne.Geometry.RegularPolygon. The AreaSection class defines a static DefaultPolygon that is the default value for this property. Please refer to "Geometry Classes" on page 134 for more details on these Tektosyne classes.

AreaSection also maintains instances of all available pathfinding algorithms that are based on the current MapGrid. These algorithms are described in "Pathfinding Classes" on page 132.

Faction and Entity Locations

AreaSection manages all locations associated with factions and entities. Since map coordinates depend on the size and contents of the game map, they must be stored with the class that defines the map, even if they logically belong elsewhere.

HomeSites — A dictionary that maps faction identifiers to the home site of each faction. This collection backs all FactionClass.HomeSite properties.

AllPlaceSites — A dictionary that maps entity class identifiers to all custom placement sites for each class. This collection backs all EntityClass.PlaceSites properties.

Hexkit Editor provides editing facilities for home sites on the Factions page, and for placement sites on the Entities page. However, since these coordinates are actually stored with the Areas section, the Areas page will be marked as changed if you edit them.

OverlayImage Class

OverlayImage describes a static overlay image that provides map graphics or serves as an editing aid, as described in "Static Overlay Images" on page 47. This class is used only by the Hexkit graphics engine, never by gameplay code. We briefly list its members for completeness.

Path and IsEmpty — The image's file path, and a flag indicating an empty path (i.e. no image).

Bitmap and Load — The actual bitmap found at Path, and the method that loads the bitmap.

AspectRatio and PreserveAspectRatio — The original aspect ratio of the loaded Bitmap, and a flag that requests preserving that aspect ratio while changing the Bounds.

Bounds — The user-defined display bounds of the loaded Bitmap.

Opacity — The opacity of the loaded Bitmap, ranging from zero to one for the OverlayImage that is intended as an editing aid and appears *above* the map grid. Always 1.0 for the AreaSection.Overlay image which appears *below* the map grid.

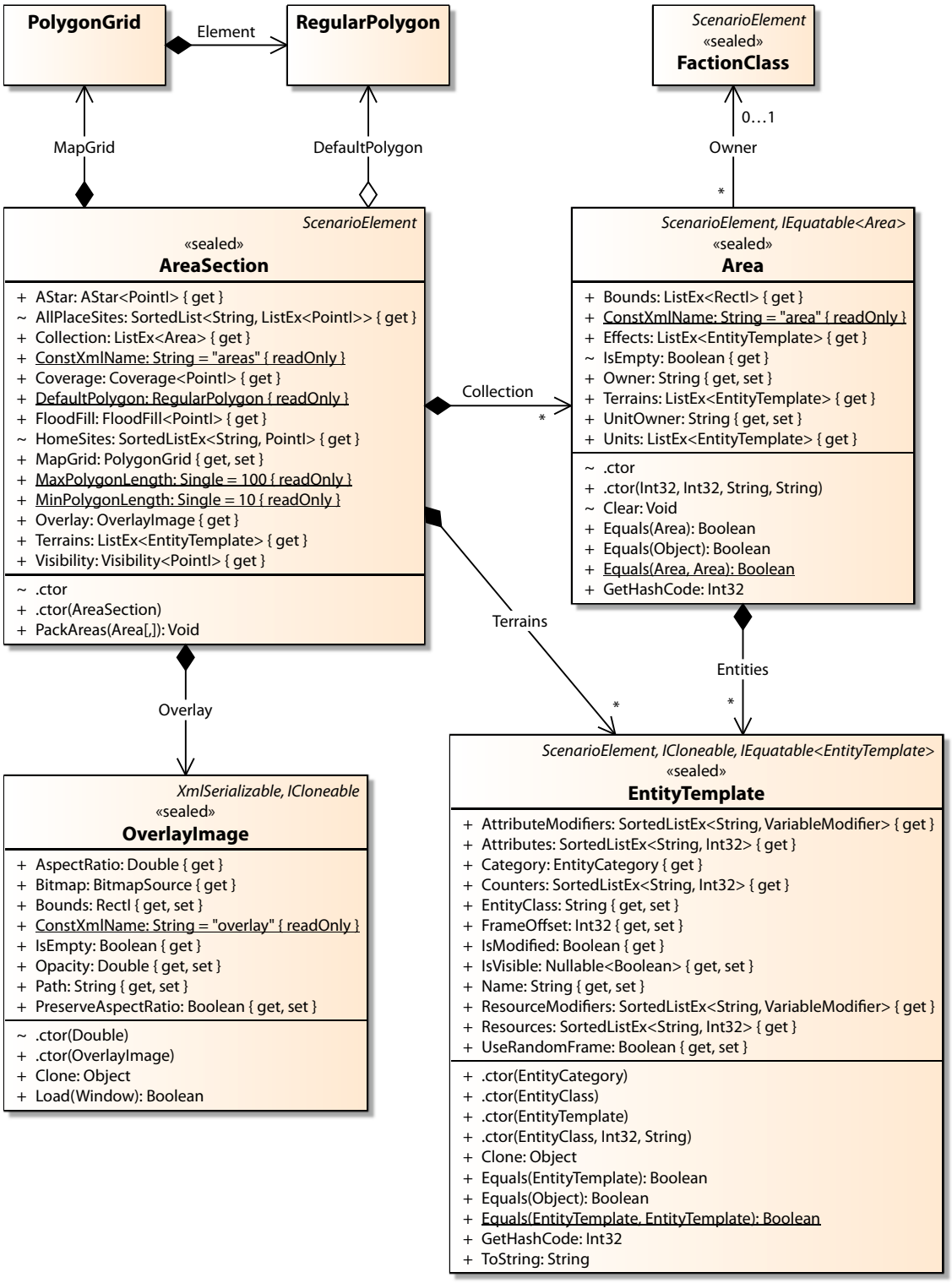


Figure 18: AreaSection Class

4.8.2 EntityTemplate Members

Each instance of the EntityTemplate class defines exactly one pre-created entity instance that belongs to a faction and/or resides on the map.

Attributes/Modifiers, Counters, Resources/Modifiers — Initial variable values for the created entity that override those defined by the associated EntityClass.

Category — The category of the EntityClass to instantiate. Used for validation.

EntityClass — The identifier of the EntityClass to instantiate.

FrameOffset — The index of the image frame to show. Only used if the instantiated entity class defines multiple bitmap catalog tiles and does not use animation.

IsModified — Indicates whether the EntityTemplate contains any custom data at all.

IsVisible — Overrides the eponymous EntityClass flag when not a null reference.

Name — An optional individual name for the entity instance.

UseRandomFrame — Indicates whether the entity shows a randomly determined image frame, rather than the one specified by FrameOffset.

The XML name of an EntityTemplate object is either terrain, unit, effect, or upgrade, as determined by its category.

4.8.3 Map Storage Format

The format used to store map contents in the Areas section, and in the corresponding XML section file, was designed to minimize the space taken up by large uniform map areas.

AreaSection.Terrains — A collection of EntityTemplate objects of the Terrain category that defines the *default terrain stack*. The equivalent terrain entities are placed on all map sites before individual Area objects are evaluated.

AreaSection.Collection — A list of Area objects that represent individual *map areas*. The contents of areas with overlapping Bounds are added on top of each other, in the order in which the areas appear in this collection. The PackAreas algorithm does not currently generate overlapping areas, however

Area.Bounds — A collection of rectangles indicating all map locations covered by the area. The rectangles are not necessarily adjacent. The PackAreas algorithm guarantees that rectangles in the same collection never overlap.

Area.Effects, Area.Terrains, Area.Units — Three collections of EntityTemplate objects that define the area's three *entity stacks*. The equivalent entities are placed on all map sites covered by the area's Bounds.

Terrain stacks that contain a background terrain *replace* the default terrain stack; otherwise, their contents are added on top of the default terrain stack.

Area.Owner and Area.UnitOwner — The identifier of the player faction that owns all map sites within the Bounds of the area, or all units placed on such sites, respectively.

Packing the Map

When Hexkit Editor saves the game map, the equivalent WorldState.Sites array (see "WorldState Class" on page 70) is first converted into an array of single-site Area objects containing all local entity stacks, minus the default terrain stack if present.

The method AreaSection.PackAreas then "packs" this array by transforming multiple areas with identical entity stacks into a single area with larger Bounds. Empty areas, i.e. those whose sites contain only the default terrain stack, are dropped altogether.

The Area members Clear, Equals, and IsEmpty are helper methods for this algorithm. The resulting Area objects constitute the new AreaSection.Collection and are serialized to the Area section XML file as usual.

Unpacking the Map

Loading a scenario reverses this process. All elements in the WorldState.Sites array first receive the default terrain stack, then any additional entity stacks defined by any Area object that covers the same site. A site's existing terrain stack is replaced whenever an Area with a new background terrain is encountered.

This behavior is not exactly optimal. Many terrain entities are created only to be immediately destroyed by a different background terrain. (They do not appear in the Event History dialog, however, because we defer all history recording until after the world state is fully established.) A future Hexkit version may do a preliminary pass over all Area objects to determine which sites should receive a default terrain stack in the first place.

Chapter 5: World State

This chapter continues our overview of Hexkit's class structure with the second big assembly that implements gameplay mechanics. While the Hexkit.Scenario assembly mirrors the static contents of a scenario file, the Hexkit.World assembly manages all the individual factions and entities as they exist and change during a game. Their totality is known as the current *world state*.

Hexkit.World also implements game commands and provides hooks for their customization by a scenario rule script. Both subjects are discussed in the following chapters.

5.1 World Assembly

The Hexkit.World assembly and namespace provide the internal representation of the state and history of a Hexkit game world, i. e. the state of a game in progress.

World states are serialized to XML files that conform to the Hexkit session schema, Hexkit.Session.xsd. As outlined in "The Command History" on page 11, only a history of game commands is serialized, not the entire world state.

Figure 19 shows the principal classes and their associations. We'll have a closer look at them in the following sections. Some types were omitted to reduce clutter, such as these:

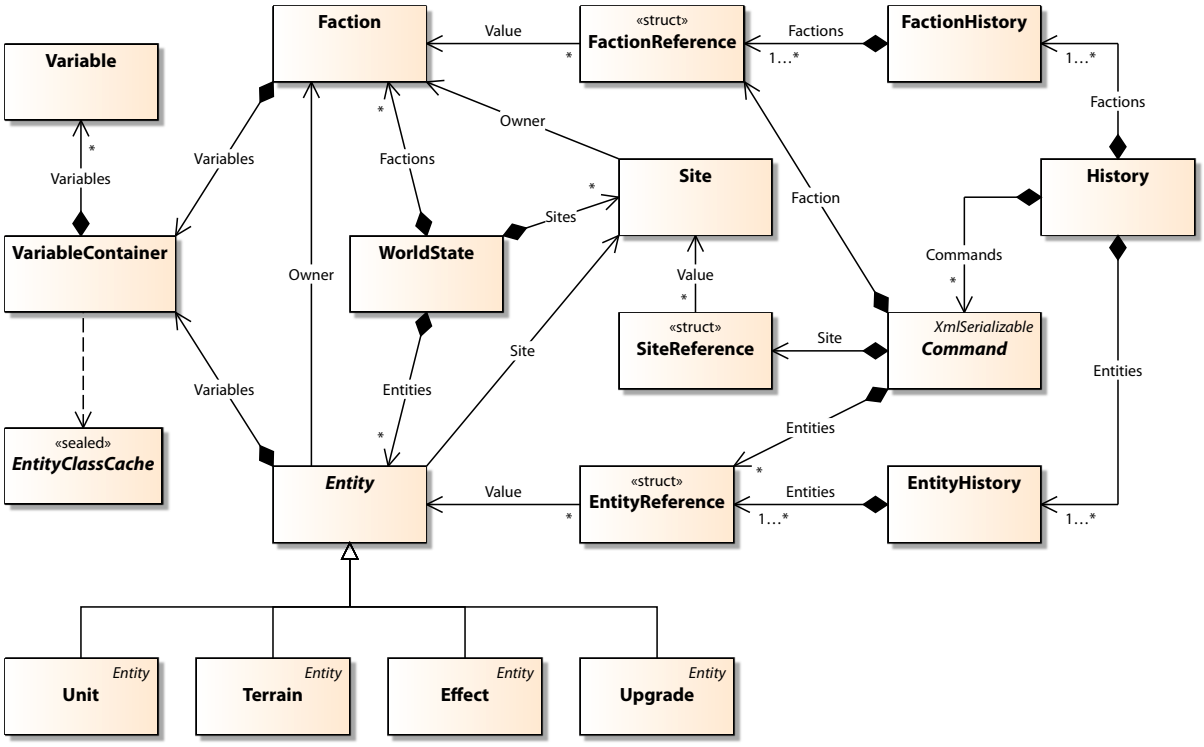


Figure 19: World Namespace

Commands and Instructions — Of the sub-namespaces that define game commands and HCL instructions, only the abstract base class Command is present. The remaining classes and related topics are covered in chapter "Game Commands" on page 90.

Finder — A static class with proxy properties and helper methods for finding movement paths and other map locations. See "Pathfinding Classes" on page 132.

InvalidCommandException — The exception that is thrown when a game command or HCL instruction contains invalid data. The reason is either an incompatible or corrupt save game or an internal error in Hexkit's command generation routines.

IRulesFactory — The interface implemented by the RulesFactory class of a rule script.

The only instance of this type is accessible through the global scenario instance. See "Rule Script Files" on page 108 for details.

RangeCategory — An enumeration that defines range categories for map distances.

This type is used by computer player algorithms to classify targets, and may be used by a rule script for similar calculations. See "Range Categories" on page 144.

ValuableComparer — A class that allows sorting IValuable instances by their valuation, as performed by a specific faction. See "Evaluating Possessions" on page 140.

WorldUtility — A static class with helper methods for common tasks such as manipulating collections of entities and variable values.

XmlSerializable — The abstract base class for all World classes that can be serialized to XML files. This includes the History class and all classes that represent game commands or HCL instructions.

5.1.1 About the Class Diagrams

The UML diagrams for the Hexkit.World namespace generally show all methods and properties with public or protected visibility. Most public properties are read-only when accessed from outside the assembly, and public collections generally expose their elements through a read-only wrapper. Rule scripts may emit HCL instructions to set the values or elements of such properties, as described in chapter "Game Commands" on page 90.

History is the only World class that supports XML serialization and thus contains a constant field named ConstXmlName. This field holds the name for the XML element that corresponds to the class, as defined in Hexkit.Session.xsd.

Most of the classes in the nested Commands and Instructions namespaces also support XML serialization. Their XML element names equal their type names, and are discovered by reflection at runtime; hence, no ConstXmlName fields are needed.

5.2 WorldState Class

The WorldState class manages all mutable data for a state of the game world. This is essentially any data described in chapter "Hexkit Game" on page 8 that is not strictly scenario data.

A WorldState instance directly stores the command and event history, an array of all map sites, and a list of all surviving player factions. Sites and factions, in turn, store all entities in the world state: units, terrains, effects, and upgrades.

5.2.1 Current World States

Hexkit Editor and Hexkit Game each maintain a *current world state* in their respective application assemblies. These world states are accessible through the following properties:

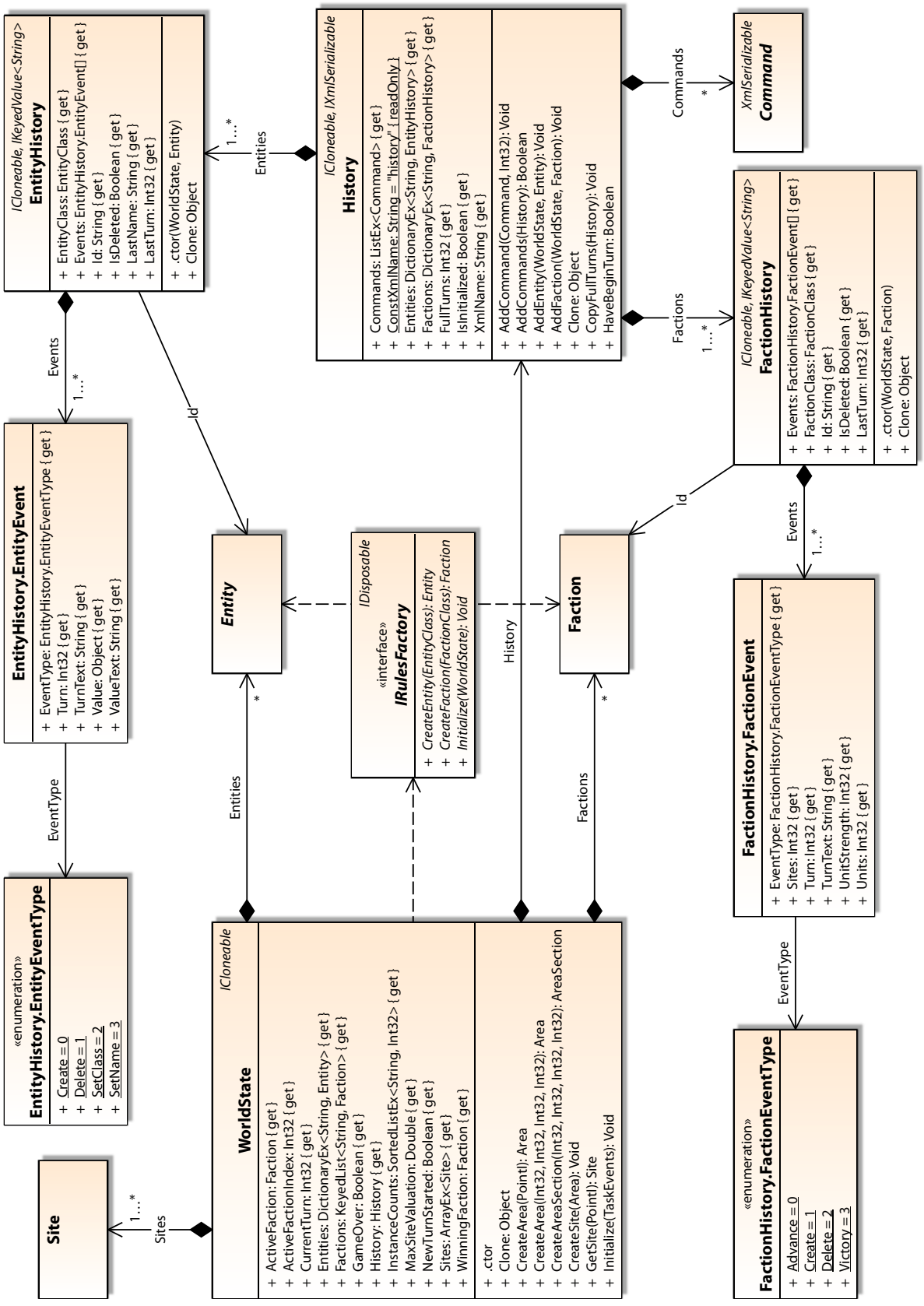


Figure 20: WorldState Class

Hexkit Editor — Hexkit.Editor.AreasTabContent.WorldState. The global instance of AreasTabContent is provided by Hexkit.Editor.MainWindow.Instance.AreasTabContent.

Hexkit Game — Hexkit.Game.Session.WorldState. The global Session instance is provided by Hexkit.Game.Session.Instance while a game is running.

Gameplay code, including rule scripts, cannot access these assemblies because they sit at the top of the dependency chain. This does not matter, however, since gameplay code must be able to operate on independent copies anyway, as we'll see in the next paragraph. Any method that might require a WorldState object therefore receives it as a parameter.

5.2.2 Copying World States

The WorldState class and all subordinate classes support *deep copies* through the ICloneable interface. Computer player algorithms always operate on a copy of the current world state, and may create more copies as required by turn prediction algorithms.

When a computer player ends its turn, the copied world state on which its commands were executed is swapped with the current world state, and automatic command replay begins with the first command that was not present in the original world state.

5.2.3 Useful WorldState Members

Here's an overview of the public members of the WorldState class, insofar as they are relevant to gameplay code and aren't covered in conjunction with another World type:

ActiveFaction and ActiveFactionIndex — The Factions element that is currently active, and its index in the Factions collection, respectively.

CurrentTurn — The zero-based index of the current game turn.

NewTurnStarted — Indicates whether the ActiveFaction is the first faction to become active during the current game turn.

This property defaults to true at the start of a new game and is updated automatically whenever a new faction becomes active. Use it to guard actions that should be executed only once per turn, usually in Faction.BeginTurn.

Entities — A hashtable that contains all Entity objects which are currently present in the game, regardless of their category. Elements are accessed by their identifiers.

Entities are automatically added to the Entities collection when they are created, and removed from the hashtable when they are deleted from the game.

Factions — A list of Faction objects that represents all surviving factions in the game. The order in which factions appear in this list determines the order in which they become active during a game turn.

Factions are automatically removed from the Factions collection when they are defeated. It is not possible to add factions to this list once a game has started.

Sites — A two-dimensional array of Site objects that represent all map locations and their current contents. This array has the same size as the Scenario.AreaSection.MapGrid of the global scenario instance, with the grid's width mapped to the first dimension.

GetSite — Returns the Sites element at the specified map coordinates, or a null reference if the coordinates are invalid.

5.2.4 Other WorldState Members

Some other WorldState members are only used internally but listed here for completeness:

GameOver and GameWasOver — Indicates whether the game has ended, and whether the game was resumed from a saved game that had already ended, respectively.

InstanceCount — The number of times each EntityClass identifier was instantiated by an Entity object. This mechanism is used to create unique internal entity identifiers.

WinningFaction — Returns the Factions element that has won the game once GameOver is true, or a null reference if all factions were eliminated.

CreateArea and CreateAreaSection — Creates Area or AreaSection objects from the contents of the specified map rectangle. Used by Hexkit Editor to save map changes, and by Hexkit Game to save a "snapshot" of the current map.

CreateSite — Creates Site objects from the specified Area object and adds them to the Sites array. Used by Hexkit Editor and Hexkit Game to create a new Sites array from the AreaSection instance of the current scenario.

Initialize — Initializes all data of the WorldState instance from the data of the current scenario. When called by Hexkit Editor, all entities and factions are created with their built-in factory methods. When called by Hexkit Game, the IRulesFactory instance provided by the Scenario.RuleScript class of the current scenario is used instead.

5.2.5 History Classes

Most of the classes shown in Figure 20 are used to record the history of the associated WorldState instance, as follows:

History — The complete history of a world state. Commands is the list of all executed game commands, Entities and Factions record all important events that affected such objects, and FullTurns is the total number of completed game turns.

Gameplay code should never call any of the public methods of the History class, as the history lists are maintained internally by Hexkit.

EntityHistory — The status and history of an Entity object. Events contains important events that affected the entity, which includes creation, deletion, and name changes.

FactionHistory — The status and history of a Faction object. Events contains important events that affected the faction, which includes creation, deletion, advancing a turn, and winning the game. Each FactionEvent also records the number of sites and the number and strength of all units owned by the faction at that point in time.

The Event History dialog shows all events recorded for any entity or faction, whether dead or alive. Moreover, the Charts tab page of the Faction Ranking dialog shows a graphical representation of the unit and site counts recorded by faction events.

5.3 Site Class

Each Site instance represents a single map site, i.e. a map location identified by a coordinate pair that may have an owner and that may contain entities – units, terrains, and effects – as described in "The Game Map" on page 13.

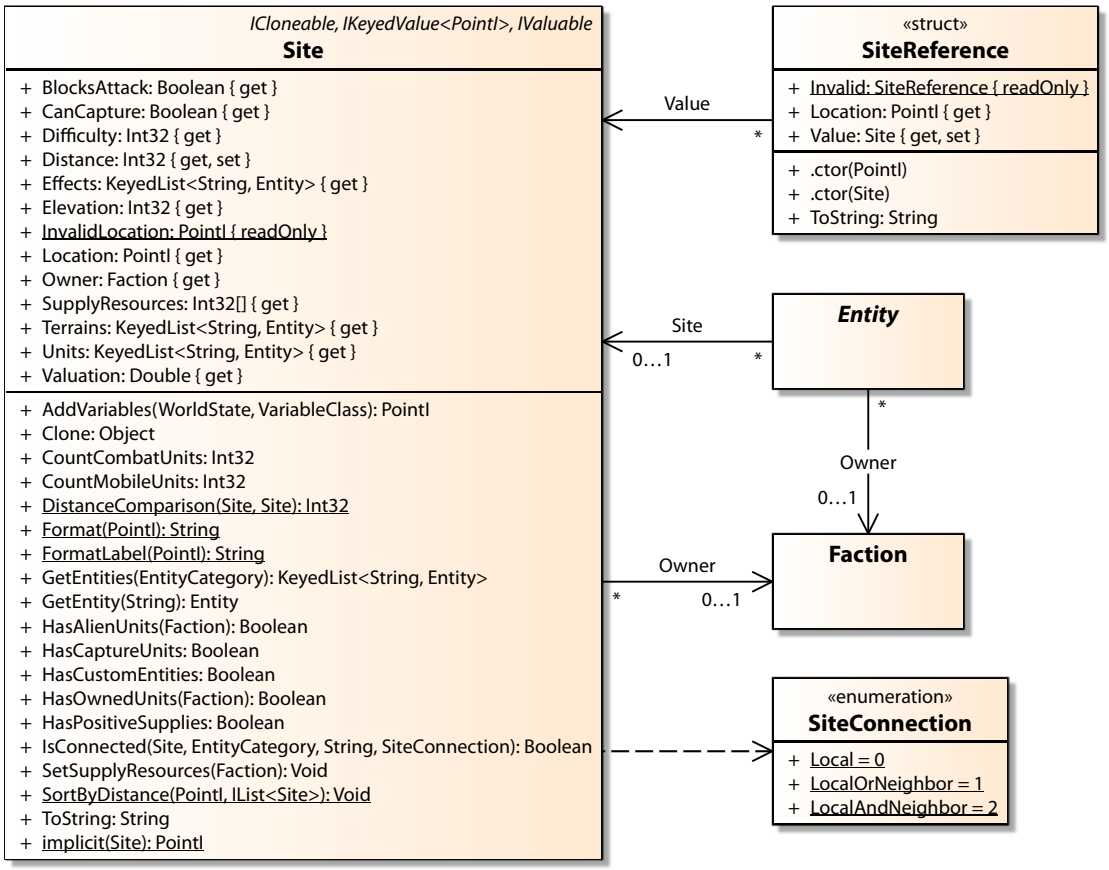


Figure 21: Site Class

The SiteReference structure is only used in arguments to game commands, so we defer its discussion to section "Command Arguments" on page 97. The SiteConnection enumeration defines the possible options for the IsConnected method, described below.

5.3.1 Site Members

Nearly all public members of the Site class are relevant to gameplay code. We start with the basics and group the rest by functionality.

Effects, Terrains, Units — The three entity stacks of the site. There is always at least one background terrain. The other two stacks may be empty.

Entity stacks are never manipulated directly. Set an entity's Site property to place the entity on a specific site, or to remove it from the map.

Location — The site's map coordinates, which are the same as its index positions in the WorldState.Sites array. Naturally, this property never changes.

Owner — The Faction object that owns the site, or a null reference if the site is unowned. Local units may belong to a different owner. Changing this property also removes the site from the Sites list of the old owner and adds it to that of the new owner, if any.

BlocksAttack — Indicates whether any local entity has the BlocksAttack flag, i.e. obstructs the line of sight for ranged attacks. See "Lines of Sight" on page 138.

Difficulty and Elevation — The difficulty of moving across the local terrains, and their elevation relative to some point of reference. See "Standard Variables" on page 119.

IsConnected — Indicates whether there is a visual connection between the instances of a given entity class in the current site and a specified adjacent site.

IsConnected tests the frame connections associated with the images of the two matching entities, as described in "Frames and Animation" on page 42. Call this method to determine if two sites are connected by road or river, for example.

SupplyResources and SetSupplyResources — The total amount of unit supplies provided by all local entities. See "Calculating Supplies" on page 149.

Valuation — The sum of the context-free valuations of all local terrains. See "Context-Free Valuation" on page 140.

GetEntity — Finds the Entity with a given identifier, in any entity stack.

GetEntities — Gets the entity stack of a given category.

5.3.2 Stack Utility Methods

The Site class provides a number of methods to extract information about local entity stacks.

When dealing with unit stacks, remember the stack invariant: all units in the same site must belong to the same faction (though not necessarily the one that owns the site).

AddVariables — Computes the sums of all basic and modifier values for the specified variable that are defined by, or may affect, any local entity. Used by the Show Variable dialog.

CanCapture — Determines whether any local terrains with the Capture ability, which means that the site itself can be captured by units with the equivalent ability.

CountCombatUnits and CountMobileUnits — Counts all local units whose IsCombat or IsMobile flag is set, respectively. Used by computer player algorithms to prioritize targets.

HasCaptureUnits — Determines whether there are any local units with the Capture ability.

HasCustomEntities — Determines whether the site contains any entities other than the default terrain stack. This method is used by Hexkit Editor for map editing.

HasOwnedUnits and HasAlienUnits — Determines whether any local units belong to a given faction or to another faction, respectively.

5.3.3 Sorting by Distance

It is often useful to sort a collection of map sites by their distance from another given location. The Site class provides several members to simplify this task.

Distance — The site's distance from an unspecified location. This is a simple variable which must set for all sites that are supplied to a DistanceComparer.

DistanceComparison — A predefined Comparison method that compares two Site objects by their Distance values. You can supply this method to a collection's Sort method.

SortByDistance — Sorts a site collection by each site's distance from a specified map location. This is the most convenient way to sort sites by distance.

The method sets the Distance properties of all sites and sorts the collection using the DistanceComparison method. The Distance values are retained after sorting is complete, and may be further used by client code as desired.

5.3.4 Terrain Value Caching

Of all the entity stacks in a given site, the terrain stack is the least likely to change. More precisely, the default rules *never* change the composition of a terrain stack, and individual terrains only by automatic resource updates (see "Attributes and Resources" on page 116).

To exploit this fact, Hexkit caches terrain-dependent values and only recalculates them when the corresponding terrain stack has changed. These cached values currently include the standard attributes Difficulty and Elevation, described in "Standard Variables" on page 119, and map site valuations, described in "Context-Free Valuation" on page 140.

The following properties govern the recalculation of cached values:

Site.TerrainChanged — Indicates whether the terrain stack of the current site was changed since its local terrain-dependent values were last recalculated.

If this flag is set when such a value is accessed, all local terrain-dependent values are updated and the flag is cleared.

Site.SetTerrainChanged — Sets the local TerrainChanged flag of the map site itself, as well as the global TerrainChanged flag of the current world state.

WorldState.TerrainChanged — Indicates whether the terrain stack of *any* site was changed since the global terrain-dependent values that are stored in the WorldState instance were last recalculated.

If this flag is set when such a value is accessed, all global terrain-dependent values are updated and the flag is cleared. Since global values are based on local values, the latter are also updated where necessary, and the corresponding local flags are cleared.

This description is merely informational since Hexkit monitors any changes to terrain stacks or their contents, and automatically sets the appropriate TerrainChanged flags. Accordingly, these properties all have internal visibility and do not appear in this chapter's UML diagrams.

5.4 Faction Class

An instance of the Faction class represents one player faction, as described in "Factions" on page 14. All factions in a given world state are stored in the WorldState.Factions collection.

The FactionReference structure is only used in arguments to game commands, so we defer its discussion to section "Command Arguments" on page 97.

The PlayerSettings structure communicates a few settings of the faction's controlling player to interested rule script code, as described in "Conditional Scripting" on page 139.

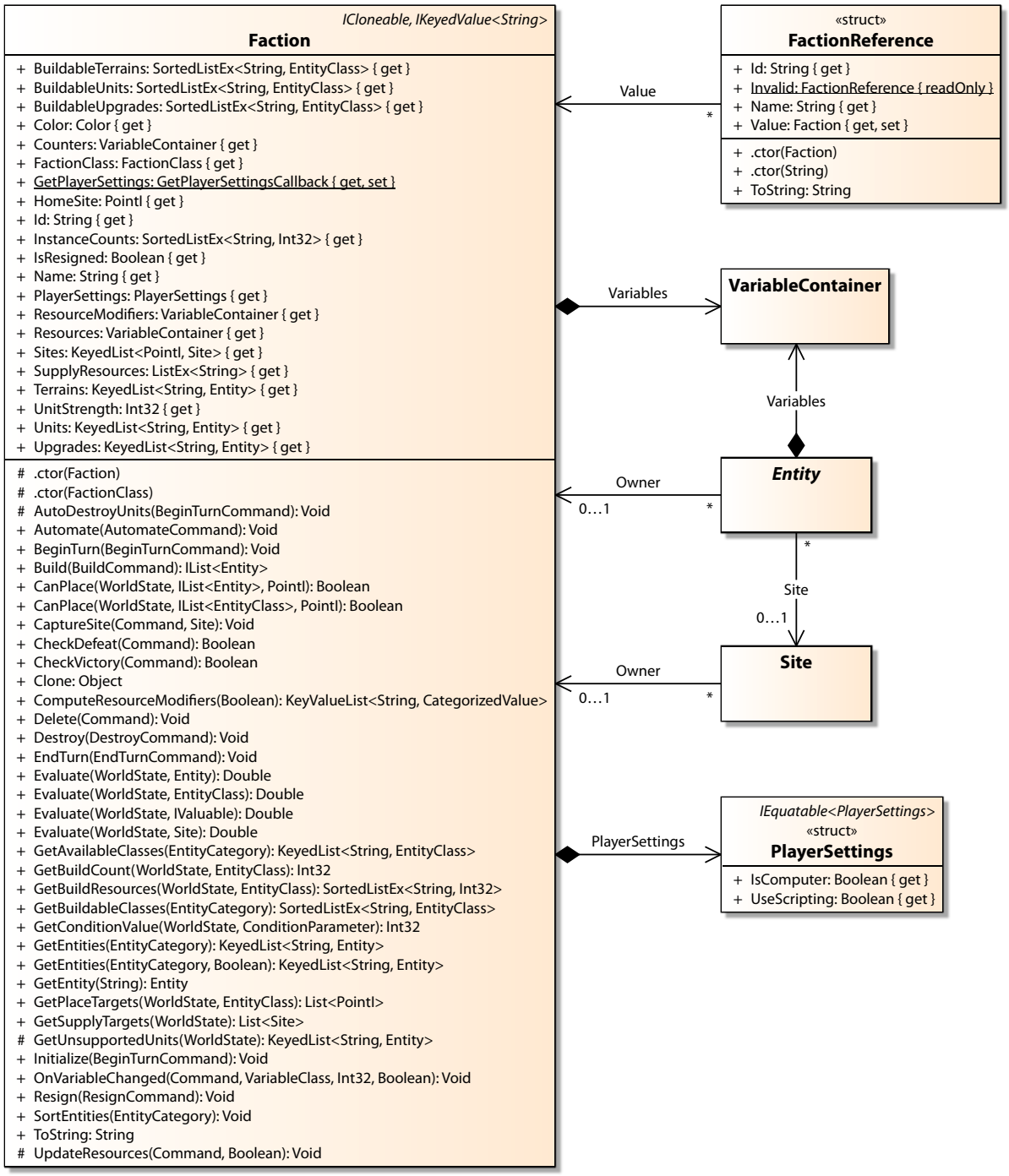


Figure 22: Faction Class

5.4.1 Faction Members

Nearly all public and protected members of the Faction class are relevant to gameplay code, but most are related to command execution or computer player algorithms, and thus described in chapter "Custom Rules" on page 108 or in chapter "Computer Players" on page 139.

Terrains, *Units*, *Upgrades* — All entities that belong to the faction. Units and terrains may be placed or unplaced whereas upgrades are by definition unplaced.

Entity collections are never manipulated directly. Set an entity's Owner property to assign the entity to a different faction, or to no faction.

InstanceCount — The number of Entity instances based on each EntityClass that belong to the faction. This dictionary is used to create unique display names for units.

UnitStrength — Computes the sum of the current Unit.Strength values of all Units.

Sites — All map sites that belong to the faction, representing its entire territory.
The site collection is never manipulated directly. Set a map site's Owner property to assign the site to a different faction, or to no faction.

Counters — The faction's internal counters. See "Data Persistence" on page 111.

Resources and ResourceModifiers — The faction's resource stockpile and resource modifiers.

GetConditionValue — Gets the current value for the specified ConditionParameter.

GetEntity — Finds the owned Entity with a given identifier, in any entity collection.

GetEntities — Gets the owned entity collection of a given category, optionally filtering the list to contain only placed or unplaced entities.

5.4.2 Proxies for Scenario Data

Every Faction instance is based on a corresponding Scenario.FactionClass object. Several Faction properties are proxies for the eponymous FactionClass properties whose values were set by the scenario designer. These values never change during a game.

FactionClass — The underlying faction class that is instantiated by the faction.

Color — The color that represents the faction on color-coded displays.

HomeSite — The map coordinates of the faction's home site, if any.

Id and Name — The unique internal identifier and display name of the faction.

BuildableTerrains/Units/Upgrades — The entity classes that a faction may build, assuming it has the required resources. See "Build Command" on page 125.

SupplyResources — Those resources used by any of the faction's units that can be resupplied on the map. See "Calculating Supplies" on page 149.

5.5 Entity Class

An instance of any class derived from Entity represents one game entity, as described in "Entities" on page 17. Each of the four classes derived from Entity corresponds to the four entity categories: Unit, Terrain, Effect, and Upgrade. The first is by far the most complex of these classes, hence we'll discuss them all in "Unit Class and Siblings" on page 81.

The EntityReference structure is only used in arguments to game commands, so we defer its discussion to section "Command Arguments" on page 97.

5.5.1 Entity Collections

All Entity objects stored in the WorldState.Entities hashtable can also be found in another entity collection which is the corresponding entity stack if they are placed on a map site, and/or the corresponding entity collection if they belong to a player faction.

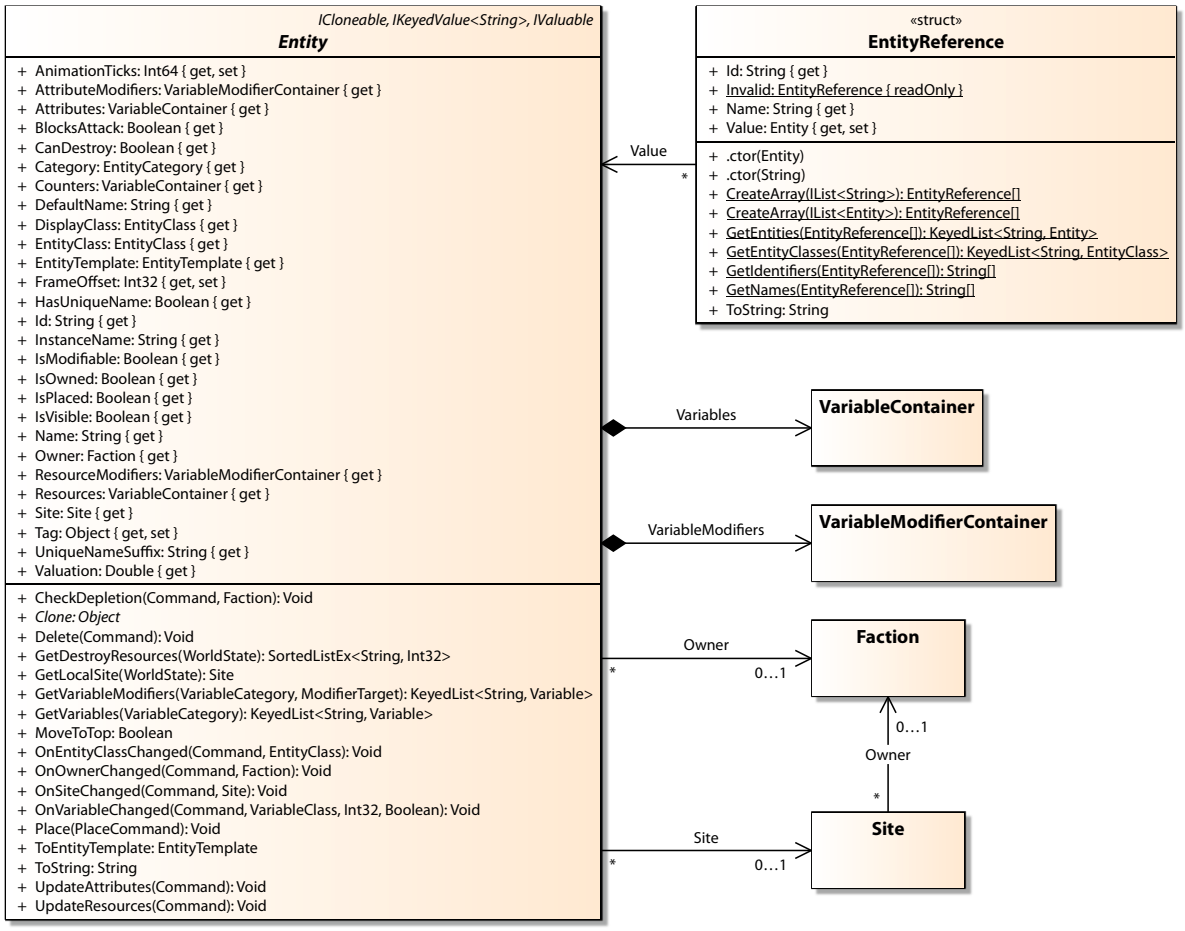


Figure 23: Entity Class

Units — Always present in a faction's Units collection, and in a site's Units stack as well when placed on the map. The unit owner may be different from the site owner.

Terrains — Present in a site's Terrains stack, in a faction's Terrains collection, or in both places at once when placed on an owned site.

Effects — Only ever present in a site's Effects stack. The effect owner does change to reflect the owner of its site, but there is no corresponding faction collection.

Upgrades — Only ever present in a faction's Upgrades collection. Upgrades are never placed on the map, hence there is no corresponding site stack.

5.5.2 Entity Members

Entity and derived classes define a large number of properties and methods are relevant to gameplay code but once again, we only cover the basics here. The remaining members are described in later chapters, within the relevant context of command planning and execution.

Id — The unique internal identifier of the entity. This property is set automatically when an entity is created by the HCL instruction CreateEntity.

Name — The display name of the entity. The default is DefaultName but units automatically append an instance count so that the full name is unique to their owner.

When set to a valid string that differs from DefaultName, the new value is also stored in the property InstanceName, which is otherwise a null reference.

HasUniqueName — Indicates whether the entity has a "unique" name which consists of the DefaultName with an appended instance count.

UniqueNameSuffix — Extracts the instance count suffix that follows the DefaultName part of the entity's "unique" name, if any.

Owner — The faction to which the entity belongs, or a null reference if unowned.
Changing this property also removes the entity from the corresponding list of the old owner and adds it to that of the new owner, if any.

Site — The map site on which the entity is placed, or a null reference if unplaced.
Changing this property also removes the entity from the corresponding stack of the old site and adds it to that of the new site, if any.

IsOwned and IsPlaced — Indicates whether the entity has a valid Owner or Site, respectively.

Tag — An arbitrary object that computer player algorithms can use as temporary storage.

Attributes and AttributeModifiers — The entity's basic attribute values and attribute modifiers.

Counters — The entity's internal counters. See "Data Persistence" on page 111.

Resources and ResourceModifiers — The entity's basic resource values and resource modifiers.

IsModifiable — Indicates whether the entity's attributes and resources are subject to automatic modification. See "Attributes and Resources" on page 116.

GetLocalSite — Gets the map site on which the entity's unit variable modifiers are centered.

GetVariables and GetVariableModifiers — Gets the entity's VariableContainer or VariableModifierContainer of a given category.

Entity variables are retrieved from a shared EntityClassCache if they are unchanged from their initial scenario values. See "Variable Class" on page 84 for details on this mechanism.

5.5.3 Proxies for Scenario Data

Every Entity object is an "instance" of a Scenario.EntityClass object (in a logical sense, not in the technical sense that applies to C# and .NET classes).

Several Entity properties are proxies for equivalent EntityClass properties whose values were set by the scenario designer. These values never change during a game. Moreover, an entity may correspond to an EntityTemplate object defined by the scenario.

EntityClass — The underlying entity class that is instantiated by the entity.

EntityTemplate — The original scenario template from which the entity was created, if any.
This property is a null reference for entities that are not based on a template.

BlocksAttack — Indicates whether the entity obstructs the line of sight for ranged attacks when placed on the map. See "Lines of Sight" on page 138.

CanDestroy — Indicates whether the entity has the Destroy ability, enabling its owner to destroy the entity at will. See "Destroy Command" on page 126.

Category — The category of the entity.

DefaultName — The default display name of the entity, which is the display name of the underlying entity class. This is also the prefix for "unique" names.

ToEntityTemplate — Returns the original EntityTemplate, if any; otherwise, creates a new EntityTemplate object that contains all representable entity data. Used to create Area objects from WorldState.Sites elements.

5.5.4 Map View Display

Lastly, the following properties and methods control the entity's appearance on a map view.

DisplayClass — The entity class that provides the image frames to be shown for the entity. This is usually the same as the EntityClass property but may be set to a different value for certain effects, such as units being transported on a ship.

Changing this property automatically resets FrameOffset to zero. Rule script code must explicitly select a different image frame if desired.

FrameOffset — The relative index of the image frame that is shown for the entity, as described in "Frames and Animation" on page 42. This is an offset from the first image frame of the entity class which itself always has a non-zero index in the bitmap tile catalog.

IsVisible — Indicates whether the entity is visible on a map view. Currently unused. Future Hexkit versions might use this property to create a "fog of war" effect.

MoveToTop — Moves the entity to the top of its local site stack. This has no gameplay effects but makes the entity's image fully visible, unless it is obscured by the topmost image of another stack with a higher display priority.

5.6 Unit Class and Siblings

An instance of one of the four classes derived from the Entity class represents one unit, terrain, effect, or upgrade, respectively. Each class may be further specialized in a rule script. Figure 24 shows an overview of the inheritance tree, and Figure 25 on page 82 shows details on the complex Unit class and several related classes.

All of the four derived classes override some of the methods defined by the common base class Entity. These methods and their specializations are described in "Entity Class" on page 78. Here we'll concentrate on the members that are new to the derived classes.

5.6.1 Proxies for Scenario Data

The Terrain and Unit classes define additional proxies for properties of the underlying EntityClass object. Most of these properties are used by various game commands – see "Command Algorithms" on page 120 for details.

Terrain.CanCapture — Indicates whether the terrain has the Capture ability. This allows units with the equivalent ability to capture map sites containing the terrain.

Unit.CanCapture — Indicates whether the unit has the Capture ability (see above).

Unit.CanDefendOnly — Indicates whether the unit has the Can Defend Only restriction, or "negative ability". This disables the unit's Attack ability during its owner's turn.

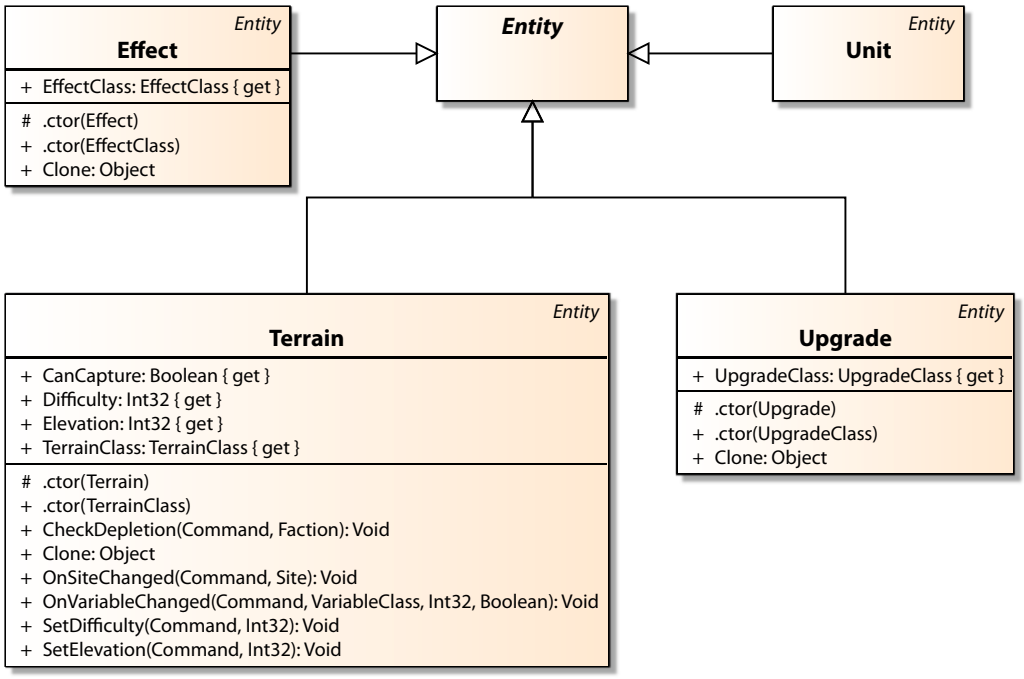


Figure 24: Entity Hierarchy

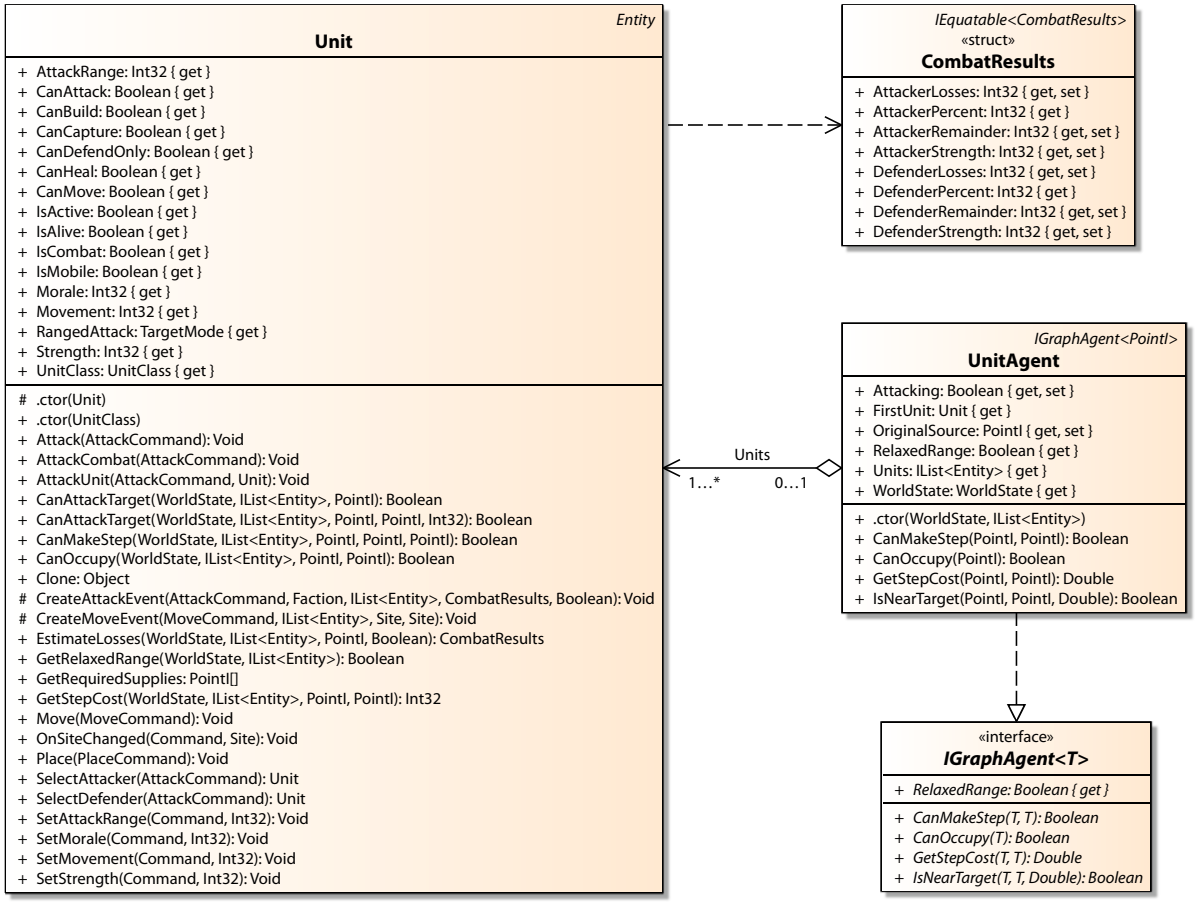


Figure 25: Unit Class

Unit.CanHeal — Indicates whether the unit has the Healing ability. This informs computer player algorithms that the unit can recover lost strength points during the game.

Unit.RangedAttack — Indicates the level of visibility required for the unit's ranged attacks.

Additionally, every derived class provides its own version of the Entity.EntityClass property with a name and type that reflects the actual category of the underlying EntityClass.

5.6.2 Standard Variables

Several properties of the Terrain and Unit classes represent "standard variables", i.e. numerical statistics with predefined semantics that can be mapped to actual variables in Hexkit Editor. See "Standard Variables" on page 119 for more information on their use.

Terrain.Difficulty — The difficulty of moving across the terrain, in terms of movement cost.

Terrain.Elevation — The terrain's elevation relative to some arbitrary point of reference.

Unit.AttackRange — The unit's maximum attack range, in map site steps. One indicates mêlée units, and zero indicates civilian units that cannot fight at all.

Unit.Movement — The unit's maximum movement range, in terms of movement cost. Zero indicates stationary units that cannot move at all.

Unit.Morale — The unit's combat morale.

Unit.Strength — The unit's combat strength, i.e. manpower or health.

All standard variable properties have corresponding setter methods prefixed with Set… that can be used to change the value of any standard variable backed by a concrete variable.

5.6.3 Other Unit Members

The Unit class provides several more abilities and helper properties in addition to the scenario proxies listed above.

CanAttack — Indicates whether the unit currently has the Attack ability. Always returns false unless the IsCombat flag is set and the unit's Morale is positive.

CanBuild — Indicates whether the unit's owner can build more units of its class, i.e. whether the owner's BuildableUnits collection contains the unit's class.

CanMove — Indicates whether the unit currently has the Move ability. Always returns false unless the IsMobile flag is set.

IsActive — Indicates whether these conditions are met: the unit is alive (IsAlive), placed on the map (IsPlaced), and can attack, move, or both (CanAttack or CanMove).

IsAlive — Indicates whether the unit is still alive, i.e. its Owner is not a null reference.

During combat, the participating units remain in the unit list parameters of any involved methods until combat is finished, even if the units were already killed. Such methods should check the IsAlive flags of all supplied units.

IsCombat — Indicates whether the unit can fight at all, i.e. its AttackRange is positive.

IsMobile — Indicates whether the unit can move at all, i.e. its Movement is positive.

5.6.4 The Pathfinding Agent

UnitAgent is an adapter class that allows complete or partial unit stacks, not just single units, to act as moving agents in the pathfinding algorithms provided by the Tektosyne library. The whole mechanism is described in chapter "Pathfinding" on page 132.

UnitAgent implements the Tektosyne interface IGraphAgent<T> and typically forwards its members to virtual Unit methods of the same name, as shown in "Customizing Pathfinding" on page 136. Here is a brief overview of the UnitAgent members not described there:

WorldState — The world state in which pathfinding takes place.

Units — The Unit objects that participate in the movement.

FirstUnit — The first element in the Units collection. All required Unit members are invoked on the FirstUnit object.[9]

Attacking — Indicates whether the Units are attacking or moving. Attacking units must get into attack range of their target site; moving units must occupy it directly.

OriginalSource — Indicates the map location where pathfinding starts. This value defaults to the site of the FirstUnit but can be set explicitly for hypothetical movements.

5.7 Variable Class

An instance of the Variable class represents one of the integer values that constitute a faction's or entity's "statistics". This includes visible attributes and resources (see "Variables" on page 19) but also hidden internal counters (see "Data Persistence" on page 111).

Figure 26 shows the Variable class and some related types which are described below. Although a variable ultimately corresponds to one integer value, Hexkit provides a lot of additional functionality for ease of use, validation, and storage efficiency.

5.7.1 VariablePurpose Enumeration

The VariablePurpose enumeration defines the purpose of a given set of variables that are stored in the same collection (i.e. VariableContainer) which is also the purpose of any individual variable within that collection.

Entity versus Faction — Indicates whether the variable is used by an entity or by a faction. Exactly one of these flags must be present.

Basic versus Modifier — Indicates whether the variable is a basic value or a modifier value. Exactly one of these flags must be present.

Scenario — Indicates whether the variable value was defined by the scenario. When this flag is absent, the variable was added to its collection by the rule script.

Some Variable methods behave differently depending on the variable's purpose. Any variable and variable collection is also associated with a VariableCategory which is stored separately.

9. This workaround is used quite often in Hexkit. As in most object-oriented languages, class (static) methods in C# are not polymorphic. Any method that a rule script might want to override must be defined as a virtual *instance* method, and so must be invoked on some instance of the class, even if it only operates on explicit parameters and completely ignores the implied this parameter.

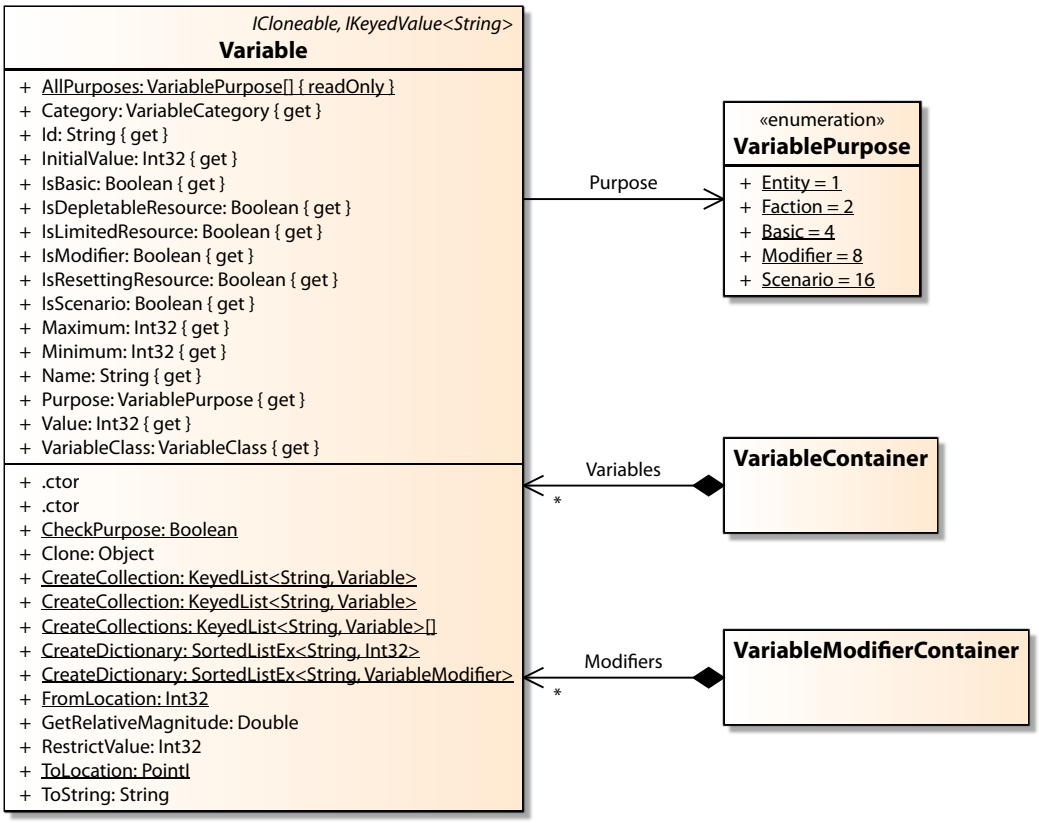


Figure 26: Variable Class

The utility method Variable.CheckPurpose checks that exactly one value of each of the flag pairs Entity/Faction and Basic/Modifier is present in a given VariablePurpose value.

5.7.2 Proxies for Scenario Data

As with factions and entities, every Variable object is an "instance" of a Scenario.VariableClass object. Several Variable properties are proxies for equivalent VariableClass properties whose values were set by the scenario designer. These values never change during a game.

VariableClass — The underlying variable class that is instantiated by the variable.

Category — The category of the variable.

Id and Name — The internal identifier and display name of the variable. Unlike factions and entities, variables do *not* possess unique identifiers; all variables based on the same variable class share the same identifier.

5.7.3 Variable Members

The remaining Variable properties vary between individual variables and collections.

InitialValue — The initial value of the variable. The definition depends on IsScenario.

Purpose — The purpose of the variable, and of all other variables in the same collection.

IsBasic — Indicates whether Purpose contains the Basic flag.

IsModifier — Indicates whether Purpose contains the Modifier flag.

IsScenario — Indicates whether Purpose contains the Scenario flag.

If so, InitialValue is the value that the current scenario defined for the variable; otherwise, it is the first value that the rule script defined for the variable.

IsDepletableResource — Indicates whether the variable represents an entity resource that is considered depletable, so that Hexkit will visualize its current depletion level.

That is, this property is true exactly if Purpose contains the Entity and Basic flags, the variable category is Resource, and IsDepletable is true for the ResourceClass.

IsLimitedResource — Indicates whether the variable represents an entity resource that is limited by its InitialValue, as reflected by the variable's Maximum value.

That is, this property is true exactly if Purpose contains the Entity and Basic flags, the variable category is Resource, and IsLimited is true for the ResourceClass.

IsResettingResource — Indicates whether the variable represents a faction resource that should be reset to its InitialValue at the start of each turn.

That is, this property is true exactly if Purpose contains the Faction and Basic flags, the variable category is Resource, and IsResetting is true for the ResourceClass.

Minimum — The minimum value for the variable.

This is the constant value VariableClass.AbsoluteMinimum if IsModifier is true, and a proxy for the Minimum value of the underlying VariableClass otherwise.

Maximum — The maximum value for the variable.

This is the constant value VariableClass.AbsoluteMaximum if IsModifier is true, the variable's InitialValue if IsLimitedResource is true, and a proxy for the Maximum value of the underlying VariableClass otherwise.

Value — The current value of the variable which is always between Minimum and Maximum.

From/ToLocation — Encodes and decodes map coordinates that are stored in a single value ranging from VariableClass.AbsoluteMinimum to VariableClass.AbsoluteMaximum.

GetRelativeMagnitude — Computes the relative magnitude of the current Value compared to a specified "normal" value which must be between Minimum and Maximum.

Ignoring special cases, the result is a floating-point quotient between the current Value minus Minimum, divided by the specified "normal" value minus Minimum. This method is helpful for comparing variables whose Minimum value is not zero.

RestrictValue — Restricts a specified value to the range from Minimum to Maximum.

This method is called automatically whenever the Value property of a Variable object is set, but rule script authors might want to explicitly restrict temporary results when performing complex variable calculations.

The three static Create… methods are used internally to convert collections of variable values between different storage formats. Gameplay code shouldn't have reason to use them.

5.7.4 Memory Efficiency

Faction and entity variables are stored in instances of the VariableContainer class which wraps a collection of Variable instances. Modifiers values similarly reside in a VariableModifierContainer that wraps a set of Variable collections, one for each possible ModifierTarget.

There are two reasons for these additional container classes:

— The container object independently stores the category and purpose of all contained Variable instances. This is useful for consistency checking, especially when a newly created container is initially empty, since category and purpose cannot be inferred from a present Variable instance in this case.

— A container's Variable collection can initially point to a pre-allocated collection that is shared among multiple container instances. A private backer collection is allocated only when a variable is changed compared to its shared value.

The second point is the more important one. Every single Entity instance holds three different variable collections for basic values, plus twelve variable collections for modifier values; and a large map might contain more than ten thousand entities.

However, many collections will always remain empty (typically non-unit attributes and most counters), and many other collections will always retain their initial scenario values as the game proceeds (typically modifier collections).

Sharing variable collections among Entity instances therefore greatly reduces the memory consumption of a given world state. The memory savings are even greater when world states are cloned for computer player calculations or interactive replays since different entities in multiple world states will reference the same backing collection.

The static class EntityClassCache, shown in Figure 27, provides such shared variable collections for all EntityClass objects in the current scenario. The built-in Entity constructors initialize their variable collections from this cache. This mechanism requires no explicit support from gameplay code.

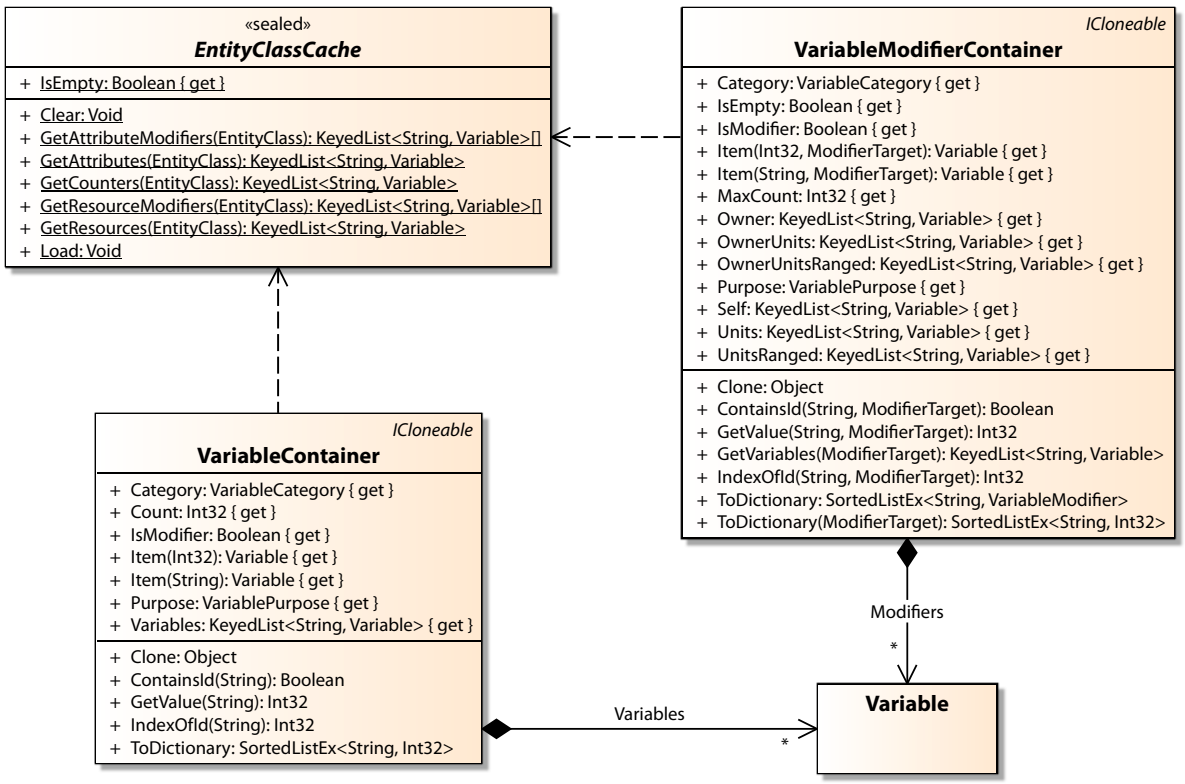


Figure 27: Variable Caching

5.7.5 VariableContainer Members

The VariableContainer class is the outermost public interface to any variable value collection stored with Faction objects, and any collection of basic values stored with Entity objects. For convenience, the class also provides proxies to members of the wrapped Variables collection.

Variables — The wrapped Variable collection. As outlined above, this property returns either a shared collection (only for entity variables) or a private backing store.

Category — The Category of all items in the Variable collection.

Count — The number of items in the Variable collection.

IsModifier — The IsModifier flag of all items in the Variable collection.

Purpose — The Purpose of all items in the Variable collection.

this[index] — Gets the item with a given index in the Variable collection.
Note that the collection is unsorted. Indices are stable since variable values are never removed from a collection once they have been added.

this[id] — Gets the item with a given identifier in the Variable collection, or a null reference if the identifier was not found. Note that the corresponding indexer of the Variable collection throws an exception for missing identifiers.

ContainsId — Determines whether the Variable collection contains an item with the specified identifier.

GetValue — Gets the Value of the item in the Variable collection that has the specified identifier, or zero if the identifier was not found. Useful for adding and subtracting.

IndexOfId — Gets the index of the item in the Variable collection that has the specified identifier, or –1 if the identifier was not found.

ToDictionary — Copies all items in the Variable collection to an independent dictionary that maps identifiers to variable values. Useful for extensive calculations that are performed on multiple values in the collection, such as resource updates.

5.7.6 VariableModifierContainer Members

The VariableModifierContainer class is the outermost public interface to any collection of variable modifier values stored with Entity objects. This class is largely identical with VariableModifier but wraps one collection per ModifierTarget rather than just a single collection.

Owner, Self, Units, etc. — The wrapped Variable collection for the corresponding target. As before, this is either a shared collection or a private backing store.

Category — The Category of all items in all Variable collections.

IsEmpty — Indicates whether all Variable collections are empty.

IsModifier — Always true since all Variable collections hold modifier values.

MaxCount — The maximum number of items in any Variable collection.

Purpose — The Purpose of all items in all Variable collections.

GetVariables — Gets the wrapped Variable collection for a given ModifierTarget.

The remaining indexers and methods work exactly as with the VariableModifier class, except that they take an additional ModifierTarget parameter to select a specific Variable collection.

Chapter 6: Game Commands

This chapter describes the internal representation of the various commands whose sequence constitutes a Hexkit game. All such data is contained in the Hexkit.World assembly.

We saw in chapter "World State" on page 69 that the Faction, Entity, and Unit classes provide many customizable properties and methods which implement the gameplay effects of most commands. Now let's take a look at how Hexkit manages those commands.

In this chapter, we'll first examine the Hexkit.World.Commands namespace that contains the internal representation of all game commands. The actual effects of any given command are encoded as a sequence of Hexkit Command Language (HCL) instructions, so we'll also look at the Hexkit.World.Instructions namespace which defines these instructions.

But before diving into implementation details, we'll lay out the fundamentals of command execution and the rationale behind our rather complex two-level mechanism.

6.1 Command Execution

Like any strategy game, Hexkit offers a small set of game commands that operate on the current world state, as described in "Commands" on page 23. A game proceeds as commands are issued by each faction. Some commands may have somewhat random effects; for instance, attacks don't always hit or may cause varying amounts of damage.

Unlike most strategy games, Hexkit provides two features that prevent us from simply "hard-wiring" game commands to world state changes:

— The precise effect of a command may be changed by the rule script associated with the current scenario (see "Custom Rules" on page 108).

— Hexkit games are stored as a history of commands, not as a snapshot of the current world state (see "The Command History" on page 11).

Both features are cornerstones of Hexkit's design. The rule script mechanism allows unparalleled flexibility in the design of game rules. The history mechanism greatly simplifies debugging and allows "post mortem" game analysis.

Customizable command effects were first implemented by associating each command with one or more virtual Faction, Entity, or Unit methods that can be overridden in a rule script. This implementation works well enough by itself, but turned out to cause complications when combined with the history mechanism, which also presents some difficulties of its own.

6.1.1 Two Levels of Execution

In the following discussion, we'll use the term *virtual command methods* to denote the virtual methods of the Faction, Entity, and Unit classes that implement the effects of a game command. The question is now, *how* are these gameplay effects realized?

Older versions of Hexkit used a *single-level execution scheme*. All relevant properties of World classes allowed write access, and the virtual command methods directly manipulated the current world state to achieve the desired gameplay effects. When a game was replayed (interactively or when loading a saved game), the virtual command methods were re-executed, in the same sequence and with the same parameters, to recreate the original world state.

This naïve implementation caused numerous problems, as we'll see below. Starting with Hexkit 3.2, a better solution was found in the present *two-level execution scheme*.

Now, all relevant World properties are read-only. The only way to manipulate a world state is through the use of certain *instructions* which are similar to game commands but operate on a lower level. The virtual command methods arrange these instructions into a mini-program that encodes the desired world state changes.

Replaying games now only requires the re-execution of these mini-programs, which are also stored with saved games. Therefore, the virtual command methods associated with a command are never again executed once the command's program has been created.

Hexkit Command Language

The totality of the new second-level instruction set is known as the *Hexkit Command Language* (HCL). Occasionally, I refer to HCL instructions as "microcode" because their relationship to game commands is similar to that of machine language to the microcode of CISC processors: one complex game command is executed as a sequence of primitive HCL instructions.

The effect of every HCL instruction is completely deterministic: executing the same instructions on identical world states produces identical results. This determinism allows Hexkit to replay a game by creating a new starting configuration based on the original scenario, and then executing all recorded instructions on that new configuration. The result is to exactly recreate the world state that has originally recorded the replayed instructions.

Customization of a game command now means changing the HCL instruction sequence that is created for the command. The instructions themselves cannot be customized; their meaning is fixed and usually correlates to a single internal property setter. Some instructions map to command events instead, and have no effect on the world state or during silent replays.

Example. Let's say the active faction issues a Build command to build a single unit. Under the default rules, the corresponding HCL program consists of the following instructions:

1. A CreateEntity instruction to create a new unit of the desired class.
2. A SetEntityOwner instruction to set the unit's owner to the active faction.
3. A SetEntityUniqueName instruction to assign the unit an individual name that is unique among the active faction's possessions.
4. Zero ore more SetFactionVariable instructions that decrease the active faction's resources to reflect the expenses of building the new unit.

6.1.2 Optimizing the Program

One notable drawback of the two-level execution scheme is the vast increase of data associated with each game command. Where the single-level scheme would merely store a command and its parameters, the two-level scheme adds at least a few instructions to each command – each with its own list of parameters.

These instructions not only take up space in memory and (especially) in saved games, they may also slow down replays due to the work required to decode instruction parameters. Several optimizations try to counter these effects.

— Two dedicated instructions were devised to enable or disable *all* of a faction's units.
This greatly reduces the space and time needed for the End Turn command which would otherwise create huge sequences of instructions for individual units.

— A newly created HCL program is optimized by removing all commands that did not actually change any world state data (*ineffectual* instructions).

Naturally, event instructions are excluded from this optimization.

— Older Hexkit versions attempted to further optimize a newly created HCL program by removing all commands whose data change was reversed by a subsequent instruction (*redundant* instructions).

However, this optimization added much extra work and consistency problems for very little gain, so it was dropped in version 3.3.

Moreover, saved games are compressed using the GZip algorithm. Unsurprisingly, the repetitive XML data of a Hexkit command history compresses extremely well – a factor of ten is quickly reached and exceeded as the raw XML file grows in size.

6.1.3 Problems Solved

You might wonder why this elaborate mechanism was necessary. This subsection describes the problems caused by the single-level execution scheme, and solved by the two-level scheme.

The Problem with Message Events

In the single-level execution scheme, the text of any generated message events is unavailable once command execution is finished. As this would greatly reduce the usefulness of the command history, older Hexkit versions maintained a separate event queue that allowed post-execution retrieval of event data, in particular message text.

This particular implementation was rather primitive and had its own issues. All events were displayed *after* the command had finished execution, which would often cause a mismatch between the visible world state and the display event shown at the same time.

While a better implementation of a dedicated event queue is conceivable, the integrated HCL program of the two-level execution scheme solves the event problem just as well.

The Problem with Random Numbers

In order to reliably recreate a world state from a command history, all game commands must have completely deterministic effects. This requirement causes two problems in the single-level execution scheme.

For one thing, some commands such as Attack may have partially random effects. Older Hexkit versions provided an auxiliary mechanism that recorded all random numbers generated during the execution of a command, and played them back as required when the command was executed again. This mechanism worked, but it was fairly complex and chained rule script authors to a particular random number generator.

The two-level execution scheme separates the virtual command methods from the instructions that actually change the world state. As a result, it is no longer required that rule script methods must be deterministic. They may use whatever random numbers they wish, and every generated instruction sequence is, by definition, fully deterministic.

The Problem with Rule Script Changes

Strict determinism at the level of virtual command methods also creates an uncomfortably close coupling between a command's predefined code for a given Hexkit version, its custom code for a given scenario version, and the saved games created for these Hexkit and scenario versions.

In the single-level execution scheme, the entire implementation code for all saved commands must be re-executed to load a saved game, and this re-execution must result in exactly the same world state as before. Any change in the implementation of any command will typically prevent Hexkit from loading a saved game.

This effectively means that the game developer or scenario author cannot simply save the game upon observing a bug or other issue, change some command methods, and reload the saved game to test the changed behavior. Instead, he must start a new game and try to manually recreate whatever situation triggered the original observation. This limitation severely damages one of the expected benefits of the history list, namely to aid development and debugging.

The two-level scheme eliminates the danger that a saved game is rendered invalid by changes to virtual command methods. All recorded commands are stored along with their HCL programs which are completely decoupled from the virtual command methods that generated them.

6.2 Commands Namespace

The Hexkit.World.Commands namespace resides in the Hexkit.World assembly and provides the internal representation of all Hexkit game commands.

Figure 28 shows the inheritance tree that starts with the Command class. The leaves of this tree implement specific commands while the trunk is formed by abstract base classes that handle command arguments and other common functionality. The concrete derived classes add specific validation code for the commands they represent, and dispatch to the appropriate Faction, Entity, or Unit methods during instruction generation.

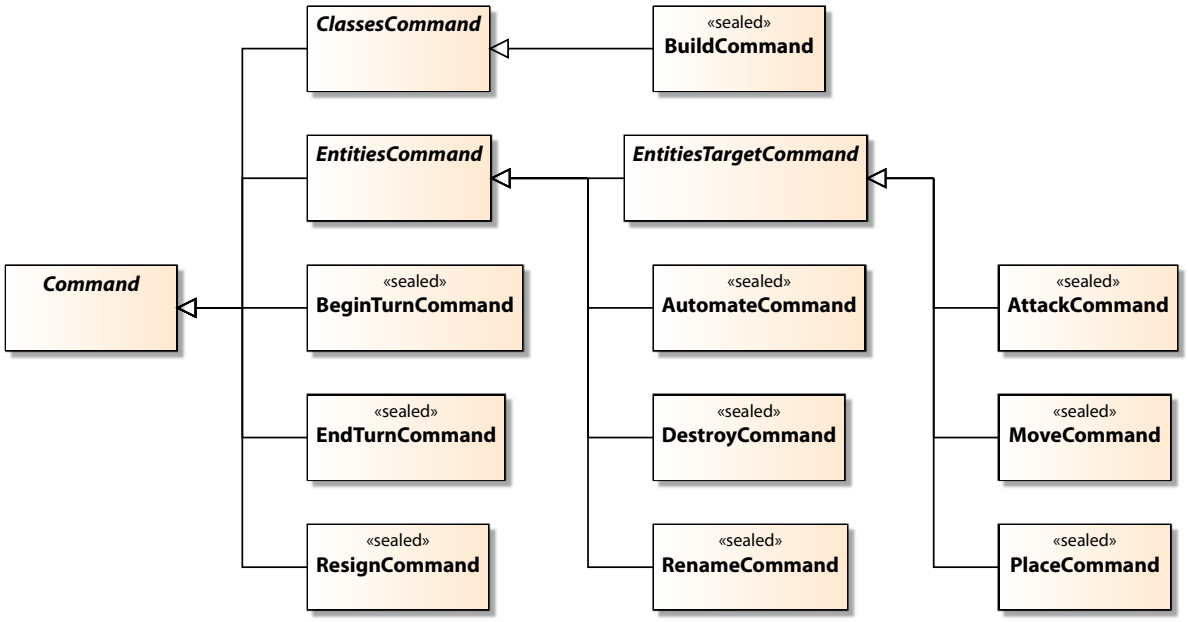


Figure 28: Command Hierarchy

We'll cover Command and some related auxiliary classes in section "Command Class" on page 94 before returning to the various derived classes in section "Derived Classes" on page 96.

6.2.1 About the Class Diagrams

The UML diagrams for the Hexkit.World.Commands and Instructions namespaces usually only show public class members. Protected and internal members may be added as needed.

All classes that represent specific commands or instructions support XML serialization. Their XML element names, defined in Hexkit.Session.xsd, equal the corresponding class names minus the Command or Instruction suffix, and are discovered at runtime via reflection rather than by a ConstXmlName field. For example, the class AttackCommand is represented by an XML element named Attack, and so on.

6.2.2 Command Class

The Command class provides the basic functionality required to manage game commands.

The Faction, Entity, and Unit methods that implement a specific game command always receive an instance of the corresponding class derived from Command. The members of this instance provide all required command arguments and access to all HCL instructions, as well as various auxiliary mechanisms.

Before we examine the Command class in detail, let's have a brief look at the other classes shown in Figure 29:

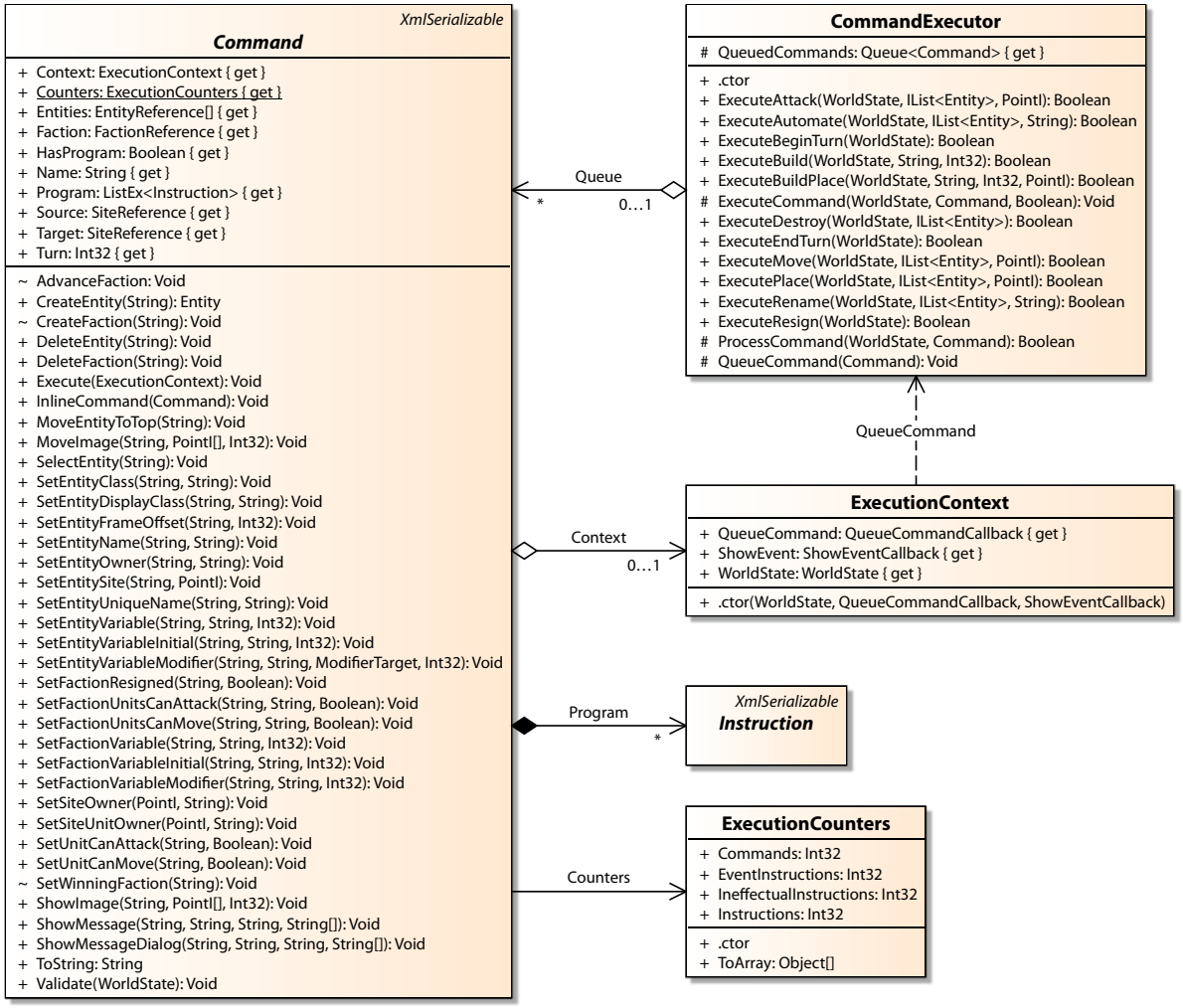


Figure 29: Command Class

CommandExecutor — Provides helper methods that simplify the use of game commands, and also manages command queuing (see "Command Queuing" on page 105). This class is only shown for completeness; it is never accessed by rule script code.

Computer player algorithms directly use the CommandExecutor class. Human player commands are handled by a derived class, Hexkit.Game.SessionExecutor, that displays status messages and error dialogs, and simulates interactive replay for queued commands.

The ExecuteBuildPlace method executes the two implied commands sequentially, as a convenience to the human player. All other Execute… methods execute the single command implied by their name.

ExecutionContext — Provides the context in which a command is executed.

This includes the current world state, a callback method for command queuing that dispatches to the current CommandExecutor (see "Command Queuing" on page 105), and another callback method used internally for event display that should be ignored by rule script code (see "Instruction Usage" on page 102).

ExecutionCounters — A simple data container with several fields that count the number of executed commands and instructions (divided into various subgroups). This data is intended for debugging and optimization only.

Instruction — The abstract base class that represents one of the instructions that constitute an HCL program. This class is described in "Instruction Class" on page 99.

Command Members

Most Command members can or must be used by gameplay code to implement the effects of a specific command, with exceptions as noted.

All public or internal Command methods – with the obvious exceptions of Execute, InlineCommand, ToString, and Validate – are *instruction methods* that generate one eponymous HCL instruction each. Please refer to "Instructions Namespace" on page 99 for a description of the instruction mechanism and of individual instructions.

Context — The command's current execution context. Command objects that are passed to Faction, Entity, or Unit methods always have a valid Context with valid WorldState and QueueCommand properties. ShowEvent may be valid or a null reference.

Counters — A shared instance of the ExecutionCounters class that accumulates data while Hexkit Game is running. Use the menu item Debug → Show Command Counters to show all current Counters values, as described in the online help.

Entities — A list of all entities affected by the command, if any.

The Command implementation always returns a null reference. Derived classes that require entity arguments must override this property. When present, all entities affected by a single command must be of the same category.[10]

Faction — The player faction that has issued the command.

This is usually the active faction in the world state on which the command is executed, but may be different if the rule script queued a command for another faction.

Name — The command's display name, as shown in the event view and in the history list.

10. Otherwise, the trick of invoking virtual command methods on the first Entity object wouldn't work since overrides, e.g. in the Unit class, could not assume that the remaining entities are also units!

This is usually the same as the class name, except for commands whose name contains a space character (Begin Turn and End Turn).

Program — A list of HCL instructions that constitute the "microcode" program for the command. HasProgram indicates whether a program has already been generated.
Please refer to "Instructions Namespace" on page 99 for further details.

Source/Target — The command's source and target locations on the map, if any.
The Command implementation always returns invalid values. Derived classes that require source or target arguments must override these properties.

Turn — The index of the game turn when the command was issued. This is always the same as the current turn in the world state on which the command is executed.

Execute — Executes the HCL program for the command. If HasProgram is false, Execute dispatches to GenerateInstructions in order to generate a new program.
You should never call this method. Use InlineCommand or Context.QueueCommand instead if you want the active faction to execute additional commands.

InlineCommand — Generates and executes the HCL program for the specified command, and then appends the resulting instructions to the program for the *current* command.
Use this method if you want to merge the effects of two commands into one.

Validate — Validates all command arguments and sets all object references.
You should never call this method. Any command that is passed to a gameplay method will always have fully validated arguments.

The Entities, Faction, Source, Target, and Turn properties represent command arguments which are validated by the Validate method. Please refer to "Command Arguments" on page 97 for details on this procedure.

Inferred Source. The Source property is an *inferred* argument which is neither specified during command creation nor serialized to XML files. It provides no significant information about the command, but serves merely to simplify interactive replay and enhance the history list.

For EntitiesCommand and all derived classes, Source stores the first valid Site value found in the Entities collection at the time *before* the command was executed. If all entities are unplaced, Source retains its inherited invalid value.

6.2.3 Derived Classes

This section's treatment of the concrete classes in the Command inheritance tree (see Figure 28 on page 93) is *very* brief. Section "Commands" on page 23 already lists the effects of all game commands, and "Command Algorithms" on page 120 will provide full implementation details, including pseudocode programs and customization options. Therefore, Table 6 merely shows an overview of the additional arguments taken by each command class.

Command Class	Additional Arguments
AttackCommand	Entities specifies all attacking units. Target specifies the map site that is under attack.

Table 6: Command Classes

Command Class	Additional Arguments
AutomateCommand	Entities specifies all entities whose actions to automate. Text specifies arbitrary internal information.
BeginTurnCommand	—
BuildCommand	EntityClasses specifies all classes to build. The same class may be specified repeatedly. One entity is built for each occurrence.
DestroyCommand	Entities specifies all destructible entities to destroy.
EndTurnCommand	—
MoveCommand	Entities specifies all moving units. Target specifies the destination of the movement.
PlaceCommand	Entities specifies all entities to place. Target specifies the map site on which all entities are placed.
RenameCommand	Entities specifies all entities to rename (usually just one). Name specifies the new display name for all entities.
ResignCommand	—

Table 6: Command Classes

These concrete classes inherit either directly from Command or from another abstract class that provides additional arguments as required. The two exceptions are the Text and EntityName arguments of AutomateCommand and RenameCommand, respectively, which are handled by these concrete classes themselves. All derived classes receive a Faction argument that is defined by the Command class and always valid. See subsection "Command Arguments" below for details on the internal representation of command arguments.

6.2.4 Command Arguments

Command arguments include the Entities, Faction, Source, Target, and Turn properties of the Command class, the EntityClasses property of the CommandClasses class, and the EntityName argument of the Rename class.

As you may recall from chapter "World State" on page 69, four of these arguments are implemented as rather complex structures:

— EntityReference for the Entities argument

— FactionReference for the Faction argument

— SiteReference for the Source and Target arguments

In this section, we'll explain the purpose and use of these structures.

The Problem

There are a number of good reasons why command arguments should only use identifiers and coordinates, but no direct object references:

1. They must be serialized to XML files which cannot store object references.

2. They must be capable of referencing different but equivalent objects in any number of world states that were created as deep copies of the original world state.
3. Related to the previous point, they must not retain references to objects which belong to outdated world state copies, or else we would have a memory leak.

Still, the Validate method needs to acquire object references for all command arguments except Turn and EntityName to test if a valid object exists, and the HCL program is generated directly after successful validation.

It would be a waste to throw away all object references we acquired during command validation, only to immediately re-acquire them during program generation. Moreover, it would be very convenient for rule script authors if each Command object came with usable references.

The Solution

Fortunately, the .NET Framework provides us with a ready-made solution for this problem.

We can create so-called "weak references" to an object which enable us to directly access the object until it has been garbage-collected, while allowing garbage collection as soon as there are no more "strong references" to the object, thus preventing memory leaks.

The above-mentioned structures are, in fact, wrappers for weak references to their associated objects. They all carry the weak reference itself in their Value component.

Value defaults to a null reference. After successful validation, Validate changes Value to the correct object reference for the current world state. Finally, when the object has been garbage-collected, Value reverts to a null reference.

The Id and Location components always provide the identifiers and coordinates, respectively, that allow Validate to re-acquire the reference from a given world state at any time. These components are also serialized to XML files.

The Name components of the FactionReference and EntityReference structures retain the referenced object's display name once Value has been set for the first time. This allows us to show proper names in the history list without having to re-acquire the references.[11]

Turn, EntityName, and EntityClasses

Obviously, no wrappers are required for the Turn and EntityName arguments, which are a simple integer value and an arbitrary string, respectively.

The EntityClasses argument does pair an extra identifier with each entity class to support XML serialization, but the referenced EntityClass object itself does not need to be wrapped in a weak reference structure. Like all scenario objects, it is immutable for the duration of a game, so the issue of incorrect references or memory leaks does not arise.

When to Validate?

Since interactive replays are always performed on deep copies of world states, the Value components of all weak reference structures must be refreshed before they are usable.

For example, the replay manager calls Validate on each command before attempting to highlight Source and Target locations, even though all commands in the history list are known to be valid according to the game rules.

Likewise, whenever you need an object reference for a command argument, you must validate the command on your current world state to ensure that Value is properly set.

---

11. In the case of destroyed entities, this would not even be possible without replaying the entire game. However, starting with Hexkit 3.4.0 we might look up display names in the event history.

Fortunately, such validation is rarely necessary in rule script code. Any Command objects that are supplied to the virtual Faction, Entity, and Unit methods are always guaranteed to have been validated on the same world state that is stored in the command's execution context.

6.3 Instructions Namespace

The Hexkit.World.Instructions namespace resides in the Hexkit.World assembly and provides the internal representation of HCL instructions.

Figure 30 shows the inheritance tree that starts with the Instruction class. The leaves of this tree implement specific instructions while the trunk contains abstract base classes that handle instruction arguments and other common functionality. The concrete derived classes dispatch to the appropriate internal methods when an instruction is executed.

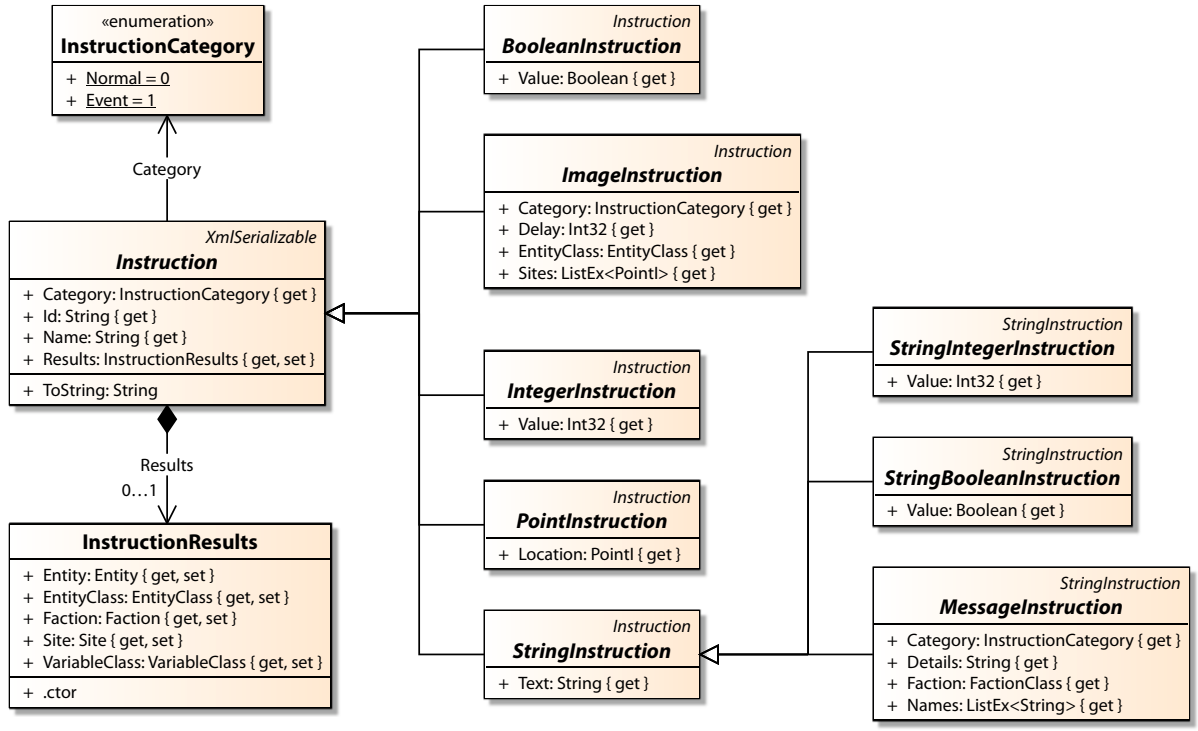


Figure 30: Instructions Namespace

You will notice that Figure 30 does not show any of these concrete classes. The Hexkit Command Language defines so many instructions that showing them all in UML diagrams would require several pages. Please refer to "Available Instructions" on page 100 for an overview of all concrete Instructions classes.

6.3.1 Instruction Class

The abstract base class Instruction provides common functionality required to manage HCL instructions. As noted above, this class and its descendants are never used directly by gameplay code, so we'll only briefly consider its members.

Category — The category of the instruction. Normal indicates a data change in the world state, Event indicates a command event; see "Instruction Usage" on page 102 for details.

Id — The identifier of some World or Scenario object that is manipulated by the instruction. This is the common instruction argument; derived classes add other arguments as required. See "Instruction Arguments" on page 103 for details.

Name — The display name of the instruction. This is the same as the class name.

Results — An optional data container holding any objects that were created or affected by the instruction. See "Instruction Results" on page 104 for details.

6.3.2 Available Instructions

We'll forgo describing the internals of the remaining Instructions classes. They are too numerous, not very interesting, and never directly used by gameplay code anyway. Instead, every HCL instruction has a corresponding method in the Command class, and gameplay code simply calls this method to generate, execute, and store the instruction.

This subsection provides a quick reference to the available instructions. Table 7 shows the name of each instruction with a brief description of its effects. Instruction categories other than Normal are indicated. Please refer to the *Hexkit Class Reference* for details on the arguments and effects of all instructions.

Instruction	Description
AdvanceFaction	Activates the next faction. For internal use only.
CreateEntity	Creates a new entity from a specified entity class, using the CreateEntity factory method of the current rule script.
CreateFaction	Completes the creation of a faction. For internal use only.
DeleteEntity	Deletes an entity from the world state. Use this instruction in the Entity.Delete method only.
DeleteFaction	Deletes a faction from the world state. Use this instruction in the Faction.Delete method only.
MoveEntityToTop	Moves an entity to the top of its local site stack, thus making it visible on the map view (unless obscured by another stack).
MoveImage	Smoothly moves an entity class image across a sequence of map sites. (Event)
SelectEntity	Selects an entity in the map view and data view. (Event)
SetEntityClass	Sets the EntityClass of an entity (see "Instruction Usage").
SetEntityDisplayClass	Sets the DisplayClass of an entity.
SetEntityFrameOffset	Sets the FrameOffset of an entity.
SetEntityName	Sets the Name of an entity.
SetEntityOwner	Sets the Owner of an entity.
SetEntitySite	Sets the Site of an entity.

Table 7: HCL Instructions

Instruction	Description
SetEntityUniqueName	Assigns a Name to an entity that is unique within the possessions of a specified faction.
SetEntityVariable	Adds or changes one current Attributes, Counters, or Resources value of an entity.
SetEntityVariableInitial	Adds or changes one initial Attributes, Counters, or Resources value of an entity, and resets the current value for attributes.
SetEntityVariableModifier	Adds or changes one AttributeModifiers or ResourceModifiers value of an entity.
SetFactionResigned	Sets the IsResigned flag of a faction.
SetFactionUnitsCanAttack	Sets the CanAttack ability of *all* units of a faction that are instances of a specified unit class.
SetFactionUnitsCanMove	Sets the CanMove ability of *all* units of a faction that are instances of a specified unit class.
SetFactionVariable	Adds or changes one current Counters or Resources value of a faction.
SetFactionVariableInitial	Adds or changes one initial Counter or Resources value of a faction.
SetFactionVariableModifier	Adds or changes one ResourceModifiers value of a faction.
SetSiteOwner	Sets the Owner of a map site.
SetSiteUnitOwner	Sets the Owner of *all* units on a map site.
SetUnitCanAttack	Sets the CanAttack ability of a unit.
SetUnitCanMove	Sets the CanMove ability of a unit.
SetWinningFaction	Sets the WinningFaction of the current world state to a specified faction and ends the game. For internal use only.
ShowImage	Shows an entity class image on one or more map sites, sequentially in the latter case. (Event)
ShowMessage	Shows a message in the event view panel. (Event)
ShowMessageDialog	Shows a message in the event view panel, and also in a modal dialog. (Event)

Table 7: HCL Instructions

6.3.3 Instruction Usage

Event Instructions

Instructions of the Event category represent command events, as described in "Command Events" on page 30. These instructions do not manipulate the current world state and have no effect when executed. Instead, they act as simple data containers for their arguments.

When a command's HCL program runs, any event instructions are passed to the ShowEvent callback method of the current execution context. The callback target must reside in the Hexkit.Game assembly and manipulate the display to achieve the desired effect.

If the execution context does not define a ShowEvent callback method, any event instructions are simply ignored. This is desirable in the following situations:

— Loading a saved game, which involves silently replaying all stored commands.

— Skipping forward during an interactive replay.

— Calculating a computer player turn in a background thread, which involves silently executing newly generated commands on a deep copy of the current world state.

Regarding the last point, a future Hexkit version might feature a secondary map view that lets human players watch the computer player's progress. In this case, a valid ShowEvent method would be supplied during computer player calculations.

CreateFaction Instruction

Hexkit currently does not support the creation of new factions after the game has begun. The CreateFaction instruction exists only to separate the creation of a new Faction instance from the start of its history recording.

Prior to Hexkit 3.7.0, a faction's first history snapshot was taken immediately after the Faction object was instantiated, and before Faction.Initialize was called. But if that method was overridden in a rule script, the first snapshot could differ significantly from the initial situation visible to the player, and also from the second history snapshot at the end of the first turn.

For example, the "Roman Empire" scenario overrides Faction.Initialize to create numerous immobile militia units that guard each faction's towns. While unproblematic as far as gameplay is concerned, this would cause an ugly "jump" in the history graph of the Faction Ranking dialog between the first and second turn.

Hexkit 3.7.0 solved this issue by delaying a faction's history recording until Faction.Initialize has returned. This required defining a new HCL instruction, however, so that the first history snapshot receives correct data even after a game had been saved and reloaded. Thus, we have the CreateFaction instruction whose only purpose is to start history recording for the faction.

SetEntityClass Instruction

SetEntityClass is a very complex instruction that requires some elaboration. Rule scripts can use this instruction to change an entity's class after the entity has been created. This is a rather tricky operation because so much of an entity's data is derived from its class.

The following list summarizes the most important points to observe, which are also listed in the documentation for the Entity.SetEntityClass method:

— You can only change the current EntityClass to another class of the same category.

— When an entity is created, its identifier is constructed from the identifier of its class and a running instance count. The identifier is immutable, however, and thus ceases to reflect either the current class or its instance count when you change the class.

— All variable collections are set to the default values for the new EntityClass and thus require you to manually transfer any desired existing values, except…

— …existing Counters values are always copied, even if matching identifiers do not exist in the Counters collection of the new class.

— …existing Resources values are copied, but only if matching identifiers already exist in the Resources collection of the new class.

The other Entity properties remain unchanged, unless of course they are merely proxies for the new EntityClass. All told, SetEntityClass is an unusually expensive instruction that should be used with care. Prefer the cheaper and simpler SetDisplayClass instruction wherever possible.

6.3.4 Instruction Arguments

The remaining subsections describe the internal storage of HCL instructions. You don't need to know about these things to write Hexkit rule scripts, but they might be interesting anyway.

Flexible Semantics

Unlike command arguments, instruction arguments do not have fixed semantics. Their actual meaning is defined by the concrete class that makes use of an argument, rather than by the abstract class that defines the argument. These "flexible semantics" are necessary to avoid multiple abstract base classes with differently named, but otherwise identical, properties.

Since the meaning of each argument is not fixed, it is always possible that a given instruction does not actually use a specific argument under certain circumstances. Therefore, all instruction arguments may be null references, empty strings, or otherwise invalid values. It is up to the constructor and execution method of each concrete class to check all arguments for validity.

Instruction.Id — The identifier of an entity class, entity, or faction.

BooleanInstruction.Value — A value for some boolean flag.

IntegerInstruction.Value — A value for some integer variable.

PointInstruction.Location — The coordinates of a map site.
Negative coordinates indicate that no map location is provided.

StringInstruction.Text — A message text, a display name, or a second object identifier.

StringBooleanInstruction.Value — A value for some boolean flag.
We need a separate class for this combination since C# does not support multiple inheritance from InstructionBoolean and InstructionString.

StringIntegerInstruction.Value — A value for some integer variable.
Again, we need a separate class due to the lack of multiple inheritance.

The semantics of the InstructionImage and InstructionMessage classes are fixed rather than flexible. These classes define map view and message events, respectively, and provide the specific arguments required for their event types. Please refer to the *Hexkit Class Reference* for a detailed description of these arguments.

Identifying Objects

HCL instructions do not store any reference-type arguments other than strings. All arguments that refer to World or Scenario objects are expressed as names, identifiers, or map coordinates. Any actual objects that are required while the instruction is executed must be retrieved by their identifiers or coordinates.

For one thing, we cannot store direct references to World objects for the same reason that we cannot store them with commands: they would cause memory leaks and become rapidly outdated as the original world state is repeatedly replaced by deep copies.

Special-casing Scenario objects, which we *could* store directly, does not seem to be worth the effort in the deep inheritance tree of the Instructions namespace. The only exception is MessageInstruction.Faction which returns the faction affected by a message event.

Using weak references to World objects, as with command arguments, seems like a waste of memory. Instructions do not have a separate validation step beyond which object references should be retained, nor are they extendable by user code that might enjoy the convenience of pre-created object references.

Therefore, most HCL instructions only store value-type arguments and strings – identifiers, names, and message text. All strings are interned to save space and to speed up searches.

6.3.5 Instruction Results

Some instructions must return the result of their execution back to their caller, or should do so as a convenience. The InstructionResults class is a simple data container that allows flexible backward communication from instructions to their containing command.

Any instruction method of the Command class that requires execution results assigns a new InstructionResults object to the Instruction.Results property of the executed instruction. The instruction's execution method then checks if the Results property is set to a valid object and if so, assigns any applicable result values before returning:

Entity — The new entity created by a CreateEntity instruction, or the existing entity affected by a DeleteEntity, SetEntityOwner, SetEntitySite, SetEntityVariable, or SetEntityVariableModifier instruction.

Faction — The faction affected by a SetFactionVariable or SetFactionVariableModifier instruction, or the *previous* owner of any objects affected by a DeleteEntity, SetEntityOwner, SetSiteOwner, or SetSiteUnitOwner instruction.

Site — The site affected by a SetSiteOwner or SetSiteUnitOwner instruction, or the *previous* site of any entities affected by a DeleteEntity or SetEntitySite instruction.

VariableClass — The class of the variable changed by a SetEntityVariable, SetEntityVariableModifier, SetFactionVariable, or SetFactionVariableModifier instruction.

Apart from the special case CreateEntity, whose newly created entity could not easily be found if it was not returned by the instruction itself, most return values are a convenience rather than a necessity. Some allow the caller to reuse object references that the instruction had to acquire anyway, and some provide the original values of changed properties that the caller would otherwise need to save before executing the instruction.

The purpose of these return values, again apart from CreateEntity which simply returns the newly created entity to the calling gameplay code, is to notify any affected factions and entities of the data changes caused by the instruction. To this end, the instruction methods of the Command class invoke dedicated handler methods on the affected objects, supplying the matching

InstructionResults properties as arguments. This mechanism and its customization is described in "Acting on Data Changes" on page 114.

Since InstructionResults uses direct references to World objects, all instruction methods must set the Instruction.Results property to a null reference before returning, in order to avoid memory leaks. Using weak references here would not only be inconvenient but impossible because DeleteEntity returns a reference to an entity that no longer exists anywhere else.

6.4 Command Automation

A custom rule script may perform any number of automated actions during the execution of a game command. Hexkit provides several mechanisms for this purpose.

6.4.1 Instruction Inlining

Since every command's effects are encoded in an HCL program, the most obvious way to perform arbitrary actions is to directly inline the desired instructions. Call the command's instruction method that matches the desired instruction, and supply the desired arguments – that's all.

Unfortunately, this is a rather laborious way to implement complex actions. Moreover, if you wished to replicate the effects of an existing command in this way, such as Attack or Move, you would want to make sure that your custom instruction sequence exactly matches the effects of the normal implementation of these commands. There are simpler ways to achieve that purpose.

6.4.2 Command Inlining

Hexkit allows the programmatic execution of entire commands, not just individual instructions. While generating a command's HCL program, additional commands may be either inlined for immediate execution, or enqueued for future execution.

To *inline* another command's HCL program with the current command's program, simply create the desired command and pass it to the InlineCommand method of the current command. The new command's HCL program will be generated and executed *immediately* before the call returns. The command itself will *not* be added to the history list; instead, the instructions in its HCL program are directly added to the current command's Program. The effect is the same as if you had manually created and inlined all the instructions of the inlined command.

Note. You cannot inline commands that control turn advancement, i.e. Begin Turn and End Turn. You *can* inline Resign commands but they will only set the executing faction's resignation flag; the faction won't actually resign until the next End Turn command is executed.

6.4.3 Command Queuing

As an alternative to the direct inlining of HCL instructions and game commands, you may also enqueue commands for later execution. This more complex mechanism provides some visual structure to long chains of automated actions.

The Problem

Automatically performing a long sequence of actions in response to a single game command is likely to confuse and annoy human players. Imagine them sitting in front the screen, watching the text in the event view grow longer and longer, seemingly without end…

Moreover, if the action takes places at a remote map location, the rule script author would have to manually add SelectEntity instructions that scroll the map view to the affected entities, allowing the user to see what's going on.

The Solution

To mitigate these effects, Hexkit provides a *command queue* linked to each game command. Just as with inlining, you first create the desired commands, but now you pass them to the QueueCommand callback method of the current command's execution context. The QueueCommand property is guaranteed to be valid in all commands received by any virtual command method.

All queued commands are executed *following* the current command, before control returns to the active player. In the case of human players, command execution is controlled by the Game.SessionExecutor class which automatically scrolls map locations into view and inserts the appropriate delays, as if during interactive command replay.

Once a queued command has been executed, it is added to the history list like any other command and so available normally when saving, restoring, and replaying games. In the latter case, the normal replay mechanism will now take care of map view scrolling and delays.

Note. As with inlining, you cannot enqueue Begin Turn or End Turn commands.

Chained Queues

When a queued command is finally executed, it may enqueue yet more commands for subsequent execution. The command queue is always worked in a FIFO (first in, first out) sequence: all previously queued commands are executed before any newly enqueued commands.

As a general rule you should avoid chained queues. They increase the danger of incompatible world states, a problem inherent with command queuing that we'll discuss next.

6.4.4 Queuing with Automate

The Problem

All queued commands and all their arguments must be specified explicitly in advance, at the time when the commands are placed in the queue. Therefore, whenever you enqueue more than one command, there is a danger that the execution of some earlier command might result in a world state that is incompatible with the arguments of some later command.

Such conflicts may cause unexpected results or even exceptions during command execution. For example, if a scenario enforces stacking limits, the execution of an earlier Move command might render the target of a later Move command unreachable.

The Solution

Since the problem is caused by the need to specify all commands and arguments in advance, the solution is to introduce a "container command" that merely hints at the desired actions, rather than specify them explicitly. That is the Automate command.

The Automate command takes an arbitrary collection of entities that are somehow related to the desired actions, and some arbitrary text that is never displayed to the player. Executing this command calls the virtual Faction.Automate method which does absolutely nothing by default.

So when you wish to enqueue a complex sequence of commands that might conflict with each other, you should create and enqueue Automate commands instead of the actual desired commands. Supply whatever entities make sense, e.g. the moving units, and a string that helps identify the desired action, e.g. "Move" or perhaps "Move (043,016)".

Now your rule script must also override Faction.Automate. Within that method, examine the supplied AutomateCommand to determine what actions to take, if any. Finally, if those actions agree with the current world state, create and inline the actual desired command, e.g. a MoveCommand with the specified entities and a target location extracted from the specified text.

The Automate command is executed as usual, except that the name of the command appears as "Automate" in the event view and in the Command History.

Chapter 7: Custom Rules

Hexkit defines a set of *default rules* which are described in the second chapter, "Hexkit Game". These rules are sufficient to provide a playable game but they are rather primitive and, for the most part, not suitable for anything but a test scenario.

This is where Hexkit's most significant feature comes in. Any .NET programmer can create scenario-specific *custom game rules* that enhance or replace the default rules, within the limits set by the Hexkit engine.

Custom rules are physically deployed in rule script files, so we discuss those first. Logically, custom rules are implemented by overriding the methods and properties of the classes that represent factions and entities. The rest of this chapter is devoted to discussing the available virtual class members, and the best strategies for overriding them.

The one game mechanism not covered here is pathfinding. This subject is complex enough to deserve a dedicated chapter, "Pathfinding" on page 132. Also, please refer to the preceding chapters for a breakdown of Hexkit's class structure on which the game rules are built.

7.1 Rule Script Files

You must provide a *custom rule script* with your scenario to change or enhance the default rules of Hexkit Game. Rule script files can be supplied in one of two formats:

— A single source code file written in C#, JScript .NET, or Visual Basic .NET. This file will be compiled automatically as Hexkit Game loads the scenario.

— A precompiled .NET assembly, written in whatever language you prefer. See the batch file Scenario\Rules\Compile.bat in your Hexkit installation directory for an example of how to set up the compilation step.

The rule script associated with a scenario is loaded (and compiled if necessary) automatically as the scenario is started. Any errors that occur during loading (and compilation) are reported to the user and prevent the scenario from starting.

Rule scripts use a factory pattern to override five central types in the Hexkit.World assembly, namely Faction, Unit, Terrain, Effect, and Upgrade. These classes provide the basic implementation for factions and entities within a world state (as opposed to a scenario).

Any modifications and extensions to the default rules that the script provides should be described in the scenario documentation so that the player knows what to expect.

Note. Despite the name, the implementation of Hexkit rule scripts does not use the dedicated scripting facilities offered by the .NET Framework. These facilities were built for maximum security in the context of web applications. They are of little use to "smart clients" that require high performance and tight integration with the host program.

7.1.1 The Default Rule Script

Hexkit Game always requires a rule script to start a scenario, so it substitutes a *default rule script* if no custom rule script is provided. This script basically does nothing – the factory methods are

present, but the derived types they return are merely empty wrappers for the corresponding base types defined in the Hexkit.World assembly.

The default rule script is written in C# and located at Scenario\Rules\Default Rules.cs in your Hexkit installation directory. Use this file as a template for your own rule scripts. The same folder contains the rule scripts for all demo scenarios which you might also wish to examine.

7.1.2 Debugging the Rule Script

Rule scripts that are provided as source code (including the default rule script) are usually compiled in memory, without creating any disk files. Unfortunately, .NET assemblies that only exist in memory are not visible to the source-level debugger of Visual Studio.[12]

The method Hexkit.Scenario.RuleScript.Compile contains conditional compilation directives that work around this problem. If the symbol DEBUG is defined, the compiled rule script assembly is written to a uniquely named DLL file in the Windows directory for temporary files (normally C:\Windows\Temp), along with a matching PDB file with debugging data.

So if you need to debug your rule script, you should get the Hexkit source package and recompile in debug mode. This is preferable anyway since it will also give you full source-level debugging for the main Hexkit code, which is closely intertwined with the rule script.

Note. Because .NET cannot unload individual assemblies, Hexkit Game cannot delete any of the DLL and PDB files created in the temporary folder! When running Hexkit Game in debug mode, you must remember to delete these files manually after closing the application.

7.2 Factory Class

Rule scripts establish communication with the Hexkit.World assembly by way of a factory class which is found by name (using reflection). Every rule script must define a public class named RulesFactory that implements the interface Hexkit.World.IRulesFactory.

This class must have a public parameterless constructor. Hexkit creates a single instance of the current rule script's RulesFactory class when a game is started or resumed, and stores this instance in the Scenario.RuleScript.Factory property[13] while the game is running.

7.2.1 Factory Methods

The constructor of the RulesFactory class may be used to perform custom initialization tasks for the rule script. A dedicated Initialize method is also available for that purpose. Conversely, IRulesFactory derives from the standard interface IDisposable, and the mandatory Dispose method may be used to perform custom shutdown tasks.

The main purpose of a factory class is to create objects, however, and IRulesFactory specifies two methods for the creation of factions and entities. All told, the RulesFactory class must implement the following methods:

12. The C# compiler actually creates a temporary DLL file even when compiling "in memory". However, this file is normally deleted after compilation and therefore unavailable for debugging.

13. Remember that you must use the global singleton instance to access scenario properties, so the actual path to this property is Hexkit.Scenario.MasterSection.Instance.Rules.Factory.

Constructor — Since the factory class must be instantiated anyway, you might as well co-opt the constructor to perform any rule script initialization that does not require a valid WorldState object. Setting pathfinding parameters is a common example.

The global scenario instance is already available at this point, including the map geometry but not the map contents – unless you want to manually traverse the data stored in the Scenario.AreaSection class!

Initialize() — Use this method to perform any rule script initialization that *does* require a valid WorldState object. This method is called once at the beginning of a new game, after all initial CreateFaction and CreateEntity calls but before the first Begin Turn command.

For faction-specific initialization you should override Faction.Initialize instead, as described in "Creating Factions and Entities" on page 112.

Dispose() — If your rule script requires any unmanaged (disposable) resources, you should release them at this point. You may also perform any other shutdown tasks that your scenario requires. This method is called once when a game is closed.

CreateFaction(FactionClass) — Creates a new World.Faction object based on the specified Scenario.FactionClass object. Presumably you'll return an instance of a custom class derived from Faction instead, such as CustomFaction in the sample rule scripts.

This method is called once for each faction, at the beginning of a new game.

CreateEntity(EntityClass) — Creates a new World.Entity object based on the specified Scenario.EntityClass object. Again, you'll probably return an instance of a custom class instead.

That class should *not* derive directly from Entity, but rather from the specialized base classes Unit, Terrain, Effect, and Upgrade. The class names used in the sample rule scripts are CustomUnit, CustomTerrain, CustomEffect, and CustomUpgrade.

This method is called whenever a new entity is created, either at the beginning of a new game or at any time during an ongoing game.

7.3 Extendable Classes

This section describes how you can and should use a rule script to extend the Faction class and the four classes derived from Entity. There are two basic ways in which you can customize a scenario by overriding virtual properties and methods in a rule script.

— Customize the data reported back to Hexkit concerning a faction or entity.

This is achieved by overriding various properties and methods that do not change the current world state. The reported data is displayed in the GUI, evaluated by the computer player, and used during the generation of a command's HCL program.

— Customize the generation of HCL instructions when a game command is executed.

This is achieved by overriding the virtual command methods of the Faction, Entity, and Unit classes, and any auxiliary methods they depend upon.

Typically, your rule script will contain some customization in both areas. The following discussion groups the available virtual properties and methods by functionality. We'll cover basic management tasks in this section, and defer command customization to the next section.

Many of the techniques described here are exemplified in the rule scripts of the scenarios that ship with Hexkit, so I suggest you take a look at these files. You should also check out the World.Finder and World.WorldUtility classes which contain many useful helper methods.

This chapter does not explain how to evaluate possessions and calculate supply resources. These mechanisms are currently mostly or exclusively used by computer players, and therefore described in "Evaluating Possessions" on page 140 and "Calculating Supplies" on page 149.

Random Numbers. The static field Tektosyne.MersenneTwister.Default provides a convenient way to generate pseudo-random numbers for non-deterministic actions. This field provides a shared instance of the "Mersenne Twister" RNG defined by the Tektosyne library. Alternatively, you are free to use the standard library class System.Random or any other RNG of your choice.

Call the Base Class. When overriding a virtual method in a rule script, remember to call the method's base class implementation, usually before doing anything else. Sometimes you may wish to perform custom actions before that call, and sometimes you may wish to omit the call entirely, for example to suppress an unwanted message event. Before doing so, however, you should study the default implementation to ensure it doesn't perform any crucial bookkeeping operations that must be executed before your custom code!

7.3.1 Pseudocode Programs

In this chapter we'll use pseudocode programs to show Hexkit's default implementations of game commands and related methods. We use actual method and property names where possible or necessary, and summary descriptions otherwise.

Compared to the actual code available in the source package and its documentation in the *Hexkit Class Reference*, these pseudocode programs serve two purposes:

— Trace gameplay algorithms across multiple methods and classes, without having to traverse a dozen hyperlinks back and forth.

— Highlight opportunities for gameplay customization. Virtual methods and properties that can be overridden by a custom rule script appear in *italics,* and properties whose values can be set only within Hexkit Editor appear in **bold.**

Generally, the programs should be self-explanatory. Method invocations are indicated by a pair of parentheses but the actual parameters are usually omitted. When such a method invocation is followed by another indented block of code, that block is the method's definition.

For command programs, we list the algorithm that generates the HCL program under the heading GenerateProgram. If the command requires its arguments to pass non-trivial validation steps, any such steps are listed under the heading Validate. There is one validation step that we omit but that might not be obvious: commands that take a list of entities always require that all such entities are of the same category.

Note. The pseudocode programs do not necessarily represent the actual instruction flow in the C# methods. Obvious conditional statements are always omitted; for example, any actions on an entity's site should be considered skipped if the entity is unplaced. Moreover, the pseudocode is generally arranged for greatest clarity while the actual C# code might express the same semantics in a different sequence for the sake of optimization or data integrity.

7.3.2 Data Persistence

As described in "Command Execution" on page 90, Hexkit persists saved games as a sequence of HCL instructions. Saved games do not directly store any of the built-in data for a world state, much less any data that you might add to the various extendable classes in a rule script.

You are free to introduce additional data fields, but you must be aware that any data that is not controlled by HCL instructions will be lost as soon as the current world state is replaced with

a clone. This happens whenever a game is replayed for any reason, including loading a saved game, but also when a computer player has finished its turn.

Therefore, you should use custom data fields – and any existing non-HCL properties such as Site.SupplyResources – strictly as a "scratch pad" while generating the HCL program for a *single* command. Between any two commands, a replay might wipe out all your custom data!

If you wish any data to persist between commands, you must be able to detect if the data was lost, i.e. replaced with the default-initialized data of a newly cloned world state. Whenever this happens, you must be able to fully regenerate the data from the current world state. This is how the SupplyResources lists are managed internally, by the way.

Guaranteed Actions

Certain creation and initialization tasks are always performed whenever a scenario is replayed, and therefore you *can* rely on the persistence (or rather, the identical automatic recreation) of any custom data established at that point. These tasks include all RulesFactory methods, the method Faction.Initialize, and the constructors of all game objects.

You should utilize these methods if you need to initialize some custom scenario data that the rest of your rule script will treat as read-only. For example, the "Battle of Crécy" and "Battle of Poitiers" demo scenarios use the RulesFactory.Initialize method to establish internal maps for the movement restrictions of computer-controlled factions.

But what if you need custom data that may be changed outside of these guaranteed actions? In that case you'll have to use entity or faction variables, which we'll discuss next.

Adding Persistent Data

The SetEntityVariable, SetEntityVariableModifier, SetFactionVariable, and SetVariableModifier instructions will add a new value to the existing variable collections of an entity or faction if the specified identifier is not already found in the corresponding collection.

You can use this feature to add persistent integer data at runtime. The new identifier must match one of the predefined variable classes in the current scenario.

Any attributes, resources, and attribute or resource modifiers that you add will subsequently appear in the Hexkit GUI, and may be subject to the automatic update mechanisms described in "Attributes and Resources" on page 116. To avoid these issues, you should use *counters* for persistent data that is strictly internal. Variables of the Counter category, introduced in "Counter Variables" on page 49, are never shown to the user and are never automatically modified.

Factions and entities may be associated with arbitrary counter values, as long as each value is based on a Scenario.CounterClass defined by the current scenario. As usual, you can only store one value per CounterClass in any given variable collection.

Note that there are no counter modifiers. The distinction between basic values and modifiers makes no sense in the case of counters, as there is no automatic update mechanism. All counters are treated as basic values as far as Hexkit is concerned, so they are added and changed with the SetEntityVariable and SetFactionVariable instructions.

Since counters are not subject to automatic updates, their initial values are unused by Hexkit. You could therefore use the SetEntityVariableInitial and SetFactionVariableInitial instructions to associate each counter variable with a second arbitrary integer value.

7.3.3 Creating Factions and Entities

You may use the constructors of any extendable class defined in your rule script to perform initialization tasks for a specific faction or entity. However, a valid world state does not yet exist when factions are created, and may not yet exist when entities are created. It is usually better to

attach any customization of entities to the actions that create them, for example the Build command (see "Build Command" on page 125).[14]

You may also override the Faction.Initialize method to perform faction-specific initialization tasks at the beginning of a game, *after* the first valid world state has been established. This is the preferred way to initialize faction data. Faction.Initialize is called during the first Begin Turn command in every game (see "Begin Turn Command" on page 123).

Creating Entities

Entities (but not factions) can be created by game commands while a game is running. Use the CreateEntity instruction to create a new "instance" of a given EntityClass. This instruction invokes the CreateEntity method of your IRulesFactory implementation which in turn should call your rule script constructor for the appropriate class. The new entity will receive a unique identifier and join the Entities hashtable of the current world state.

The new entity is unplaced and unowned. Use the SetEntitySite or SetEntityOwner instructions to assign a new map site or owning faction, respectively.

The new entity's name defaults to the display name of its EntityClass. You can use the SetEntityName instruction to assign a new name. Alternatively, if the entity belongs to a faction, you might want to use the SetEntityUniqueName instruction to create a name with a numerical suffix that is unique among the faction's possessions.

Cloning Objects

You *must* override the copy constructor and the Clone method for any custom faction or entity class in your rule script so that deep copies of world states are populated with the correct derived types. The default rule script already contains the required code, but if you must remember to copy any new fields of your rule script types in the type's copy constructor. Keep in mind that the values of these fields are not persistent, as discussed in "Data Persistence" on page 111.

7.3.4 Deleting Factions and Entities

There are several ways to delete existing factions and entities. Players can issue a Resign command to effectively delete their own faction (see "Resign Command" on page 130), and Destroy commands to delete any of their faction's entities that are marked as destructible (see "Destroy Command" on page 126).

The Resign and Destroy commands eventually call the two virtual Delete methods shown in Figure 31. The default implementation of Entity.Delete is trivial, and that of Faction.Delete is complicated only by the deletion of any remaining faction possessions.

To programmatically delete a faction or entity, your rule script can either issue a Resign or Destroy command, or directly invoke the corresponding Delete method. Override the following methods to customize the deletion of game objects:

— Override Entity.Delete to perform additional actions when deleting an entity.

— Override Faction.Delete to perform additional actions when deleting a faction.

Note. The Delete methods are supposed to generate any HCL instructions required to "clean up" after a deleted faction or entity. You should *never* directly execute a DeleteEntity or DeleteFaction

14. Theoretically, the Entity class might define another virtual Initialize method, just like the Faction class. I'm skeptical of this approach, however, because the initial state of entities tends to depend on the context in which they were created.

```
Entity.Delete()
        instruction DeleteEntity(this)

Faction.Delete()
        foreach Site in Sites
                instruction SetSiteOwner(null)
        foreach Entity in Units, Terrains, Upgrades
                Entity.Delete()
        instruction DeleteFaction(this)
```

Figure 31: Delete Programs

instruction, except when overriding a Delete method, and then only to delete the one object on which it was invoked. Always call Delete or issue commands to destroy game objects!

7.3.5 Acting on Data Changes

Many HCL instructions change some property of a game object. The Faction and Entity classes define several virtual methods that allow you to react to such data changes, after the fashion of .NET event handlers. The Command methods that create and execute these instructions call the appropriate data change handler immediately after executing an instruction. The call is skipped if the instruction was redundant, i.e. did not actually alter the value of its target property.

Table 8 shows a list of all HCL instructions that have a corresponding data change handler, along with the virtual methods that are invoked for each instruction. The methods are invoked on the same Entity or Faction that is the target of the instruction, unless otherwise noted.

Instruction	Invoked Methods
DeleteEntity	Entity.OnOwnerChanged, Entity.OnSiteChanged
SetEntityClass	Entity.OnEntityClassChanged
SetEntityOwner	Entity.OnOwnerChanged
SetEntitySite	Entity.OnSiteChanged
SetEntityVariable	Entity.OnVariableChanged
SetEntityVariableModifier	Entity.OnVariableChanged
SetFactionVariable	Faction.OnVariableChanged
SetFactionVariableModifier	Faction.OnVariableChanged
SetSiteOwner	Entity.OnOwnerChanged (on all Site.Terrains and Site.Effects)
SetSiteUnitOwner	Entity.OnOwnerChanged (on all Site.Units)

Table 8: Data Change Handlers

Table 9 shows the default implementation of each handler method as a pseudocode program. Several Entity methods have specialized overrides which are shown as well. Overrides always call the base class implementations before doing anything else; these calls are not shown.

Method	Pseudocode Program
Entity.OnEntityClassChanged	UpdateSelfAndUnitAttributes()
Entity.OnOwnerChanged	UpdateSelfAndUnitAttributes()
Entity.OnSiteChanged	UpdateSelfAndUnitAttributes()
— Terrain.OnSiteChanged	OldSite.SetTerrainChanged() NewSite.SetTerrainChanged()
— Unit.OnSiteChanged	if NewSite.Owner <> Unit.Owner *Owner.CaptureSite()*
Entity.OnVariableChanged	UpdateSelfAndUnitAttributes()
— Terrain.OnVariableChanged	Site.SetTerrainChanged()
Faction.OnVariableChanged	/* empty virtual method */

Table 9: Data Change Programs

The various Entity methods call Entity.UpdateSelfAndUnitAttributes (described in "Attributes and Resources" on page 116) to update the attributes of any entity that could possibly be affected by the data change. This might result in one or more recursive calls to OnVariableChanged, but a properly designed attribute system should quickly reach a stable state that produces only redundant SetEntityVariable instructions (and thus no longer triggers OnVariableChanged).

Unit.OnSiteChanged also attempts to capture the unit's new map site, if possible. This action is implemented by the virtual method Faction.CaptureSite, shown in Figure 32.

```
Faction.CaptureSite(Site)
    if any Site.Terrains with Terrain.CanCapture
    and any Site.Units with Unit.Owner = this and Unit.CanCapture
        instruction SetSiteOwner(this)
        instruction ShowMessage
```

Figure 32: Capture Program

— Override Faction.CaptureSite to determine if and how units capture map sites. This method is by default invoked only after a unit's site has changed.

The Terrain class does not perform any actions in response to data changes but needs to set the TerrainChanged flags of the affected sites to trigger an update of any cached terrain properties – see "Terrain Value Caching" on page 76. Lastly, Faction.OnVariableChanged does nothing at all by default and exists only for the benefit of custom rule scripts.

7.3.6 Entity Invariants

When changing the Site and Owner properties of entities, you must observe various category-specific invariants. Hexkit will stop command execution if any invariant is violated.

Units — Owner must be valid unless the unit has been destroyed. Site may be valid or a null reference, but if valid, all units in the same Site must have the same Owner.

Terrains — Site may be valid or a null reference, but if valid, Owner must equal Site.Owner.

Effects — Site must be valid, and Owner must equal Site.Owner.

Upgrades — Site must be a null reference, and Owner must be valid.

Due to the unit stack invariant, you cannot transfer of an entire unit stack (all units in the same location) to another faction by simply looping through the units and changing their Owner. As soon as the first unit's Owner would change, the stack invariant would be violated.

You could set each unit's Site to a null reference, change the Owner of all units, and then re-deploy all units on the original Site. However, it is much faster and simpler to use the dedicated SetSiteUnitOwner instruction which changes the Owner of all units in a stack through an internal helper method that temporarily bypasses the stack invariant.

7.3.7 Attributes and Resources

The management of attributes and resources is largely automatic. Variables are updated based on the total sum of applicable modifiers. You can use the instructions SetEntityVariableModifier and SetFactionVariableModifier to change any desired modifiers, and the changes will be directly reflected in the user interface and in the automatic variable updates.

You can also use the instructions SetEntityVariable and SetFactionVariable to directly set any variable of an entity or faction to the desired value, but you should be aware that your value may be changed again or completely wiped out by the next automatic update. You may wish to change a variable's initial value instead, as described below.

Automatic variable updates are performed by several UpdateAttributes and UpdateResources methods that are called under the following circumstances:

— Changing certain entity data updates the attributes of any possibly affected entities, as described in "Acting on Data Changes" on page 114.

— All entity variables and faction resources are updated at the beginning of each turn, as described in "Begin Turn Command" on page 123.

— Resetting faction resources are also updated at the end of each turn, as described in "End Turn Command" on page 127.

As of Hexkit 4.1.0, automatic variable management is no longer customizable by rule script code (with one exception), so the following sections merely illustrate the built-in mechanism.

Initial Values

Aside from its current value, every variable is also associated with an *initial value*. This is either the value that was specified in Hexkit Editor, or the value supplied to the rule script instruction that originally created the variable. The automatic variable management ignores the initial value of most variables, except in the following three cases:

Entity Attributes — Current value always equals initial value plus modifiers.

Limited Entity Resources — Current value cannot exceed initial value.

Resetting Faction Resources — Current value is reset to initial value at the start of each turn, before applying modifiers.

You should *never* use SetEntityVariable to change attribute values, or SetFactionVariable to change resetting resource values, because your new current value will be obliterated by the automatic variable update process – possibly after some delay, to the great confusion of the player! You have two options to reliably change variable values that are subject to automatic updates:

1. Change or define modifiers that apply to the desired variable, using the SetEntityVariableModifier or SetFactionVariableModifier instruction. The player will see the changed modifier along with the changed value. This is the natural choice if the new value is considered temporary or otherwise "abnormal".
2. Change the initial value of the desired variable, using the SetEntityVariableInitial or SetFactionVariableInitial instruction. The player will only see the changed value, with no indication that it has ever been different. Choose this option if the new value is considered permanent, or the reason for its change is completely obvious.

In the case of limited entity resources, there is no such choice: the current value is always limited by the initial value, and you must increase the initial value before you can assign a greater current value. Of course you can also decrease the initial value, in which case a greater current value is automatically restricted to the new limit.

Modifier Maps

Unit variables may be modified by other entities in the same or a different location, including faction inventories. Since aggregating all applicable modifiers is quite expensive, every faction maintains *modifier maps* to cache the sums of all attribute and resource modifiers (other than self-modifiers) that currently apply to its units anywhere on the map.

These modifier maps are the two-dimensional arrays Faction.UnitAttributeModifiers and Faction.UnitResourceModifiers which store the aggregate modifiers for every map site. The maps are updated by any instruction that could conceivably change a relevant modifier.

The global flag WorldState.UnitAttributeModifiersChanged indicates whether UnitAttributeModifiers were changed and must be re-applied to all placed units. Any such change must trigger an attribute update for a specific entity which gives us an opportunity to examine this flag, as described below. No flag is required for UnitResourceModifiers since the resources of all entities are updated unconditionally at the start of each turn.

All properties related to modifier maps have internal visibility and therefore do not appear in the UML diagrams of chapter "World State" on page 69, but we do show them in the following pseudocode programs since they are important parts of the algorithms.

Entity Attributes

Figure 33 shows the methods that perform automatic entity attribute updates. The Entity.UpdateSelfAndUnitAttributes method is called by all data change handlers. This method always updates the attributes of the entity itself, and also those of all placed units if the current world state has marked the unit attribute modifier maps as changed.

For entities other than units, Entity.UpdateAttributes merely applies self-modifiers. For units, this method also applies the aggregate modifier value for the current site stored in the owner's attribute modifier map. In either case, each attribute's value is reset to its initial scenario value before applying the current modifiers.

Automatic attribute and resource updates are only performed for entities whose IsModifiable flag is set. This is the case for placed and owned units, placed terrains or effects, and owned upgrades. In particular, this means that variable modifiers are not automatically applied to units and terrains which reside in off-map faction inventories awaiting placement.

```
Entity.UpdateSelfAndUnitAttributes()
    UpdateAttributes()
    if WorldState.UnitAttributeModifiersChanged
        WorldState.UnitAttributeModifiersChanged := false
        foreach Site in WorldState.Sites
            foreach Unit in Site.Units
                Unit.UpdateAttributes()

Entity.UpdateAttributes() where Entity.IsModifiable
    foreach Attribute in Attributes
        value := Attribute.InitialValue + AttributeModifiers.Self
        if this is Unit
            value += Owner.UnitAttributeModifiers[Site][Attribute]
        instruction SetEntityVariable(Attribute, value)
```

Figure 33: Entity Attribute Programs

Entity Resources

Figure 34 shows the methods that perform automatic entity resource updates. As before, Entity.UpdateResources merely applies self-modifiers to non-unit entities. For units, this method also adds the aggregate modifier value for the current site in the owner's resource modifier map.

```
Entity.UpdateResources() where Entity.IsModifiable
    foreach Resource in Resources
        value := Resource.Value + ResourceModifier.Self
        if this is Unit
            value += Owner.UnitResourceModifiers[Site][Resource]
        instruction SetEntityVariable(Resource, value)
    if this is Unit
        Unit.AddSiteResources()

Unit.AddSiteResources()
    foreach Resource in Resources
        value := Resource.Value
        foreach Entity in Site.Terrains, Site.Effects
            if Entity.ResourceTransfer <> None
                supply := Entity.Resource.Value
                transfer := supply restricted by
                    Resource.Maximum/Minimum – value
                value += transfer
                instruction SetEntityVariable(Entity.Resource, supply - transfer)
                Entity.CheckDepletion()
        instruction SetEntityVariable(Resource, value)

Entity.CheckDepletion()
    if ResourceTransfer = Delete and all Resources = 0
    and this is not Terrain with Terrain.IsBackground = true
        Entity.Delete()
        instruction ShowMessage
```

Figure 34: Entity Resource Programs

Unit resources are potentially affected by automatic resource transfer. Unit.AddSiteResources transfers any available and required resources from eligible terrains and effects in the same map site to the unit's own resources. Where applicable, depleted entities are then automatically deleted by Entity.CheckDepletion, the only public virtual method here.

— Override Entity.CheckDepletion to determine whether an entity should be deleted, and to perform additional actions when that happens.

Faction Resources

Figure 35 shows the methods that perform automatic faction resource updates. Factions apply their own resource modifiers as well as the owner modifiers of any owned entities. Except for upgrades, the modifiers of unplaced entities are ignored. The Faction.ComputeResourceModifiers method returns its results as a CategorizedValue collection so that the Hexkit GUI can show a faction's income broken down by its various sources.

```
Faction.UpdateResources(type: accumulating or resetting)
    modifiers := ComputeResourceModifiers(type)
    foreach Resource in Resources
        value := Resource.Value if accumulating, Resource.InitialValue if resetting
        modifier := modifiers[Resource].Total
        instruction SetFactionVariable(Resource, value + modifier)
    if accumulating and any Upgrades
        AddUpgradeResources() /* not shown here */

Faction.ComputeResourceModifiers(type: accumulating or resetting)
    returning modifiers := empty collection of CategorizedValue
    foreach Resource in Resources where Resource.IsResettingResource = type
        modifiers += new modifier for current Resource
        modifier.Other := ResourceModifier
        foreach Upgrade in Upgrades
            modifier.Upgrade += Upgrade.ResourceModifier.Owner
        foreach Unit in Units where Unit.Site is valid
            modifier.Unit += Unit.ResourceModifier.Owner
        foreach Site in Sites
            foreach Terrain in Site.Terrains
                modifier.Terrain += Terrain.ResourceModifier.Owner
            foreach Effect in Effects
                modifier.Effect += Effect.ResourceModifier.Owner
```

Figure 35: Faction Resource Programs

Faction resources may be affected by automatic resource transfer, just like units, except that supplies are drawn from owned upgrades rather than local terrains and effects. We don't show Faction.AddUpgradeResources since this method is otherwise identical to Unit.AddSiteResources, including the transfer calculations and the call to Entity.CheckDepletion.

7.3.8 Standard Variables

Hexkit predefines semantics for a number of terrain and unit variables that are likely to appear in most scenarios. Each of these variables is represented by a property which maps to the value of the actual variable whose identifier is specified by the underlying EntityClass.

Table 10 shows the available standard variables and their default values. This is the attribute or resource with the indicated identifier, or else zero or one if no valid identifier is defined.

Property	Default Value	Command
Terrain.Difficulty	0 or TerrainClass.DifficultyAttribute	"Move Command"
Terrain.Elevation	0 or TerrainClass.ElevationAttribute	—
Unit.AttackRange	1 or UnitClass.AttackRangeAttribute	"Attack Command"
Unit.Movement	1 or UnitClass.MovementAttribute	"Move Command"
Unit.Morale	1 or UnitClass.MoraleResource	"Attack Command"
Unit.Strength	1 or UnitClass.StrengthResource	"Attack Command"

Table 10: Standard Variables

The Terrain and Unit classes also provide helper methods for changing the backing value of a standard variable. Each method prefixes the corresponding standard variable name with Set, e.g. Unit.SetStrength. These methods throw an exception if the underlying TerrainClass or UnitClass does not define a valid identifier for the standard variable.

Most standard variables are referenced by the Attack and Move commands under the default rules, except for Elevation which is currently used only by the Crécy and Poitiers scenarios. The two terrain attributes are also aggregated by eponymous Site properties. The aggregate values are cached using the mechanism described in "Terrain Value Caching" on page 76.

Lastly, the classes AttributeClass and ResourceClass define constant self-instances to represent all standard variables. Internally, these "pseudo-variables" are used only by GUI code, not by the default game rules, but they might come in handy as a means to refer to standard variables in custom rule scripts. See "Standard Variables" on page 59 for details.

7.4 Command Algorithms

This section shows pseudocode programs for the default behavior of all game commands, and describes how a scenario designer can customize that behavior.

Several commands rely on the static utility class World.Finder which provides helper methods to find map locations based on various criteria. The relevant methods are presented here while the entire class is shown in "Pathfinding Classes" on page 132.

7.4.1 Attack Command

Figure 36 shows the default implementation of the Attack command, described in "Attack Site" on page 25. Much of the complexity lies in the target finding methods shown in Figure 37. Ranged combat may run the Visibility algorithm, discussed in chapter "Pathfinding" on page 132. The Finder methods are also used when players attempt to determine possible attack targets.

Override the following methods to customize the Attack command:

— Override Unit.Attack to perform additional actions when conducting an attack, or to leave the Attack ability enabled.

```
Validate
        foreach Entity in specified Entities
                require Entity.Category = Unit
                require valid Entity.Site
        targets := Finder.FindAttackTargets(specified Entities)
        require targets contain specified Target

GenerateProgram
        Unit.Attack()
                attackers := specified Units
                defenders := all Site.Units of specified Target

                Unit.AttackCombat()
                        foreach Unit in attackers
                                defender := Unit.SelectDefender()
                                        return random Unit in defenders
                                Unit.AttackUnit(defender)
                                        defender.Delete()
                        foreach Unit in surviving defenders
                                attacker := Unit.SelectAttacker()
                                        targets := Finder.FindAttackTargets(this)
                                        return random Unit in both attackers and targets
                                Unit.AttackUnit(attacker)
                                        attacker.Delete()

                Unit.CreateAttackEvent(attackers)
                        instruction ShowMessage(losses)
                Unit.CreateAttackEvent(defenders)
                        instruction ShowMessage(losses)
                foreach Unit in surviving attackers
                        instruction SetUnitCanAttack(Unit, false)

Unit.EstimateLosses(Units, Target)
        AttackerStrength := sum of Unit.Strength of all Units
        DefenderStrength := sum of Unit.Strength of all Site.Units in specified Target
        DefenderLosses := AttackerStrength
        AttackerLosses := DefenderStrength – AttackerStrength
```

Figure 36: Attack Program

— Override Unit.CreateAttackEvent to change the after-action reports generated for both attackers and defenders after a battle.

— Override Unit.AttackCombat to change the fundamental combat system. By default, this method executes a series of duels: all attackers attack one defender each, then all surviving defenders counter-attack one attacker each.

— Override Unit.SelectDefender and Unit.SelectAttacker to change how each attacker or defender selects an opponent for a duel, assuming you keep the default system.

— Override Unit.AttackUnit to change how each duel between attacker and defender is resolved, assuming you keep the default system.

— Override Unit.EstimateLosses to change the estimates of hypothetical Unit.Strength losses on both sides so that they (roughly) match your customized combat system.

```
Unit.CanAttackTarget(Units, Target)
    succeed if any Target.Units with Unit.Owner <> Owner
        and Finder.AreUnitsInAttackRange(Units, Target)

Unit.CanAttackTarget(Units, Source, Target)
    succeed if any Target.Units with Unit.Owner <> Owner
        and Finder.AreUnitsInAttackRange(Units, Source, Target)

Finder.AreUnitsInAttackRange(Units, Target)
    foreach Unit in Units
        source := Unit.Site
        distance := MapGrid.GetDistance(source, Target)
        fail if distance > Unit.AttackRange
        if distance > 1 and Unit.RangedAttack = Line-of-Sight
            scale distance by world diameter of map polygon
            visible := Visibility.FindVisible(Site.BlocksAttack, source, distance)
            fail if not visible contains Target
    succeed

Finder.AreUnitsInAttackRange(Units, Source, Target)
    distance := MapGrid.GetDistance(Source, Target)
    fail if distance > minimum Unit.AttackRange of all Units
    if distance > 1 and any Unit.RangedAttack = Line-of-Sight
        scale distance by world diameter of map polygon
        visible := Visibility.FindVisible(Site.BlocksAttack, Source, distance)
        fail if not visible contains Target
    succeed

Finder.FindAttackTargets(Units)
    if any Unit in Units without Unit.CanAttack
        return empty collection
    targets := all map sites
    foreach Unit in Units
        restrict targets by MapGrid.GetNeighbors(Unit.Site, Unit.AttackRange)
    remove target from targets where not Unit.CanAttackTarget(Units, target)
    return targets

Finder.FindUnitsInAttackRange(Units, Target)
    return any Unit in Units where Finder.FindAttackTargets(Unit) contains Target
```

Figure 37: Attack Target Programs

— Override Unit.CanAttackTarget (both overloads) to impose additional restrictions on the possible attack targets of the specified units.

Practically every scenario will need a customized combat system, and that system is likely to consume the majority of its rule script's code. Aside from implementing more complex rules you will also want to show some visual indication of attacks and hits during combat, and you must override Unit.EstimateLosses to give decent estimates for your new system. I recommend that you take a look at the sample scenarios to get an idea how to go about this task.

Attack Targets

The standard variable Unit.AttackRange denotes each unit's maximum attack range in steps, with zero indicating civilian units that cannot attack (or counter-attack) at all, and one indicating mêlée units that can only attack adjacent map sites.

The Finder methods shown in Figure 37 define the set of possible attack targets of a group of units is the intersection of map circles around each unit whose radii equal their attack ranges. This set may be further restricted by CanAttackTarget, and also by visibility obstructions if any ranged attackers require a clear line of sight – see "Lines of Sight" on page 138 for details. You should always set AttackRange to the greatest possible range at which a unit could conceivably attack, then restrict target selection as needed.

Attack Planning

Unit.CanAttackTarget comes in two flavors: one that uses the current site of the specified units as the source location, and one that specifies an explicit hypothetical source location. The latter variant is used during movement planning by the computer player algorithms. When overriding one overload you must override the other as well, to ensure that the same combat system applies.

One variant doesn't simply forward to the other because the hypothetical variant should be more "optimistic" and succeed even if some additional conditions must be met to conduct the hypothetical attack, such as the presence of additional helper units. The idea is that the computer player will first move the specified units into place, and then look for help.

7.4.2 Automate Command

Figure 38 shows the default implementation of the Automate command, described in "Queuing with Automate" on page 106. Override the following method to customize its behavior:

```
Validate
        require specified Text not empty

GenerateProgram
        Faction.Automate()
                /* empty virtual method */
```

Figure 38: Automate Program

— Override Faction.Automate to perform the desired automatic action, either by directly executing HCL instructions or by inlining other game commands.

The default implementation does absolutely nothing because the Automate command is merely a hook for arbitrary actions defined by the scenario rule script.

7.4.3 Begin Turn Command

Figure 39 shows the default implementation of the Begin Turn command, described in "Begin Turn" on page 25. Override the following methods to customize its behavior:

— Override Faction.Initialize to perform additional actions at the start of the game, after the world state has been fully established but before the first Begin Turn command.

```
Validate
    require specified Faction = WorldState.ActiveFaction

GenerateProgram
    if first command during game
        foreach Entity in WorldState
            Entity.UpdateAttributes()
        foreach Faction in WorldState
            Faction.Initialize() /* empty virtual method */

    Faction.BeginTurn()
        foreach UnitClass in EntitySection where UnitClass.CanDefendOnly
            instruction SetFactionUnitsCanAttack(UnitClass, false)
        if first turn
            if first command during turn
                foreach Faction in WorldState
                    Faction.UpdateResources(type: resetting)
        else
            if first command during turn
                foreach Terrain, Effect, Upgrade in WorldState
                    Entity.UpdateResources()

            Faction.AutoDestroyUnits()
                candidates := Faction.GetUnsupportedUnits()
                if any candidates
                    instruction ShowMessageDialog
                    inline command Destroy(candidates)

            Faction.UpdateResources(type: accumulating)
            foreach Unit in Faction
                Unit.UpdateResources()

    WorldState.CheckVictory()
        victorious := ActiveFaction.CheckVictory()
            if only Faction in WorldState
            or any FactionClass.VictoryCondition.Threshold met
            or any Faction.Resource >= ResourceClass.Victory
                instruction ShowMessageDialog
                succeed
        if victorious
            instruction SetWinningFaction(ActiveFaction)
```

Figure 39: BeginTurn Program

— Override Faction.BeginTurn to perform additional actions at the start of each turn. You should not attempt to change the resource update logic in this method – change the variable definitions in Hexkit Editor instead.

— Override Faction.AutoDestroyUnits to change or suppress the Destroy command that is generated automatically to disband unsupported units. Normally, you would only override this method to change the generated message events.

— Override Faction.GetUnsupportedUnits to create a different list of units that should be disbanded by AutoDestroyUnits. You could allow a faction to keep unsupported units, or select the destroyed units by different priorities.

— Override Faction.CheckVictory to check for additional or different victory conditions than those defined by the scenario, or to change the generated message event.

The default implementation of Faction.GetUnsupportedUnits examines all units that contribute to the resource deficit, and selects those with the lowest valuation by Faction.Evaluate(Entity). This is the only use of valuations outside of computer player algorithms. The valuation system is described in "Evaluating Possessions" on page 140.

The Begin Turn command also calls some of the methods described in "Attributes and Resources" on page 116, and possibly those in "Destroy Command" on page 126.

7.4.4 Build Command

Figure 40 shows the default implementation of the Build command, described in "Build Entities" on page 26. Override the following methods to customize its behavior:

```
Validate
    foreach EntityClass in specified EntityClasses
        require Faction.GetBuildCount(EntityClass) > 0
            require Faction.GetBuildableClasses(Category) contains EntityClass
                case Unit: return FactionClass.BuildableUnits
                case Terrain: return FactionClass.BuildableTerrains
                case Effect: return empty collection
                case Upgrade: return FactionClass.BuildableUpgrades

            returning count := 999
            foreach buildResource in Faction.GetBuildResources(EntityClass)
                if buildResource > 0
                    restrict count to Faction.Resource / buildResource

GenerateProgram
    Faction.Build()
        buildResources := empty collection
        foreach EntityClass in specified EntityClasses
            buildResources += Faction.GetBuildResources(EntityClass)
                return EntityClass.BuildResources
            entity := instruction CreateEntity(EntityClass)
            instruction SetEntityOwner(entity, this)
            if Category is Unit
                instruction SetEntityUniqueName(entity)

        foreach buildResource in buildResources
        where Faction.Resources contain buildResource
            value := Faction.Resource – buildResource
            instruction SetFactionVariable(Resource, value)
```

Figure 40: Build Program

— Override Faction.Build to perform additional actions when building an entity.

— Override Faction.GetBuildCount to adjust the maximum number of entities of a given class that the faction can build. This count is also checked by the human player GUI and by computer player algorithms before issuing a Build command.

— Override Faction.GetBuildResources to change the types or amounts of resources that the faction must expend to build one entity of the specified class. Resources that have no entry (not even a zero entry) in the faction's resource collection are ignored, i.e. the entity is "free" with regards to those resources.

You should normally rely on Hexkit Editor to define any buildable classes and build resources. Override GetBuildCount and GetBuildResources only if you need to dynamically adjust this data in response to certain game events, for example to make specific units available.

7.4.5 Destroy Command

Figure 41 shows the default implementation of the Destroy command, described in "Destroy Entities" on page 26. Override the following methods to customize its behavior:

```
Validate
    foreach Entity in specified Entities
        require EntityClass.CanDestroy

GenerateProgram
    Faction.Destroy()
        destroyResources := empty collection
        foreach Entity in specified Entities
            destroyResources += Entity.GetDestroyResources()
                resources := Faction.GetBuildResources(EntityClass)
                    return EntityClass.BuildResources
                return resources divided by five
            Entity.Delete()

        foreach destroyResource in destroyResources
            value := Faction.Resource + destroyResource
            instruction SetFactionVariable(Resource, value)
```

Figure 41: Destroy Program

— Override Faction.Destroy to add non-resource effects to the destruction of an entity, such as the creation of deletion of another entity.

— Override Entity.GetDestroyResources to adjust the resources returned to the owning faction when an entity is destroyed.

All entities specified in a Destroy command must have the Destroy ability. This is always the case for units. Non-background terrains can be made destructible in Hexkit Editor.

The virtual method Faction.GetBuildResources is primarily intended for the Build command and discussed in that context (see "Build Command" on page 125). The virtual method Entity.Delete is described in "Deleting Factions and Entities" on page 113.

7.4.6 End Turn Command

Figure 42 shows the default implementation of the End Turn command, described in "End Turn" on page 26. Override the following methods to customize its behavior:

```
Validate
    require specified Faction = WorldState.ActiveFaction

GenerateProgram
    Faction.EndTurn()
        foreach Faction in WorldState
            Faction.UpdateResources(type: resetting)
        instruction SetFactionUnitsCanAttack(all, true)
        instruction SetFactionUnitsCanMove(all, true)

    instruction AdvanceFaction
    WorldState.CheckDefeat()
        defeated := select each Faction in WorldState
            where Faction.IsResigned or Faction.CheckDefeat()
                if any FactionClass.DefeatCondition.Threshold met
                or any Faction.Resource <= ResourceClass.Defeat
                    instruction ShowMessageDialog
                    succeed

        foreach Faction in defeated
            Faction.Delete()
        if no more Faction in WorldState
            instruction SetWinningFaction(none)
```

Figure 42: EndTurn Program

— Override Faction.EndTurn to perform additional actions at the end of each turn. You should not attempt to change the resource update logic in this method – change the variable definitions in Hexkit Editor instead.

— Override Faction.CheckDefeat to check for additional or different defeat conditions than those defined by the scenario, or to change the generated message event.

The End Turn command also calls some of the methods described in "Attributes and Resources" on page 116, and possibly those in "Deleting Factions and Entities" on page 113.

7.4.7 Move Command

Figure 43 shows the default implementation of the Move command, described in "Move Units" on page 27. Movement relies on the pathfinding algorithms discussed in chapter "Pathfinding" on page 132. They are usually run by the helper methods shown in Figure 44 which are also called when players attempt to determine possible movement targets.

Override the following methods to customize the Move command:

— Override Unit.Move to perform additional actions when moving units, or to leave the Move ability enabled.

— Override Unit.CreateMoveEvent to change the map view event that illustrates a Move command.

```
Validate
    foreach Entity in specified Entities
        require Entity.Category = Unit
        require valid Entity.Site
    targets := Finder.FindMoveTargets(specified Entities)
    require targets contain specified Target

GenerateProgram
    Unit.Move()
        foreach Unit in specified Units
            instruction SetEntitySite(Unit, null)

        Unit.CreateMoveEvent(original Unit.Site, specified Target)
            path := Finder.FindMovePath(Units, Source, Target)
            class := topmost Unit.DisplayClass
            instruction MoveImage(class, path)

        foreach Unit in specified Units
            instruction SetEntitySite(Unit, specified Target)
            instruction SetUnitCanMove(Unit, false)
```

Figure 43: Move Program

```
Finder.FindMovePath(Units, Source, Target, isAttacking)
    agent := new UnitAgent(Units, Source, isAttacking)
    if AStar.FindBestPath(agent, Source, Target) succeeds
        return AStar as IGraphPath
    return null

Finder.FindMovePath(Units, Source, Targets, isAttacking)
    agent := new UnitAgent(Units, Source, isAttacking)
    foreach Target in Targets by increasing distance from Source
        if AStar.FindBestPath(agent, Source, Target) succeeds
            return AStar as IGraphPath
    return null

Finder.FindMoveTargets(Units)
    if any Unit in Units without Unit.CanMove
        return empty collection
    agent := new UnitAgent(Units)
    movement := minimum Unit.Movement of all Units
    return Coverage.FindReachable(agent, Units.Site, Units)
```

Figure 44: Move Target Programs

— Override the various Unit methods that are called by the UnitAgent class to define the actual movement system. See "Customizing Pathfinding" on page 136 for details.

The arrival of moving units on the target site might cause other changes, such as capturing the site. These actions are not encoded in the movement methods proper but in the Unit.OnSiteChanged handler, described in "Acting on Data Changes" on page 114. This event is triggered twice by Unit.Move: once for removing the units from their original site, and then a second time for placing them on the target site.

The helper method Finder.GetMoveTargets finds all map locations that the specified units can reach within one turn. The search range is restricted to a number of movement steps that equals the smallest Movement value of any moving unit because every step has a minimum cost of one, as required by the A* algorithm (see "Customizing Pathfinding" on page 136).

7.4.8 Place Command

Figure 45 shows the default implementation of the Place command, described in "Place Entities" on page 27. The fairly complex validation process relies on the target finder methods shown in Figure 46 which are also used when players attempt to determine possible placement locations.

```
Validate
        foreach Entity in specified Entities
                require invalid Entity.Site
        foreach PlaceTargets in Finder.FindAllPlaceTargets(specified Entities)
                require PlaceTargets contain specified Target
        if number of specified Entities > 1
                require Faction.CanPlace(specified Entities, specified Target)
                        return Faction.CanPlace(EntityClasses, Site)
                                succeed if Site.Owner = this
                                        and any Site.Units have Unit.Owner = this

GenerateProgram
        Entity.Place()
                foreach Entity in specified Entities
                        instruction SetEntitySite(Entity, specified Target)

Unit.Place()
        foreach Unit in specified Entities
                instruction SetUnitCanAttack(Unit, false)
                instruction SetUnitCanMove(Unit, false)
```

Figure 45: Place Program

Override the following methods to customize the Place command:

— Override Entity.Place to perform additional actions when placing entities.

— Override Unit.Place to perform additional actions when placing units, or to leave the Attack and Move abilities enabled.

— Override Faction.GetPlaceTargets to determine the set of valid placement locations for a single hypothetical entity of the specified class. You must always filter the results by Faction.CanPlace to ensure they are immediately valid for single entities.

— Override Faction.CanPlace to impose restrictions on placing one or more hypothetical entities on the same concrete site. The *location* of the specified site is guaranteed to be valid for all individual classes, so CanPlace should check its *contents*, and whether a specified group of more than one entity could be placed simultaneously.

The Place command requires that the target site is a GetPlaceTargets element for every single class of the placed entities, *and* that CanPlace succeeds for the combined group of entities. The result is that combining multiple entities in a single Place command can only restrict, never extend, the location sets defined for the participating entity classes.

```
Faction.GetPlaceTargets(EntityClass)
        returning targets := EntityClass.PlaceSites
        if EntityClass.UseDefaultPlace
                if first turn
                        targets += Faction.Sites
                else if Faction.HomeSite valid
                        targets += Faction.HomeSite
        remove target from targets
                where not Faction.CanPlace(EntityClass, target)

Finder.FindAllPlaceTargets(Faction, Category)
        returning targets := empty collection
        classes := Faction.GetAvailableClasses(Category)
                return Faction.GetBuildableClasses(Category)+Faction.GetEntities(Category)
        foreach EntityClass in classes
                targets += (EntityClass, Faction.GetPlaceTargets(EntityClass))

Finder.FindAllPlaceTargets(Faction, Entities)
        returning targets := empty collection
        foreach unique EntityClass in Entities
                targets += (EntityClass, Faction.GetPlaceTargets(EntityClass))
```

Figure 46: Place Target Programs

The default implementation of CanPlace is as quite liberal and only excludes sites that contain enemy units, which is an invariant required by Hexkit, and unowned sites. You might want to lift the latter restriction for special units such as paratroopers. On the other hand, you might want to impose additional restrictions such as a unit stacking limit.

Since placing an entity automatically invokes Entity.OnSiteChanged (see "Acting on Data Changes" on page 114), placing a unit can also capture the site or perform other actions.

Note. Placement targets are based on entity classes rather than entities because computer players must know if and where an entity can be placed before they build it – otherwise they would waste a lot of resources on building unplaceable entities! This also means that entities of the same class always have identical placement sites within the same world state.

7.4.9 Rename Command

Figure 47 shows the default implementation of the Rename command, described in "Rename Entities" on page 28. This command does not allow any customization whatsoever, and is listed here only for completeness.

```
GenerateProgram
        foreach Entity in specified Entities
                instruction RenameEntity(specified EntityName)
```

Figure 47: Rename Program

7.4.10 Resign Command

Figure 48 shows the default implementation of the Resign command, described in "Resign Game" on page 28. Override the following method to customize its behavior:

```
GenerateProgram
    Faction.Resign()
        instruction SetFactionResigned(true)
        instruction ShowMessageDialog
```

Figure 48: Resign Program

— Override Faction.Resign to change the generated message event or to prevent specific factions from resigning the game (as in the "Roman Empire" scenario).

The default implementation is extremely simple because the Resign command itself merely sets the Faction.IsResigned flag. The resigned faction is not actually deleted until the next End Turn command (see "End Turn Command" on page 127).

Chapter 8: Pathfinding

Moving units around is the most common action in a Hexkit scenario, and aside from combat it is also the most complex and customizable task in the game. This chapter describes the internal map geometry on which pathfinding and other graph algorithms operate, and how they can be controlled in a rule script.

8.1 Pathfinding Classes

Hexkit relies on the general-purpose pathfinding algorithms provided by the Tektosyne library. These algorithms operate on interfaces which are implemented by other Tektosyne and Hexkit classes that represent the actual game data. Figure 49 shows the types and their relationships.

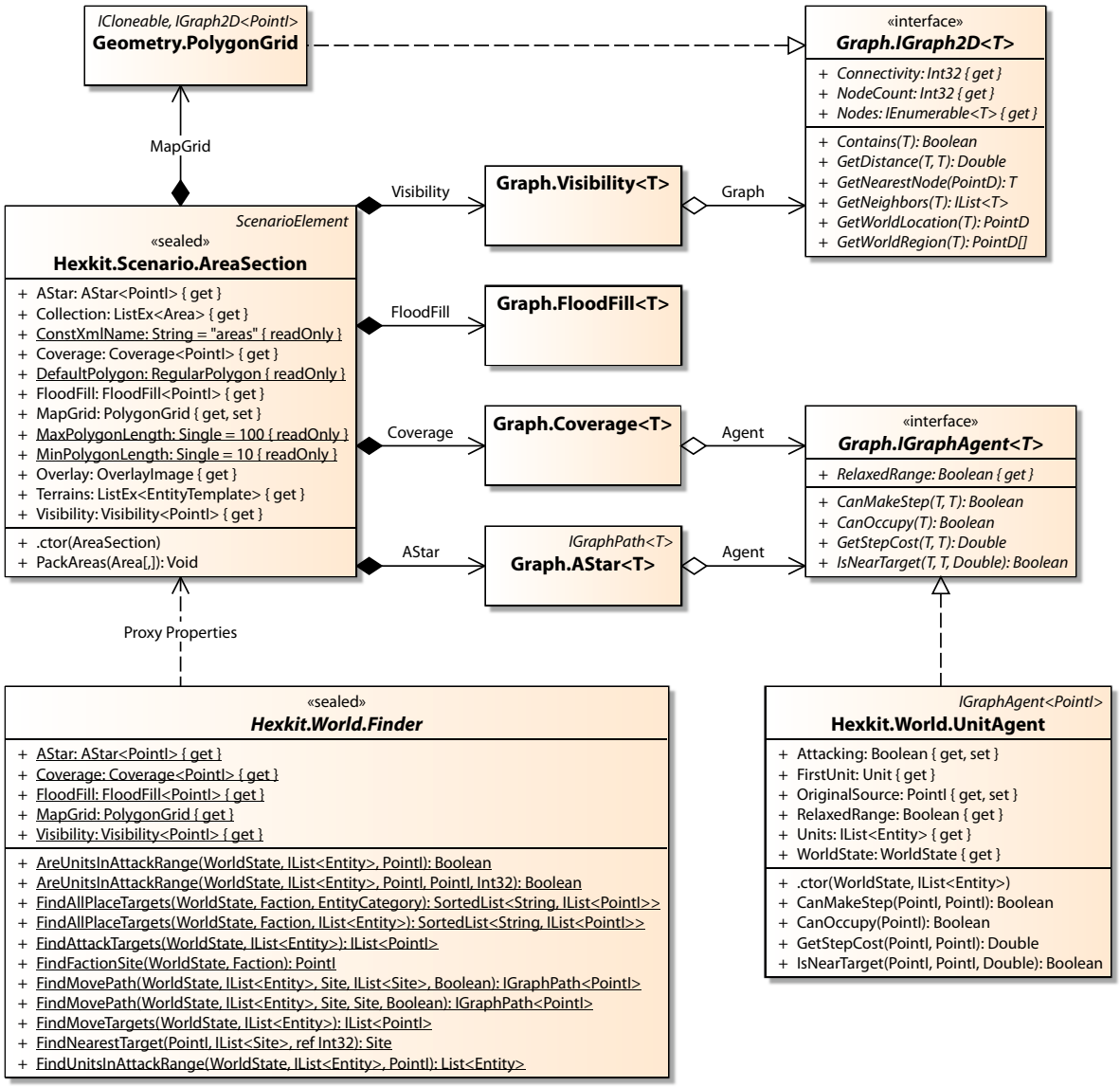


Figure 49: Pathfinding Classes

The following discussion covers the types in the Tektosyne.Graph namespace only insofar as they relate to Hexkit types. Please refer to the *Tektosyne User's Guide* for a detailed description of the Tektosyne graph interfaces and algorithms.

8.1.1 Tektosyne Types

All Tektosyne graph algorithms require an implementation of IGraph2D<T>, representing the graph itself. The A* pathfinding and path coverage algorithms also require an implementation of IGraphAgent<T>, representing some agent that moves across the graph.

In Hexkit, the graph is always the current MapGrid defined by the Scenario.AreaSection class. This is an instance of the PolygonGrid class, described in "Geometry Classes" on page 134. For space reasons, only the Visibility algorithm shows a Graph reference in Figure 49, but the other three algorithms likewise refer to PolygonGrid via IGraph2D<T>.

The moving agents are individual units or stacks of units. The class World.UnitAgent, briefly mentioned in "The Pathfinding Agent" on page 84, maps IGraphAgent<T> members to virtual Unit methods. The exact mapping is described in "Customizing Pathfinding" on page 136.

The Tektosyne library provides four graph algorithms: AStar, Coverage, FloodFill, Visibility. The class Scenario.AreaSection maintains an instance of each algorithm for the current MapGrid. Note that the FloodFill algorithm is not currently used by Hexkit Game, but Hexkit Editor runs FloodFill to replace the contents of all identical adjacent map sites.

8.1.2 Finder Class

The World.Finder class provides proxies for MapGrid and the four algorithm instances. Rule script authors should use these proxies since they are considerably shorter than the AreaSection names, e.g. Finder.AStar instead of MasterSection.Instance.Areas.AStar.

Finder also contains a number of helper methods that find paths and other (collections of) map locations. These methods are used by command methods and computer player algorithms, and might come in handy for rule scripts as well.

AreUnitsInAttackRange — Determines whether the specified units are in range for an Attack command on the specified target location, given their actual current locations or the specified hypothetical source location.

AreUnitsInAttackRange compares each unit's AttackRange to its distance from the target location, and also runs the Visibility algorithm to check that a clear line of sight exists for any ranged attackers that require one.

FindAllPlaceTargets — Finds all valid target locations for a Place command with the specified entities, or with entities of any class available to the specified faction. Those are the locations returned by Faction.GetPlaceTargets for the desired entity classes.

FindAttackTargets — Finds all valid target locations for an Attack command performed by the specified units. Those are all locations within the combined AttackRange radii of all units for which Unit.CanAttackTarget succeeds.

FindFactionSite — Finds the location on which the map view is centered when the specified faction becomes active, as described in "Home Site and Alternatives" on page 15.

FindMovePath — Finds the best path for a Move command performed by the specified units, with the specified source and target locations. FindMovePath runs the AStar algorithm with an automatically created UnitAgent.

FindMoveTargets — Finds all valid target locations for a Move command performed by the specified units, starting at their current location. FindMoveTargets runs the Coverage algorithm with an automatically created UnitAgent.

FindNearestTarget — Finds the map site, among a list of possible choices, that is nearest to the specified source location, in terms of movement steps.

FindUnitsInAttackRange — Finds the units, among a list of possible choices, that can perform individual Attack commands on the specified target location. Those are the units for which FindAttackTargets returns collections that contain the target location.

Section "Command Algorithms" on page 120 shows the pseudocode programs for those Finder methods that are used in conjunction with specific game commands.

8.2 Geometry Classes

The IGraph2D<T> interface is implemented by the PolygonGrid class, representing a rectangular grid of regular polygons. Each node in the graph is a PointI instance that represents the grid location of a polygon, connected to all adjacent polygons in the grid as its immediate neighbors. The shape and orientation of a polygon is defined by the RegularPolygon class. Figure 50 shows all related types in the Tektosyne.Geometry namespace.

In Hexkit terms, the PolygonGrid instance stored in AreaSection.MapGrid represents the game map, and each polygon in the MapGrid represents one map site. Most of the numerous members of PolygonGrid and RegularPolygon handle the translation between graph (= map) coordinates and world (= display) coordinates. They are used internally by the Hexkit graphics engine. We'll focus instead on the few members that are relevant to rule script authors:

Size — The width and height of the polygon grid, i.e. the number of columns and rows.

CreateArray and CreateArrayEx — Creates a two-dimensional Array or ArrayEx of the same Size as the polygon grid. Used to create the Hexkit game map, and to associate map sites with markers or counters.

GetNeighbor and GetNeighborIndex — Convert between neighboring graph nodes and their neighbor indices. See "Neighbor Indices" on page 136.

GridShift — A PolygonGridShift value indicating the "shifting" of the second row or column in the polygon grid. AreColumnsShifted and AreRowsShifted indicate whether columns or rows are shifted. See "Map Geometry" on page 37 for details on this subject.

VertexNeighbors — Indicates whether the graph supports vertex neighbors in addition to edge neighbors. See "Edge and Vertex Neighbors" on page 135.

Element — The RegularPolygon that constitutes one graph node or map site.

RegularPolygon.Connectivity — The maximum number of immediate neighbors of each polygon. This is also the value returned by PolygonGrid.Connectivity.

RegularPolygon.Orientation — A PolygonOrientation value indicating whether each polygon is lying on a side (= edge) or standing on a corner (= vertex).

RegularPolygon.Sides — The number of sides in each polygon. This is always either four or six.

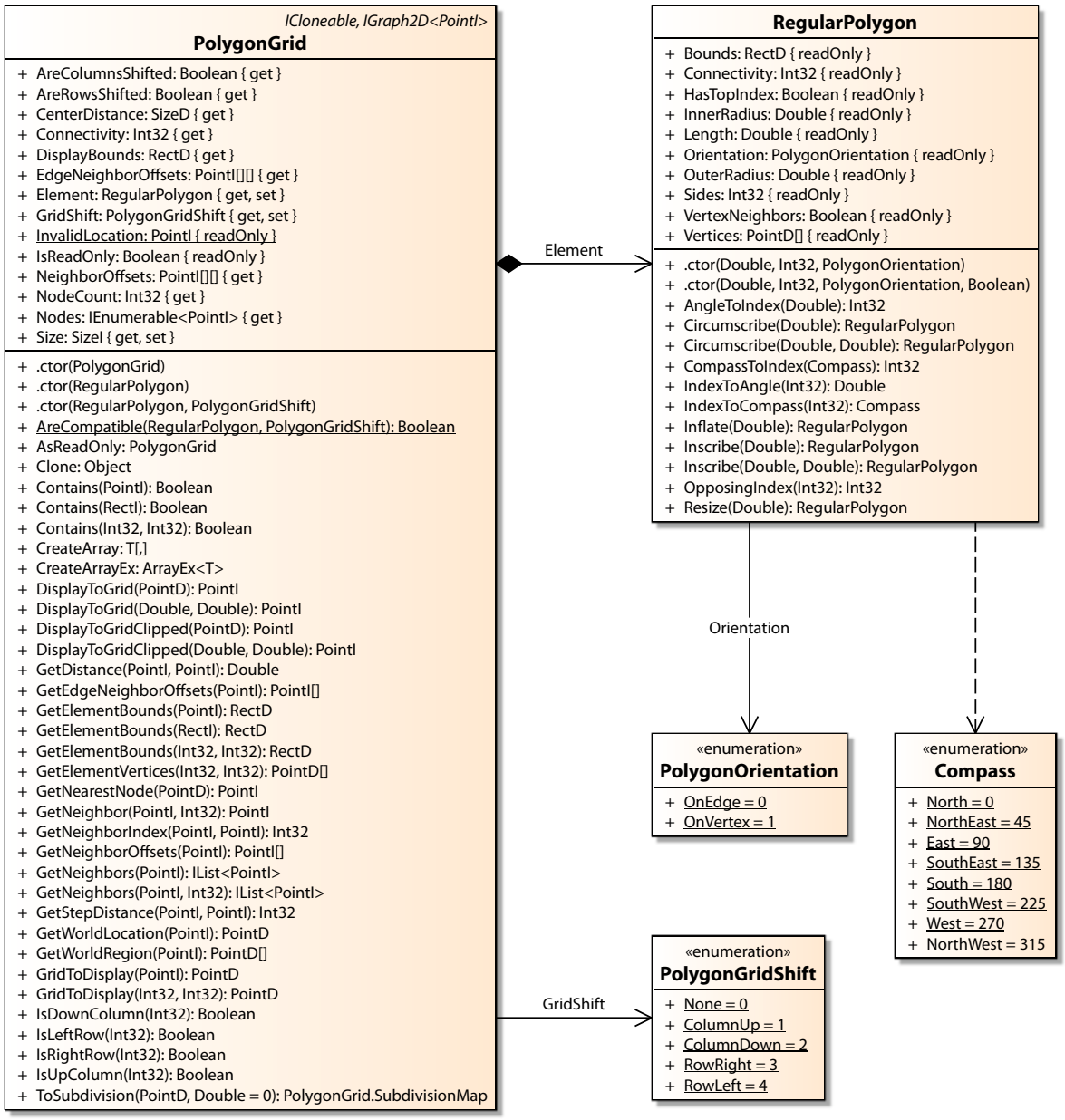


Figure 50: Geometry Classes

8.2.1 Edge and Vertex Neighbors

PolygonGrid supports polygons with four or six sides. This creates some uncertainty regarding the concept of immediate neighbors. Adjacent hexagons always share a common edge; that's what makes them so useful for wargames! But adjacent squares might share a common edge or just a common vertex. We introduce a flag to handle the latter case.

Polygons that share a common edge (*edge neighbors*) are always considered immediate neighbors. However, squares that share a common vertex (*vertex neighbors*) are considered immediate neighbors only if the RegularPolygon.VertexNeighbors flag is set. When true, units can move across a shared vertex in a single movement step; when false, they must take two steps across shared edges to reach the same destination.

8.2.2 Neighbor Indices

Depending on the current topology, each polygon that is not located on the edges of the polygon grid has four, six, or eight immediate neighbors, as shown in "Polygon Connections" on page 43. The total number is reported by the Connectivity property.

Each neighbor transition across an edge or vertex (only if the VertexNeighbors flag is enabled) is identified by a *neighbor index*. The GetNeighbor method takes such an index as a parameter to indicate which of the neighboring map locations to return.

You can use GetNeighborIndex to determine the index of the edge or vertex across which two neighboring sites are connected. Moreover, the associated RegularPolygon offers various members that deal with neighbor indices, floating-point angles, and Compass values:

AngleToIndex — Converts the specified angle to the nearest neighbor index.

CompassToIndex — Converts the specified Compass value to the nearest neighbor index.

IndexToAngle — Converts the specified neighbor index to the corresponding angle.

IndexToCompass — Converts the specified neighbor index to the Compass value nearest to the corresponding angle.

OpposingIndex — Determines the neighbor index opposite to the specified neighbor index.

HasTopIndex — Indicates whether neighbor index zero is located at an angle of zero degrees.

8.3 Customizing Pathfinding

The AStar and Coverage algorithms rely on several virtual Unit methods called by the UnitAgent class, and also directly expose a few options to rule script code. Moreover, the Visibility algorithm usually inspects an Entity property that can be set in Hexkit Editor. This section describes how to use these mechanisms to customize the algorithms' behavior.

Table 11 shows how the various UnitAgent members map to virtual Unit methods. Most of them are documented here, except for Unit.CanAttackTarget which is called only when planning an Attack command and thus described in "Attack Command" on page 120.

UnitAgent Member	Pseudocode Program
RelaxedRange	*Unit.GetRelaxedRange()* return false
CanMakeStep(Source, Target)	*Unit.CanMakeStep()* return Target.Difficulty > 0 and Unit.Owner = Owner for all Target.Units
CanOccupy(Target)	*Unit.CanOccupy()* return true
GetStepCost(Source, Target)	*Unit.GetStepCost()* return maximum of 1 and Target.Difficulty

Table 11: Unit Agent Programs

UnitAgent Member	Pseudocode Program

IsNearTarget(Source, Target)	if UnitAgent.Attacking
	  return *Unit.CanAttackTarget()*
	else
	  return Source = Target

Table 11: Unit Agent Programs

8.3.1 Limited Search Range

Set RelativeLimit for scenarios with very large maps, to speed up movement calculations. The "Roman Empire" scenario uses the value 3.0, for example.

This is a global parameter for the entire scenario. In the constructor of your rule script's RulesFactory class, set Finder.AStar.RelativeLimit to the desired value.

8.3.2 World Distance Comparison

Set UseWorldDistance for scenarios that offer many opportunities for strange path variants, especially relatively "empty" maps based on squares, such as in the "Troll Chess" scenario.

This is a global parameter for the entire scenario. In the constructor of your rule script's RulesFactory class, set Finder.AStar.UseWorldDistance to the desired value.

8.3.3 Transient vs. Permanent Occupation

UnitAgent forwards CanMakeStep and CanOccupy to the virtual Unit methods of the same name.

The default implementation of CanMakeStep fails if the Site.Difficulty of the target site is zero, indicating impassable terrain, or if the target site contains any units owned by another faction.

The default implementation of CanOccupy always succeeds. The Hexkit default rules do not place any additional restrictions on permanent as opposed to transient occupation.

Override CanMakeStep in your CustomUnit class to restrict some or all units from leaving the specified source site or from entering the specified target site. For example, the Crécy scenario adds the size of all moving units to that of the units already present in the target site, and fails if the sum exceeds a stacking limit of 800 size points (traffic congestion!).

Override CanOccupy in your CustomUnit class to restrict some or all units from permanently occupying the specified map site. For example, the Crécy scenario allows at most 100 archers to deploy in the same site, even though that number may fall well below the stacking limit imposed on transient occupation (archers must deploy in a thin line to get a clear shot).

You should assume that any target site that is supplied to CanOccupy allows the temporary presence of the specified units. Any related tests in CanMakeStep have already succeeded, so there is no need to recheck such conditions as the presence of enemy units. Test only for new conditions that apply exclusively to permanent occupation.

8.3.4 Movement Costs

UnitAgent forwards GetStepCost calls to the virtual Unit method of the same name. The default implementation returns the Site.Difficulty of the target site, if that value is positive; otherwise, the constant value one. A site's Difficulty defaults to the summed Difficulty values of all local terrains.

The total cost of a movement path is then compared to the smallest Movement value of any moving unit to determine their actual movement range. Terrain.Difficulty and Unit.Movement are

virtual "standard variables" that can be mapped to concrete variables either using Hexkit Editor settings, or by overriding their property getters in a rule script. See "Standard Variables" on page 119 for details on this subject.

Override GetStepCost in your CustomUnit class to define a different movement cost. Typically you'll start with the Difficulty of the target site, and then adjust the cost depending on the moving units and/or the specified source and target sites. Remember that the final cost must be positive.

Note. You cannot dynamically change a unit's total movement *allowance* based on the context of a specific Move command. Instead, you must override GetStepCost and adjust the *cost* of each movement step based on the specified parameters.

A unit's Movement value should reflect its maximum possible movement range, under any circumstances. Units with a Movement of zero are considered immobile and cannot participate in a Move command, although the rule script may reposition them programmatically.

Note. Units such as paratroopers or teleporters are difficult to realize in this system. You must set their Movement values to the greatest possible distance they can cover in a single movement, and then override GetStepCost to return either the constant value one or the regular step cost, depending on whether the unit is performing its special "jump" or a normal movement. You might even need to increase the regular step cost if the jump range is too great. This is all very inconvenient, and a future Hexkit version should offer a new game command that bypasses the pathfinding mechanism entirely and directly repositions such units.

8.3.5 Relaxed Range

UnitAgent initializes RelaxedRange to the result of Unit.GetRelaxedRange when the UnitAgent is created, i.e. at the start of a pathfinding operation.

Override GetRelaxedRange in your CustomUnit class to extend the range of the moving units. You can return a constant value to use the same option for all Move commands, or you can return different values depending on the specified combination of moving units.

8.3.6 Lines of Sight

The Visibility algorithm can be run with an arbitrary predicate to determine which graph nodes block the line of sight to nodes behind them, but in practice this algorithm is usually run by the helper method Finder.AreUnitsInAttackRange which inspects the Site.BlocksAttack property of every visited map site. This property in turn inspects the Entity.BlocksAttack properties of all local entities (of any category) and succeeds exactly if any of them returns true.

Use Hexkit Editor to set the Blocks Attack option of any entity class (usually terrains and units) whose instances should block the line of sight for ranged attacks. There is currently no way to further customize line-of-sight calculations in the rule script, although you can always run the Visibility algorithm yourself and supply any predicate you desire. If you do so, however, note that the maximum search distance must be specified in world coordinates, not step counts!

You can also change the threshold at which a site is considered visible, in terms of the visible fraction of its tangential arc. This is a global parameter for the entire scenario. In the constructor of your rule script's RulesFactory class, set Finder.Visibility.Threshold to the desired value.

Chapter 9: Computer Players

This chapter documents the various computer player algorithms (also known as "artificial intelligence" or AI) and the related infrastructure provided by Hexkit Game.

Creating a computer player for a strategy game construction kit is a rather unique challenge as many basic assumptions that could otherwise be hard-coded into the AI algorithms are simply not available, or must be inferred from the current scenario and world state, or acquired from dedicated rule script properties and methods.

For this reason, and also to provide easy-to-use "building blocks," all computer player algorithms rely on a number of general support algorithms that are built into Hexkit. Some of these algorithms aggregate and simplify the customized data and behavior that is stored in scenarios and rule scripts, while others provide "canned" actions that should be at least somewhat useful in any Hexkit scenario.

We start with the "infrastructure" provided by the various support algorithms, highlighting opportunities for customization via Hexkit Editor or the rule script, and then move on to the computer player algorithms built on top of this infrastructure.

Namespace Conventions. To improve legibility, namespaces are only explicitly stated when introducing a new type, or if eponymous types exist in different namespaces.

All Hexkit types reside in the outer namespace Hexkit which is always omitted. Moreover, all computer player algorithms are understood to reside in the Hexkit.Players namespace.

9.1 Conditional Scripting

The ultimate goal of our computer player algorithms is to generate dynamic behavior based on the current world state, without any programmatic aid by the scenario designer. However, such aid may sometimes be desirable to help the computer play a stronger game, or else the scenario designer may wish to restrain the computer player for the sake of historical accuracy.

In such cases we resort to *scripted behavior*, the poor cousin of true "artificial intelligence" as it were, but useful nonetheless. Since Hexkit already supports rule customization via script code it's a simple thing to programmatically execute game commands as well. "Command Automation" on page 105 describes how to unconditionally perform scripted actions that should apply to any player as part of the scenario rules, such as units routing when their morale is low.

But what if you wish to perform certain actions only when the active faction is controlled by a computer player? That's where the Faction.PlayerSettings property comes in. Its value reflects the current settings of the faction's controlling player, insofar as they are relevant to scripting. You can use Hexkit Editor and the Player Setup dialog to change these settings.

Query the PlayerSettings property in your rule script code to perform conditional scripting. The returned object contains the following boolean flags:

IsComputer — Indicates whether the faction is currently controlled by a computer player.

UseScripting — Indicates whether the faction is currently controlled by a computer player *and* should use any optional scripting provided by the scenario. The default is true for any faction under computer player control.

The intended purpose of the UseScripting flag is to switch between historical and optimal actions in historical scenarios. Your rule script should enforce historically accurate behavior when this flag is set, and let the computer player operate without such restraints when it is cleared. See the Crécy rule script for an example of such scripting. Conversely, you should check the IsComputer flag to enforce behavior that should be scripted under any circumstances, as long as the faction is controlled by a computer player.

Note. Remember that the local human player may access the Player Setup dialog and change its settings at any time during a game. Therefore, you must not buffer any PlayerSettings beyond the scope of the method in which you retrieved them. Moreover, factions are not assigned to players until the first Begin Turn command, so you cannot use PlayerSettings within the constructor or the Initialize and CreateFaction methods of your IRulesFactory instance.

9.2 Evaluating Possessions

Every actual and potential possession of a player faction is associated with a *valuation* that indicates its desirability to computer players. Possessions include entities and map sites. Valuations are also assigned to the entity classes that a faction can instantiate.

Valuations are non-negative floating-point numbers, usually within the standard interval $[0,1]$ but possibly greater than one. Their exact meaning depends on the type of possession:

Entity Classes — The desirability of building a new entity of that class.

Entities — The desirability of protecting an entity that the evaluating faction owns, and of acquiring or destroying an entity that it does not own.

Map Sites — The desirability of defending a map site that the evaluating faction owns, and of capturing a site that it does not own.

When a computer player examines a set of possessions for prioritization, the one with the highest valuation is interpreted as the most desirable. Worthless possessions should always have a valuation of zero so that computer players will not waste any effort on them.

However, the set of possessions that is actually subject to such examination depends on the the rule script, the computer player algorithm in use, and the concrete game situation. The total set of possessions that are potential agents or targets for some action is almost certainly going to be pruned significantly by other considerations before valuations are considered – if at all.

Figure 51 shows the World and Scenario classes that participate in the evaluation of possessions. Their interaction is described in the following subsections.

9.2.1 Context-Free Valuation

All possessions provide *context-free valuations* that can be determined solely by examining the possession itself, without referring to a specific faction or world state.

Entity Classes — Every entity class has a constant valuation in the standard interval $[0,1]$ that defaults to zero. The scenario may define higher valuations for specific classes.

Entities — Every entity receives the valuation of its underlying entity class, multiplied by the relative magnitude of all current resources compared to their initial values, if any. The resulting value is capped at one. For example, a unit with that has 9 hit points left out of 10 will receive a valuation that equals 90% of its unit class valuation.

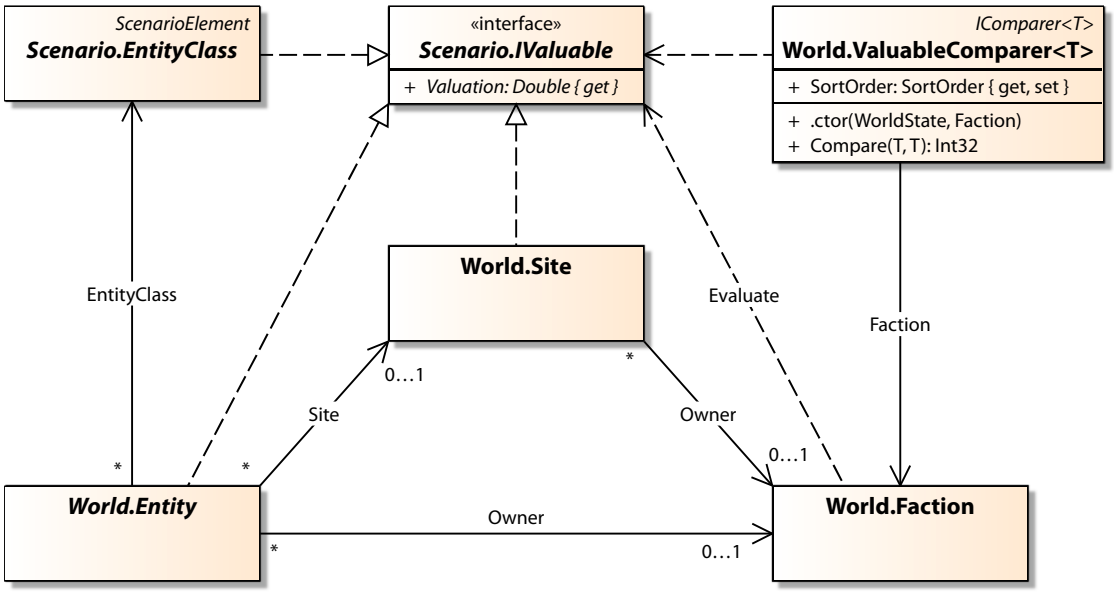


Figure 51: Valuable Classes

Map Sites — Every map site receives a valuation that equals the sum of the valuations of all terrains on that site. This value may be greater than one.

Terrain valuations are cached since terrain stacks rarely change. Please refer to "Terrain Value Caching" on page 76 for details on this mechanism.

The current sum of all context-free valuations for a faction's entire territory and army is shown by the Site Values and Unit Values selections, respectively, of the Info → Faction Ranking dialog. Entity class valuations are not visible in Hexkit Game.

Implementation

All Hexkit types that represent valuable possessions implement the Scenario.IValuable interface. This interface defines the Valuation property which returns the context-free valuation of the possession as a 64-bit floating-point value.

Figure 52 shows the pseudocode programs for entity and site valuations. Use Hexkit Editor to define the context-free valuation of an entity class. The context-free valuations of entities and map sites are based on these entity class valuations but cannot themselves be customized.

```
Entity.Valuation
    valuation := EntityClass.Valuation
    foreach Resource in Resources where Value <> InitialValue
        valuation *= (Value – Minimum) / (InitialValue – Minimum)
    return min(1.0, valuation)

Site.Valuation
    return sum of Terrain.Valuation for all Site.Terrains

WorldState.MaxSiteValuation
    return maximum Site.Valuation for all WorldState.Sites
```

Figure 52: Basic Valuation Programs

The property WorldState.MaxSiteValuation returns the maximum valuation of any map site and can be used to scale individual site valuations (which may be greater than one) to a standard

interval. Individual and maximum map site valuations are buffered for repeated access, using the mechanism described in "Terrain Value Caching" on page 76.

9.2.2 Contextual Valuation

Context-free valuations, taken by themselves, are not terribly useful in practice. A map site that produces victory points is certainly more valuable than one that produces timber; an enemy unit that occupies a placement site must be destroyed more urgently than a more expensive one that is in no position to do any harm; etc.

Unfortunately, accurate *contextual valuations* depend on the semantics of entity properties and map locations, and semantics are not part of the formal scenario description. The default implementations work well enough for simple scenarios but you should consider overriding them in a rule script to calculate more appropriate contextual valuations.

Entity Classes — By default, the contextual valuation equals the context-free valuation if the evaluating faction does not own any entities of the same category.

Otherwise, the valuation is twice the CFV minus the share of owned instances of the class among all owned entities of the same category (see below).

Entities — The contextual valuation equals the context-free valuation by default. Note that this valuation already takes the entity's current state into account.

Map Sites — By default, the contextual valuation equals 1.0 for the evaluating faction's home site, 0.9 for another faction's home site, and otherwise the context-free valuation scaled to the interval [0, 0.8].

Entity Classes

The default contextual valuation for entity classes interprets the context-free valuation (CFV) of an entity class as an indicator of its *most desirable share* among the entities of the same category owned by the evaluating faction.

More precisely, the most desirable share is the ratio between the class's CFV and the sum of the CFVs of all buildable classes of the same category. If there is only one buildable class, its desired share is 100% regardless of its actual CFV. If there are two classes with an equal CFV, the desired share of either class is 50%, and so on.

If the faction owns relatively fewer or more entities based on a specific class than is indicated by its most desirable share, the class's contextual valuation is adjusted upward or downward in proportion to the deviation. The valuation is reduced to a minimum of zero if there are at least twice as many instances as desired. Conversely, the valuation is doubled if the faction does not own a single instance of the class.

This adjustment compels the computer player to build more or fewer entities of that class, respectively, so that their distribution gravitates back towards the most desirable share of each class in the long run.

Implementation

Contextual evaluation is performed by the set of Evaluate methods shown in Figure 53. Since the context of a player faction is required, all methods are defined by the Faction class.

The overload that takes an IValuable parameter simply forwards to one of the other overloads, depending on the concrete type of its argument. Override any or all of the three other overloads to adjust the contextual valuation of the corresponding game object.

```
Faction.Evaluate(IValuable)
        return Evaluate(concrete type of IValuable)

Faction.Evaluate(Entity)
        return Entity.Valuation

Faction.Evaluate(EntityClass)
        valuation := EntityClass.Valuation
        if valuation = 0.0 return 0.0

        buildableClasses := Faction.GetBuildableClasses(EntityClass.Category)
        if not buildableClasses contain EntityClass
                return valuationvaluation /= sum of all EntityClass.Valuation
                where EntityClass in buildableClasses

        entities := Faction.GetEntities(EntityClass.Category)
        specifiedCount := count of Entity in entities
                where Entity.EntityClass = specified EntityClass
        buildableCount := count of Entity in entities
                where buildableClasses contains Entity.EntityClass

        valuation := 2 * valuation – specifiedCount / buildableCount
        return valution restricted to [0.0, 1.0]

Faction.Evaluate(Site)
        if WorldState.MaxSiteValuation = 0.0
                return 0.0
        else if Site = HomeSite
                return 1.0
        else if Site = Faction.HomeSite where Faction <> this
                return 0.9
        else
                return 0.8 * Site.Valuation / WorldState.MaxSiteValuation
```

Figure 53: Faction Valuation Programs

To simplify sorting by valuation, World.ValuableComparer provides an IComparer that sorts IValuable instances by their Faction.Evaluate results. For each comparison, the Compare method invokes Evaluate on the Faction and with the WorldState that were supplied to the constructor of the ValuableComparer instance.

9.2.3 Selecting by Valuation

Valuation provides a simple and general way to select between two possessions, although this is usually a method of last resort when no better criteria are available.

SelectValuable attempts to choose between two IValuable objects first by contextual valuation, then by context-free valuation, and finally by a random roll of the dice. Selection thus proceeds by decreasing quality if a better method does not give conclusive results.

```
AlgorithmGeneral.SelectValuable(IValuable x, y)
        select by greater Faction.Evaluate(IValuable) result
        if equal select by greater IValuable.Valuation
        if equal select by 50% random chance
```

9.2.4 Threat Assessment

Units that belong to enemy factions are threats rather than assets. From our perspective, their valuation determines how much of a threat they pose to our possessions.

Given a list of potential targets of enemy attacks, which one is currently facing the gravest threat? EvaluateThreats provides a simple heuristic that considers the valuation of all enemy units on the map, weighted by each unit's proximity to each potential target.

```
AlgorithmGeneral.EvaluateThreats(Targets)
    foreach Target in Targets
        threat(Target) := 0.0
        foreach Site in WorldState.Sites
            distance := 1 + MapGrid.GetDistance(Site, Target)
            foreach Unit in Site.Units where Unit.Owner <> this
                threat(Target) += Faction.Evaluate(Unit) / distance
```

When overriding Faction.Evaluate(WorldState, Entity), check the owner of a specified unit to determine whether you are evaluating friendly or enemy units. Evaluate enemy units to estimate their threat level, i.e. their priority as attack targets for your own units.

9.3 Target Selection

Selecting a target site for an Attack or Move command often involves choosing between several potential targets by their respective range from the active units' current position.

In this section, we define a generalized terminology for target ranges that is suitable for either command, and describe the process of selecting targets by range.

9.3.1 Reaching a Target

First we define what it means for a unit to have "reached" a target location. This statement has different implications depending on the target type we're talking about.

Attack Targets — A unit has "reached" an attack target if it occupies a site from which it can attack the target. There are usually multiple such locations.

Move Targets — A unit has "reached" a movement target if it directly occupies the target site.

Note. In the current Hexkit version, attack targets always contain enemy units. This means that a unit can never directly occupy an attack target, and that the locations from which it can attack the target never include the target site itself.

9.3.2 Range Categories

Now that we know what it means to "reach" a target site, we can define *range categories* that are valid for Attack and Move commands alike.

Short Range — Targets that a unit has already reached.

Medium Range — Targets that a unit can reach within the current turn.

Long Range — Targets that the unit can reach no earlier than the next turn.

The enumeration World.RangeCategory mirrors these categories.

Note. When considering attack targets, it is understood that a unit can attack short and medium range targets during the current turn. This implies that the unit must still possess its Attack and Move abilities. Otherwise, range categorization will be distorted.

9.3.3 Comparing Ranges

Once the range categories of two targets are known, we can define a reflexive ordering by range that allows computer players to select the closest target from a list of candidates.

Given a unit and two potential target locations A and B, we say that the unit is closer to A than to B if any of the following conditions hold:

— A is a short-range target, and B is a medium-range or long-range target.

— A is a medium-range target, and B is a long-range target.

— A and B are both medium-range targets, and the distance (in map locations) between the unit's current location and A is less than its distance to B.

— A and B are both long-range targets, and the total cost for the movement path that would allow the unit to reach A is less than the total path cost to reach B.

This leaves the following situations in which the two targets are considered to have the same range from the unit in question:

— A and B are both short-range targets.

— A and B are both medium-range targets, and their distance (in map locations) from the unit's current location is identical.

— A and B are both long-range targets, and the total cost for the movement path that would allow the unit to reach them is identical.

In the case of two medium-range targets, we compare map distances rather than movement path costs to create more "realistic" looking actions, and to prevent units from spreading out too far, in a situation where optimal movement does not matter anyway.

There is currently no method that implements a full range comparison. Instead, the target selection algorithms for the Attack and Move commands test the stated conditions individually, interspersing command-specific conditions according to the desired priority sequence.

9.3.4 Target Limit

In particularly target-rich scenarios, examining all potential targets for a command can consume a great deal of time. To speed up computer player calculations under such circumstances, Hexkit provides a basic option for all computer players to limit the number of examined targets.

This option is accessible in the AlgorithmOptions.TargetLimit property which defaults to eight. You can use Hexkit Editor and the Player Setup dialog to set a different value for each player. The SelectAttackTarget and SelectMoveTarget methods (see below) each take a target limit parameter for which the "Seeker" algorithm passes the current TargetLimit setting.

Naturally, any method that observes the TargetLimit setting is expected to start with the most promising targets so that the quality of the target selection won't degrade too much. We cannot know for sure which targets are promising without actually examining them, but a target's distance from the units trying to reach it should provide a good heuristic. We can likely ignore very distant targets without doing too much damage to our analysis.

9.4 Attack Command

This section describes the support algorithms for the Attack command. Please refer to "Attack Command" on page 120 on how to customize this command.

9.4.1 Comparing Losses

One important criterion for deciding which target to attack are the estimated combat losses for each target, i.e. the expected total loss of Unit.Strength points for each side. Such estimates are computed by Unit.EstimateLosses and can be expressed in two metrics:

Absolute Losses — The absolute total amount of strength points lost on each side.

Relative Losses — The share of lost strength points, compared to the total amount of strength points that each side had before the battle. 100% means complete annihilation.

The CombatResults structure returned by EstimateLosses conveniently packages all related data: pre-battle and post-battle total strength as well as absolute and relative losses for each side.

Figure 54 shows the support methods that process this data. Two methods compare either absolute or relative losses for two CombatResults instances, and a third method chooses between them based on whether a majority of units in the current world state have the Healing ability.

```
AlgorithmGeneral.CombatComparisonAbsolute(CombatResults x, y)
    return (x.DefenderLosses – x.AttackerLosses) –
        (y.DefenderLosses – y.AttackerLosses)

AlgorithmGeneral.CombatComparisonRelative(CombatResults x, y)
    return (x.DefenderPercent – x.AttackerPercent) –
        (y.DefenderPercent – y.AttackerPercent)

AlgorithmGeneral.SelectCombatComparison()
    if Unit.CanHeal for less than half of all WorldState.Units
        return CombatComparisonAbsolute
    else
        return CombatComparisonRelative
```

Figure 54: Losses Comparison Programs

You can use Hexkit Editor to set the Healing ability for any unit class. This does not actually heal such units but merely informs computer player algorithms that they have a way to restore lost strength points during the game. We use this information as a heuristic to determine how to compare combat results:

— If units can be expected to recover their losses we must focus on destroying them as quickly as possible. Thus, we compare relative losses since reducing a unit from one point to zero points – a 100% loss resulting in permanent destruction – is preferable to reducing another from 100 points to 10 points – a 90% loss that may be healed.

— If units can be expected to retain their lowered strength we focus instead on doing as much damage as possible in each battle. Thus, we compare absolute losses.

Currently, the "Seeker" algorithm calls SelectCombatComparison just once, whenever it begins execution for the active faction, and uses the resulting comparison method for all attack targets during the faction's turn. Dynamically choosing a comparison method based on the two targets

that are currently being compared would yield better results in scenarios that have both healing and non-healing units. This is a possible optimization for a future Hexkit version.

9.4.2 Selecting a Target

Simple computer player algorithms attack with individual unit stacks, regardless of the strategic context. Given a full or partial stack of combat-ready units, we walk through a list of potential target sites and determine which one we should attack.

The two variants of this algorithm discussed below are implemented by the two AlgorithmGeneral.SelectAttackTarget overloads.

Stationary Attack

The basic variant is an attack from the attacking units' current location. This algorithm is used with stationary units, or those that have already exhausted their movement capacity.

We examine all targets that are within attack range of all units. The potential target sites are traversed in order of increasing distance from the units' location.

We apply a sequence of prioritized conditions to choose between any two targets:

1. Prefer a target that contains mobile units over one that does not, as indicated by each unit's Unit.IsMobile flag.
2. Prefer the target with the greater estimated difference between enemy and friendly losses, as determined by Unit.EstimateLosses and a specified comparison method.
3. Prefer the target with the higher contextual valuation, according to SelectValuable.

The first condition is a heuristic to ensure that mobile units cannot run away or break through our lines. There is always time to deal with stationary units later on.

Mobile Attack

The advanced variant is an attack from any location that would bring the attacking units within range of a target site. This algorithm is used with mobile units.

We examine all targets that all units could possibly attack, even if it would take one or more Move commands. The potential target sites are traversed in order of increasing distance from the units' location, and classified by range categories (see "Target Selection" on page 144).

We apply a sequence of prioritized conditions to choose between any two targets:

1. Prefer a short-range or medium-range target over a long-range target.
2. Between two long-range targets, prefer the one that can be attacked from a position which can be reached with the lower movement path cost, according to A*.
3. Prefer a target that is found in an optional list of preferred targets over one that is not.
4. Prefer a target that contains mobile units over one that does not, as indicated by each unit's Unit.IsMobile flag.
5. Prefer the target with the greater estimated difference between enemy and friendly losses, as determined by Unit.EstimateLosses and a specified comparison method.
6. Prefer a short-range target over a medium-range target.
7. Between two medium-range targets, prefer the one at the smaller distance from the units' current location.

8. Prefer the target with the higher contextual valuation, according to SelectValuable.

Condition 3 is used to concentrate forces on targets on which an attack is already planned but that require additional attackers to improve the combat odds.

9.4.3 Performing Attacks

Performing an attack is as easy as issuing the appropriate Attack command. We still provide two helper methods for this task in order to simplify the management of *active units*.

We assume that an AI will only consider units which are still active, i. e. which can still attack or move during the current turn, and that all such units are stored in a collection. This collection is passed to the helper methods so that they can remove any units that have become inactive as a result of the attack – either because they have exhausted their actions for the current turn, or because they have been destroyed in combat.

Standard Attack

Given a list of attacking units and a list of active units which contains all attackers:

1. Issue an Attack command with the specified attackers *only*.
2. Remove any units whose IsActive flag is now false from the list of active units.

This algorithm is implemented by AlgorithmGeneral.PerformAttack.

Group Attack

Given a *single* attacking unit (called the "leading" unit) and a list of active units which contains the leading unit:

1. Issue an Attack command with the leading unit and *all* active units that can also attack the specified target from their current locations.
2. Remove any units whose IsActive flag is now false from the list of active units.

This algorithm is implemented by AlgorithmGeneral.PerformGroupAttack.

9.4.4 Conditional Attacks

Computer players planning an attack may consider the estimated outcome, and only perform the attack if the odds are good enough; that is to say, if some absolute or relative condition holds regarding the estimated losses on either or both sides.

Such evaluations allow "smart grouping" of units for an attack, and prevent suicidal or ineffective attacks. Note that grouping is not mandatory; units may attack individually if they cannot suffer any losses, typically because they outrange the defenders.

Note. The conditional attack algorithms receive and manipulate a list of active units, in the same way as the unconditional attack algorithms described above.

Conditional Attack

Given a list of attacking units, a list of active units, and required minimum odds of n, perform a standard attack if both of the following conditions hold:

— The defender's relative losses equal at least $10\% \times n$, or the attacker's relative losses do not exceed 10%. That is, we must either inflict damage as required by the specified odds, or we must at least *not* suffer any substantial damage ourselves.

— The attacker's relative losses equal at most the defender's relative losses divided by n. This condition attempts to ensure a favorable rate of attrition for our units.

Attacker and defender losses are calculated by Unit.EstimateLosses as usual. Since we compare the estimates to required minimum odds rather than absolute numbers, we always look at relative losses rather than absolute losses.

This algorithm is implemented by AIgorithmGeneral.AttemptAttack which calls PerformAttack if the required odds are met, or if the specified n is zero or negative. This means you may specify an invalid value for the required odds to perform an *unconditional* attack.

Conditional Group Attack

Given a *single* attacking unit (called the "leading" unit), a list of active units, and required minimum odds of n, attempt to compose an effective attack group by drawing as many support units as required from the list of active units.

1. Create a list of attacking units that initially contains the leading unit only.
2. Attempt a conditional attack with the current list of attacking units.
   We succeed if an attack could be performed.
3. For each active unit that can also attack the specified target, add the unit to the list of attacking units and attempt another conditional attack.
   We fail if there are no active units left to add.

This algorithm is implemented by AIgorithmGeneral.AttemptGroupAttack which usually calls AttemptAttack repeatedly. However, if the specified n is zero or negative, PerformGroupAttack is called instead to perform an *unconditional* group attack with *all* active units.

Note. We do not compute all possible permutations of attacking units. The initial unit always participates in the attack. Support units are added in the order in which they are specified, and once added they, too, are never removed from the list of attacking units.

9.5 Move Command

This section describes the support algorithms for the Move command. Please refer to "Move Command" on page 127 on how to customize this command.

9.5.1 Calculating Supplies

If a scenario defines unit supply resources for any faction, we expect computer players to take them into account and move depleted units to a resupply location as necessary.

Figure 55 shows the virtual Faction and Unit methods that compute provided and required supplies, respectively. Later we'll see how the AI makes use of this data.

```
Faction.GetSupplyTargets()
    returning targets := empty collection
    foreach Site in WorldState.Sites
        if any Site.Units with Unit.Owner <> this
            Site.SetSupplyResources(null)
        else
            Site.SetSupplyResources(this)
            if any Site.SupplyResources > 0
                targets += Site

Site.SetSupplyResources(Faction)
    Site.SupplyResources := empty collection
    foreach Resource in Faction.SupplyResources
        Site.SupplyResources += Faction.UnitResourceModifiers[Site][Resource]

    foreach Entity in Site.Terrains, Site.Effects where Entity.ResourceTransfer <> None
        foreach Resource in Faction.SupplyResources
            Site.SupplyResources += Entity.Resource.Value

Unit.GetRequiredSupplies()
    returning requiredSupplies := empty collection
    foreach Resource in Owner.SupplyResources
        requiredSupplies += zero supply for current Resource

        if Unit.Resources contains Resource
        and Resource.InitialValue > Resource.Value > 0
            supply := Resource.InitialValue – Resource.Value
            priority := 100 * supply / Resource.InitialValue
```

Figure 55: Supply Programs

Available Supplies

The entire supply mechanism is ultimately based on Faction.SupplyResources. This is a collection of resource identifiers defined in Hexkit Editor. SupplyResources indicates any resources that could be used by any of the faction's units, and that can be resupplied on the map.

To figure out which of these resources are currently available, Faction.GetSupplyTargets calls Site.SetSupplyResources on all map sites that are not blocked by enemy units. That method in turn sets the site's own SupplyResources collection to the sum of all resource modifiers that apply to local units, plus any supply resources contributed by local entities.

The local unit resource modifiers are retrieved from the faction's modifier map, which is a caching mechanism described in "Attributes and Resources" on page 116. The available supply resources of local entities comprise any resources held by terrains and effects that participate in the automatic resource transfer mechanism (ibidem).

To find a potential supply target, we can now examine each site's SupplyResources for positive values. Faction.GetSupplyTargets conveniently returns a collection of all such map sites that were encountered during the calculation.

Note. The Site.SupplyResources collections only apply to the faction on whose behalf they were calculated, and they are lost when a game is saved (see "Data Persistence" on page 111). So when calculating available supplies in your rule script, you must repeat the calculation whenever there is a chance that the active faction has changed, or that the game was saved and reloaded.

Required Supplies

Only units can consume supplies; other entities merely produce them. Unit.GetRequiredSupplies calculates the amount of desired supplies as the difference between its initial and current values for each of the faction's SupplyResources.

That is, we expect a unit to resupply at most to its initial resource levels. If you wish to start a unit with depleted resources you must set the desired maximum as the initial value in Hexkit Editor, and then programmatically lower the current value in the rule script. This is somewhat awkward and might find a better solution in a future Hexkit version.

Unit.GetRequiresSupplies also calculates a *resupply priority* for each deficient resource. This is the desired amount divided by the initial value, expressed in percent. Thus, the priority increases with the deficit and reaches 100% when all of the initial stockpile is gone.

Computer player algorithms usually call GetRequiredSupplies on each unit, and send off a unit to the nearest site with matching SupplyResources if any resource's resupply priority exceeds a given threshold.

Customization

— Use Hexkit Editor to the define the potential supply resources for each faction, and the matching resources and/or unit resource modifiers for any desired entities.

— Use Hexkit Editor set the desired resource transfer mode for any entities that should automatically supply resources from their stockpile to local units.

— Override Faction.GetSupplyTargets to change the calculation of available supply sites, for example to exclude additional locations.

— Override Unit.GetRequiredSupplies to change the calculation of required supply resources, for example to assign different priorities.

9.5.2 Selecting a Target

Simple computer player algorithms move individual unit stacks to occupy desirable locations, regardless of the strategic context. Given a full or partial stack of mobile units, we walk through a list of potential target sites and determine which one we should occupy.

We examine all targets that all units can trace a path to, even if they cannot be reached in one turn. Optionally, the caller may specify a supply resource that the target site should provide. In this case, all targets that cannot supply this resource are ignored.

The potential target sites are traversed in order of increasing distance from the units' current location, and classified by range categories (see "Target Selection" on page 144).

We apply a sequence of prioritized conditions to choose between any two targets:

1. Prefer a short-range or medium-range target over a long-range target.
2. Between two long-range targets, prefer the one that can be attacked from a position which can be reached with the lower movement path cost, according to A*.
3. If we require a supply resource, prefer the target that provides more of it.
4. Prefer a short-range target over a medium-range target.
5. Between two medium-range targets, prefer the one at the smaller distance from the units' current location.
6. Prefer the target with the higher contextual valuation, according to SelectValuable.

This algorithm is implemented by AlgorithmGeneral.SelectMoveTarget.

9.6 Build Command

This section describes the support algorithms for the Build command. Please refer to "Build Command" on page 125 on how to customize this command.

All computer players can be forced to build entities randomly rather than strategically. This option is accessible in the AlgorithmOptions.UseRandomBuild property which defaults to false. You can use Hexkit Editor and the Player Setup dialog to set a different value for each player.

Note. The current computer player algorithms cannot build entities other than units.

9.6.1 Random Building

Build a random number of units of any class that the active faction is allowed to build, excluding classes without valid placement sites. Continue until the faction's resources are depleted.

This algorithm is implemented by AlgorithmGeneral.BuildRandom.

9.6.2 Value-Guided Building

Distribute the available resources according to the faction's valuation of each unit class.

Note that the contextual valuation of a unit class is adjusted by its share among the faction's existing units by default (see "Contextual Valuation" on page 142).

1. Consider all unit classes that the faction can build and which have at least one valid placement site.
2. For each unit class, the number of reserve units to build is the maximum possible number multiplied with the faction's contextual valuation of the class.

   The result is rounded up unless the valuation is exactly zero, indicating that the faction desires absolutely no new units of that class.
3. Iterate over all unit classes by decreasing valuation, building as close to the calculated number of units of each class as the remaining faction resources allow.

This algorithm is implemented by AlgorithmGeneral.BuildByValue.

9.7 Place Command

This section describes the support algorithms for the Place command. Please refer to "Place Command" on page 129 on how to customize this command.

All computer players can be forced to place entities randomly rather than strategically. This option is accessible in the AlgorithmOptions.UseRandomPlace property which defaults to false. You can use Hexkit Editor and the Player Setup dialog to set a different value for each player.

Note. The current computer player algorithms cannot place entities other than units.

9.7.1 Maximum Placement

A scenario might enforce placement restrictions, for example a stacking limit that restricts the maximum number of units on a single map site. Since we will often want to place more than one unit at once, a dedicated algorithm takes care of observing such restrictions.

1. Check if Faction.CanPlace succeeds for the entire list of units to be placed.
   If so, place all units and return.
2. Otherwise, remove the last unit from the list.
3. Check if CanPlace succeeds for the remaining list.
   If so, place all remaining units and return.
4. Otherwise, repeat from step 2 until the list is empty.

This algorithm is implemented by AlgorithmGeneral.AttemptPlace. The placement algorithms described below both rely on AttemptPlace to perform the actual unit placement. This ensures that all placement restrictions imposed by the rule script are obeyed.

Note. As with the conditional group attack, we do not test all possible permutations of placed vs. reserve units, nor do we attempt to choose among several possible permutations.

Custom placement restrictions may therefore lead to suboptimal unit placement decisions, with important units ending up at the wrong location or remaining in reserve.

9.7.2 Random Placement

Place each unit on a map location that is randomly selected among all valid placement sites for its class. Units without valid placement sites remain unplaced.

This algorithm is implemented by AlgorithmGeneral.PlaceRandom.

9.7.3 Threat-Guided Placement

Prefer those placement sites that are exposed to the greatest threat by enemy units, adjusted by the site's contextual valuation.

1. Iterate over all unit classes on which at least one unplaced unit is based.
2. If there are no valid placement targets, skip the class.
3. If there is exactly one valid placement target, place all units of the current class on that map location. Continue with the next class.
4. Otherwise, call EvaluateThreats to determine the threat level of each placement target. Multiply each threat level with the contextual valuation of the map site, and normalize the resulting list so that their sum equals one.
5. For each placement target, multiply its normalized threat level with the number of unplaced units of the current class, rounding up the result.
6. Place the resulting number of units on each target, starting with the one that receives the greatest number of units. Continue with the next class.

This algorithm is implemented by AlgorithmGeneral.PlaceByThreat.

9.8 Seeker Algorithm

The basic computer player algorithm, nicknamed "Seeker", performs no strategic planning, does not maintain states or use a prediction tree, and provides no specific options.

"Seeker" tries to wreak as much havoc as possible without thinking too hard. Each unit is considered individually and sent against the nearest or most valuable target of any kind, preferably enemy units. A few heuristics limit the most outrageous stupidity and allow for some ad-hoc cooperation between nearby units.

Due to its lack of strategic planning, "Seeker" is easily distracted by worthless but closer targets when it should pursue more valuable but more remote targets. In scenarios with more than two factions, it does not recognize when one enemy's units threaten another enemy, so its choice of target may inadvertently aid a stronger and more dangerous foe. Moreover, it may scatter its forces when fighting multiple opponents, although the various cooperation heuristics counteract this weakness to some degree.

On the upside, the algorithm is usually very fast – the actual time spent on calculations is so short that computer players appear to move instantaneously. Slowdowns are possible if units must find paths over very long distances, or if custom scenario rules frequently prevent units from acquiring a target. Thanks to its blind aggressiveness, "Seeker" works well for a "barbarian horde" type of faction, particularly when random building and placement are enabled.

9.8.1 Overview

"Seeker" implements a custom unit cycle to generate Attack and Move commands, and employs standard algorithms to generate Build and Place commands.

1. Initialize all required data for the unit cycle in steps 2–3:
    a) Create a list containing all active units. Set the current unit index to zero.
    b) Create lists containing all potential target sites of various types.
    c) Initialize the minimum required attack odds to four.
2. Perform the following steps with the active unit at the current unit index:
    a) Select an attack target if possible.
    b) Select a movement target if possible.
    c) If there are two different valid targets, use heuristics to select one.
    d) Execute an attack if a valid attack target is in range and the odds are met.
    e) Execute movement if a valid movement target is defined.

    After command execution, remove any units that have become inactive from the list of active units, and adjust the current unit index if necessary.
3. Set the current unit index to the next active unit. If we have cycled through all active units without issuing any commands, decrease the minimum attack odds.

    Return to step 2 if there are any active units left that have at least one valid target, and the odds are not negative.
4. Call either BuildRandom or BuildByValue to build new units, depending on the value of the UseRandomBuild option.
5. Call either PlaceRandom or PlaceByThreat to place any unplaced units, depending on the value of the UseRandomPlace option.

This algorithm is implemented by Seeker.FindBestWorld. In the rest of this section, we'll examine the first three steps that comprise the unit cycle.

9.8.2 Initialization

In the first step we set up the collections for all active units and their potential targets.

Active Units

The list of active units initially contains all units owned by the active faction that are placed on the map and that can either attack or move. If there are any unplaced units, they'll have to wait until the algorithm terminates before they are placed on the map.

The unit cycle always examines a single unit at a time, which is the element in the active units collection that is indicated by the current unit index. This index is initialized to zero.

Potential Targets

The lists of potential targets contain the following map sites:

Attack Targets — All sites that contain enemy units. We want to destroy these units.

Engaged Attack Targets — Initially an empty collection. During the unit cycle, this list will contain all attack targets that were selected by any of our units, even if an attack was not executed yet. We will use this list to arrange group attacks.

Capture Targets — All sites that can be captured and belong to another faction. We want to occupy these targets with units that have the Capture ability.

Free Capture Targets — All capture targets that do not contain enemy units. Same as above, but we actually *can* occupy these sites.

Garrison Targets — All sites that can be captured and already belong to the active faction. We want to place idle units on these targets to prevent their capture by enemy units.

During the unit cycle, these lists will be continually updated to reflect the changes to contents and ownership of any targets due to our Attack and Move commands.

Minimum Attack Odds

The minimum attack odds are initialized to four, allowing us to execute attacks with very little danger to our own units where possible. On the other hand, we don't want our units to remain inactive forever, so the cycle control will eventually lower these odds to zero if we were unable to take any actions at higher odds.

If any attack opportunities remain when the odds have reached zero, they will be exploited unconditionally, even if the results are suicidal. Various target selection heuristics attempt to minimize such cases, however.

9.8.3 Unit Actions

The core of the unit cycle operates on the *current unit*, which is the element in the list of active units that is indicated by the current unit index.

Select Attack Target

Select an attack target for the current unit if it is a combat unit, and if it either still has the Move ability or if the required minimum attack odds are still positive.

This condition has the effect to prevent unconditional attacks (odds equal zero) if the current unit has already moved. Unconditional attacks are usually suicidal, so if we just arrived within attack range, we would rather wait another turn for support units to join us, or for better targets to appear in our new movement range.

If the current unit still has the Move ability, call SelectAttackTarget to pick a target for a mobile attack among all remaining attack targets. All currently engaged attack targets are treated as preferred targets to increase the chance for group attacks. We do *not* care if the unit still has the Attack ability because we may want it to move towards a remote enemy right away.

Otherwise, the current unit must still have the Attack ability, or else it would not be an active unit. Call SelectAttackTarget to pick a target for a stationary attack, with the same parameters.

If a valid attack target was found, add it to the list of engaged attack targets.

Select Move Target

Select a movement target for the current unit if it still has the Move ability, and if there is either no valid attack target already in range, or if the required minimum attack odds are less than two.

This condition has the effect to suppress movement target selection until we have reached even attack odds, *unless* we need to move in order to attack. In other words, we delay the calculation of "peaceful" movement targets until we have performed all possible "safe" attacks. Since successful attacks usually open new and better movement possibilities, this optimization allows us to move more intelligently and saves a lot of redundant calculation.

Call SelectMoveTarget to pick a target among the various lists of potential movement targets, in a sequence defined by target priorities and the unit's abilities:

1. If the unit has the Capture ability, select a target among all free capture targets.
2. If the unit requires a supply resource with a priority of 60% or more, check whether we have already found a nearby high-priority target.

   If we have a valid movement target at medium range, immediately execute the movement. This ensures that we do not waste our slim resources on an attack that might render us incapable to reach the high-priority target.

   Otherwise, recalculate all supply targets for the active faction and select one that provides the required resource as the new movement target. Immediately execute any attack and then move, but skip the attack if the odds already equal zero since a unit that needs supplies is even less likely than usual to survive a suicide attack.
3. If we still don't have a movement target, select one among all garrison targets.

Apply Target Heuristics

Now that our current unit has one attack target, one movement target, or both, we must decide what to do about them.

1. If we have no valid targets at all, go to cycle control to activate the next unit.
2. If we have an attack target but no movement target, attempt to attack if possible, or else move towards the attack target if necessary.
3. If we have a movement target but no attack target, simply execute the movement.

4. Otherwise, we have two valid targets. If they are identical, or if the current unit has already reached *both* targets, attempt to attack if possible, or else move if necessary.
5. Otherwise, we have two *different* valid targets, and movement is required to reach one or both. So we must apply heuristics to choose which target to pursue.

Note. In these heuristics, "switching targets" means that we adopt the attack target as our new movement target. If the attack target is already in range, the movement target is simply deleted. Otherwise, we execute a movement that brings us into (or closer to) attack position.

Improve Attack Position. If the attack target is at short or medium range, attempt to find a good attack position and try not to run off in pursuit of a less important movement target.

If the movement target is also at short or medium range, check if it is a valid attack position, and immediately execute the movement if so. (Note that both targets cannot be at short range!)

If the movement target is a free capture target at long range, or a garrison target that is not a valid attack position, find a nearby garrison target that is a valid attack position. Execute attack if one was found at short range, or movement if one was found at medium range.

Drop Garrison Targets. Fighting enemy units is more important than garrison duty. If the movement target is a garrison target, switch targets and attempt to attack and move.

Prefer Capture Targets. Capturing sites is more important than engaging enemies at the same or a greater range. Capturing *nearby* sites is more important than engaging enemies at *any* range, *except* if those enemies could capture a more valuable site if we moved away.

If the movement target, which must be a free capture target at this point, is at least as close as the attack target, immediately execute the movement.

Otherwise, if the movement target is at medium range, immediately attempt to attack and move, *except* if the current unit is the last combat unit in its location, and that location can be captured, and the attack target contains enemy units with the Capture ability, and the current site has a higher contextual valuation than the movement target.

Note that this exception it is not terribly reliable because it only considers the attack target, and ignores any others nearby enemy units that could capture the current unit's location.

Prefer Attack Target. If none of the previous heuristics applied, our movement target is either too far away or not important enough. Switch targets and attempt to attack and move.

Execute Attack

If a valid attack target is in range and the current unit has the Attack ability, call AttemptGroupAttack to attempt a conditional group attack at the current minimum odds. All remaining active units may be drafted to support the attack.

If there is no valid target, or the unit cannot attack, or the required odds were not met, skip this step and attempt movement instead. Otherwise, update the list of active units and all target lists to reflect the changes caused by the Attack command.

If the current unit has become inactive, set the current unit index to –1 and go directly to cycle control in order to resume the unit cycle with the new first active unit. Otherwise, we must recalculate the current unit index because support units may have become inactive.

If all enemy units in the target location were eliminated and the current unit can still move, go back to target selection *without* changing the current unit. This may allow us to capture a site immediately after it has been "liberated."

Otherwise, the available movement targets are either unchanged or the current unit couldn't profit from them anyway. We continue with the current movement target, if any.

Execute Movement

If a valid movement target exists that does not equal the current unit's location, and the unit has the Move ability, issue a Move command towards that target.

Determine the *actual* new location of the current unit in case we didn't make it all the way, and update all target lists to reflect the changed world state. (For example, we may have captured a site that lay on our way but that we did not plan on capturing!)

If the current unit has become inactive, update the list of active units, set the current unit index to –1, and go directly to cycle control in order to resume the unit cycle with the new first active unit. We assume that a Move command cannot render any other units inactive.

If the current unit can still attack, go back to target selection *without* changing the current unit. This may allow us to immediately attack some enemies that we were trying to reach.

Otherwise, we are done with this unit and enter cycle control to activate another unit.

9.8.4 Cycle Control

The unit cycle ends immediately if there are no active units left.

Otherwise, the current unit index is incremented with a wraparound to zero. If the index does reach zero again, we check if any commands were issued since the last time the index was zero. If so, we clear this flag and resume the unit cycle at the *current* minimum attack odds.

Otherwise, we check if the current odds are already zero, or if we failed to find any valid targets for any units in the previous loop. If so, the unit cycle ends.

Otherwise, we *decrease* the required minimum attack odds and resume the unit cycle. This may allow us to exploit attack opportunities that were previously considered too dangerous.

Note. The index may also reach zero prematurely if the current unit became inactive. However, this can only happen if a command was issued, so we don't need to check for this special case.

If the index does reach zero because the current unit became inactive, we may re-examine a number of active units that we already examined at the current minimum attack odds.

While not always optimal behavior, this is actually desirable most of the time because the actions that caused the current unit to become inactive may have changed the world state in such a way that we should reconsider all units anyway.