

Domain Specific Techniques for Creating Games

A Major Qualifying Project Report:

submitted to the faculty of the

WORCESTER POLYTECHNIC INSTITUTE

in partial fulfillment of the requirements for the

Degree of Bachelor of Science

by:

Michael Anastasia

Jeremiah Chaplin

Micah Gaulin-McKenzie

April 25, 2007

Approved:

Professor Gary F. Pollice, Major Advisor

1. Domain-Specific Languages
2. Strategic games
3. Code Generation

Abstract

Modern game development projects rely on specialized tools for physics, graphics, and interface building in order to abstract common game features and reduce the cost of development. However there remains a notable lack of domain-specific tools for encapsulating and expressing game rules. This paper examines strategic games and presents the Game Design Language as a language for expressing game rules.

Acknowledgements

Thanks to WPI Computer Science students, WPI Interactive Media and Game Development students, the WPI Game Development Club, the Boulder Game Developer's Club, and the WPI Science Fiction Society and anyone else who participated in our user study.

Many thanks belong to Professor Gary Pollice for his advice and guidance throughout the completion of this project.

Table of Contents

Domain Specific Techniques for Creating Games	1
Abstract	i
Acknowledgements	ii
List of Illustrations.....	iii
List of Listings	iv
1. Introduction	1
2. Background.....	3
Domain Specific Techniques.....	3
Game Development Tools.....	5
3. Methodology.....	9
Domain Analysis	9
Strategic Game Example.....	11
Game Design Language Schema.....	12
GDL Parsing and Template-Based Generation.....	15
Library Architecture	19
Rule Package	20
Battle Package	21
Game Package	23
4. Results and Analysis	27
Metrics	27
Survey	27
5. Future Work.....	29
Language.....	30
Studies.....	32
6. Conclusions	33
Appendix A.....	34
References.....	35

List of Illustrations

Figure 1 Package Overview	19
Figure 2 Rule Structure Diagram	20
Figure 3 The Results of a Stratego Battle	22
Figure 4 An Example Phase-Turn Tree	24

List of Listings

Listing 1 Common Elements of strategic board games	10
Listing 2 Definition of strategic board game	11
Listing 3 GDL Document Example.....	15
Listing 4 BattleRule.tmpl.....	17
Listing 5 Stratego.xml (LowWins Rule).....	17
Listing 6 LowWins.java.....	18

1. Introduction

The goal of this project was to apply domain specific programming techniques to create a language for generating turn based strategy games. The language is written in XML while the templates and runtime libraries for the architecture are written in Java. The programmer writes the players, boards, units, rules, and turn structure of the game in XML. The XML is given to a parser which fills in the templates to produce a game architecture. The programmer then needs to create a user interface to the architecture to have a game.

A domain specific language (DSL) is a programming language that describes a specific problem domain. Ruby on Rails is an example of a DSL for creating web-based applications that use databases. The Rails framework is used to generate and test the Ruby code for the architecture. This allows web developers to generate applications quickly using minimal coding.

The background section of this paper includes an overview of domain specific techniques in general, including a discussion of libraries, code generation, and domain specific languages. The background section also includes taxonomy of existing game development tools.

The methodology section includes an analysis of the game domain, an explanation of the language structure, and the process used to collect about the project. The domain analysis includes definition and examples of game elements. The language section specifies how these elements are expressed in the domain specific language. The data collection section includes uses cases for the language as well as an explanation of the

survey conducted. The survey was done to determine which features were regarded as useful and what could be improved.

The results section covers the results of the survey. The survey consisted of a fifteen minute video demonstration of the making a game and then a series of questions based on the video. The questions cover ease of use, comprehension, future work, and comments.

The future work section describes what improvements could be made to the project. This includes what could have been done given more time, or what modifications could be done to replace existing components. It also covers what could studies could be done to using the existing project.

2. Background

Domain Specific Techniques

Within any area of knowledge, or application domain, problems to be solved often have many aspects in common. Domain specific techniques rely on commonalities between applications in a similar domain to improve quality and efficiency of work within that domain. Integrating existing knowledge of the domain with the software engineering process makes such improvements possible. Many of these techniques re-use existing solutions to common problems in their domain. Others allow programmers and domain experts to develop new applications in the language of the domain rather than the language of computers. In either case domain specific tools abstract the commonalities of their domain while allowing developers to express and control the concepts and procedures that vary from application to application.

A problem, once solved, need not be solved again. This is the guiding principle behind libraries. Existing solutions to well understood common problems in a domain can be grouped together and encapsulated in a library. Using libraries developers can save themselves the time and effort of reinventing the wheel and can instead rely on expert solutions to common tasks.

Code generators are software tools that read some input and produce a body of software code. Often the input is expressed in a domain specific language. Prime examples of this are Lex and Yacc. Lex and Yacc are domain specific tools used for creating compilers. They take input in BNF format, a language for describing grammars, and produce output in the form of a program to parse the grammar described in the input.

Ruby on Rails™ is a web-application framework on the cutting edge of code generation. Web applications are sufficiently well understood at this point that there are conventional ways of putting them together, specifically the model-view-controller framework. Rails can automatically generate Ruby source code for applications that follow the convention. This leaves the developers only with the tasks of adding content and attaching a database server. Using Rails to generate the framework developers save time and ensure that the conventions of Web development are followed correctly.

A Domain Specific Language (DSL) allows domain experts to express knowledge from their domain, in the language of their domain, to a computer. Compared with general purpose programming languages, DSLs are capable of expressing far fewer programs, but they exchange this capability to express specific programs very clearly. DSLs are not always executable languages. (Mernik, Heering and Sloane 316-344) Non executable languages describe objects or concepts from a domain, executable languages describe procedures within a domain.

The important feature of domain specific tools is not the form they take but that they have a vocabulary and structure created from a complete understanding of the problem domain. Without a thorough understanding of the domain it is impossible to speak in the language of that domain. (Mernik, Heering and Sloane 316-344) If there are no existing solutions or patterns of solutions for a common set of problems it is impossible to create a framework for recreating those solutions time and time again. In domains with well-known best practices or long standing conventions domain-specific tools can save considerable time and expense.

Game Development Tools

The need to simplify the process of creating computer games is not a new problem. Game development projects require a wide variety of specialized knowledge: artists, designers, writers, and programmers all contribute. Effective tools that simplify game development for any of these contributors are a valuable asset as they reduce the time and investment required to bring a project to its fruition. This paper focuses solely on tools that reduce or simplify the task presented to the game programmer.

It is possible to place game development tools in a spectrum based on how much restriction they place on the games to be created. On one end of the spectrum lay tools and libraries that focus on specific computer functions common to games rather than specific game functions. On the far end are tools that can only alter existing games. In between are tools that provide specific kinds of rules and languages or tools for creating or describing entire games.

These tools can also be organized into a spectrum of how much programming knowledge they require. This spectrum is bounded by programming languages themselves with function/object libraries meant for games on one end and graphical point-and-click game builders on the other end.

Allegro© is a C library that contains functions for graphics, sound, matrix math, keyboard input, and manipulation of configuration files. ("Allegro - Introduction."n.p.) Because it © encapsulates only the most generic of capabilities since it does not constrain programmers to any type of game or rules set. Allegro requires a solid background in C or C++ programming in order to use so it is inaccessible to non-programmers.

Torque 2D© is one of a small series of engines that provide the back-end rules logic for games written in Torque Script. Torque 2D focuses on the needs of two-dimensional sprite-based games. ("Products: Torque: TGB."n.p.) Torque provides abstractions for animated sprites, physics, collision detection, networking, keyboard and mouse I/O, sound, and other common features of games. As long as game developers want to use Torque's features 'as-is' Torque can save time. If there is a need to change the internals of the engine, the developer must edit the Torque source code which cancels out much of the reason for using an existing engine in the first place.

Unreal® script is a language that allows users to create games or game modifications using the Unreal® Engine. The creators of Unreal® Tournament 2003 keep the internals of the Unreal Engine private but provide mod creators with UnrealScript. Using this language programmers can alter existing game objects such as weapons, player avatars, maps, scoring rules, and even create entirely new objects. ("UnrealWiki: Unreal Engine."n.p.) The programmers are constrained to the capabilities and constraints laid down by the engine: the rules of physics, networking capability, control schemes, key bindings, the 1st person perspective, and others. Within these constraints, programmers are able to create a wide variety of modifications and even entire new games that have nothing in common with Unreal Tournament on the surface.

The Klick line of authoring software provides a code-free avenue for game developers. Klick presents users with a graphical environment into which they can produce game content without any experience in software. Users compose the game environment on the screen and assign game logic and behavior to on-screen objects through a simple point-and click interface. ("Klik - Wikipedia."n.p.) Klick is a very

intuitive and programming-free tool for creating several common types of game. By necessity the behaviors and objects available are limited to those provided with the software and there are no backdoors for experienced programmers. This limits the extent of what kinds of game can actually be produced using Klick software.

Game Maker© is similar to Klick in many ways but sacrifices some user-friendliness for greater robustness. Objects in the game are composed using a point-and-click, drag-and-drop interface. ("Game Maker Pages."n.p.) The rules that govern their behavior and interactions are then written in Game Maker Language (GML) which is a built-in scripting language. GML is simple in comparison to many programming languages but nonetheless it is not easily accessed by non-programmers.

ViGL is a language for describing video games that was developed in a previous MQP. (Sutman, Schementi and Pollice 53) Syntactically, ViGL is a collection of XML types the fields of which are made up of strings of ruby code. Each of the XML blocks describes a common component of a video game. ViGL 'compilers' pass over each of these components and generate code in another programming language. The ViGL specification focuses largely on graphical needs of video games and integrating game rules with objects displayed on the screen. The language is extensible and while the original project team did not get far beyond creating objects that can be displayed and respond to input the language could be extended to support more common elements of videogames in the future.

Most existing game development tools can be placed into one of three categories: GUI tools, physics engines, or tools for developing specific styles of games. Board games receive little support from any of these tools. The challenging part of

computerizing board games is not the graphical interface nor does it require complicated physics calculations. Board games do not have complicated interfaces because their traditional form must be manufactured. Board games do not require complicated computations because players must do any math without a computer. Board games rely on simple rules and the enforcement of those rules.

There are many different kinds of board games. Board games can be categorized by common themes or by common game mechanics. Arranged by themes there are trivia games, historical games, economic games, race games, political games, word games, and many others besides. Organizing games by mechanics is more useful for this discussion. Games use many common rules archetypes. Area-Control, Auction, Capture, Light War, War, Progress, Race, and Tile-Laying each imply a certain style of rules structure. (Phillies and Vasel 222) These game types have physical components in common: boards, pieces, cards, dice, etc. Many of them also have rules components in common: turns, phases, victory conditions, movement of and battle between pieces, etc. Currently tools to reproduce those structures in software do not exist, and programmers must work from scratch each time.

3. Methodology

Domain Analysis

The first and potentially most important step in creating a domain-specific language is a clear and consistent definition of the domain itself. Without a good domain definition the language is difficult to create at best; flawed and useless at worst. The largest difficulty in defining a domain is determining the domain's scope. If the scope is too wide then the language risks being too general and ends up doing very little for the user. If the scope is too narrow then the language becomes inflexible and potentially useful applications are harder to create as users must spend additional time working around the language's narrow assumptions. The key is a definition that encompasses the majority of potential application's within the domain without weighing down the definition with additional constraints.

The definition of a game from an academic standpoint is contested, especially as relates to modern board and electronic games. Questions of goal, avatar, and other elements muddy the question of “What is a game?”. (Costikyan n.p.) While the discussion is an interesting and important academic debate it is far beyond the scope of this project to attempt to tackle such questions. The Game Design Language needs a functional definition that is understood instantly by any game designer that chooses to use it. To define ‘strategic board game’, we examine a list of games and decide which games are and are not within the domain. Using these choices as a guide, the elements of the games that were decided to be in the language are compared and similarities noted. These similarities form the basis for what constructs need to exist within the language.

Once the list of elements was compiled the list of games that were excluded were examined again to determine if they were in fact outside the scope of the language as it had been defined. After several iterations of this process, the language encompasses all the elements that are important without forcing the user into a strict definition of what a game was. The final list of games that were included or excluded is in Appendix A.

The common elements of the games selected are listed below. All the games have these elements in some variation and they provide the basis for the definition of a strategic board game.

- The game is represented by pieces arranged on boards. Everything about the current game can be represented either as a piece on a board or as a value associated with either a piece or a player
- Player actions alter the state of the game by moving a piece on a board or altering a value associated with a game object (piece, board, or player).
- Player's actions are sequential. While this does not mean that a single player cannot take many actions in sequence it does mean that game actions are atomic. One action completes before the next is processed. While the game can process the repercussions of these actions can be calculated at any time to give the illusion of simultaneous action the actions themselves are separate from any other action.
- The game has an ending condition. At some point the entire entity terminates because some player has achieved victory.
- Complex interactions between pieces that are caused by player actions are termed “battles” and their resolution form a subset of the games rules.

Listing 1 Common Elements of strategic board games

From these elements the following definition is derived:

A strategic board game consists of a collection of boards upon which game pieces are placed. Players sequentially take actions which alter these boards and pieces until an ending condition is reached.

Listing 2 Definition of strategic board game

While this might seem to be a limited view of games it actually encompasses a wide variety of games.

Strategic Game Example

The simplest way to examine the elements of a strategic board game is by examining games that fall within the definition. The game of Stratego is a good example of a strategic board game that is still of a reasonable size in terms of scope. This discussion assumes that the reader knows the rules to the game of Stratego; if the reader does not, they can be found at this website. (Collins n.p.)

The game of Stratego fits the definition of a strategic board game; there are players, a board, and pieces as well as an ending condition. The game's actions are moving a piece or revealing a scout. Both actions are atomic although the reveal of a scout is always followed by a move action. Play alternates between players after each player takes a move action until one player achieves the victory condition of revealing the flag piece. A battle in Stratego is triggered when a piece moves into a square occupied by an enemy piece.

Game Design Language Schema

The syntax of a document in the Game Design Language is defined by an XML Schema definition. The Schema format was chosen over the Document Type Definition, because it is more expressive, and is also in written in XML. This allows both the Schema and XML document to both be validated. Schema also allows the data types of various tags to be specified.

The Schema has a well defined syntax. The core tags are required to have a valid document. There are no default values for the required tags, so each tag needs to contain data. The values contained in the tags are substituted directly into Java code, so anything in the tags also has to be valid in Java where it is replaced.

The root of the document is the Game tag. The only attribute that the Game tag contains is the Title of the game. The AuthorInfo and LicenceInfo tags are meant to be substituted into the templates for documentation purposes. The Include tag was meant to be a way to split the XML code into separate files. However, the Include tag is an optional component and was not implemented with the current parser. Players, Boards, Units, Round, and Rules tags are where the game components are described.

The Players tag simply contains PlayerName tags. Each PlayerName tag contains the name of one of the players of the game. There must be at least two PlayerName tags, since the architecture does not allow for less than two players. There is no maximum number of players specified.

The Boards tag contains all of the information necessary to generate the boards and tiles for the game. The Tiles tag contains the TileDefault and Tile tags. The TileDefault tag is a tile with a TileType and a list of TileProperties. The Board tag is used

to describe the board as well as the initial setup of the game. The board dimensions are given in the BoardRows and BoardCols tags. The AllowedDirections tag specifies the directions that units are allowed to travel. This can be square directions, the cardinal directions, diagonal only, or the directions to represent a board as a hex grid. The board has a name, specified by the BoardName tag, that is used to identify the board within the architecture. The FillSpace tag references the TileDefault tag in order to tell the game what tiles to fill the board with by default. Any tiles that are not the default are given by the CustomSpace tag, which contains the coordinate and type of a tile. The board also contains a list of StartingUnit tags that are used to generate the initial setup of the board.

The Unit tag contains a list of all of the units in the game. The DefaultUnit tag is used to define the default properties of a unit and their values. Each Unit tag contains a UnitType tag, and a list of UnitProperty tags if their properties are different than the default. This is similar to the type and property setup for the tiles.

The Round tag is what describes the order that players take actions in the game. A list of Phase tags determines which phases are generated. Each Phase tag contains a PhaseName, which should be unique, and may be referenced by the SubPhase tag. The PrimaryPhase tag is used to determine if the Phase should be at the root of the phase/turn structure. After that, there are zero or more SubPhase tags which contain the name of the sub phases. The phase also has a list of ActionName tags within the AllowableActions tag, which specify the actions allowed during that phase. The Turn tag contains a list of PlayerName tags in the AllowablePlayers tag. The allowable players are the names of the players that are able to take actions during the phase by taking turns.

The Rules tag is where all of the ActionRule and BattleRule tags are described for

the game. Each rule has a RuleName and a Condition tag. The rules can also have a PrimaryRule tag which is used to determine whether the rule is the root of a rule tree that has SubRule tags. The SubRule tags reference the name of the sub-rules. The ActionRule tags have a TriggerAction tag that contains the name of an action that causes the rule to be checked. A BattleRule tag does not contain a TriggerAction tag. The BattleRule tags contain a Results tag instead. The Results tag is only used for BattleRules, since it describes what actions should be taken after a battle has been resolved. After a battle a score is generated and compared between the attacker and defender. The Score tag specifies how the score is calculated, while the AttackerWins, DefenderWins, and Tie tags are used to specify the actions to do based on those three outcomes.

The Condition and Score tags both use a system of nested predicate statements. These statements are generated as nested predicate objects when translated into Java code in the templates. The predicates include both logical and numeric operators. They also have boolean, numeric, and string constants as well as specific tags that represent function calls in the Java libraries.

```
<Game Title="The game title goes here">
  <AuthorInfo />
  <LicenceInfo />
  <Players>
    <!-- PlayerNames go here... -->
  </Players>
  <Board>
    <!-- Boards go here... -->
    <Tiles>
      <!-- Tiles go here... -->
    </Tiles>
  </Board>
  <Units>
    <!-- Units go here... -->
  </Units>
  <Round>
    <!-- Round structure goes here... -->
  </Round>
  <Rules>
    <!-- Rules go here... -->
  </Rules>
</Game>
```

Listing 3 GDL Document Example

GDL Parsing and Template-Based Generation

While the architecture of the run-time library was being assembled we used Stratego as a reference game. For each of our abstract game components we hand-wrote the concrete components that would be needed to implement Stratego with our library. This allowed us to pinpoint problems in the design and features we had not addressed. When the library was finished our implementation of Stratego was also complete.

Using the concrete classes built for Stratego it was possible to reverse engineer a series of template files. In each of these classes some portion of the code is ‘game specific’. All game specific code in the Java files is replaced in the template files by a tag. During the generation process, the parser replaces each tag with code generated from

the user's input. There are two styles of tag, a direct replacement and a function call, each with different meaning to the parser. Direct replacement tags take the form “##value##” and indicate to the parser that it should get the property named ‘value’ from the users input and insert the value of that property in place of the tag. Function call tags take the form “@@value@@” and indicate that the parser should invoke that function.

The parser is designed and written to provide maximum flexibility for the library architecture to change. For each major component of the architecture (Units, Battle rules, Boards, and so on) there is a Perl module that loads each of the templates related to that component, finds all the tags in those templates, and reads the XML language file to obtain and compute the appropriate values to replace each tag. For schedule reasons all the file paths and most of the architectural knowledge are hard-coded into the parser itself. This means that any changes in the library implementation are likely to require changes to the corresponding part of the parser. On the other hand if an entire set of related components is replaced the parser for that component can be re-written without impacting other modules in the parser.

This listing taken from BattleRule.tmpl and the following listing taken from Stratego.xml are examples of the data used by the parser to generate the rules of Stratego. These are taken from the template for the BattleRule concrete class and the definition of Stratego in the Game Design Language respectively.

```

public class ##RuleName## extends BattleRule {

    public ##RuleName##() {
        super(false,false);

        /*
        SubRuleConstructor creates, for each SubRule tag, the
following code:
        this.subRules.add(new *RuleName*());
        where *RuleName* is the name given for the SubRule.
        */
        @@SubRuleConstructor@@
    }

    @Override
    public boolean isCondition(ArrayList<AbstractUnit> Attacker,
ArrayList<AbstractUnit> Defender, ArrayList<Tile> Tiles) {
        /*
        ConditionConstructor creates the nested predicates for
the condition.
        */
        return @@ConditionConstructor@@
    }
}

```

Listing 4 BattleRule.tmpl

```

<BattleRule>
    <RuleName>LowWins</RuleName>
    <PrimaryRule>Yes</PrimaryRule>
    <SubRule>MinerMines</SubRule>
    <SubRule>SpyAssasinates</SubRule>
    <Condition>
        <BooleanConstant Value="true" />
    </Condition>
    ...
</BattleRule>

```

Listing 5 Stratego.xml (LowWins Rule)

While parsing the battle rules module of the parser loads `BattleRule.tmpl` into a buffer and gets each of the `BattleRule` elements from the description of the game. Then for each `BattleRule` element it performs each of the substitutions in the `BattleRule` template and outputs a complete Java class. The `##RuleName##` tags are replaced directly from the XML `RuleName` tag, in this case with the value “LowWins”. The `@@SubRuleConstructor@@` tag invokes a function within the parser that uses the

SubRule tags to generate constructors, in this case for the MinerMines and SpyAssasinate rules. The @@ConditionConstructor@@ tag invokes another function within the parser that creates a set of nested predicate objects from the contents of the Condition tag, in this case just the value true. Listing 6 contains the final LowWins.java

```
public class LowWins extends BattleRule {

    public LowWins() {
        super(false, false);

        /*
        SubRuleConstructor creates, for each SubRule tag, the
following code:
        this.subRules.add(new *RuleName*());
        where *RuleName* is the name given for the SubRule.
        */
        this.subRules.add(new MinerMines ());
        this.subRules.add(new SpyAssasinate ());
    }

    @Override
    public boolean isCondition(ArrayList<AbstractUnit> Attacker,
ArrayList<AbstractUnit> Defender, ArrayList<Tile> Tiles) {
        /*
        ConditionConstructor creates the nested predicates for
        the condition.
        */
        return new PredicateBoolean(true).computeResult();
    }
}
```

Listing 6 LowWins.java

Library Architecture

Visual Paradigm for UML, Standard Edition (Worcester Polytechnic Institute)

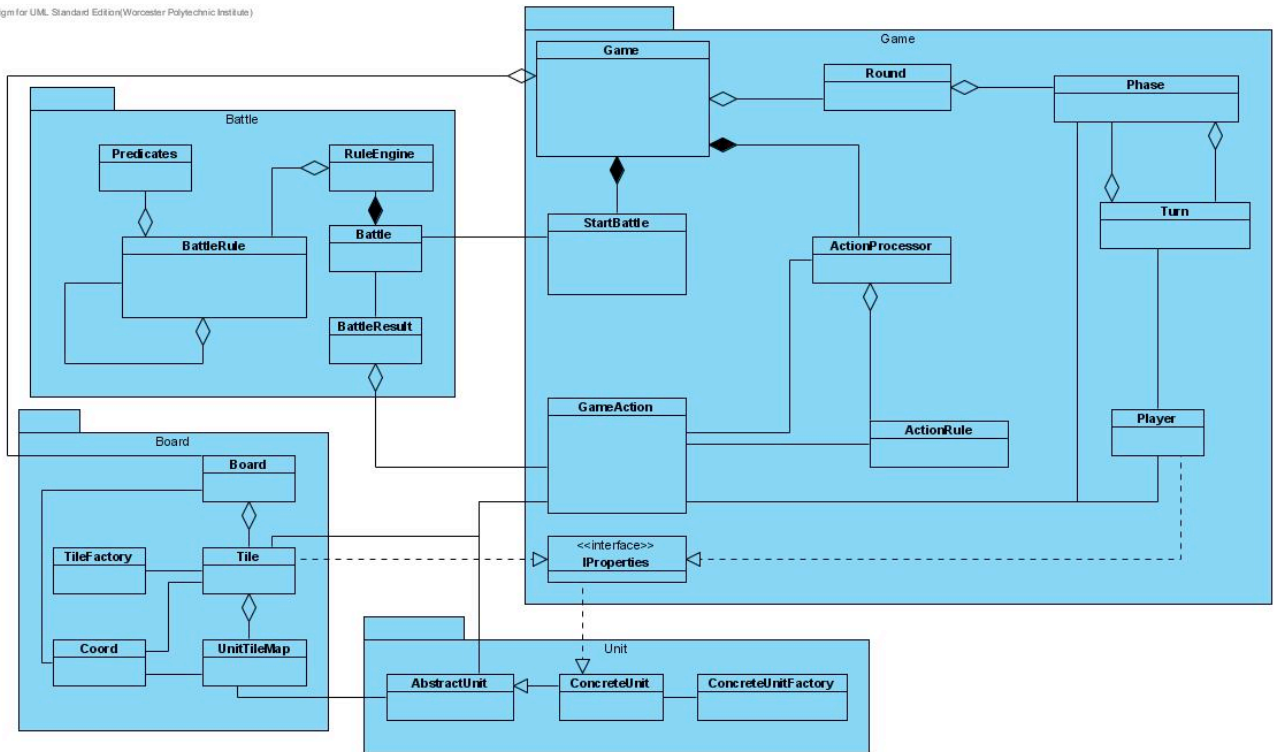


Figure 1 Package Overview

The biggest challenge when designing the architecture is finding a balance between an architecture which is general enough to be a useful development tool. The architecture itself is a combination of library functions for completing simple tasks and framework into which the templates and parser could fill in working code. The architecture is divided into 4 packages; battle, game, unit, and board. Each package consists of a single representational class which the other packages interact with. This representational class aggregates the other elements of the package and provides a clean facade for the other packages to interact with that data. The four representational classes are the Game class, the AbstractUnit class, the Board Class, and the Battle Class.

The Battle and Game packages are coded from scratch since these are the packages that need the most flexibility and power. Board and Unit while important are

not incredibly complex constructs so the base code for these packages were taken from a wargame implementations coded by Software Engineering students in a previous course. These packages are edited and expanded so that they fit cleanly with the rest of the system and have cleaner implementations but their basis remained intact.

Rule Package

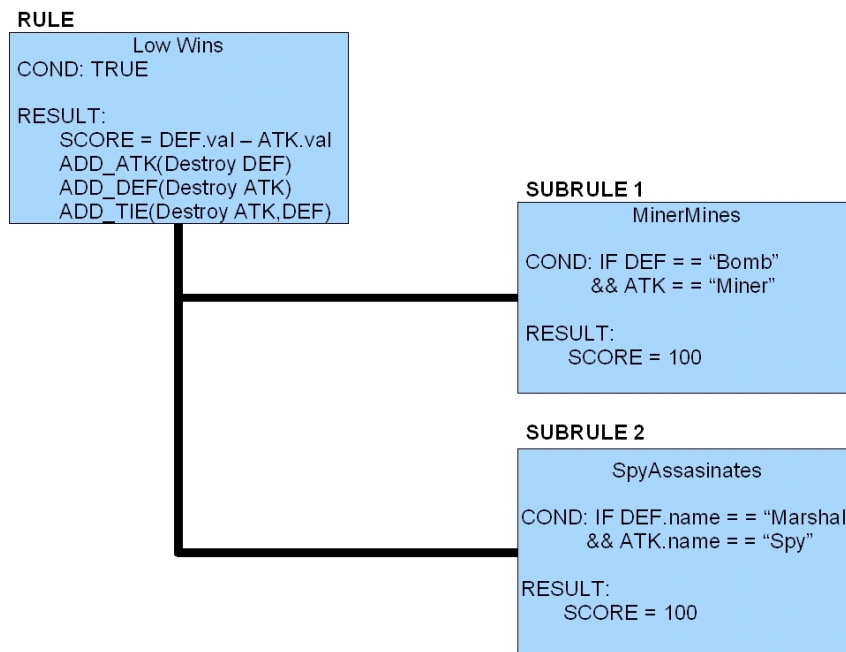


Figure 2 Rule Structure Diagram

Games are defined by the rules they present more than any other aspect. Rules are also the area of most variability among games and thus require a powerful and flexible representation to allow game designers the largest degree of freedom. The rules are nested class structures like the one depicted in Figure 2. Each rule is represented by a rule class which aggregates rules themselves. A rule is composed of a conditional and a statement which differs depending on the rule in question. The conditional determines whether any given rule is executed in a given situation while the statement contains the code to be executed for the rule. In the above example, “Low Wins” is the primary rule

and it contains two subrules, “Miner Mines” and “Spy Assassinates”. These subrules could conceivably contain subrules themselves.

There are two kinds of rules in the architecture; battle rules and action rules. Battle rules are used to determine the actions that result from any given battle. Action rules are used to determine whether an action taken by a player is legal. The conditionals for both rules are identical although their statements vary. An action simply calculates a boolean to determine whether the given action is legal and returns that boolean. A battle rule is slightly more complicated as it needs to return a series of actions for the game to take based on the rules for battles. This algorithm is described in more depths in the discussion of the battle package.

Battle Package

The battle package handles all direct interactions between different pieces on the board. Whether we are discussing the simple capturing of a piece in chess or a more complicated structure requiring die rolls and table look ups the same resolution method is used. The battle packages consists of the battle class which represents the battle, a rules engine which applies the rules supplied by the user, and a battle result which contains the game actions that will be executed by the game itself.

The battle class itself simply contains the elements of the battle; the attacking pieces, the defending pieces, and possibly tiles which might affect the result. What precisely constitutes an attacking or defending piece is decided by the battle sorter class which is discussed in the section about the game package. The rules engine takes the battle class and resolves the battle by iterating through the rules it contains and compiling the battle results from the rules in a battle result class for processing by the game.

Battle rules, like all rules, have a conditional and an action. The actions of battle rules operate on four elements; three lists of actions describing what happens if the battle is a win, lose, or tie for the attacker and a score that determines whether the attacker won or lost. Each rule can add results to any of the piles or adjust the score. The score is calculated as the sum of the scores of each subrule and passed back up the tree where the final score is calculated by the top-most rule. A positive score indicates the attacker won, while a negative score indicates the attacker lost and a score of zero indicated a tie. In Figure 3 we can see the results of a simple situation in the game of Stratego where a Miner piece attacks a Bomb piece.

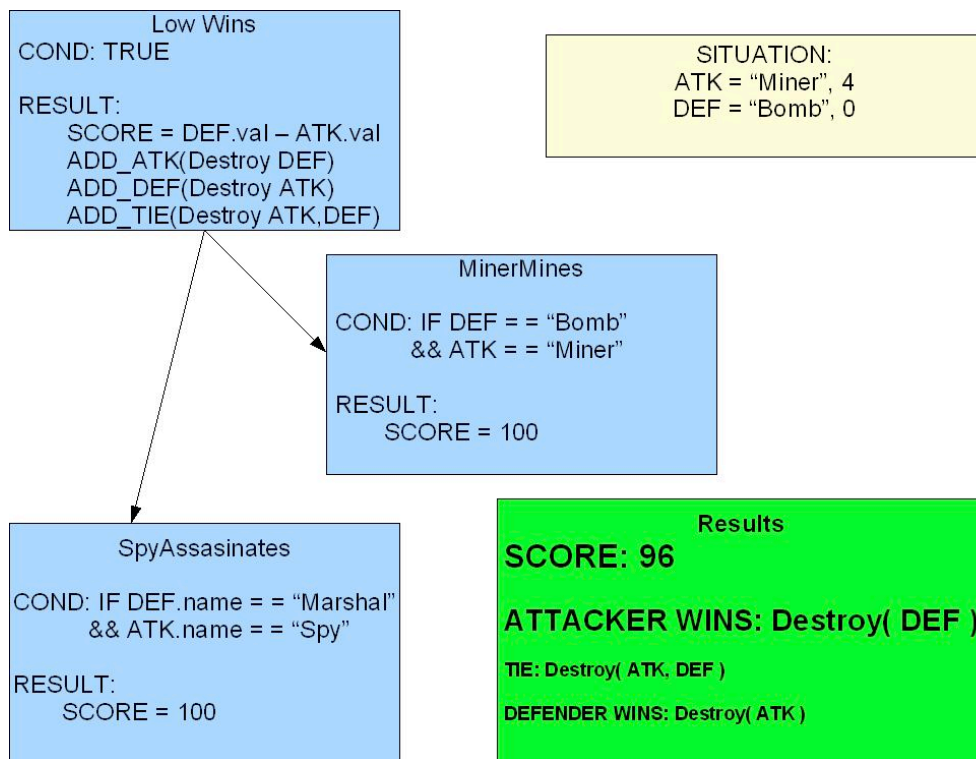


Figure 3 The Results of a Stratego Battle

In the figure above the “Low Wins” rule fires first since it is the topmost rule. This rule is designed to always evaluate (since it defines the default behavior of a battle in Stratego.) This rule gives us a score of 4 and places the appropriate actions as dictated

by the game of Stratego into the appropriate piles. Then we begin executing the subrules; the “Spy Assassinates” rule never fires since its conditional is not met. “Miner Mines” does fire and adjusts the score so that the miner is guaranteed to win the battle. When we compile the battle result class we see that the score is positive and return the actions listed in the attacker wins pile for execution by the game package.

It should be noted that if a rule's conditional is not met the rules engine will still continue on to check the subrules unless explicitly told otherwise by the user. This is to allow special case rules to execute properly and return whatever alternate result is described by the user.

Game Package

What actions players take or even result from battle interactions are one of the biggest areas of variability in games. As such this is an area in which the architecture sacrifices structure for flexibility. Actions vary so wildly between games that a simple library was constructed to handle the most common actions such as adding, removing, or moving a piece, changing a value associated with a piece or tile, etc. While the action library suffices for many games there is large number of special cases a particular game might wish to implement. For this a trapdoor class known as UserHook was created so that a user could easily go into the architecture and extend this class it to add his specific functionality. Actions are presented to the Game class which hands them off to the ActionProcessor. The ActionProcessor is responsible for determining the type of action and updating the game as the actions dictate. Actions that are generated from player input are filtered through the ActionRuleEngine in order to ensure the legality of the submitted actions. If an action in the set is not legal the game simply refuses to execute

any actions in the offending action list and waits for new input until a legal set of actions are given.

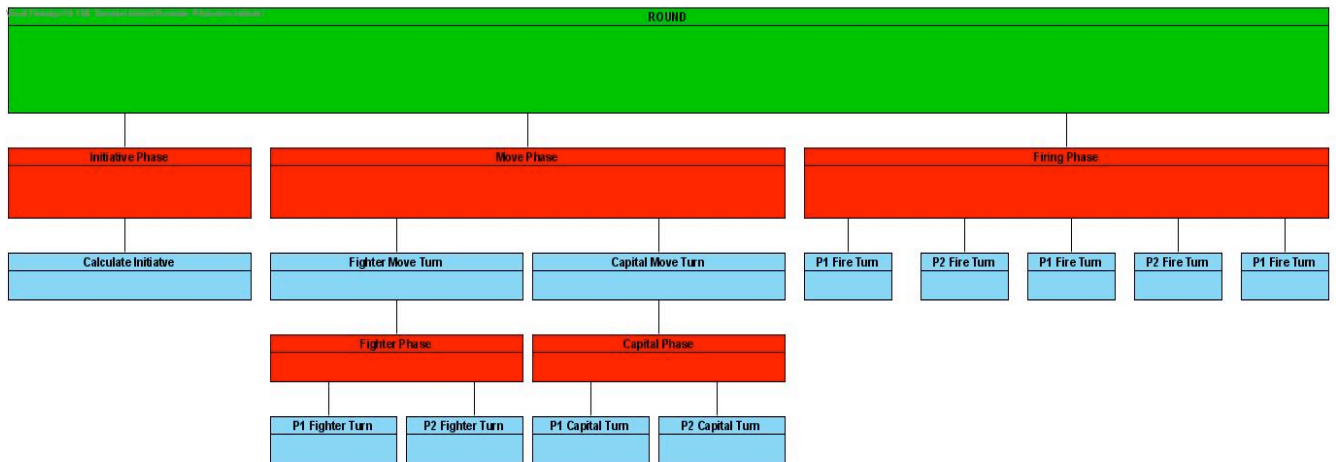


Figure 4 An Example Phase-Turn Tree

All strategic board games have a turn structure of some kind used to decide which player is taking actions and what actions they can take. These turn structures vary wildly from game to game and thus a structure is needed, much like rules, to capture the variability inherent in turn structures. The end result of this is the creation of the phase-turn tree, a structure which captures a large number of potential turn structures.

The phase-turn tree is created using three classes, Round, Phase, and Turn. Other than the root node the phase-turn tree defines alternating layers of turn and phases which define the turn order. Turns and phases aggregate each other so the resulting tree contains turns and phases in alternating levels. Phases are used to define what actions a player will take while turns define which players will act at any given time. The phase-turn always ends in a turn which defines a set of players who take actions defined by the tree.

In practice the user defines specific turn and phase elements that will make up his turn structure. The GDL then takes the configuration described and subclasses the

Round, Turn, and Phase abstract classes to create the array of objects that will form the phase-turn tree. Each user defined class executes some operation on either a list of possible actions or a list of possible players for Phases and Turns respectively.

The general structure of a turn phase tree is shown in figure 4. The root node, colored in green, is the Round. A Round represents the cyclic nature of most strategic board games; the game is defined as a series of Rounds until a termination condition is reached. The tree executes through a leftmost tree walk stopping at the leaves to prompt a player for input or perform some game state calculation.

The example above shows the structure for a theoretical space combat game. The first leaf is Calculate Initiative. Here we do some calculations for the game and the move on, no player's act during this turn. Next we visit the move phase which defines two actions, "Move Fighter" and "Move Capital". The subturn here is simply a container for the next subphase, the "Fighter Phase". Here we remove the "Move Capital" action from the list of potential actions. At the leaf we prompt player 1 to take an action, in this case "Move Fighter" actions are what is available to him. Once this turn is complete we walk to the Capital Move Turns and execute in a similar fashion. Once the move phase is complete the walk continues to the firing phase in which the firing actions are defined and then the players alternate turn taking firing actions.

While the does not include interface elements within the architecture itself some kind of hook is needed such that a user could easily attach their own interface to the system. Input from players is handled in the Player class which has a `getActions()` method which can be edited to hook out to a user interface.

When creating both the Stratego and Tic-Tac-Toe demos an interface needed

to be created for testing purposes. These are simple console based text interfaces that use a simple event passing system for communication between the interface and the model. This works well and serves as a foundation for how other UI designers might choose to extend the system.

The board package and the unit package are not developed from scratch. Professor Pollice provided several game frameworks at the start of this project that had been developed by his Object Oriented Analysis and Design class the previous term. Team Knight Rider's implementation of units is the basis for the Unit package and team A-Team's implementation of boards is the basis for the Board package, however each is altered to suit the generality needs of the GDL.

The largest edit to the Battle and Board packages other than some additional helper functions was the addition of the UnitTileMap class and IProperties interface. In the code we took from Object Oriented Analysis and Design the Board and the Unit classes had a bidirectional aggregation. While simple and easy to code this presented potential synchronization problems so an association class is needed to unhook the Units from the Tile.

The IProperties interface standardizes how the system attaches values to game elements. Any given piece or tile on the board could have game specific that a user would use in some game specific calculation. IProperties simply states that any class that implements should be able to return a value given a specific enumeration. This is how a user can implement elements like hit points, money, or elevation in their games. Once the value is defined the user then must simply reference it and the interface will retrieve the appropriate value for him.

4. Results and Analysis

Metrics

The source code for the project is divided into 5 components: GDL XML documents, the perl parser, the fixed library files, the generated game files, and sample user interface code. The perl parser contains 1517 lines of perl code in 11 modules. The fixed library contains 1497 lines of Java code in 51 classes. The sample user interface is made up of four classes with two hundred lines of Java code.

Tic-Tac-Toe and Stratego are both implemented in GDL. Tic-Tac-Toe uses 182 lines of XML, and includes 23 lines of custom code. The generated code for Tic-Tac-Toe contains 15 classes and 994 lines of code, giving a compression ratio of 5 times.

Stratego requires 304 lines of XML, and 15 lines of custom Java code. The generated code for Stratego contains 16 classes and 1006 lines of code, giving a compression ratio of 3 times.

Survey

In order to demonstrate that the GDL, parser, and library actually ease the burden on programmers, we conducted a study. The goals of the study were threefold. First, to determine if the language was easy to understand and use. Second, to determine if developing games with the GDL compared favorably with developing games in the most widely used general-purpose programming languages. Third, to determine the level of interest in continuing the project in future MQPs or IQPs.

In order to accomplish these goals, programmers who have worked on game projects, or at least large programming projects, need to evaluate our language. WPI Computer Science students and Interactive Media and Game Design students make an excellent test population for this project. Studying a team of programmers using the GDL in a complete game development project is beyond the scope of this MQP, so the study took a less hands-on approach.

Subjects watched a video-walkthrough of the GDL. The demonstration used video taken with the Camtasia screen capture software and voiceover recorded audio. The video walked through each step in the project architecture, beginning with the XML description of Tic-Tac-Toe, then running the parser and integrating the library with the generated source, and finally adding a user interface and playing the game. The video was roughly eighteen minutes in length.

After viewing the walkthrough of the GDL, subjects were asked a series of questions about each of the study's goals. To determine if the language was easy to understand and use subjects answered two multiple-choice questions about the system itself. These questions were meant to show whether or not the survey-taker actually understood what he had been taught while watching the video. Subjects were also asked how strongly they agreed with the statement "I feel after watching the video that I understand the language." To determine if GDL is easier to use than general purpose programming languages, subjects were asked to compare the ease of making games with GDL to the ease of making games with C++, Java, and the language of their choice, and if they felt that Rules and Turns were easy to understand and use in GDL. Lastly, the

survey asked if the subject would be interested in continuing work on the project as an MQP, or studying the way people work with the GDL as an IQP.

The survey results are mixed. There are a total of 24 responses to the survey although one of these we chose to discard since the participant simply did not fill out the survey properly. The 23 valid responses give a mixed picture of the system. 22% of those surveyed answered both of the comprehension questions correctly and 91% answered at least one of the two questions correctly. This shows that a large portion of the participants could, after watching the video, start using the system without much additional instruction. Interest in using the system over Java or C++ is strong with 78% of the survey participants preferring the system to a traditional programming language. 65% of participants feel that the turn and rule structures are intuitive and understandable which is important since these concepts lay at the core of the GDL. Interest in extending the project is not strong; only 26% of those surveyed display interest in working on the project in some fashion.

5. Future Work

The language currently includes the XML language written as a Schema, the Java templates and fixed runtime libraries and the parser written in Perl. Each of these components can be expanded to improve the language. Improving the parser would be necessary after improving each part except for the runtime libraries. Other future work includes studies on how well the system works.

Language

The Schema has a well defined syntax that has little flexibility. Creating an XML document that conforms to the Schema requires the majority of the tags to be in a specific order. The Schema could be modified to include each of the main tags to be specified in whatever order is convenient for the user. Very few tags are optional and do not have default values which bloats the GDL. Expanding on the Schema could be done by including defaults for certain tags which would make creating games easier by only requiring the user to write tags that deviate from the default set of values. More expressive or easier to understand tags could also be created to improve upon the language. The parser could be extended to use some of the tags that currently do not get used such as the “Include” tag. The original purpose of the tag was to allow the user to split the XML code into multiple files and have the substituted in by the parser. Time constraints prevented implementation of the “Include” tag was not implemented.

A further expansion of the language may include moving away from XML and instead creating a scripting language specific to creating games. This could be done using an existing scripting language such as Javascript, Python, or Tcl. This would make certain tasks in the language easier for the user although it also increases language complexity.

Libraries

The Java libraries and templates could also be expanded to include more useful elements specific to games. Creating a user specified artificial intelligence system for creating computer opponents would be useful to developers. Also providing the user with a graphical user interface library and set of commands in the language would

remove the need for the user to define output themselves in Java code.

Templates

The templates are currently Java code with annotations where code generation is needed. The parser simply finds the annotations within the file and replaces them with Java code generated based on what the user specifies. Modifying the language that the templates are written in would also require modification of the parser. However this means that if the parser were modified to be able to generate code in languages other than Java then the templates could also be written in that language. Having templates written in various languages would mean that a user could generate the game using the same architecture but using a programming language they were more familiar with than Java.

The templates could also be modified to include embedded Perl code. Embedding executable Perl code would make the parser less complicated. The templates would generate the code themselves, instead of requiring the parser to determine what language should be generated. This would make generating games using other languages easier.

Parser

The parser could also be improved to recognize keywords within the XML. This would eliminate the need for Java code to be embedded in the XML. The parser currently uses direct substitution for certain tags. It would also make it easier to generate code for templates written in languages other than Java without changing the appearance of the language on the XML end.

The parser could also be modified to make use of the templates with embedded Perl code within the templates. The parser would be more compact, since it would not

have code to generate code from specific languages. It would just search through the templates to find executable Perl code and generate the necessary game code from there.

Studies

Doing a study on the use of this Game Design Language is another way to expand upon the work done in this project. Determining if using domain specific languages instead of general purpose languages actually reduces the time it takes to create a complex game, such as a strategy game, could be measured. The time to create a game using the Game Design Language could be compared to how much time it takes to code an equivalent game in another language from scratch.

6. Conclusions

There are many domain specific tools that make creating games easier. However, many of these tools focus on creating graphics and interfaces. The Game Design Language was created to help make rules and game logic easier to generate.

The Game Design Language is an XML language meant for generating strategic games. The XML is given to a parser which generates code using templates and runtime libraries in Java. This allows the architecture to be flexible, since code can be reused to generate various strategy games.

The survey determined that people can understand the Game Design Language and how it works. The results of the survey are favorable. The majority of the participants were able to understand at least part of the system. Also, most of the participants like the idea of a language that they can create games with.

The Game Design Language is a viable tool for creating games. However, it needs more work. Based on the comments from the survey, there are many components that could be expanded upon. Using the domain specific programming techniques to create the Game Design Language was successful.

Appendix A

List of example strategic games:

- Battlespace
- Chess
- Risk
- Starcraft
- Homeworld
- Moving Tank Game
- Dungeons and Dragons (combat aspects)
- Stratego
- Diplomacy
- Warhammer 40,000
- Robo Rally
- Avalon Hill Wargames
- Alpha Centauri

List of games which are clearly not strategic games:

- Harvest Moon
- Final Fantasy I-XII
- Myst
- Combat Flight Simulators
- Empire Builder
- Babylon 5 CCG
- Ricochet Robots
- Burn in Hell
- Dungeons and Dragons (roleplaying aspects)

References

- "Allegro - Introduction." 12 Feb 2007 2007.
<<http://www.talula.demon.co.uk/allegro/readme.html>>.
- Collins, Ed. "Ed's Stratego Site." <<http://www.edcollins.com/stratego/stratego-rules.htm>>.
- Costikyan, Greg. "I Have No Words & I Must Design." 1994.
<<http://www.costik.com/nowords.html>>.
- "Game Maker Pages." 22 Apr 2007 2007. <<http://www.gamemaker.nl/>>.
- "Klik - Wikipedia." 26 Mar 2007 2007. <<http://en.wikipedia.org/wiki/Klik>>.
- Mernik, Marjan, Jan Heering, and Anthony M. Sloane. "When and how to Develop Domain-Specific Languages." ACM Comput. Surv. 37.4 (2005): 316-44.
<<http://doi.acm.org/10.1145/1118890.1118892>>.
- Phillies, George, and Tom Vasel. Design Elements of Contemporary Strategy Games. INTERNET: Third Millenium Publishing, 2006.
- "Products: Torque: TGB." GarageGames.
<<http://www.garagegames.com/products/torque/tgb/>>.
- Sutman, Andrew Burge, James M. Schementi, and Gary F. Pollice. Video Game Language., 2006.
- "UnrealWiki: Unreal Engine." 22 Jan 2007 2007.
<http://wiki.beyondunreal.com/wiki/Unreal_Engine>.