ABSTRACT

CREATING A DOMAIN-SPECIFIC MODELING LANGUAGE
FOR EDUCATIONAL CARD GAMES

by Kaylynn Nicole Borror

Domain-specific modeling languages abstractly represent domain knowledge in a way such that non-technical users can understand the information presented in the model. These languages can be created for any domain, provided the necessary knowledge is available. This thesis uses the domain of educational game design as a demonstration of the ability of domain-specific modeling. Games are useful tools in supplementing the traditional education of students. While games are an effective learning aid, educators often do not possess the design or technical skills to develop a game for their own use. MOLEGA (the Modeling Language for Educational Games) is a domain-specific modeling language that enables guided model design and code generation. Using MOLEGA, users can create abstract models inspired by UML class diagrams to represent card games of two selected variants. User models are then used to generate executable source code for a mobile compatible, browser-based game that can be deployed on a server by following provided instructions. MOLEGA is evaluated for validity and correctness using a suite of example models.

# CREATING A DOMAIN-SPECIFIC MODELING LANGUAGE FOR EDUCATIONAL CARD GAMES

A Thesis

Submitted to the
Faculty of Miami University
in partial fulfillment of
the requirements for the degree of
Master of Science
by

Kaylynn Nicole Borror
Miami University
Oxford, Ohio

2021

Advisor: Eric J. Rapos, PhD
Reader: Matthew Stephan, PhD
Reader: Karen C. Davis, PhD

This Thesis titled

CREATING A DOMAIN-SPECIFIC MODELING LANGUAGE
FOR EDUCATIONAL CARD GAMES

by

Kaylynn Nicole Borror

has been approved for publication by

The College of Engineering and Computing

and

The Department of Computer Science & Software Engineering

---

Eric J. Rapos, PhD

---

Matthew Stephan, PhD

---

Karen C. Davis, PhD

# Table of Contents

# List of Tables

# List of Figures

# Acknowledgements

First, I would like to express my gratitude to my thesis advisor, Dr. Eric Rapos. Without his constant support and encouragement, this thesis never would have been finished. In what felt like crisis after crisis to me in the past year, Eric helped pull me up and I'll always be grateful for that.

Also, I must thank my thesis committee, Dr. Matthew Stephan and Dr. Karen Davis, not only for being willing to support my work, but also for being amazing professors in my academic career at Miami University.

Finally, I owe a great bit to my friends, who I can certainly say have been like family to me the entire time I've known them. Especially those who were also in the masters program, who we all shared our struggles and achievements with. I'll miss my boys and our donut trail runs as we all start going our separate ways.

# Chapter 1

# Introduction

Software engineering is the application of engineering principles to create software systems. These systems range from banking software to video games to the code that controls aircraft.

One sub-domain of software engineering is model-driven software engineering (MDSE). MDSE is the use of models to represent software systems [3]. This approach to software design and development provides an abstract representation of what originally was a complex problem and solution. One significant aspect of MDSE is the use of domain-specific modeling languages (DSML). These languages are often specialized to a specific domain. DSMLs abstractly represent software systems inside a domain in a way where non-technical users can understand the information the model is presenting [4]. One specific domain that has the ability to leverage the power of DSMLs is that of game design.

Within game design, one focused area of interest is educational games [2]. These are games which are meant to educate the user about a specific topic, while also providing a fun and engaging experience. This is often done through the gamification of an activity that is not traditionally considered a game, or building a game around an educational concept. Those that design games may not possess the technical knowledge to implement their designs in an amusing and entertaining format. People who want to use games or gamified experiences to their benefit often do not possess either design skills or implementation skills to write code for a game. Educators often fall into this category, wanting to use games to enhance their students' learning experiences but not being experienced developers themselves. Having a way for an educator to create their own classroom aids without requiring the knowledge on how the aids work would be extremely helpful for the educator.

This thesis presents the creation of MOLEGA (Modeling Language for Educational Games), a domain-specific modeling language that allows educators to create web-based card games for usage in their classrooms. User models are created using MOLEGA inside a language-specific web editor that represent the specific type and style of game desired. From these models, fully functional code for the games can be generated in a format where it can be deployed in a web-based environment with minimal technical knowledge required, following detailed instructions. This model-to-text code generation is done in a way where the user is not required to have an understanding of the backend code of the game.

## 1.1 Motivation

The combination of a DSML that allows the abstract representation of a game with a code generator to generate said game allows a user, typically one of limited technical knowledge, to fully develop a

functional game based only on their specifications. Usage of the DSML makes it so these educators are not required to possess the knowledge on how to implement the underlying functionality, but rather need to focus only on their desired customizations.

Limited research exists on the usage of DSMLs in educational games. Further, no published research exists for the usage of DSMLs specifically for card-based educational games. Two published instances of DSMLs in educational games do not specialize on the type of game to be created, but rather focus on allowing a story or series of valid actions to be represented [5] [6] [7] [8]. Also, neither of these languages have any official implementation of the transformation between abstract model and working game code between them.

## 1.2 Contributions

This thesis aims to answer the following research questions:

> **RQ1:** Can domain-specific modeling be used to create web-based educational card games?
>
> **RQ2:** Does a guided framework ensure the generation of consistently correct executable game code?

These research questions are answered through the following contributions:

- The creation of a domain-specific modeling language for the definition of custom web-based educational card games (Chapter 4)
- The implementation of a model-to-text code generation engine to produce executable web-based code (Chapter 5)
- The implementation of a complete web-based framework integrating the DSML and code generation processes, applied to two example game types (Chapter 3)
- A systematic evaluation of the code generation process covering all aspects of the two example games (Chapter 6)

# Chapter 2

# Background & Related Work

Three main subject areas make up the background of this thesis: Domain-Specific Modeling Languages, Educational Game Design, and Web Application Development. Each of these main topics overlap to have interdisciplinary topics of their own, which can be seen in Figure 2.1.

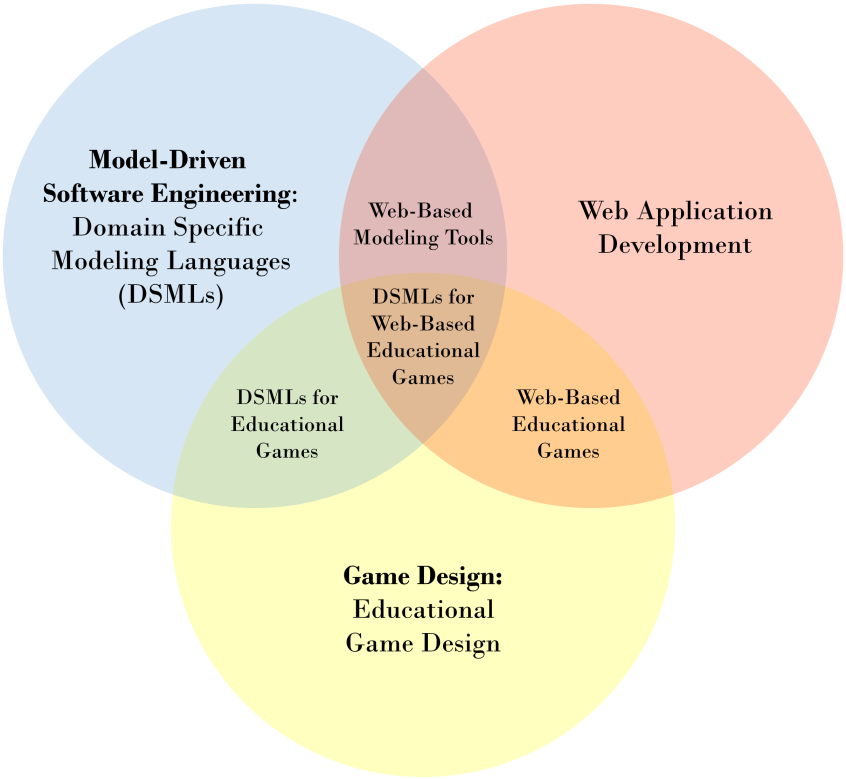


Figure 2.1: Topic Overview

## 2.1 Domain-Specific Modeling Languages

In this section, model-driven software engineering is introduced, along with some challenges associated with it, followed by the steps involved in creating a domain-specific modeling language.

### 2.1.1 Model-Driven Software Engineering

Model-driven software engineering (MDSE) is a subset of model-driven engineering that is focused on the usage of models for representing software systems. Using models instead of code to represent a software system creates more abstraction for a system. In this case, abstraction is purposefully leaving out details that serve no purpose in describing the function of the system. This allows a greater variety of people to be able to understand the system without requiring a working knowledge of programming [3].

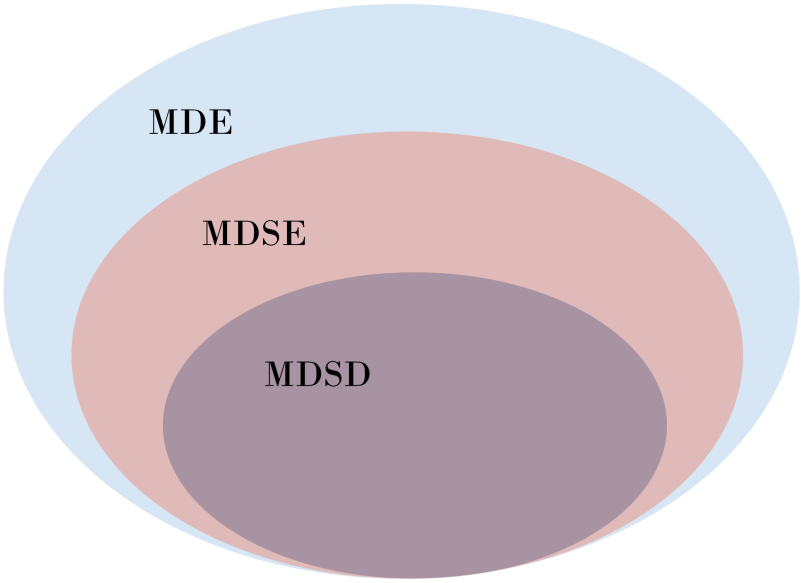


Figure 2.2: Relationships between MDE, MDSE, and MDSD

Model-driven software development (MDSD) is a subset of MDSE that focuses primarily on using high-level models in place of programming [9]. MDSD is commonly used to mean the same thing as MDSE. However, they are not completely the same. Figure 2.2 shown above shows the relationships between model-driven engineering, model-driven software engineering, and model-driven development, each labeled as MDE, MDSE, and MDSD respectively.

Models are created for one of two reasons: to represent a whole or part of a software system, and to represent a real-world situation affected by a software system. A model of a software system is referred to as a system model while a model of a situation affected by a software system is called a domain model [10]. Due to the limited nature of some modeling languages, a system should be represented by more than one model to capture its full functionality [11].

These models can be used for a variety of different activities already existing in software engineering. This includes prototyping a possible system [12], analyzing the system for design flaws, code generation to transform a model into executable code, and creating automated tests [11].

Just like every other area of software engineering, MDSE has challenges. These challenges can be separated into five different categories [13]:

- **Foundation**: e.g., Updating new models while allowing old ones to exist functionally; bidirectionality in model transformations
- **Domain**: e.g., Transformation of implicit to explicit knowledge for use in modeling
- **Tool**: e.g., Lack of consistent, human-readable tools; lack of autonomy/intelligence in modeling tools
- **Social**: e.g., Difficulty of getting non-technical people involved; confidentiality issues concerning using a modeling standard
- **Community**: e.g., How to teach MDSE; sharing large amounts of quality models

### 2.1.2 Creating Domain Specific Modeling Languages

Domain specific modeling languages (DSML) are modeling languages that are created for usage in a specific domain. Development of a DSML requires knowledge of the target domain along with knowledge of language development. To create a language, domain experts are required to work with language development experts in an iterative manner, which leaves room for errors and takes up a lot of time [4].

#### Steps for Creating a DSML

1. Identify the Domain

   The first step to creating a DSML is to identify the domain. This will determine a lot of the features required for use in the language along with setting the rules for how the language will work.

2. Create a Domain Model (Metamodel)

   The next step in the creation of a DSML is creating a domain model, also known as a metamodel. A metamodel defines the syntax and semantics of the model to declare what the model is able to display along with what it cannot display [14]. Due to this fact, the metamodel acts similar to a grammar for a language, containing a specific syntax and semantics.

   Many tools exist for the purpose of defining and managing metamodels in a way where any graphical representation can be used alongside it. For example, MetaEdit+ [15] [16] is a tool that allows easy storage and editing power of metamodels without requiring a user to know how to code. The Eclipse Modeling Framework (EMF) is a framework that can use metamodels to generate Java output using an XMI model specification [17]. Unified Data Model (UDM) framework is another tool that works similarly to EMF, but focuses on C++ rather than Java [18].

3. Create a Graphical Representation of the Domain Model (Metamodel)

   There are many different ways to create the graphical representation to be used by the metamodel. Many of the tools mentioned in the previous step use Unified Modeling Language

(UML) diagrams to represent systems. The Eclipse Graphical Modeling Framework (GMF) works by being layered with EMF so that UML created can easily be passed along to the metamodel for transfer into XMI model specification code then ultimately Java code [17].

These graphical model editors all involve a 2D coordinate system and a way to show relationships between elements in the model [19]. This often comes in the form of rectangles with text in them and line edges to connect the rectangles.

4. Create Instances Using Created Graphical Editor

   The instances of models do not have to be UML. UML is not always the best choice for representing a system. That is one of the points of creating a domain-specific language. For example, Instructional Modeling Language (IML) looks similar to UML, but has less complex options for relationships. This makes it more suitable for a beginning learning environment than a larger, more complex modeling language [20].

5. Leverage DSML features, code generation, model transformation

   Code generation is the purpose for creating many DSMLs. However, DSMLs can be used for other purposes as well. Model transformation is one example. Different types of model transformations include Model-to-Model, Model-to-Text, and Text-to-Model [21]. It is important to understand what is needed most from a DSML so that more accuracy can be put into that purpose.

## 2.2 Educational Game Design

This section begins with principles of game design and gamification, followed by a summary of educational games and what makes a game educational.

### 2.2.1 Game Design

Game design is a very large topic that, for digital games, involves the human-computer interaction (HCI) between player and gaming device. However, video game HCI and traditional software HCI are different. For example, games are focused on the experience of use rather than the final results. Games also constrain and challenge the user on purpose, where a traditional software system would seek to eliminate challenges. In addition, many features such as sounds and images seek to enhance a user's emotional experience rather than indicate functionality [22]. Games are also created for the purpose of enjoyment for a player. These differences result in the need for a different approach to game design than traditional software design.

#### Classification

Games are often informally classified in one of two ways: by genre or by resemblance to other games. Both of these can result in faulty classifications depending on the comparison made. For

example, comparing "The Elder Scrolls V: Skyrim" [1] to "Dungeons and Dragons" [2] because they both fall under the "Role-Playing Game" category is faulty due to the fact that one is digital while the other is almost strictly a pen-and-paper game. In addition, comparing "Candy Crush" [3] to "Doodle Jump" [4] because they both can be played on a mobile device also presents issues due to the fact that being based on mobile devices is one of the only things they share in common [23].

Table 2.1: Game Classification Parameters and Examples, Adapted from [1]

| **Classification** | **Examples** |
|---|---|
| Genre | Role-Playing, Action-Adventure, First-Person Shooter, Puzzle |
| Player Type | Single-Player, Multiplayer (competitive), Multiplayer (cooperative) |
| Device | Mobile Device, PC, Console |
| Game Lifetime | Story-based (end), Story-based (endless), Sandbox (endless) |

Games can also be classified by many different qualities to them, including but not limited to number of players, how the players are meant to interact with one another, how much time is meant to be spent in real-time hours playing the game, and many other qualities [23]. This thesis is focused on using the chosen sub-genre of educational-type games. Genre is a broad categorization that is often broken down into multiple combinations of genres for games, creating sub-genres. Some examples of game classification parameters can be seen in Table 2.1 [1].

### Gamification

Gamification is defined as the application of gaming concepts in places they are not usually found. With the popularity of digital games, along with how accessible gaming is with the existence of mobile devices, businesses and other organizations adopting some aspects of gamification has become more common [24].

There are three main principles for gamification: mechanics, dynamics, and emotions. Mechanics involve the goals, rules, and interactions that players have with the game or each other. In solitaire, a mechanic includes the goal of sorting all of the cards into stacks separated by card suit from smallest to largest number. Another mechanic is only being able to sort cards via largest number to smallest when not stacking. Dynamics are behaviors in players that arise from the act of playing a game. These are often behaviors related to strategies meant to increase chances of winning. While more easily seen in competitive-based multiplayer experiences, dynamics also exist in single-player experiences to a degree. For example, someone playing solitaire might not like the card that they drew out of the draw pile and cheat by pretending they didn't draw the card, placing it at the bottom of the pile. Emotions are both states of mental being and reactions drawn from participants in a gamified experience. The simplest examples of this include joy at winning a game and disappointment or anger at losing.

[1] `https://elderscrolls.bethesda.net/en/skyrim`
[2] `https://dnd.wizards.com/`
[3] `https://play.google.com/store/apps/details?id=com.king.candycrushsaga&hl=en_US`
[4] `https://play.google.com/store/apps/details?id=com.lima.doodlejump&hl=en_US`

**Serious Games**

Serious games are a broad category of game that range from displaying some sort of real-life social issue to educating the player in an interactive manner. While some are made for the purpose of learning and are intended to be fun, others do not aim for player enjoyment and rather seek to invoke critical thinking [25]. Many individuals and organizations, such as *Games for Change* [5], encourage the creation and spread of serious games.

### 2.2.2 Educational Games

Educational games, or "edugames," are a type of serious game that are used for the purpose of learning. While traditional games have the goal of creating a solely enjoyable experience, edugames have the primary goal of educating the players while also offering an enjoyable experience as a secondary effect.

In order to be useful, edugames must follow both traditional game design principles and pedagogical principles. This can be difficult, since game designers and education experts often do not possess enough knowledge about each others' domain areas to work completely independently [26].

The RETAIN model is a combination of other well-supported game design models that was developed to aide in evaluating how effective educational content is. The individual terms contained in the RETAIN model along with examples of their usage can be found in Table 2.2 [2].

Card-based activities are a common educational tool, often seen in the form of memorizing flash cards. However, card-based games have been shown to have a positive effect on learning as well. A 1998 study involved teaching students about gastrointestinal physiology through the use of modified versions of Go Fish and Gin Rummy [27]. A similar study in 2011 required pharmacy students to play games based off the same two card games three times each over a six-week period. The pharmacy study found that the student participants had an overwhelmingly positive reception to the card games and felt that it contributed to their learning [28].

## 2.3 Web Application Development

The accessibility of mobile devices with increasing computational power and memory storage has made web applications very popular in recent years. Web applications are, at the very least, websites that are optimized for usage on mobile devices. This term also includes web-based browsers displayed on a specific installable application on a mobile device [29], such as the Twitter social media app.

Website design principles are mainly based around the concept of usability, or user-based design. This comes in the form of heuristics lists that cover a variety of requirements that a website should follow to be considered usable. Some examples include [30]:

- The website layout is organized in a way that is predictable and similar to other websites (e.g., putting a navigation menu along the top or left side of the screen)

[5] `http://www.gamesforchange.org/`

Table 2.2: RETAIN Model Terms and Examples, Adapted from [2]

| | Definition | Example |
|---|---|---|
| Relevance | The media is relevant not only to a player's needs and learning styles, but also to previous content, i.e., the content builds off of itself | Contains relevance to a player's life using familiar themes or non-player-character types |
| Embedding | The combination of gameplay and educational content in a way where they are indistinguishable from one another | Teachable moments do not interrupt gameplay flow and challenges are not too easy or hard |
| Transfer | Evaluating a player's ability to transfer knowledge from one situation to another similar, but not quite the same, situation | Offering an opportunity to teach other players (real or non-player-characters) what they have learned |
| Adaptation | Evaluating a player's ability to take existing knowledge to find patterns, along with the creation of new knowledge to make sense of something that does not fall into those patterns | Requires a player to identify something they haven't encountered before and learn from it |
| Immersion | The high level of engagement a player has with the media that results in higher information retention | Allows a player to act in response to an event and promotes active participation |
| Naturalization | Spontaneous knowledge; the point at which a player can recall information without significant mental strain | Requires a player to draw conclusions about an event in accordance to previously obtained knowledge |

- The website layout and navigation style remains consistent throughout the entire site
- The website design is simple
- The website is aesthetically pleasing

A variety of tools for creating web applications exist, each with varying levels of abstraction. Adobe PhoneGap [6] and jQuery Mobile [7] are tools that are used through direct manipulation of HTML5. While HTML is usually quicker for a user to learn than a back-end programming language, these tools are aimed more towards users who already have an understanding of other front-end languages, such as Javascript.

Other more abstract tools have been proposed and created. For example, a nameless development system created in 2014 only defined as a 'mobile mashup' system has three different versions: a PC version, a mobile (pad-device) version, and a mobile (smartphone) version. The PC version offers full functionality and is intended for a user familiar with creating mobile web applications. The mobile version is aimed towards casual users and has more limited functionality than the PC version [31].

## 2.4 Multi-Disciplinary Research

These three large topics have some overlap between all of them. Each overlap between topics has some degree of research that has been conducted. The following topics are based off of the overlap between Sections 2.1 through 2.3, which can be seen in Figure 2.1 at the beginning of this chapter. Each related sections are listed at the beginning of the following topics for the reader's convenience.

### 2.4.1 Web-Based Modeling Tools

Web-based modeling tools, a combination of domain from Sections 2.1 and 2.3, allow developers to make use of MDSE in an easily-accessible web-based format. One example of such a tool is Umple Online.

Umple [8] is described as a model-oriented programming language that includes both class diagram and state machine functionality. It also supports integration into a variety of modeling/development environments, including Eclipse and Rational Software Architect [32]. Due to this, Umple can be used to generate models from the textual interface or directly inserted into programming languages to speed up the implementation process [33].

Umple Online [9] is a web-based environment that allows a user to either type Umple directly into the editor or draw a UML diagram. When the Umple text is updated on the left side of the web application, the UML diagram on the right side responds in real-time. The same occurs when the UML diagram is updated [33].

[6] `https://phonegap.com/`
[7] `https://jquerymobile.com/`
[8] `https://cruise.eecs.uottawa.ca/umple/`
[9] `https://cruise.eecs.uottawa.ca/umpleonline/`

Another web-based modeling tool makes use of a modeling language called Instructional Modeling Language (IML) [10]. IML is intended to be an educational tool to assist students in understanding MDSE without being overwhelmed by existing complex tools [20] [34]. The web-based nature of this tool makes it accessible to any user who has a web browser. An image of the model transformations web interface built into the IML website can be see in Figure 2.3.

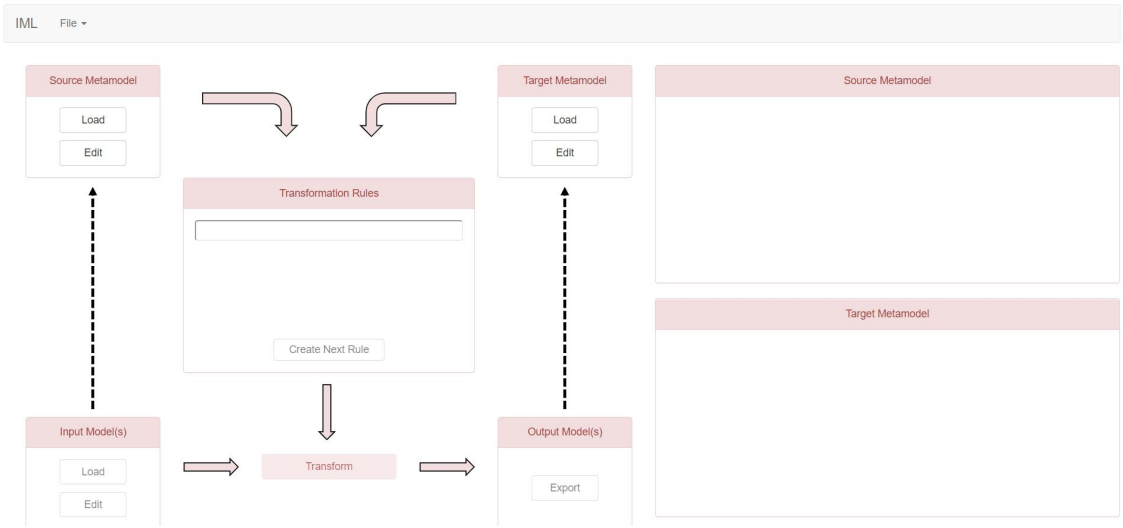


Figure 2.3: IML Model Transformations Web Interface

### 2.4.2 Web-Based Educational Games

Web-based educational content contains elements from Sections 2.2 and 2.3. Hosting educational content online allows for greater accessibility of the content to students. However, online content often fails to be engaging to students. Web-based instructional courses must be careful to not only provide information, but also to get students' attention [35].

Web-based educational games are one way to gain student attention. These types of games exist in all formats. An example of a single-player type of edugame is CodeCombat[11], which aims to teach students Javascript through the use of a RPG-style adventure game.

A multiplayer game that is commonly used in classroom settings is Kahoot[12]. This quiz-based game works by each student connecting to the live game hosted on the Kahoot website with their personal device. Students get points for answering the multiple-choice quiz questions quickly and correctly. Kahoot is popular among instructors due to its customizability. It is popular among students due to the competitiveness of the game via a 'Top 5' scoreboard.

[10] `http://iml.cec.miamioh.edu/`
[11] `https://codecombat.com/`
[12] `https://kahoot.com/`

### 2.4.3 Domain-Specific Modeling Languages for Educational Games

DSMLs for educational game development involves domain knowledge from Sections 2.1 and 2.2. Research exists for using DSML for general game development. Eberos GML2D is a DSML meant for modeling 2D games [36]. SharpLudus Game Modeling Language (SLGML) is a language focused on visualizing action-adventure games [37]. Emanuel Montero Reyno and José Á Carsí Cube explore the usage of modeling language to generate 2D platformer game prototypes [38]. However, not much research exists for usage of DSML for edugames.

#### Previous Works

A. T. Prasanna created a DSML for edugame usage using various images to represent game elements in 2012. For example, the image of a calculator indicates the presence of a math mini-game and a giftbox image indicates a reward. This language has models based around game storyline, separating the game into different levels. Each level consists of one or many objectives and occur sequentially. The player moves through these levels with a buddy element acting as an emotional motivator to keep the player moving through the game, along with having specific goals at the end of levels to encourage the player to keep moving towards the end of the game [5].

Another DSML approach based around serious games was created in 2013. This approach focused on meeting a list of influencing factors in serious games determined by the paper. Some of these factors include user freedom, assessment/measurement of progress, and adjusting to player skill level [7]. The language created, dubbed GLiSMo, consists of structure models (resembling UML class diagrams) and logic models (resembling UML statechart diagrams) [8]. This language is intended to be used for point-and-click graphical adventure genre games. The paper states that, though the language was created with the integration of educational content in mind, those elements would still be developed and implemented using the language by educators. Such educational content includes, but is not limited to, math problems to solve in order to proceed in-game [6].

#### Downfalls of DSML for Educational Games

Using DSMLs for game creation is considered to be one way to avoid the learning curve of using existing game development engines. However, they are often considered too specific for usage of modeling out any kind of game. First, there are many different game genres and types of play. While a DSML could be used to cover any genre of game, designers and other non-technical members may benefit more from a representation specific to a genre, which cannot be displayed in just one DSML [39].

Another challenge with using DSMLs for edugame design is the existence of different learning styles and teaching strategies [39]. A modeling language created specifically for role-playing story-based styles may not work for creating a game based around direct instruction with automatic correct-incorrect feedback.

It is important to take into account how detailed a DSML should be for representing the game. If the models include everything needed to generate the code and more, it quickly becomes overwhelming to view and loses any benefits the abstraction was meant to give. If the model has too

little information, then the development may not turn out the way originally envisioned when creating the model [39].

## 2.5 Related Work

As mentioned previously, due to how specific DSMLs are, it is difficult to propose a general-purpose modeling language to cover one entire genre of game. Doing research in domain-specific modeling (DSM) for web-based educational games involves bringing together all of the previously covered topics in varying degrees. Since there is no research previously published for DSMLs for edugames in a web-based format, the intersections of these topics contain many opportunities to create new solutions.

Work by A. T. Prasanna [5] and the DSML GLiSMo[6] [7] [8] are related to this work. For both works, DSMLs were created to represent different aspects of game development, such as choices that the player character can make at each stage of the game, along with areas where the insertion of mathematical problems is valid. Neither of these languages offer any code generation capabilities in their published research. Rather, they are meant to be used as visualization tools for a user to follow along with in order to understand the progression of game events from beginning to end.

Zahari et al. proposed an extension to the GLiSMo DSML, called FA-GLiSMo [40]. This extended DSML indends to represent educational adventure games while adopting elements to encourage Flow Theory: a learning theory that describes the state of complete engagement to an activity. FA-GLiSMo intends to build upon GLiSMo's drawbacks, aiming to embed elements in the learning theory into the educational games represented by the language.

Another DMSL, created by T. Eterovic et al., offers an abstract visualization of the connections between Internet of Things (IoT) technologies [41]. This approach, based off of UML diagrams, allows both technical and non-technical users to configure the plan of their own IoT systems. This language was tested through use of human interaction, evaluations done on two types of user groups: those who had UML experience and no IoT experience and those who had experience in neither topic.

SharpLudus, is a code generation environment intended for generating action-adventure games through use of domain-specific languages (DSL) [37]. This environment's DSL, SLGML, is focused around defining the game world, allowing representation elements like rooms and their design, non-player characters and their actions, and specifications for when a player character lives or dies. SharpLudus generates C# classes in response to receiving valid SLGML diagrams.

The research in this thesis differs from these related works in a few ways. Unlike previous works that define DSMLs for edugames, this research not only defines a DSML for a different type of edugame (i.e., card games), but also incorporates a code generation algorithm which allows the user to use the DSML to represent a game they want to exist, then to actually be able to create it. Rather than allow a user to create a game's objects and flow of gameplay, the language created in this thesis allows a user to specify a type of game included in the DSML's metamodel, along with customize a variety of features involved with the chosen variation of game.

# Chapter 3

# MOLEGA - Modeling Language for Educational Games

MOLEGA is a domain-specific modeling language that allows users to create models representing educational card games. The web editor for creating MOLEGA models checks that the model conforms to the metamodel, along with allowing generation of the game the model is representing all from one webpage. This section begins with discussing some of the technologies and rules in the code generation targets MOLEGA supports: Community Judge and Relations. This is followed by an overview of the MOLEGA language, ending with a summary of the code generator that transforms models to the code target.

## 3.1 Target Game Selection

Two possible target game rulesets are represented in the MOLEGA metamodel: Community Judge and Relations.

The rules of Community Judge games are almost identical to those of Cards Against Humanity [1] or Apples to Apples [2]. During a game turn, one player is designated the Judging player, having a black card which displays a prompt. All other players must play one card. After all other players have submitted their card choice, the Judging player chooses which of the played cards they feel best fits the black card's prompt. Upon that decision, the player who submitted that card gains a point and the Judging player title is moved to another player. One round has passed when all players have had a chance to be the Judging player. The player with the most points at the end of a certain number of rounds, or the player who reaches a certain score first, wins.

The rules of Relations games are a modified mixture of Gin Rummy and Go Fish. During a player's turn, they can choose cards in their hand that are related to one another. They can do this as many times as they see fit during their turn. The player can also click on their opponent's names in order to see their related card collections during their turn. When they have made all of their decisions, the current player can then either pass the turn or discard a card in their hand while passing their turn. This power then moves on to the next player in line. The player with with the most points at the end of a certain number of rounds, or the player who reaches a certain score first, wins.

Community Judge is a good choice for the first type of game due to the popularity of nearly

[1] https://cardsagainsthumanity.com/

[2] https://www.mattelgames.com/games/en-us/family/apples-apples

identical games like Cards Against Humanity. Additionally, this type of game doesn't have a strict rule structure, since the winner of a turn is determined by player opinion and not an in-game mechanic. This makes it easier to customize the content on both card decks. Relations, however, being a mixture of games with stricter rule structures, needs a little more attention to ensure that the game behaves correctly to player input. This is done by making sure related cards are listed correctly in the card file. While the card file setup may be a little more complex than Community Judge, Relations is another valid target for MOLEGA. Previous literature shows that when used in an educational context, modified versions of Go Fish and Gin Rummy are beneficial for student learning [28] [27].

As it is not possible to create a DSML to model every possible educational card game variant in order to answer **RQ1**, "Can domain-specific modeling be used to create web-based educational card games?", the selection of these two types of games, along with their included variants, aim to provide a representative sample that is sufficient in supporting the research question. By choosing two significantly different ruleset choices with multiple variations, MOLEGA serves as a proof of concept realization and demonstration of the power of domain-specific modeling to represent educational games without the need to provide full coverage.

The web game code for both types of games is built in a client-server model architecture. One code file in the bundle (app.js) acts as the server and contains all of the global information required for hosting multiple rooms of clients. The remainder of the HTML, CSS, and Javascript files contain the information that each individual client uses to connect to and interact with the server. This allows clients to connect and disconnect freely without impacting other clients' activities with the server.

The implementation of this architecture is done in a Node.js environment. In order to handle client interactions simply and smoothly, the Socket.io library [3] is used. Node.js was chosen for this reason. While Socket.io has been adapted for other languages such as Java, C++, and Python, it was originally written as a Javascript framework.

The games are coded in a way where both the mobile and browser versions are readable and scaled to size. The difference between browser play and mobile play can be seen in Figure 3.1, Figure 3.2, and Figure 3.3. Figure 3.1 displays the game interface for a computer browser player, while Figures 3.2 and 3.3 display the game interface for a mobile player, where the first figure shows the game table and players while the second is scrolled down on the mobile device, showing the player's hand of cards.

In order to know what sort of customizable attributes that the modeling language should include, these target games were created first. By having example targets complete and working first, it is easier to find the pieces that are customizable without breaking the flow of gameplay.

## 3.2 Modeling Language

MOLEGA, which stands for ***Mo**deling **L**anguage for **E**ducational **Ga**mes*, is a domain-specific modeling language that allows representation of the two selected types of games. MOLEGA sup-

[3] https://socket.io/

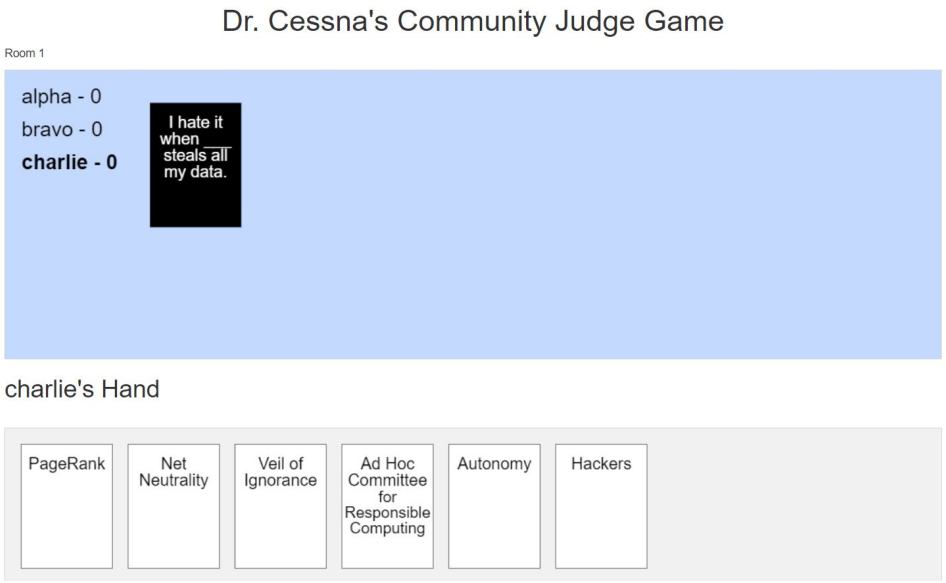


Figure 3.1: Browser Community Judge Play


Figure 3.2: Mobile Community Judge Play, Top of Page

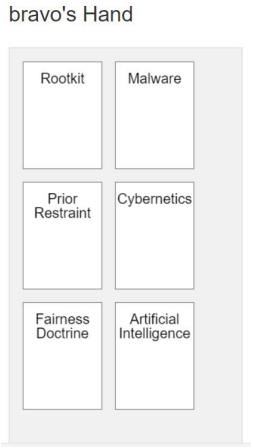


Figure 3.3: Mobile Community Judge Play, Scrolled Down

ports customization of both game setup elements as well as game content.

Game setup elements are defined by the attributes for each class contained in a model. The attributes *numOfRooms*, *maxNumOfPlayers*, and *minNumOfPlayers* allow a user to control the client capacity of their game server. For example, a user with a need for a larger number of clients, such as a class size of 70+ students, can balance these attributes to fit their client numbers. A «Theme» inheriting class can be customized to fit the user's color preferences or needs. If a user knows that some of the intended clients to the game are colorblind, they have the power to customize the colors as they see fit.

Game content customization is controlled by the contents of csv (comma seperated value) files

generated by the usage of a «Deck» inheriting class. For Community Judge type games, two of these files exist. For Relations type games, only one csv file exists. While these files come pre-loaded with example content, they are able to be modified using a spreadsheet software, such as Microsoft Excel.

The web editor layout and *File* menu options are seen in Figure 3.4. More details about the design of the domain-specific modeling language are presented in Chapter 4.

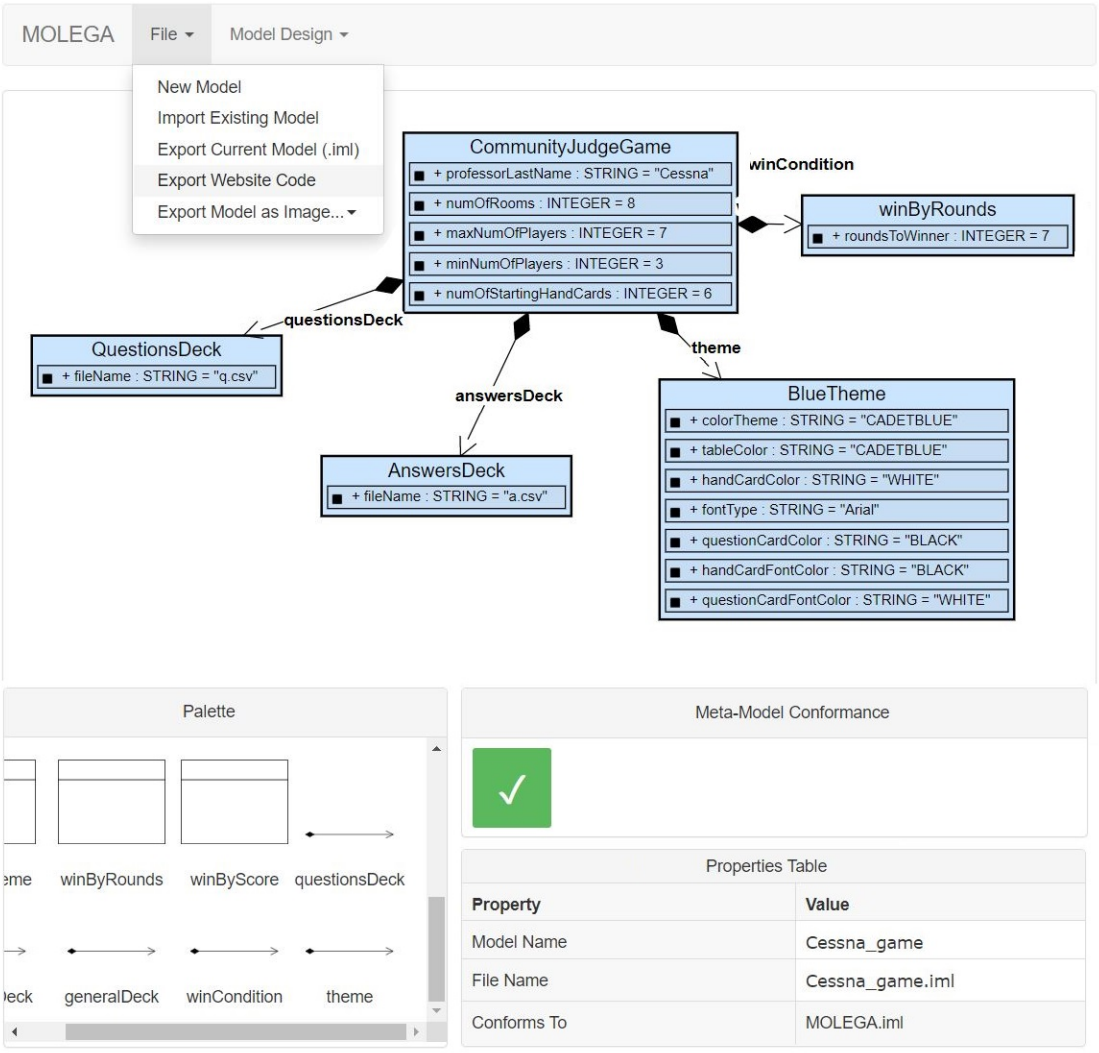


Figure 3.4: MOLEGA Web Editor with File Menu Icon Expanded (control panels typically appear to the right, as seen in Figure 4.2)

## 3.3 Code Generator

When a model is created that adheres to MOLEGA's metamodel rules, the code generation procecss is enabled. This feature is available through the usage of the "Generate Website Code" option in the web editor.

When this button is pressed, the model is parsed in to a String then read in an XML format. Using the contents of the XML items, customized content declared in the model attributes is parsed through and transformed into text. Constant files for the intended target, such as code controlling client to server communications, rule conformance, and winner checking, are added when the type of target is parsed and declared by the code generator.

The code generation process and output is further explained in Chapter 5.

# Chapter 4

# Domain Specific Modeling Language Design

**RQ1** in this thesis is "Can domain-specific modeling be used to create web-based educational card games?" This chapter defines such a language to represent these web-based educational card games. This language is called MOLEGA (Modeling Language for Educational Games). This chapter is organized by explaining the metamodel design for MOLEGA, followed by the web editor for MOLEGA and the choices associated with it.

## 4.1 Metamodel Design

The MOLEGA metamodel is composed of a modified, domain-specific, version of UML class diagrams. It is designed to represent two instances of card game: Community Judge and Relations. This metamodel can be seen in Figure 4.1.

The first major element of a MOLEGA model consists of a «Game» class. This abstract class encompasses all shared attributes of the different game types. Since «Game» is abstract, it cannot be used itself in a model but rather the different game types inherit the attributes from it. These inheriting classes are *CommunityJudgeGame* and *RelationsGame*. Every model must have only one of these classes.

The «winCondition» class is another abstract class. The *winByRounds* and *winByScore* classes inherit from this class. While they do not share any common attributes, they both serve the same purpose of acting as defining the win condition of the game object. *winByRounds* defines an attribute that sets the number of rounds a game is played before declaring a winner, while *winByScore* defines a different attribute that sets the number at which a winner is declared when reaching that score. Every model must have only one of these conditions.

The «Deck» class is an abstract class that has three inheriting classes: *QuestionsDeck*, *AnswersDeck*, and *GeneralDeck*. The *QuestionsDeck* and *AnswersDeck* classes are unique to Community Judge games, while the *GeneralDeck* class is unique to Relations games. Every model must have only one of each deck related to its game type. A model containing *CommunityJudgeGame* must have one *QuestionsDeck* and one *AnswersDeck*. Similarly, a model containing *RelationsGame* must have one *GeneralDeck*.

The «Theme» class is an abstract class that has multiple inheriting classes. Each inheriting class contains all of the same attributes, but with different default values. Three of these classes (*BlueTheme*, *RedTheme*, *GreenTheme*) have all values set for default, while one inheriting class called *CustomTheme* contains blank attribute values made with the intent of being filled in by a user creating the model. Three different preset color themes are provided in order to show multiple examples of how CSS colors can be used in the models, along with provide more than one choice

of color theme in the event that a user does not want to make their own custom theme, but does not want the default theme. Theme classes are the only optional part of MOLEGA's definition, a model being valid if containing one or none instances of these classes. When no instance of a «Theme» class exists in a model, a default light blue-based theme is applied to the target when code is generated.

This metamodel's design assists in answering the first research question presented in this thesis. Domain-specific modeling languages are meant to be useful in accurately representing domain software in an abstract way. Attributes in the model should be encompassing of the domain it is representing. For MOLEGA to be a useful DSML, attributes for the different class types were determined by first creating the example target games. Each example game contains variation points at which customization of the system can occur. These include colors of specific in-game elements and the way that a player wins a game. The classes and attributes of MOLEGA were designed based on those variation points.

This metamodel design allows all customizable components of the target to be represented in any model generated by a user. For either of the classes that inherit from «Game», the listed attributes (professor's name, number of rooms, maximum number of players, etc.) can be customized in order to meet the user's needs. Similarly, in the «Theme» classes different colors of different pieces of the target are also fully customizable, with multiple attributes allowing multiple different colors if so desired.

The design also makes it easier to add to the metamodel in the future. So long as the new addition of a game's ruleset requires a specific win condition and a deck of cards, the usage of inheritance structures in the metamodel only requires that the new class inherit from the «Game» class. Additionally, if this new game requires a different win condition than the ones already provided in the metamodel, then a new win condition class can be created to inherit from «winCondition».

One example of how the model could be extended is adding more game ruleset types. For example, the card game Pig works where all players start with 4 cards. The objective of the game is to get four of a kind before anyone else. If one person gets four of a kind, they recognize it by putting a finger on their nose. The last person to recognize this action gets a letter in the word Pig. When a player collects all letters in the word, they are eliminated from the game. A modified version of this game could easily be added to this metamodel by having a *PigGame* class inherit from the «Game» object, then adding a new «winCondition» subclass specifying a winner by last player standing.

## 4.2 Web Editor

The web editor for creating MOLEGA model instances is a modified version of the structural modeling web UI for IML [34]. While the IML web editor allows for the import of any IML-type metamodels to use for creating models, the MOLEGA web editor has the MOLEGA metamodel as the default and only metamodel to use. Therefore, MOLEGA's web editor can only be used to generate games as defined by MOLEGA's syntax. The UI for MOLEGA's web editor with no models created is seen in Figure 4.2.

This decision was made for a few reasons. IMLs structural modeling editor already had a

finished built-in model conformance check. This saved time in generating MOLEGA's web editor since more time could be spent on implementing error checking from elements such as user input into the models rather than building the entire conformance check from scratch. The *Metamodel Conformance* panel in the editor makes identifying model errors convenient and allows for a dynamic report of errors rather than returning all model conformance issues at the time of code generation.

The browser-based nature of the editor also makes it a good decision for use. Users are not required to download a piece of software they may not be familiar with and any required plugins or modules to make it work. They can instead go to the website where MOLEGA is hosted and do everything intended in the model creating and code generating process directly from the webpage. Since the target of the code generator is executable via web technologies, making the modeling editor as a whole web-based keeps the inputs and outputs consistent with its technologies.

Aside from restricting the IML editor to only support MOLEGA-defined models and related functions, other modifications were made in order to make the MOLEGA modeling editor more useful. These modifications were made in order to ensure that models created in the editor not only conform to the MOLEGA metamodel, but also that the details inside the model accurately represent a valid game. For example, checks that only one game object is defined in the modeling environment at a time were coded into the conformance checking system. Other modifications made to the web editor include checking that all numeric values are positive, making it impossible that the maximum number of players is less than the minimum number of players, and forcing any colors entered into the modeling environment to be in a format interpretable by a web program (e.g., a CSS or hexadecimal color code).

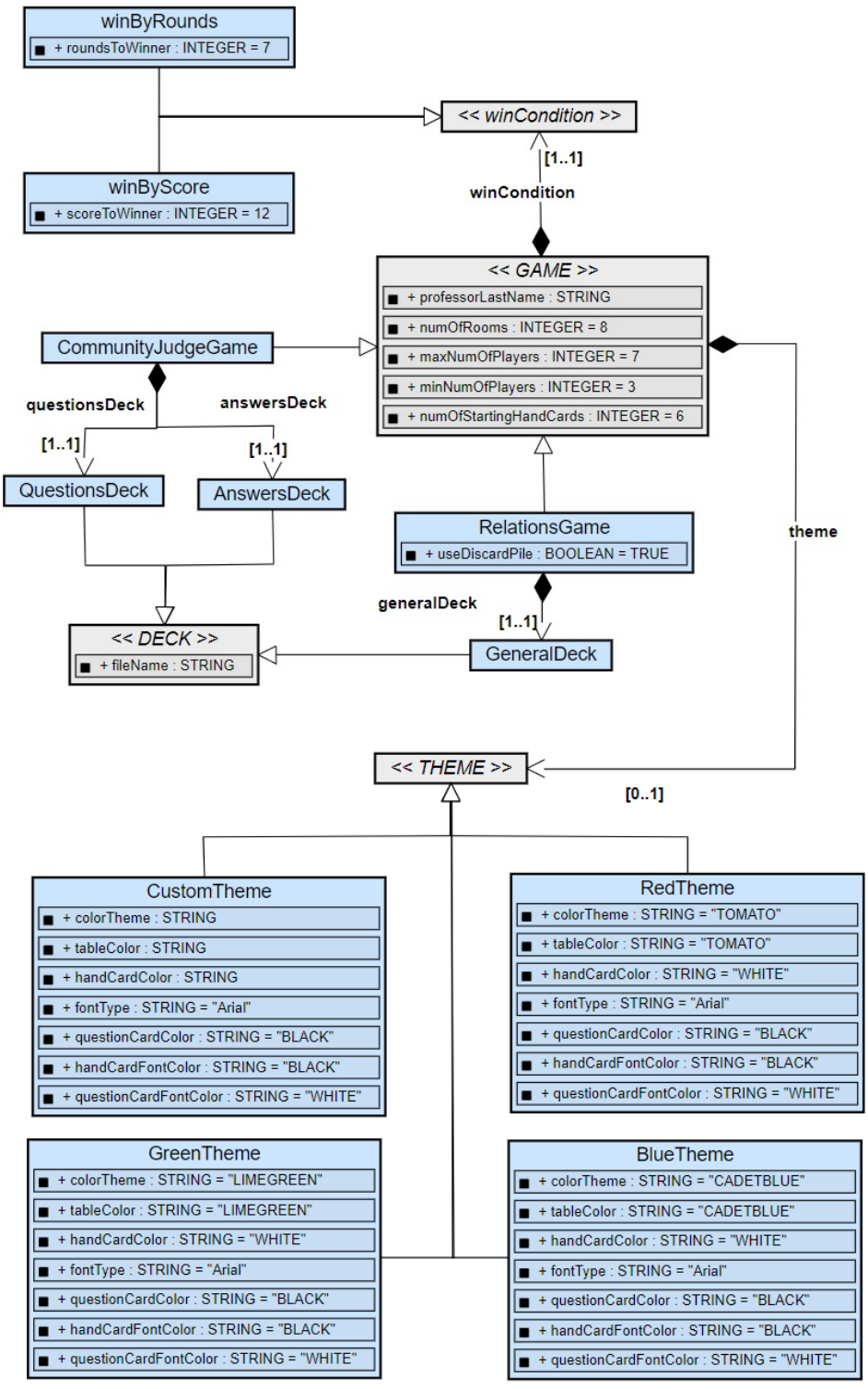


Figure 4.1: MOLEGA Metamodel

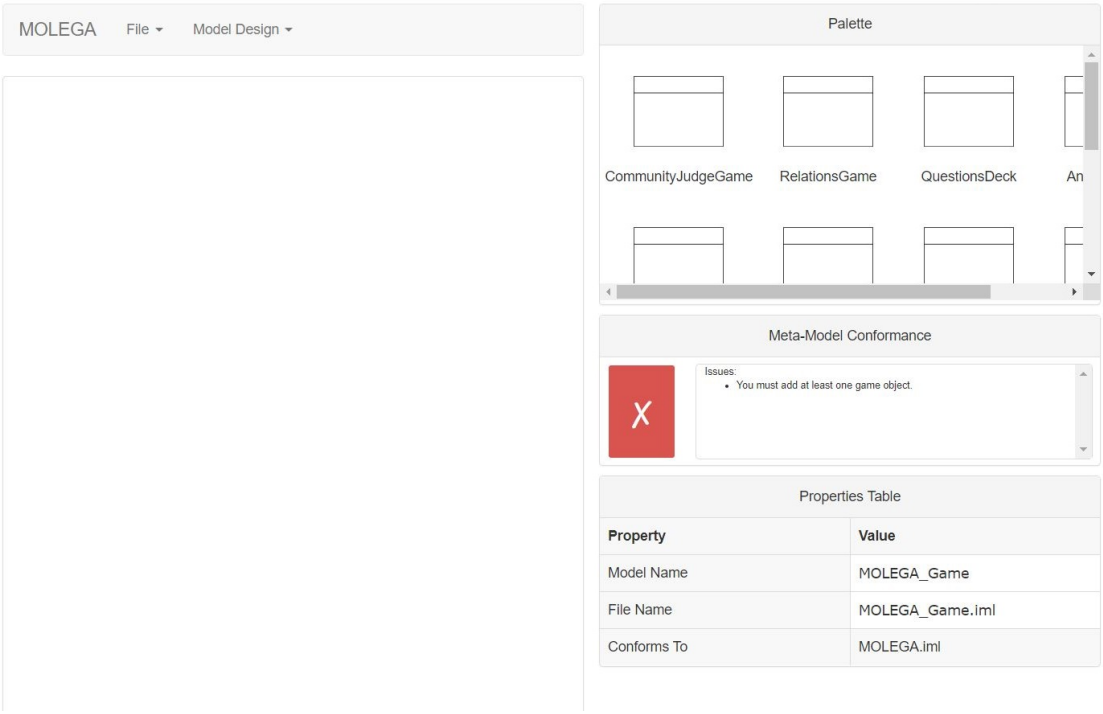


Figure 4.2: MOLEGA Web Editor

# Chapter 5

# Code Generation

This chapter explains the set up for a model-to-text code generator in order to answer the first part of **RQ2**, "Does a guided framework ensure the **generation** of consistently correct executable game code?" Code generation is implemented using the backend Javascript in the modeling software, generating web code in accordance to the model built in the editor. MOLEGA is currently able to generate web code for the two game types it supports: Community Judge and Relations. This section covers the prerequisites of a model in order to successfully generate code, then explains how the code generator works.

## 5.1 Prerequisites

In order for successful code generation to take place, a valid model must be created in the modeling software. A valid model is designated as such to a user through use of the *Metamodel Conformance* panel on the model editor's UI. When this panel displays a red X and a list of issues, the model does not adhere to the MOLEGA metamodel. When the panel displays a green checkmark, the model is considered valid and code generation is possible.

Another possible conformance issue that would result in an invalid model includes impossible attribute values, such as two card decks with the same name, the maximum number of players being an integer number less than the specified minimum number of players, or any integer value being zero or less than zero.

A valid Community Judge model consists of four required components and one optional component. A Community Judge game must have a *CommunityJudgeGame* object, a *winCondition* class, a *QuestionsDeck* class, and an *AnswersDeck* class. The optional component is a *Theme* class.

A valid Relations model consists of three required components and one optional component. A Relations game must have a *RelationsGame* object, a *winCondition* class, and a *GeneralDeck* class. The optional component is a *Theme* class.

## 5.2 Transformation Process

In this section, the process of code generation from model to code is explained. This explanation is followed by a summary of elements included in the final output of the code generator.

### 5.2.1 Code Generator

The code generator is triggered by a Javascript function when the "Export Website Code" menu option is clicked. A short psudocode of the code generation process from after xml searlization to zip file generation is seen in Figure 5.1. The first step in this process is the transform the model in the editor into a format where the contents can be parsed for class and attribute names and values. The serialization of the models leverages the existing IML algorithm. This serialization is then parsed into an xml format using jQuery's built-in *parseXML()* function in order to use the xml tags to navigate the model's elements rather than manually parsing the String output.

In order to assure that the classes being parsed are connected to one another using the appropriate composite relationships, the *Relation* tags are especially important. Each *Relation* tag is composed of multiple elements, the two most important for this task being the *source* element and the *destination* element. The *source* element lists the id for the source class of the composite, in the case of MOLEGA models this class is always either a *CommunityJudgeGame* class or a *RelationsGame* class. The *destination* element's value is the id for the connecting class. If a class exists in the model, but is not connected (see Appendix A) then there is no *Relation* tag with the id of that class. However, if the class is connected (see Appendix B), then the id will be present in a tag. This is important for determining which classes in the model conform to the metamodel (i.e., are connected with composites) and which classes are free-floating in the modeling editor, as free-floating classes are not considered conformance errors in accordance to the MOLEGA metamodel.

The code generator stores the ids of models in the active, non-floating classes in order to assure that only these classes are represented in the code generated. Attributes contained in the model are sorted into one of three String variables: a style variable, an app variable, or a constants variable. The style variable contains all of the text meant to make up the style.css file for the code output This mostly includes background colors of webpage elements and other colors that do not change. The app variable contains the global variables required for the server to run the game code. This consists of attributes such as the number of rooms, the player limits for each room, and the file names for the csv card files. Finally, the constants variable contains the client side variables which are independently kept for each connecting client. These include the card color attributes and the font which is used for displaying information to the client.

When all active classes have been parsed, the above String variables are then turned into code files in order to be added to a zip file for export. This is done using JSZip [1], a Javascript-specific library for generating and modifying zip files.

While the customized files are generated from the xml parsing, other files required for the target to function correctly are constants that persist for any version of model. These constant files are copied from a directory on the MOLEGA server containing example versions of the complete target game codes. The code generator only generates Community Judge-specific files for models that have the *CommunityJudgeGame* class. Similarly, Relations-specific files are only generated for models that contain the *RelationsGame* class. This is done using XMLHttpRequest objects in order to access the bodies of these example files. In order to avoid these asynchronous requests from ending after the export has taken place, these requests are nested on one another in order to

[1] https://stuk.github.io/jszip/

assure that all required files are called for and generated into code files for the code generator before a full export occurs.

```
activeClassIds = []
constants = ""
style = ""
app = ""

for relation in all Relation elements:
  if ! activeClassIds includes relation.destination:
    activeClassIds.push(relation.destination)
  if ! activeClassIds includes relation.source:
    activeClassIds.push(relation.source)

for class in all Class elements:
  if activeClassIds includes class.id:
    if class.name includes "CommunityJudge":
      save "CommunityJudge" as gameType
    else if class.name includes "Relations":
      save "Relations" as gameType
    for attr in class.attributes:
      format each attr in "var " + attr.name + " = " + attr.value + ";" format
      place attribute in the correct String (constants, style, app)
if no Theme class exists:
  apply default theme to constants and style

if gameType == "Community Judge":
  generate Community Judge-specific static files
else:
  generate Relations-specific static files

generate remaining files (readme, package.json, constants, style)
export zip file
```

Figure 5.1: Code Generation Pseudocode

### 5.2.2 Generation Target - Web Game Code

The result of the code generation is a zip file of executable web-based game code. This file is automatically started as a download on the local machine of the user using the web editor when the editor is prompted to generate code. In the case of these games, executable means that all information required to host the code on a web server is included in the generated zip file. The main file that ensures this is a package.json file, which contains information for all external npm packages required to run the Node.js app.js server located at the root of the zip file. A short readme (see Appendix C) is also included in the root of the zip file, containing information about how to host the app.js file on a server along with information about how to edit the csv card files properly.

Due to the way that code generation takes place, there are three code files in the generated zip file in the total eight required for game play that code differs in from one game to another. A majority of customization is implemented in the constants.js file, which controls elements such as colors that change in accordance to client interaction in-game, along with font type and the professor's name displayed at the top of the webpage. When the model is transformed during code generation, a majority of the attributes under the «Game» inheriting class are put in the app.js server file, while the other customizable attributes are stored in either constants.js or the style.css stylesheet. An example of a valid model and the corresponding constants.js generated by the code generator are visible in Figures 5.2 and 5.3. Figure 5.2 displays a valid model due to the presence of all four required classes for a Community Judge model (*CommunityJudgeGame*, *winCondition*, *QuestionsDeck*, and *AnswersDeck*, along with these required classes being connected with composition relationships. The addition of the *CustomTheme* class is done in accordance to the MOLEGA metamodel, so the theme does not break the validity of this model.

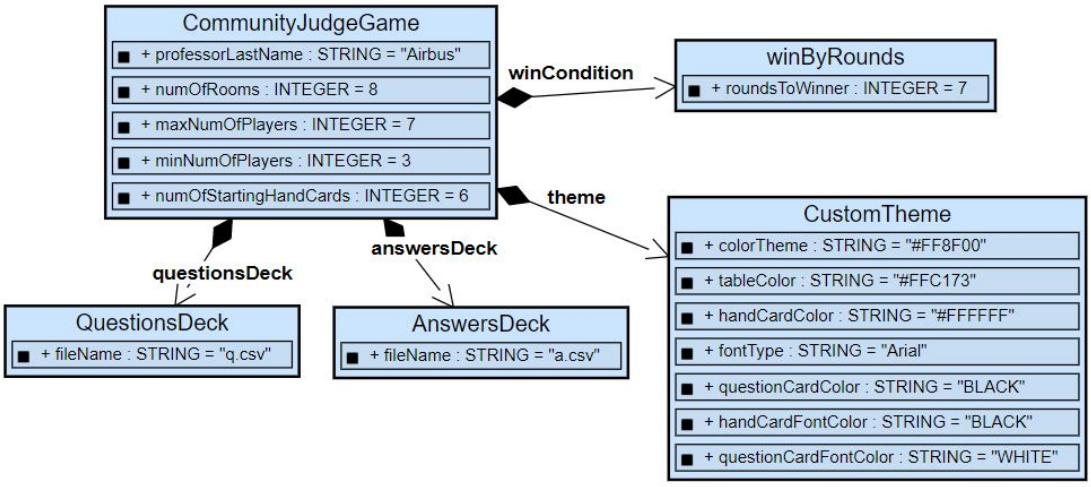


Figure 5.2: Valid Model Example

```
1  // A good card width-height ratio is 8:11 in my opinion
2  var cardWidth = 110;
3  var cardHeight = 150;
4  var spaceBetweenCards = 20;
5  var cardFontSize = 20;
6  var tableFontSize = 25;
7  var typeOfGame = "Community Judge Game";
8  var professorLastName = "Airbus";
9  var numOfCardsInHand = 6;
10 var winByRounds = true;
11 var numOfRounds = 7;
12 var roomTableSelectColour = "#FF8F00";
13 var handCardColour = "#FFFFFF";
14 var fontType = "Arial";
15 var questionCardColour = "BLACK";
16 var fontColour = "BLACK";
17 var questionFontColour = "WHITE";
```

Figure 5.3: Valid Model's constants.js Code

# Chapter 6

# Evaluation

The second part of **RQ2** "Does a guided framework ensure the generation of consistently correct executable game code?" is asking whether the code generated in the previous chapter is consistently correct and executable. To achieve this, the evaluation of MOLEGA focused on verification, or the correctness of the code generator's output in comparison to the expected output. A systematic evaluation plan was created in order to test and verify both positive and negative error cases. This chapter first explains the design of the MOLEGA evaluation, transitions into the results of this evaluation, and ends with a short discussion of the benefits and drawbacks of this type of evaluation.

## 6.1 Experimental Design

The design of MOLEGA's evaluation is split into two major categories: valid model outputs and invalid model outputs. These two categories could then further be split again by specifying the target type to be tested: Community Judge-type models and Relations-type models. This taxonomy covers all cases of possible models able to be generated using MOLEGA. It also makes it easier to see if any tests are missing, since splitting the tests by the intended model target allows target-specific tests to be included while ensuring the duplicate tests are properly differentiated.

In valid models, three boolean criteria were ensured in each model: "Is it Valid?", "Is it Correct?", and "Does it Run?". A model is considered valid if it is successfully generated by the code generator when the web editor is instructed to generate website code. A model is considered correct if it outputs the expected code. A model must first be valid in order for it to be correct. If a model is valid, but not correct, then code is generated that is not customized properly in the way that the model intended. A model that is not valid has no need to be compared to expected output, since no code is output from an invalid model.

In invalid models, only one criterion was tested: "Is it Valid?". In these tests, the passing answer to this question should be "No". Furthermore, since invalid models should not generate code, they have no need to be correct. If the modeling editor is instructed to generate website code, it passed this test if it refused to output code and instead threw an error message to the user.

For each type of target multiple tests were written out in order to test the correctness of all possible valid model components. Similarly, actions that would result in invalid models were also recorded and tested. For each valid model, one component was tested at a time in order to assure that the component change in the model accurately resulted in the corresponding target code. For example, the base model for a Community Judge game is one that has a game object, a win condition of *winByRounds*, a *QuestionsDeck*, and an *AnswersDeck*. In this model, all default values are kept the same while adding valid input to the default missing attributes in order to make the model

valid. This model can be seen in Figure 6.1. For the valid model test "Change Num of Rooms," the *CommunityJudgeGame* attribute *numOfRooms* should be changed to another valid number. In order to pass the valid models test, not only should successful code generation occur, but the variable in the code that controls the number of game rooms should accurately reflect the change in the model.

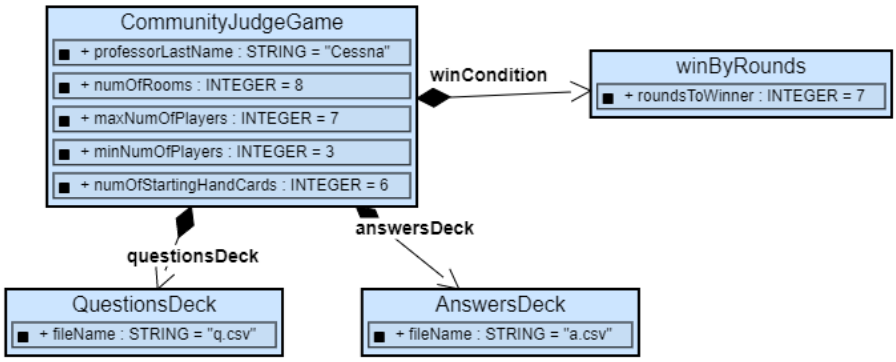


Figure 6.1: Community Judge Base Model

In order to test that generated code is in fact correct, the expected output for each model transformation was created manually. While tedious, this ensures that the expected files align with what a human user would expect to encounter rather than relying on another machine-generated file that could be generating incorrect content.

Any files that are intended to be customized were physically created in order to compare generated files to the manual expected files. A bash script (see Appendix D) simplifies this comparison task: unzipping the generated code and comparing all custom files in the generated code to the manually created expected files. If a conflict arises, the script notes as such along with which files are not the same. No matter if all tests pass or fail for that model, after completing one generated zip file the script then moves on to the next one, only requiring one execution of the script in order to check all valid model code generations. This removes a step of possible human error. So long as the generated zip file and the expected files directory have the same name then the correct files are checked for discrepancies.

In order to assure that valid models produced *executable* web game code, after passing the model validity and code correctness tests, each code file was launched in a local host environment. This tests the "Does it Run?" criterion. While a game file with code correctness should theoretically launch and play with no issues, physically launching each game gives extra reassurance that each valid model creates a working target.

Table 6.1: Valid Community Judge Model Results

| **Valid Community Judge** | | | |
|---|---|---|---|
| **Test Case** | **Is it Valid?** | **Is it Correct?** | **Does it Run?** |
| **Base Model** | | | |
| Base Valid Model (no theme, win by rounds) | ✓ | ✓ | ✓ |
| **Class Changes** | | | |
| Use Blue Theme | ✓ | ✓ | ✓ |
| Use Red Theme | ✓ | ✓ | ✓ |
| Use Green Theme | ✓ | ✓ | ✓ |
| Use Custom Theme (CSS, all valid) | ✓ | ✓ | ✓ |
| Use Custom Theme (hexadecimal, all valid) | ✓ | ✓ | ✓ |
| Use Win By Score | ✓ | ✓ | ✓ |
| Extra Theme (Not Connected) | ✓ | ✓ | ✓ |
| Extra Win Condition (Not Connected) | ✓ | ✓ | ✓ |
| Extra Questions Deck (Not Connected) | ✓ | ✓ | ✓ |
| Extra Answers Deck (Not Connected) | ✓ | ✓ | ✓ |
| **Attribute Changes** | | | |
| Change Prof Name | ✓ | ✓ | ✓ |
| Change Num of Rooms | ✓ | ✓ | ✓ |
| Change Max Players | ✓ | ✓ | ✓ |
| Change Min Players | ✓ | ✓ | ✓ |
| Change Num Starting Cards | ✓ | ✓ | ✓ |
| Change Deck Name (Questions) | ✓ | ✓ | ✓ |
| Change Deck Name (Answers) | ✓ | ✓ | ✓ |
| Change Color Theme | ✓ | ✓ | ✓ |
| Change Table Color | ✓ | ✓ | ✓ |
| Change Hand Card Color | ✓ | ✓ | ✓ |
| Change Font | ✓ | ✓ | ✓ |
| Change Question Card Color | ✓ | ✓ | ✓ |
| Change Hand Card Color | ✓ | ✓ | ✓ |
| Change Question Card Font Color | ✓ | ✓ | ✓ |
| Change Rounds to Winner | ✓ | ✓ | ✓ |
| Change Score to Winner (use Win by Score) | ✓ | ✓ | ✓ |

Table 6.2: Valid Relations Model Results

| **Valid Relations** | | | |
|---|---|---|---|
| **Test Case** | **Is it Valid?** | **Is it Correct?** | **Does it Run?** |
| **Base Model** | | | |
| Base Valid Model (no theme, win by rounds) | ✓ | ✓ | ✓ |
| **Class Changes** | | | |
| Use Blue Theme | ✓ | ✓ | ✓ |
| Use Red Theme | ✓ | ✓ | ✓ |
| Use Green Theme | ✓ | ✓ | ✓ |
| Use Custom Theme (CSS, all valid) | ✓ | ✓ | ✓ |
| Use Custom Theme (hexadecimal, all valid) | ✓ | ✓ | ✓ |
| Use Win By Score | ✓ | ✓ | ✓ |
| Extra Theme (Not Connected) | ✓ | ✓ | ✓ |
| Extra Win Condition (Not Connected) | ✓ | ✓ | ✓ |
| Extra General Deck (Not Connected) | ✓ | ✓ | ✓ |
| **Attribute Changes** | | | |
| Change Prof Name | ✓ | ✓ | ✓ |
| Change Num of Rooms | ✓ | ✓ | ✓ |
| Change Max Players | ✓ | ✓ | ✓ |
| Change Min Players | ✓ | ✓ | ✓ |
| Change Num Starting Cards | ✓ | ✓ | ✓ |
| Change Use Discard Pile to False | ✓ | ✓ | ✓ |
| Change Deck Name (General) | ✓ | ✓ | ✓ |
| Change Color Theme | ✓ | ✓ | ✓ |
| Change Table Color | ✓ | ✓ | ✓ |
| Change Hand Card Color | ✓ | ✓ | ✓ |
| Change Font | ✓ | ✓ | ✓ |
| Change Question Card Color (used for Selected Card Color) | ✓ | ✓ | ✓ |
| Change Hand Card Color | ✓ | ✓ | ✓ |
| Change Question Card Font Color (used for Selected Card Color) | ✓ | ✓ | ✓ |
| Change Rounds to Winner | ✓ | ✓ | ✓ |
| Change Score to Winner (use Win by Score) | ✓ | ✓ | ✓ |

Table 6.3: Invalid Community Judge Model Results

| **Invalid Community Judge** | | | |
|---|---|---|---|
| **Test Case** | **Is it Valid?** | **Is it Correct?** | **Does it Run?** |
| **Missing Classes** | | | |
| Missing Game | ✗ | N/A | N/A |
| Missing Questions Deck | ✗ | N/A | N/A |
| Missing Answers Deck | ✗ | N/A | N/A |
| Missing Win Condition | ✗ | N/A | N/A |
| **Missing Attribute Values** | | | |
| Missing Prof Name | ✗ | N/A | N/A |
| Missing Questions Deck Name | ✗ | N/A | N/A |
| Missing Answers Deck Name | ✗ | N/A | N/A |
| Missing Values from Custom Theme | ✗ | N/A | N/A |
| Missing/Deleted Other Attributes (e.g., numOfRooms) | ✗ | N/A | N/A |
| **Surplus Relations** | | | |
| Surplus Themes | ✗ | N/A | N/A |
| Surplus Win Conditions | ✗ | N/A | N/A |
| Surplus Questions Decks | ✗ | N/A | N/A |
| Surplus Answers Decks | ✗ | N/A | N/A |
| Surplus Games | ✗ | N/A | N/A |
| **Invalid Inputs** | | | |
| Max Players Less Than Min Players | ✗ | N/A | N/A |
| Both Decks the Same File Name | ✗ | N/A | N/A |
| CommJudge Attributes Zero or Negative | ✗ | N/A | N/A |
| Win Rounds Zero or Negative | ✗ | N/A | N/A |
| Win Score Zero or Negative | ✗ | N/A | N/A |
| Invalid CSS/Hex Color | ✗ | N/A | N/A |
| Invalid Font Type | ✗ | N/A | N/A |
| Invalid Composite Connection | ✗ | N/A | N/A |

Table 6.4: Invalid Relations Model Results

| **Invalid Relations** | | | |
|---|---|---|---|
| **Test Case** | **Is it Valid?** | **Is it Correct?** | **Does it Run?** |
| **Missing Classes** | | | |
| Missing Game | ✗ | N/A | N/A |
| Missing General Deck | ✗ | N/A | N/A |
| Missing Win Condition | ✗ | N/A | N/A |
| **Missing Attribute Values** | | | |
| Missing Prof Name | ✗ | N/A | N/A |
| Missing General Deck Name | ✗ | N/A | N/A |
| Missing Values from Custom Theme | ✗ | N/A | N/A |
| Missing/Deleted Other Attributes (e.g., numOfRooms) | ✗ | N/A | N/A |
| **Surplus Relations** | | | |
| Surplus Themes | ✗ | N/A | N/A |
| Surplus Win Conditions | ✗ | N/A | N/A |
| Surplus General Decks | ✗ | N/A | N/A |
| Surplus Games | ✗ | N/A | N/A |
| **Invalid Inputs** | | | |
| Max Players Less Than Min Players | ✗ | N/A | N/A |
| Relations Attributes Zero or Negative | ✗ | N/A | N/A |
| Win Rounds Zero or Negative | ✗ | N/A | N/A |
| Win Score Zero or Negative | ✗ | N/A | N/A |
| Invalid CSS/Hex Color | ✗ | N/A | N/A |
| Invalid Font Type | ✗ | N/A | N/A |
| Invalid Composite Connection | ✗ | N/A | N/A |

## 6.2 Results

Tables 6.1 through 6.4 display the outcomes of all tests conducted. These tests were generated in order to cover all edge and error cases possible.

Table 6.1 displays the valid Community Judge models tested in accordance to adhering to the MOLEGA metamodel, along with their correctness in expected output. Table 6.2 does the same but with the valid Relations models. Table 6.3 lists tests concerning the conformance checking on the model editor, testing different attributes contributing to invalid Community Judge models. Table 6.4 does the same conformance checking concerning invalid Relations models.

## 6.3 Discussion

All tests for valid models passed. These tests evaluated the code generator's ability not only to export code in response to a valid mode, but also that the code generated was correct. This supports an affirmative answer both of this thesis's research questions. Since the purpose of these verification tests was to ensure the that code generator behaved as expected for the usage of each modeling ele-

ment separately, the fact that all tests passed the generated versus expected tests demonstrates that this process is not only successfully transforms a domain-specific modeling language into a web-based educational game, but that the code for this game is consistently correct. Also, considering that all tests for generated code successfully ran, this code can also be claimed as executable.

In addition, all tests passed for invalid models as well. Since these tests were intended to end in failure, the presence of all "X" answers in the invalid models tables qualifies as a pass. In reference to the research questions presented by this thesis, this supports an affirmative answer to the second research question. In order for the code generator to generate a correct executable game, the model must first be correct. If this model is not correct, and does not align with its metamodel, then no transformation should occur in order to assure that only consistently correct code is generated.

With respect to **RQ1**, it is clear from the evaluation experiments that MOLEGA can be used to create web-based educational card games, thus affirming that domain-specific modeling is a useful tool for this application. MOLEGA is capable of representing every possible variation of the selected game variants, indicating there are no significant aspects lost through the abstraction provided through modeling. Further, with respect to **RQ2**, the evaluation confirms that through the guided editor and strict metamodel conformance requirements, every valid model where a game was expected to be produced led to consistently correct and executable code. Furthermore, any models that contained invalid game elements would not lead to broken or incomplete code, preventing users from attempting to generate code that would not work. In summary, through this evaluation, it is shown that not only is it possible to use domain specific modeling to create educational games (**RQ1**), but the provision of a guided modeling approach ensures that only correct and executable code will ever be provided to users (**RQ2**).

# Chapter 7

# Conclusion

This chapter discusses some limitations of MOLEGA in its current form, followed by future work to build upon this project. To end, a summary of this project's motivations and work is provided.

## 7.1 Limitations

As mentioned before, MOLEGA is limited to representing two specific rulesets for the target game code. Considering the sheer number of card game rulesets that exist, this is a major limitation to this work. MOLEGA's metamodel is quite simple with some directions for expansion, but also rather restricted in how it can be expanded upon. While adding a new multiplayer game type or a new kind of win condition would be rather simple, updating the code generator to include this new variant would require a lot of work, since the ruleset web game code would need to be built from scratch. While the constant files could be kept and built upon, it would take someone familiar with Javascript development and client-server communications to implement a full new ruleset.

Another limitation of this work is that it verifies the functionality of the web-based framework and code generation processes, but does not involve any user studies for testing usability or enjoyability of either the web editor or the target games generated. The addition of user tests and modification of the modeling editor and the target games in accordance to feedback from those tests would make this project claimable as a user tool rather than a proof of concept.

Verification tests for this project were done by comparing generated code to manually created expected code. As with any work with manually created elements, the possibility of human error exists. This was considered a risk worth taking, however, since writing a script to generate an expected output not only would take more resources than available at the time, but would also be acting as a code generator itself. Two code generators compared against one another could lead to consistent errors if the expected output generator is set up incorrectly.

## 7.2 Future Work

This thesis implemented a modeling language for representing two examples of educational card games, along with a code generation process to transform a domain-specific model into web-based game code. This work could be improved upon, and therefore lead to future work, in a few different ways.

Usability tests and modifications would certainly add value to this project. Being able to claim the usability of a modeling language editor intended for non-technically inclined users would be

beneficial. Alongside usability tests for the web editor, usability and user play experience of the target games from the code generation process would also improve the impact of this work.

Work to further encompass more varieties of card game rulesets would also be valid future work. Refactoring the current MOLEGA metamodel to fit multiplayer and single player games or to allow the combination of game rulesets would increase the customizable aspects of this algorithm, allowing more control over the output to the user creating a model.

## 7.3 Summary

Model-driven software engineering is a sub-domain of software engineering that is often not leveraged to its full potential, especially in areas where it could be most useful. In specific domain areas, such as game design, models can help to abstract problems in a way where a person with limited technical abilities can understand the problem. While not directly involved in game design, educators can benefit from the accessibility of games for use in their classrooms. However, educators often do not have the technical design or implementation skills to create these games themselves. Through domain-specific modeling, these potential users can leverage MDSE to obtain functional games having only had to specify the custom aspects of their game via a graphical editor.

MOLEGA's web editor serves as this graphical editor, possessing the ability to represent two types of card game while providing various customizable features to the games. Most importantly, the content on the cards in the game is customizable, allowing games generated by MOLEGA's code generator to be used not only as generic card games, but also as educational tools by replacing the default content with specific educational content. This generated code is formatted in a way where, with access to web hosting services, the code can be hosted with very little outside packages or modules and can be accessed by any device with browser capabilities.

This code generation process has been tested using a systematic approach in order to cover all edge cases for both valid and invalid models. For models that are valid in accordance to the conformance check, code is shown not only to be successfully generated, but also tested to be correct in accordance to a manually created expected output. For models that should not generate code, they fail to run the code generator and give the user an error message as intended. These tests ensured that not only MOLEGA can be used to represent and generated web-based educational card games, but also that any code generated by the algorithm is consistently correct and executable in a server environment.

While still requiring a little technical knowledge or access to a technical person in order to host the generated game code, MOLEGA provides an abstract approach to customized web code generation that requires little to no technical knowledge to generate an educational card game. This research displays that a domain-specific modeling language can be used to represent and create web-based educational games. It also proves that given a guided framework, this code generation is capable of generating consistently correct and executable game code.

# Appendix A

# IML Output for Model with a Theme Class Not Connected

```
<iml version="0.1">
   <StructuralModel name="no-theme" conformsTo="MOLEGA.iml" routingMode="simpleRoute" >
     <Classes>
     <Class name="CommunityJudgeGame" isAbstract="FALSE" x="318" y="46" id="a49356ce-1be4-47a6-b689-5eba607642b4">
       <Attribute visibility="PUBLIC" name="professorLastName" type="STRING" value="Cessna" lowerBound="1" upperBound="1" position="1" />
       <Attribute visibility="PUBLIC" name="numOfRooms" type="INTEGER" value="8" lowerBound="1" upperBound="1" position="2" />
       <Attribute visibility="PUBLIC" name="maxNumOfPlayers" type="INTEGER" value="7"  lowerBound="1" upperBound="1" position="3" />
       <Attribute visibility="PUBLIC" name="minNumOfPlayers" type="INTEGER" value="3"  lowerBound="1" upperBound="1" position="4" />
       <Attribute visibility="PUBLIC" name="numOfStartingHandCards" type="INTEGER" value="6" lowerBound="1" upperBound="1" position="5" />
     </Class>
     <Class name="QuestionsDeck" isAbstract="FALSE" x="47" y="213" id="e1dd48bb-7149-4ebf-9e24-628bfd0e6e89">
       <Attribute visibility="PUBLIC" name="fileName" type="STRING" value="q.csv" lowerBound="1" upperBound="1" position="1" />
     </Class>
     <Class name="AnswersDeck" isAbstract="FALSE" x="271" y="351" id="2af3f384-d6fb-4d50-a295-f30cb1b5ded8">
       <Attribute visibility="PUBLIC" name="fileName" type="STRING" value="a.csv" lowerBound="1" upperBound="1" position="1" />
     </Class>
     <Class name="winByRounds" isAbstract="FALSE" x="703" y="110" id="58732736-36b2-4858-b71f-ebd6f76e30d8">
       <Attribute visibility="PUBLIC" name="roundsToWinner" type="INTEGER" value="7"  lowerBound="1" upperBound="1" position="1" />
     </Class>
     <Class name="BlueTheme" isAbstract="FALSE" x="588" y="261" id="ca806aea-ccfc-4447-8ab1-ad36799181f7">
       <Attribute visibility="PUBLIC" name="colorTheme" type="STRING" value="CADETBLUE" lowerBound="1" upperBound="1" position="1" />
       <Attribute visibility="PUBLIC" name="tableColor" type="STRING" value="CADETBLUE" lowerBound="1" upperBound="1" position="2" />
```

```
23          <Attribute visibility="PUBLIC" name="handCardColor" type="STRING" value="WHITE
                ↪ " lowerBound="1" upperBound="1" position="3" />
24          <Attribute visibility="PUBLIC" name="fontType" type="STRING" value="Arial"
                ↪ lowerBound="1" upperBound="1" position="4" />
25          <Attribute visibility="PUBLIC" name="questionCardColor" type="STRING" value="
                ↪ BLACK" lowerBound="1" upperBound="1" position="5" />
26          <Attribute visibility="PUBLIC" name="handCardFontColor" type="STRING" value="
                ↪ BLACK" lowerBound="1" upperBound="1" position="6" />
27          <Attribute visibility="PUBLIC" name="questionCardFontColor" type="STRING"
                ↪ value="WHITE" lowerBound="1" upperBound="1" position="7" />
28        </Class>
29      </Classes>
30
31      <Relations>
32        <Relation source="a49356ce-1be4-47a6-b689-5eba607642b4" destination="2af3f384-
              ↪ d6fb-4d50-a295-f30cb1b5ded8" type="COMPOSITION" name="answersDeck"
              ↪ lowerBound="1" upperBound="1" nameDistance="0.33" boundDistance="0"
              ↪ nameOffset="-15" boundOffset="0" />
33        <Relation source="a49356ce-1be4-47a6-b689-5eba607642b4" destination="58732736-36
              ↪ b2-4858-b71f-ebd6f76e30d8" type="COMPOSITION" name="winCondition"
              ↪ lowerBound="1" upperBound="1" nameDistance="0.33" boundDistance="0"
              ↪ nameOffset="-15" boundOffset="0" />
34        <Relation source="a49356ce-1be4-47a6-b689-5eba607642b4" destination="e1dd48bb
              ↪ -7149-4ebf-9e24-628bfd0e6e89" type="COMPOSITION" name="questionsDeck"
              ↪ lowerBound="1" upperBound="1" nameDistance="0.33" boundDistance="0"
              ↪ nameOffset="-15" boundOffset="0" />
35      </Relations>
36    </StructuralModel>
37  </iml>
```

# Appendix B

# IML Output for Model with a Theme Class Connected

```
<iml version="0.1">
   <StructuralModel name="has-theme" conformsTo="MOLEGA.iml" routingMode="simpleRoute ">
     <Classes>
     <Class name="CommunityJudgeGame" isAbstract="FALSE" x="318" y="46" id="a49356ce-1be4-47a6-b689-5eba607642b4">
       <Attribute visibility="PUBLIC" name="professorLastName" type="STRING" value="Cessna" lowerBound="1" upperBound="1" position="1" />
       <Attribute visibility="PUBLIC" name="numOfRooms" type="INTEGER" value="8" lowerBound="1" upperBound="1" position="2" />
       <Attribute visibility="PUBLIC" name="maxNumOfPlayers" type="INTEGER" value="7"  lowerBound="1" upperBound="1" position="3" />
       <Attribute visibility="PUBLIC" name="minNumOfPlayers" type="INTEGER" value="3"  lowerBound="1" upperBound="1" position="4" />
       <Attribute visibility="PUBLIC" name="numOfStartingHandCards" type="INTEGER" value="6" lowerBound="1" upperBound="1" position="5" />
     </Class>
     <Class name="QuestionsDeck" isAbstract="FALSE" x="47" y="213" id="e1dd48bb-7149-4ebf-9e24-628bfd0e6e89">
       <Attribute visibility="PUBLIC" name="fileName" type="STRING" value="q.csv" lowerBound="1" upperBound="1" position="1" />
     </Class>
     <Class name="AnswersDeck" isAbstract="FALSE" x="271" y="351" id="2af3f384-d6fb-4d50-a295-f30cb1b5ded8">
       <Attribute visibility="PUBLIC" name="fileName" type="STRING" value="a.csv" lowerBound="1" upperBound="1" position="1" />
     </Class>
     <Class name="winByRounds" isAbstract="FALSE" x="703" y="110" id="58732736-36b2-4858-b71f-ebd6f76e30d8">
       <Attribute visibility="PUBLIC" name="roundsToWinner" type="INTEGER" value="7"  lowerBound="1" upperBound="1" position="1" />
     </Class>
     <Class name="BlueTheme" isAbstract="FALSE" x="588" y="261" id="ca806aea-ccfc-4447-8ab1-ad36799181f7">
       <Attribute visibility="PUBLIC" name="colorTheme" type="STRING" value="CADETBLUE" lowerBound="1" upperBound="1" position="1" />
       <Attribute visibility="PUBLIC" name="tableColor" type="STRING" value="CADETBLUE" lowerBound="1" upperBound="1" position="2" />
```

```
23          <Attribute visibility="PUBLIC" name="handCardColor" type="STRING" value="
                ↪ WHITE" lowerBound="1" upperBound="1" position="3" />
24          <Attribute visibility="PUBLIC" name="fontType" type="STRING" value="Arial"
                ↪ lowerBound="1" upperBound="1" position="4" />
25          <Attribute visibility="PUBLIC" name="questionCardColor" type="STRING" value="
                ↪ BLACK" lowerBound="1" upperBound="1" position="5" />
26          <Attribute visibility="PUBLIC" name="handCardFontColor" type="STRING" value="
                ↪ BLACK" lowerBound="1" upperBound="1" position="6" />
27          <Attribute visibility="PUBLIC" name="questionCardFontColor" type="STRING"
                ↪ value="WHITE" lowerBound="1" upperBound="1" position="7" />
28        </Class>
29      </Classes>
30
31      <Relations>
32        <Relation source="a49356ce-1be4-47a6-b689-5eba607642b4" destination="2af3f384-
              ↪ d6fb-4d50-a295-f30cb1b5ded8" type="COMPOSITION" name="answersDeck"
              ↪ lowerBound="1" upperBound="1" nameDistance="0.33" boundDistance="0"
              ↪ nameOffset="-15" boundOffset="0" />
33        <Relation source="a49356ce-1be4-47a6-b689-5eba607642b4" destination="58732736-36
              ↪ b2-4858-b71f-ebd6f76e30d8" type="COMPOSITION" name="winCondition"
              ↪ lowerBound="1" upperBound="1" nameDistance="0.33" boundDistance="0"
              ↪ nameOffset="-15" boundOffset="0" />
34        <Relation source="a49356ce-1be4-47a6-b689-5eba607642b4" destination="e1dd48bb
              ↪ -7149-4ebf-9e24-628bfd0e6e89" type="COMPOSITION" name="questionsDeck"
              ↪ lowerBound="1" upperBound="1" nameDistance="0.33" boundDistance="0"
              ↪ nameOffset="-15" boundOffset="0" />
35        <Relation source="a49356ce-1be4-47a6-b689-5eba607642b4" destination="ca806aea-
              ↪ ccfc-4447-8ab1-ad36799181f7" type="COMPOSITION" name="theme" lowerBound="
              ↪ 0" upperBound="1" nameDistance="0.33" boundDistance="0" nameOffset="-15"
              ↪ boundOffset="0" />
36      </Relations>
37    </StructuralModel>
38  </iml>
```

# Appendix C

# Example Readme File for Community Judge

```
Thanks for generating and downloading <profName>'s 's Community Judge Game!

Important information:
The app.js file is the backend server which controls most game functions. All other
    files are intended for use from the client perspective (index.html).

Included with this game in the cardFiles directory are example csv files that contain
     card information. For the Community Judge game, each card is its own row. In
    order for the game to work properly, any card information should be placed one
    per row in column 1, as seen in the example card file. The labels "Questions" and
     "Answers" MUST stay at the top of each card file column.

Do not change the name of any files in the cardFiles directory.
Contents may be changed so long as they follow the above rules.

To Host on a Server:
Link to instructions - https://www.digitalocean.com/community/tutorials/how-to-set-up
    -a-node-js-application-for-production-on-ubuntu-14-04
Note: Above instructions should only include "Install Node.js" (be sure to install
    version 12.18.2), "Install PM2", and
"Manage Application with PM2" (using app.js instead of hello.js). If running app.js
    results in missing package errors, run the command "npm install"

This set of game files uses Node.js version 12.18.2
package.json is a file that contains all dependency packages required to run this
    game. Use npm install before launching the node app to ensure that everything
    runs as intended.
```

# Appendix D

# Output vs. Expected Comparison Script

```
#!bin/bash

ALLPASS=0;

cd ~/Downloads/zips

for f in *; do
       PASS=0
       FILE=$f

       if [ ${#FILE} -eq 0 ]
       then
              echo "No game zip files found"
       fi

       echo ${FILE##*/}

       unzip -q -d ~/Downloads/actGame/ ~/Downloads/zips/${FILE##*/}

       cd ~/Desktop/exp/
       EXPFILE=$(find . -type d -name "${FILE##*/}")
       if [ ${#EXPFILE} -eq 0 ]
       then
              echo "Expected file ".${FILE##*/}."not found"
       else
              cd ${FILE##*/}
       fi


       if ! diff -b ~/Downloads/actGame/js/constants.js constants.js; then
              PASS=1
              ALLPASS=1
              echo "FAIL: constants.js"
       fi

       if ! diff -b ~/Downloads/actGame/css/style.css style.css; then
              PASS=1
              ALLPASS=1
              echo "FAIL: style.css"
       fi

```

```
42          if ! diff -b -q ~/Downloads/actGame/index.html index.html; then
43                  PASS=1
44                  ALLPASS=1
45                  echo "FAIL: index.html"
46          fi
47
48          if ! diff -b ~/Downloads/actGame/app.js app.js; then
49                  PASS=1
50                  ALLPASS=1
51                  echo "FAIL: app.js"
52          fi
53
54          cd ~/Downloads/actGame/cardFiles
55          for c in *; do
56                  CARD=$c
57                  if [ ${#CARD} -eq 0 ]
58                  then
59                          PASS=1
60                          ALLPASS=1
61                          echo "FAIL: card File Not found"
62                  else
63                          if ! diff -b -q ${CARD##*/} ~/Desktop/exp/${EXPFILE##*/}; then
64                                  PASS=1
65                                  ALLPASS=1
66                                  echo "FAIL: cardFile name"
67                          fi
68                  fi
69          done
70
71          cd ../..
72          rm -r ~/Downloads/actGame/*
73
74          if [ $PASS -eq 0 ]
75          then
76                  printf "Pass\n\n"
77          else
78                  printf "FAIL\n\n"
79          fi
80  done
81
82  if [ $ALLPASS -eq 0 ]
83  then
84          echo "All Pass"
85  else
86          echo "Some Tests Failed"
87  fi
```

# References


[1] Stephanie Heintz and Effie Lai-Chong Law. The game genre map: A revised game classification. In *Proceedings of the 2015 Annual Symposium on Computer-Human Interaction in Play*, CHI PLAY '15, page 175–184, New York, NY, USA, 2015. Association for Computing Machinery.

[2] Glenda A. Gunter, Robert F. Kenny, and Erik H. Vick. Taking educational games seriously: using the retain model to design endogenous fantasy into standalone educational games. *Education Tech Research Dev*, 2008.

[3] Marco Brambilla, Jordi Cabot, and Manuel Wimmer. *Model-Driven Software Engineering in Practice*. Morgan & Claypool, 2017.

[4] H. Cho, J. Gray, and E. Syriani. Creating visual domain-specific modeling languages from end-user demonstration. In *2012 4th International Workshop on Modeling in Software Engineering (MISE)*, pages 22–28, 2012.

[5] Akhila Tirumalai Prasanna. A domain specific modeling language for specifying educational games. Master's thesis, Vrije Universiteit Brussel, August 2012.

[6] Niroshan Thillainathan. A model driven development framework for serious games. *Available at SSRN 2475410*, 2013.

[7] Niroshan Thillainathan, Holger Hoffmann, Eike M. Hirdes, and Jan Marco Leimeister. Enabling educators to design serious games – a serious game logic and structure modeling language. In Davinia Hernández-Leo, Tobias Ley, Ralf Klamma, and Andreas Harrer, editors, *Scaling up Learning for Sustained Impact*, pages 643–644, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[8] Niroshan Thillainathan and Jan Marco Leimeister. Serious game development for educators - a serious game logic and structure modeling language. In Louis Go?mez Chova, editor, *EduLearn 14 publications : 6th International Conference on Education and New Learning Technologies, Barcelona*, pages 1196–1206. IATED Academy, July 2014.

[9] Johannes Schropfer and Thomas Buchmann. Unifying modeling and programming with Valkyrie. *MODELSWARD*, 2019.

[10] Gonzalo Génova, Maria C. Valiente, and Mónica Marrero. On the difference between analysis and design, and why it is relevant for the interpretation of models in model driven engineering. *Journal of Object Technology*, 2009.

[11] John Klein, Harry Levinson, and Jay Marchetti. Model-driven engineering: Automatic code generation and beyond. Technical Report AD1046652, CARNEGIE-MELLON UNIV PITTSBURGH, March 2015.

[12] Steffen Becker, Tobias Dencker, and Jens Happe. Model-driven generation of performance prototypes. In Samuel Kounev, Ian Gorton, and Kai Sachs, editors, *Performance Evaluation: Metrics, Models and Benchmarks*, pages 79–98, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[13] Antonio Bucchiarone, Jordi Cabot, Richard F. Paige, and Alfonso Pierantonio. Grand challenges in model-driven engineering: an analysis of the state of the research. *Software and Systems Modeling*, 2020.

[14] Thomas Stahl and Markus Völter. *Model-Driven Software Development*. John Wiley & Sons Inc, 2006.

[15] Juha-Pekka Tolvanen and Steven Kelly. MetaEdit+: Defining and using integrated domain-specific modeling languages. In *Proceedings of the 24th ACM SIGPLAN Conference Companion on Object Oriented Programming Systems Languages and Applications*, OOPSLA '09, page 819–820, New York, NY, USA, 2009. Association for Computing Machinery.

[16] Juha-Pekka Tolvanen and Matti Rossi. MetaEdit+: Defining and using domain-specific modeling languages and code generators. In *Companion of the 18th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA '03, page 92–93, New York, NY, USA, 2003. Association for Computing Machinery.

[17] Jean Bézivin, Christian Brunette, Régis Chevrel, Frédéric Jouault, and Ivan Kurtev. Bridging the generic modeling environment (GME) and the eclipse modeling framework (EMF). In *Proceedings of the Best Practices for Model Driven Software Development at OOPSLA*, volume 5, 2005.

[18] Endre Magyari, Arpad Bakay, András Láng, Tamas Paka, Attila Vizhanyo, Aditya Agarwal, and Gabor Karsai. UDM: An infrastructure for implementing domain-specific modeling languages. In *The 3rd OOPSLA Workshop on Domain-Specific Modeling, OOPSLA*, 2003.

[19] Hanns-Alexander Dietrich, Dominic Breuker, Matthias Steinhorst, Patrick Delfmann, and Jörg Becker. Developing graphical model editors for meta-modelling tools - requirements, conceptualisation, and implementation. *Enterprise Modelling and Information Systems Architectures - An International Journal*, 8(2):42–78, 2013.

[20] Eric J. Rapos and Matthew Stephan. IML: towards an instructional modeling language. In *MODELSWARD*, 2019.

[21] Thomas Buchmann, S Hammoudi, M van Sinderen, and J Cordeiro. Valkyrie: A UML-based model-driven environment for model-driven software engineering. In *ICSOFT*, pages 147–157, 2012.

[22] Randy J. Pagulayan, Keith R. Steury, Bill Fulton, and Ramon L. Romero. Designing for fun: User-testing case studies. *Funology Human-Computer Interaction Series*, page 137–150, 2003.

[23] Christian Elverdam and Espen Aarseth. Game classification and game design: Construction through critical analysis. *Games and Culture*, 2(1), January 2007.

[24] K. Robson, K. Plangger, J. H. Kietzmann, I. McCarthy, and L. Pitt. Is it all a game? understanding the principles of gamification. *Business Horizons*, 2015.

[25] Stewart Woods. Loading the dice: The challenge of serious videogames. *The International Journal of Computer Game Research*, 4(1), November 2004.

[26] Francesco Bellotti, Riccardo Berta, Alessandro de Gloria, Michela Ott, Sylvester Arnab, Sara de Freitas, and Kristian Kiili. Designing serious games for education: from pedagogical principles to game mechanisms. *HAL*, April 2014.

[27] Cynthia M. Odenweller, Christopher T. Hsu, and Stephen E. DiCarlo. Educational card games for understanding gastrointestinal physiology. *Advances in Physiology Education*, 20(1), December 1998.

[28] Sean M. Barclay, Meghan N. Jeffres, and Ragini Bhakta. Educational card games to teach pharmacotherapeutics in an advanced pharmacy practice experience. *American Journal of Pharmaceutical Education*, 75(2), March 2011.

[29] Alexander Zibula and Tim A. Majchrzak. Cross-platform development using html5, jquery mobile, and phonegap: Realizing a smart meter application. In José Cordeiro and Karl-Heinz Krempels, editors, *Web Information Systems and Technologies*, pages 16–33, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[30] Grant Warren Sherson. Website design principles: Researching and building a website evaluation tool. Master's thesis, Victoria University of Willington, 2002.

[31] Yoon-Seop Chang, Seong-Ho Lee, Jae-Chul Kim, and Young-Jae Lim. Study on mobile mashup webapp development tools for different devices and user groups. In *The International Conference on Information Networking 2014 (ICOIN2014)*, pages 433–438, 2014.

[32] Omar Badreddin. Umple: A model-oriented programming language. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 2*, ICSE '10, page 337–338, New York, NY, USA, 2010. Association for Computing Machinery.

[33] Andrew Forward, Omar Badreddin, Timothy C. Lethbridge, and Julian Solano. Model-driven rapid prototyping with umple. *Software: Practice and Experience*, 42(7):781 – 797, July 2012.

[34] Nicholas John DiGennaro. Intuitive model transformations: A guided framework for structural modeling. Master's thesis, Miami University, May 2021.

[35] Kristian Kiili. Digital game-based learning: Towards an experiential gaming model. *The Internet and Higher Education*, 8(1):13 – 24, 2005.

[36] Frank E. Hernandez and Francisco R. Ortega. Eberos GML2D: A graphical domain-specific language for modeling 2D video games. In *Proceedings of the 10th Workshop on Domain-Specific Modeling*. ACM, 2010.

[37] André WB Furtado and André LM Santos. Using domain-specific modeling towards computer games development industrialization. In *The 6th OOPSLA workshop on domain-specific modeling (DSM06)*. Citeseer, 2006.

[38] Emanuel Montero Reyno and José Á Carsí Cubel. Automatic prototyping in model-driven game development. *Comput. Entertain.*, 7(2), June 2009.

[39] Olga De Troyer and Elien Paret. Challenges in designing domain-specific modeling languages for educational games. In *Proc. Int. Work. Involv. End Users Domain Expert. Des. Educ. Games*, 2011.

[40] Ana Syafiqah Zahari, Lukman Ab Rahim, Nur Aisyah Nurhadi, and Mubeen Aslam. A domain-specific modelling language for adventure educational games and flow theory. *International Journal on Advanced Science, Engineering and Information Technology*, 10(3):999–1007, 2020.

[41] Teo Eterovic, Enio Kaljic, Dzenana Donko, Adnan Salihbegovic, and Samir Ribic. An internet of things visual domain specific modeling language based on UML. In *2015 XXV International Conference on Information, Communication and Automation Technologies (ICAT)*, pages 1–5, 2015.