Hindawi
International Journal of Computer Games Technology
Volume 2017, Article ID 9626710, 13 pages
https://doi.org/10.1155/2017/9626710

Research Article

MEnDiGa: A Minimal Engine for Digital Games

Filipe M. B. Boaventura and Victor T. Sarinho

State University of Feira de Santana, Feira de Santana, BA, Brazil

Correspondence should be addressed to Filipe M. B. Boaventura; fmbboaventura@gmail.com

Received 15 February 2017; Revised 29 May 2017; Accepted 7 June 2017; Published 11 July 2017

Academic Editor: Michael J. Katchabaw

Game engines generate high dependence of developed games on provided implementation resources. Feature modeling is a technique that captures commonalities and variabilities results of domain analysis to provide a basis for automated configuration of concrete products. This paper presents the Minimal Engine for Digital Games (MEnDiGa), a simplified collection of game assets based on game features capable of building small and casual games regardless of their implementation resources. It presents minimal features in a representative hierarchy of spatial and game elements along with basic behaviors and event support related to game logic features. It also presents modules of code to represent, interpret, and adapt game features to provide the execution of configured games in multiple game platforms. As a proof of concept, a clone of the *Doodle Jump* game was developed using MEnDiGa assets and compared with original game version. As a result, a new *G-factor* based approach for game construction is provided, which is able to separate the core of game elements from the implementation itself in an independent, reusable, and large-scale way.

1. Introduction

Game engines allow game developers to reuse significant portions of key software components [1]. A game engine is defined as "extensible software that can be used as the foundation for many different games without major modification" [1] and represents "the collection of modules of simulation code that do not directly specify the game's behavior (game logic) or game's environment (level data)" [2].

Although game engines are reusable across multiple game projects, they generate high dependence of the game on implementation resources provided by the chosen engine [3]. Referring collectively to game logic, object model, and game state elements as *G-factor*, BinSubaih and Maddock [3] provided a service-oriented based architecture able to separate the core of game objects from the implementation itself and to support the game portability among game engines.

As an attempt to identify commonalities and variabilities in the digital game domain, Sarinho and Apolinário [4] proposed the *Narrative*, *Entertainment*, *Simulation*, and *Interaction* (NESI) feature model. It is a feature-based approach capable of representing the *G-factor* according to game concepts found in the literature. Sarinho and Apolinário also proposed the *GameSystem*, *DecisionSupport*, and *SceneView* (GDS) feature model [5], which is based on game features that describe generic configurations and behavioral aspects found on game implementation resources identified in the literature. GDS was also used as a reference model to a proposed generative approach of digital games, showing as a result the feasibility of using features in the production of concrete digital games [5].

Although NESI and GDS models represent digital games without relying on the structure of game engines, the large number of proposed features became a difficulty in the design of small and casual games such as platform, quiz, or maze games [6]. This difficulty was confirmed during the production of the *SimplifiedPacMan* version [5] using Feature-based Environment for Digital Games (FEnDiGa) [7], a game production environment based on a combined representation of NESI and GDS features via Object Oriented Feature Modeling (OOFM) approach [8].

This paper presents the *Minimal Engine for Digital Games* (MEnDiGa), an extensible collection of representative classes based on a simplified set of NESI and GDS features that can be used as the foundation for small and casual games without major modification. MEnDiGa also provides a collection of

modules of code for the interpretation and adaptation processes of represented features that do not directly specify the game's behavior or game's environment.

This paper is organized as follows. Section 2 describes important papers related to game domain engineering and game feature modeling. Section 3 presents the proposed MEnDiGa feature model and the resulting framework [9] with representative classes and modules of code. Section 4 describes the development steps to provide a *Doodle Jump* clone using MEnDiGa assets. Section 5 presents the software metrics analysis for *Doodle Jump* game versions. Finally, Section 6 presents the conclusions and future work of this project.

2. Related Work

Many types of reusable approaches have been applied in game development in recent years. They can simplify and accelerate the production of gaming systems, focusing on game modeling artifacts, digital game components, game product lines, and reusable aspects in game development, for example.

Considering the usage of software components on game development, Folmer [10] stated that developers could reuse specific game components to reduce the cost of building games. Folmer [10] also proposed a Reference Architecture for digital games as an attempt to possibly identify areas of reuse.

Zhang and Jarzabek [11] proposed the RPG Product Line Architecture (RPG-PLA), a group of common and variable features of four distinct RPG games. As a result, any of the original RPGs as well as similar ones could be derived from feature configurations interpreted by the RPG-PLA.

Albassam and Gomaa [12] proposed the use of Software Product Lines (SPL) in the video games domain. They have built a feature dependency model to describe the variability in multiplatform video games (such as different input/output devices, user interface, and CPU) and a variable component-based SPL suited for any video game in the product line.

Furtado et al. [13] proposed an improvement of the Sharp-Ludus project [14], replacing the previous ad hoc approach with a customized DSM approach called Domain-Specific Game Development. In this approach, feature models are used to improve the domain vocabulary and to help the identification of specific subdomains of the SPL domain.

Müller [15] presented the DGiovanni project, an open-source multiagent architecture for building interactive dramas. It makes use of ontologies to support the creation of different stories and to feed the system with story-related information.

Finally, Machado et al. [16] proposed a generic representation to model virtual agents in digital games. It allows the implementation of adaptable behaviors for game agents according to different features of the game environment. Agents are modeled using a linear combination of different variables, which are used to represent specific game features.

3. The Minimal Engine for Digital Games (MEnDiGa)

This section presents the proposed MEnDiGa feature model, based on a simplified collection of NESI and GDS features, and the resulting framework, based on representative classes and modules of code able to work with MEnDiGa features. Together, they can configure, represent, perform, and adapt feature specifications of digital games according to *G-factor* portability for distinct game platforms.

3.1. MEnDiGa Feature Model. Originally presented by Kang et al. [17] as part of the Feature Oriented Domain Analysis (FODA), feature modeling allows the identification of system properties during the domain analysis. According to them, a feature model represents "the *standard* features of a family of systems in the domain, and the relationships between them," and features are "aspects or characteristics of a domain which are visible to the user."

Following the feature modeling perspective to identify similarities or differences between the products of a product line [18], MEnDiGa presents digital games as collections of three main features: *Spatial*, *Behavior*, and *Observer* (Figure 1).

Regarding the *Spatial* feature, it is a collection of *Node* features that represent the elements of the game. *Spatial* feature can be also an *Environment* feature with a collection of *Location* features. Each *Node* feature contains a *CurrentLocation* feature to determine its current position in a game. Each *Node* feature also has a *BoundingVolume* feature to delimit the collision detection space. *Node* features can also contain *AudioNode*, *GraphicNode*, and *PhysicsNode* features, like *SceneNode* feature from GDS model [5].

The *AudioNode* feature represents information about sound effects to be used later in a digital game. It holds the path to an audio file that contains the desired sound effect or background music. *AudioNode* can also contain information about the state of the audio file and its play mode (*normal* or *looping*).

GraphicNode feature represents configurations related to the graphical modeling of a certain *Node* feature. As a simple *Text* feature, it holds information about the font and the alignment to draw a string. As a *Sprite* feature, it holds information related to how a texture, or even a region of a texture, should be rendered. Various regions of the same texture can be used to compose animations. As a *Camera* feature, it contains information about the *viewport* of a *Spatial*.

The *PhysicsNode* feature contains the physical attributes of a *Node* feature, such as its *density*, *mass*, and *restitution* coefficient. It also allows the setting of some constraints for physical simulation, such as the amount of *gravity* that acts on the *Node* feature and whether it is *solid* or not.

Element features are specializations of *Node* features. They contain one or more *Property* features represented with identifier *names* and representative *values* (*Speed*: 50 m/s; *Life*: 2 lives). An *Element* that is responsible for user *Behaviors* is defined as a *Player* feature. *Player* behaviors (*Jump* and *Crouch*, for instance) are related to default commands

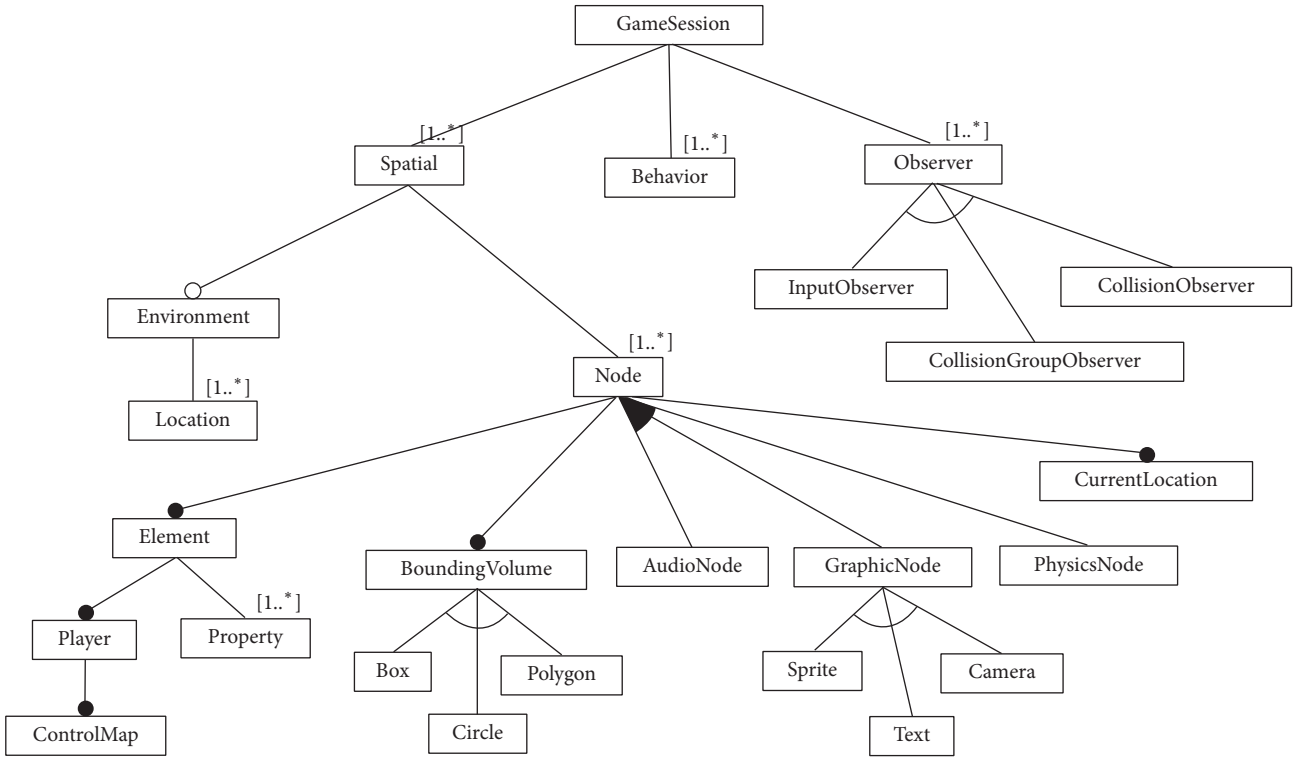

Figure 1: MEnDiGa feature diagram.

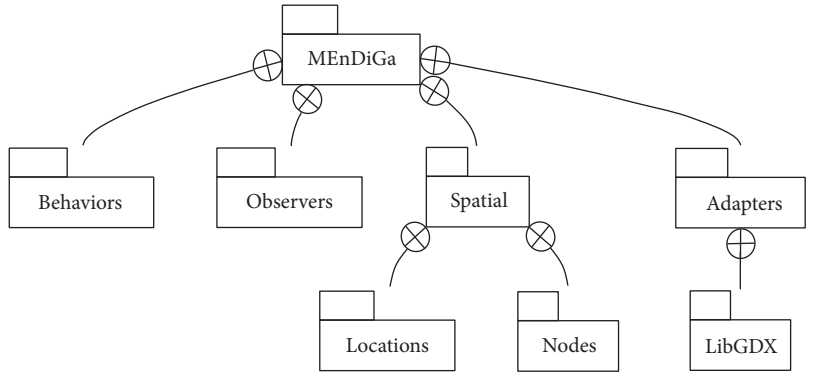

Figure 2: Package diagram of the MEnDiGa framework.

(*MOVE_UP, MOVE_DOWN*, etc.) defined on its *ControlMap* feature.

Observer features are responsible for performing *Behavior* features according to the evaluation of monitored resources. For instance, an *InputObserver* feature can be set to evaluate the command activation from an input device, executing the related *Behavior* feature of the *Player* after firing it. As another example, a *CollisionObserver* feature can trigger the execution of an *IncreaseScoreBehavior* feature when the *Player* collides with an item or execute a *LoseLifeBehavior* feature when it collides with an enemy.

3.2. MEnDiGa Framework. According to Czarnecki [19], a proposed feature model can be used to define a set of classes capable of configuring various software systems. It is the project stage of a software domain [20], whose objective is to represent a feature value in each instantiated class based on a derived framework from the associated feature model.

In this sense, a collection of classes was proposed to represent, interpret, and adapt feature configurations derived from MEnDiGa feature model. The resulting framework was organized according to the following list of packages and classes (Figures 2 and 3):

(i) *Spatials:* represents the collection of spaces and game elements using *Spatial*, *Environment*, and *Node* classes

(ii) *Nodes:* defines specializations and internal components of the *Node* class, such as *AudioNode*, *GraphicNode*, *BoundingVolume*, *Element*, and *Player* classes

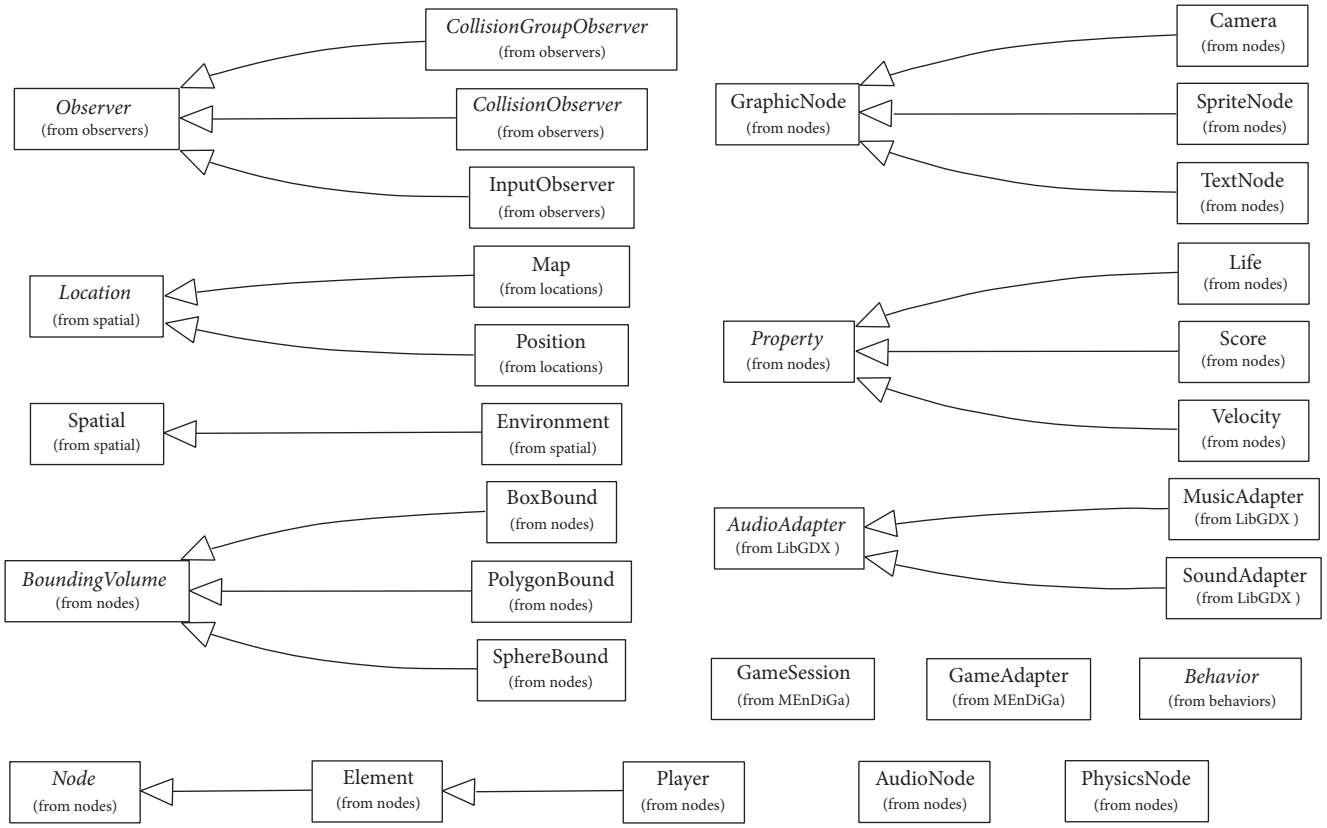

Figure 3: Classes and type hierarchies of the MEnDiGa framework.

(iii) *Locations:* defines internal components of an *Environment* using *Location*, *Map*, and *Position* classes

(iv) *Behaviors:* defines the abstract class *Behavior* able to be specialized in actions to be performed by a game

(v) *Observers:* defines abstract structures (*Observer*, *InputObserver*, *CollisionObserver*, and *CollisionGroupObserver*) to handle fired events in a game

(vi) *Adapters:* declares concrete classes able to render MEnDiGa objects in distinct game platforms

As the main class of the MEnDiGa framework, *GameSession* (Figure 3) has the responsibility of integrating the *Spatial*, *Observer*, and *Behavior* objects derived from the proposed MEnDiGa features. The *Spatial* object contains a collection of *Node* instances, such as *AudioNode*, *GraphicNode*, *PhysicsNode*, *BoundingVolume*, and *Location* instances. The *Element* class extends the *Node* class, including a collection of *Property* instances represented in a *HashMap* structure. The *Player* class has been defined as an *Element* subclass. It contains a control map with a collection of *Behavior* instances related to player commands. The *Environment* class was also defined as a *Spatial* specialization. It contains a collection of *locations* objects such as *Map* and *Position* instances (the game space).

For each *Observer* instance, there is a collection of *Behavior* instances. They are executed according to the *Observer* evaluation approach programmed for each possible event of a game. Each *Behavior* instance is programmed to execute a determined game action, modifying attributes in *Node* instances and changing the current *Spatial* instance of the *GameSession* instance in each renderization cycle of the game.

Considering the *adapters* classes, they are responsible for the execution of the instantiated MEnDiGa objects using implementation resources of a chosen game platform. For each worked game platform, a new set of adaptation classes is modeled only once for the first game, allowing the *G-factor* portability of other MEnDiGa games for the same game platform.

4. Case Study: A Doodle Jump Clone

A clone of the Doodle Jump game was developed to demonstrate the feasibility of producing games with MEnDiGa assets. The MEnDiGa version is based on a Doodle Jump clone called Super Jumper, developed using the LibGDX game framework [21]. Designed to faithfully reproduce the Super Jumper version, the MEnDiGa clone was developed using the same textures and audio files from the original game. It also presents a similar set of features in comparison with the original game.

This section presents an explanation of the Super Jumper development process, showing the necessary configuration, implementation, and adaptation steps to provide a MEnDiGa game. For the configuration step, it illustrates some identified Super Jumper features according to proposed MEnDiGa

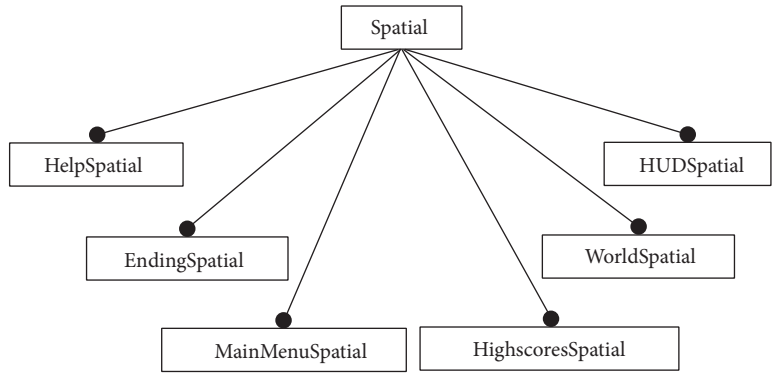

Figure 4: *Spatial* subfeatures for the Super Jumper game.

feature model. The implementation step presents developed classes and source codes based on designed MEnDiGa framework which represent structural and dynamics aspects of the Super Jumper game. Finally, the adaptation step describes developed classes able to interpret MEnDiGa objects and perform the Super Jumper game execution using LibGDX resources.

4.1. Super Jumper Configuration. According to Super Jumper game, the players must guide Bob (the playable character) to the end of the level. Bob must bounce on platforms to reach the castle, at the top of the game world. Along the way, Bob can collect coins for points and hit springs to jump higher. There are also obstacles, such as crumbling platforms and flying squirrels. The game ends in victory when Bob reaches the castle or defeat if he collides with a squirrel or falls of a platform.

For the MEnDiGa version, each screen of the Super Jumper game is represented using *Spatial* configurations, where each one holds an appropriate set of *Node* features. A *Position* feature contains the Cartesian coordinates of a *Node* instance, and it is used to locate nodes in a *Spatial* feature. *Position* also holds depth information used to define the overlapping between images on *Spatial*. Figure 4 illustrates this descriptive mapping among Super Jumper game elements and *Spatial* subfeatures for the Super Jumper game.

Figure 5 shows the *Spatial* configuration of the main menu screen. Each option in the main menu, as well as any clickable object in the game, is represented using *Node* features. These clickable nodes hold an appropriate *GraphicNode* feature (*Sprite* or *Text*), as well as a rectangular *BoundingVolume* feature to determine the clickable area. *BackgroundNode* and the *TitleNode* features are simply placeholders for the background and title of the game.

Figure 6 also shows the game world represented by the *WorldSpatial* feature. This spatial configuration contains *Node* features that build the game level as a whole. Since platforms and squirrels need the *speed* property, *Platform* and *Squirrel* subfeatures of the *Element* feature are used to represent them. *Bob*, the *Player* character represented as a feature, also contains a property for holding *score* information. Stationary objects such as *Coins, Springs*, and the *Castle* are modeled as subfeatures of the *Node* feature.

When *Bob* reaches the *Castle* at the end of the level, the *EndingSpatial* feature is used to display the game's ending. Figure 7 shows *Bob* and the *Princess* having a conversation in front of the *Castle*. The *MessageNode* feature contains the *GraphicNode* feature responsible for displaying the line of the dialog. Clicking anywhere on the screen will advance to the next message or return the game to the main menu when the last message is displayed.

Regarding the support of MEnDiGa game events, there is a series of collision events among the Super Jumper game elements and the *Player* feature itself that must be monitored. Figure 8 illustrates some *Observer* features configured to be responsible for this collision detection and player monitoring during the game execution. There are also *Observers* features dedicated to monitoring interactions with the user interface components, such as clicks on the screen and button press.

The actions responsible for changing *Spatial* feature values are represented using *Behavior* features, such as *Player* commands (MOVE_UP, MOVE_LEFT, etc.) or game dynamics (increase/decrease score, win/lose game, etc.). Figure 9 presents some of these *Behavior* features, showing game dynamics that were implemented in the MEnDiGa version. It is noteworthy that *Observer* features will activate such *Behavior* features upon confirmation of a monitored event (ex.: *BUTTON1_PRESS* → *JUMP*).

4.2. Super Jumper Implementation. By the definition of Super Jumper game features, the next step consists of representing the Super Jumper configuration according to the MEnDiGa framework hierarchy. In this sense, *Node* subclasses are used to represent the characteristics of *Spatial* components, such as starting *Position*, *Element* properties, and *GraphicNode* instances.

In case of *Sprite* objects used to represent *GraphicNode* instances in the game, it is necessary to define the desired region of the image that will be shown. If this information is omitted, the whole image will be used. For each selected region in the image, it will contain subregions called *frames*, where each one will have the size set on the *SpriteNode* constructor (Algorithm 1). Each frame can be referenced using an

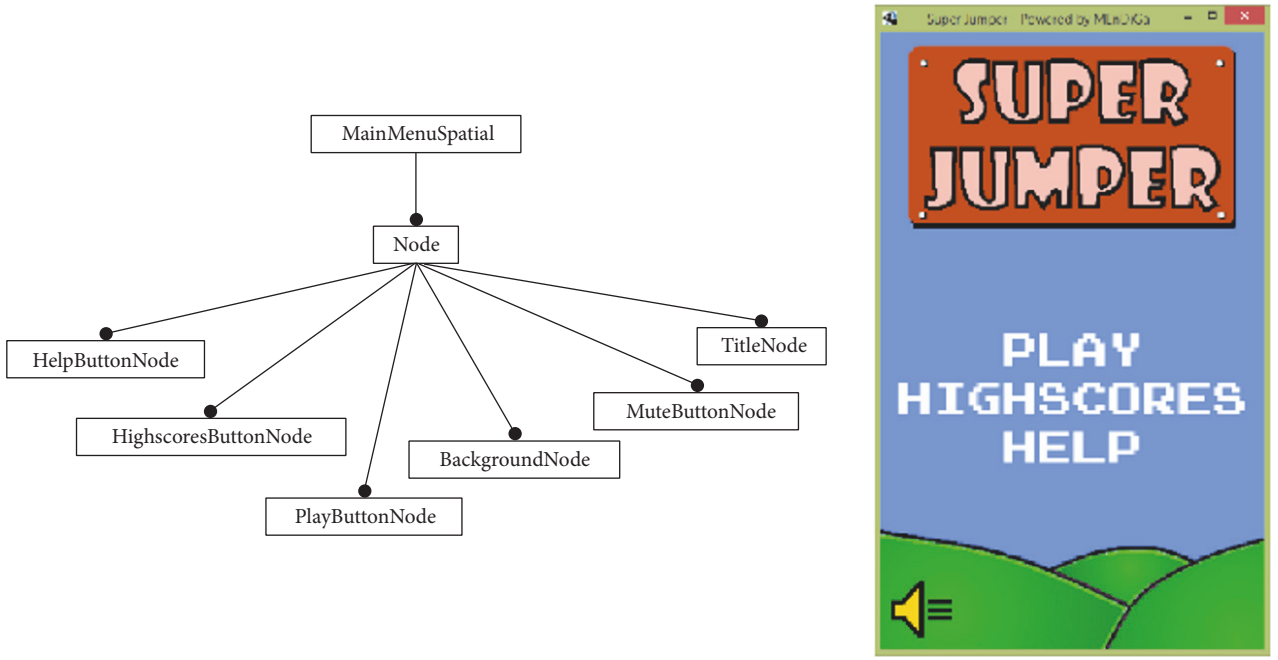

Figure 5: Main menu screen and respective *Spatial* configuration.

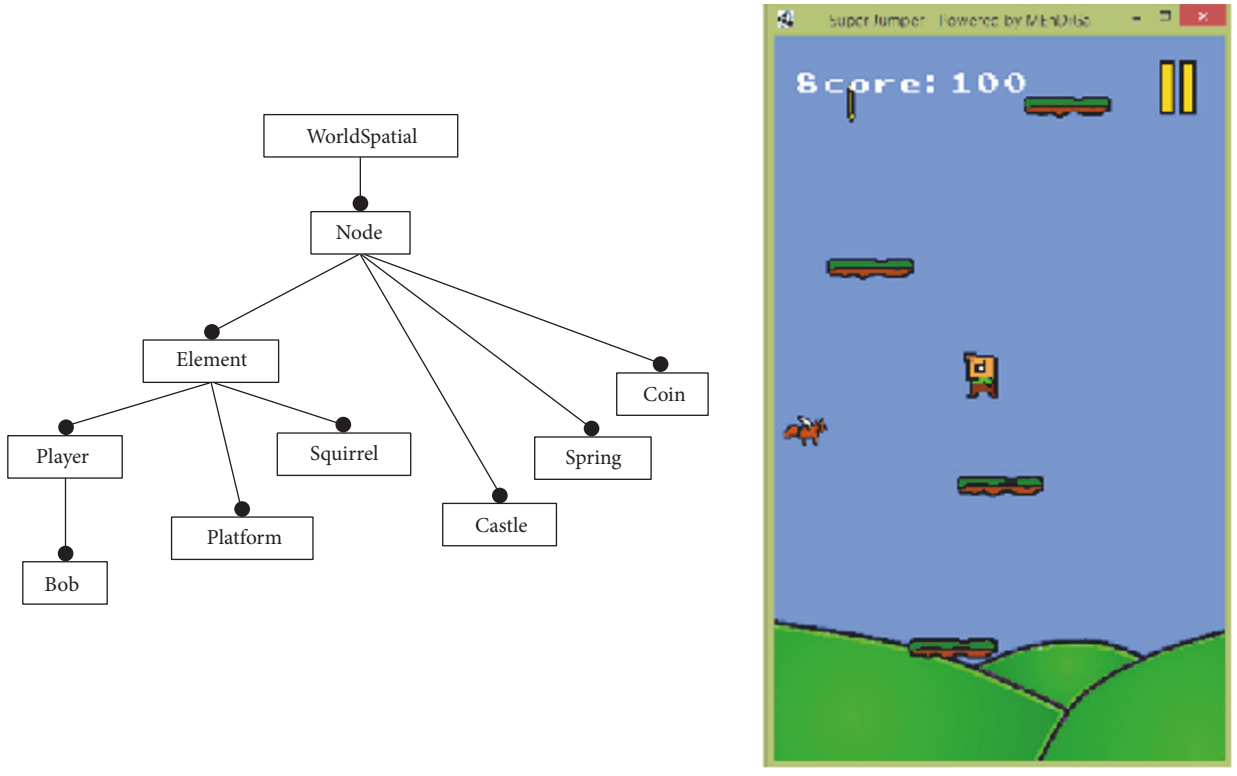

Figure 6: Spatial configuration of the gameplay screen.

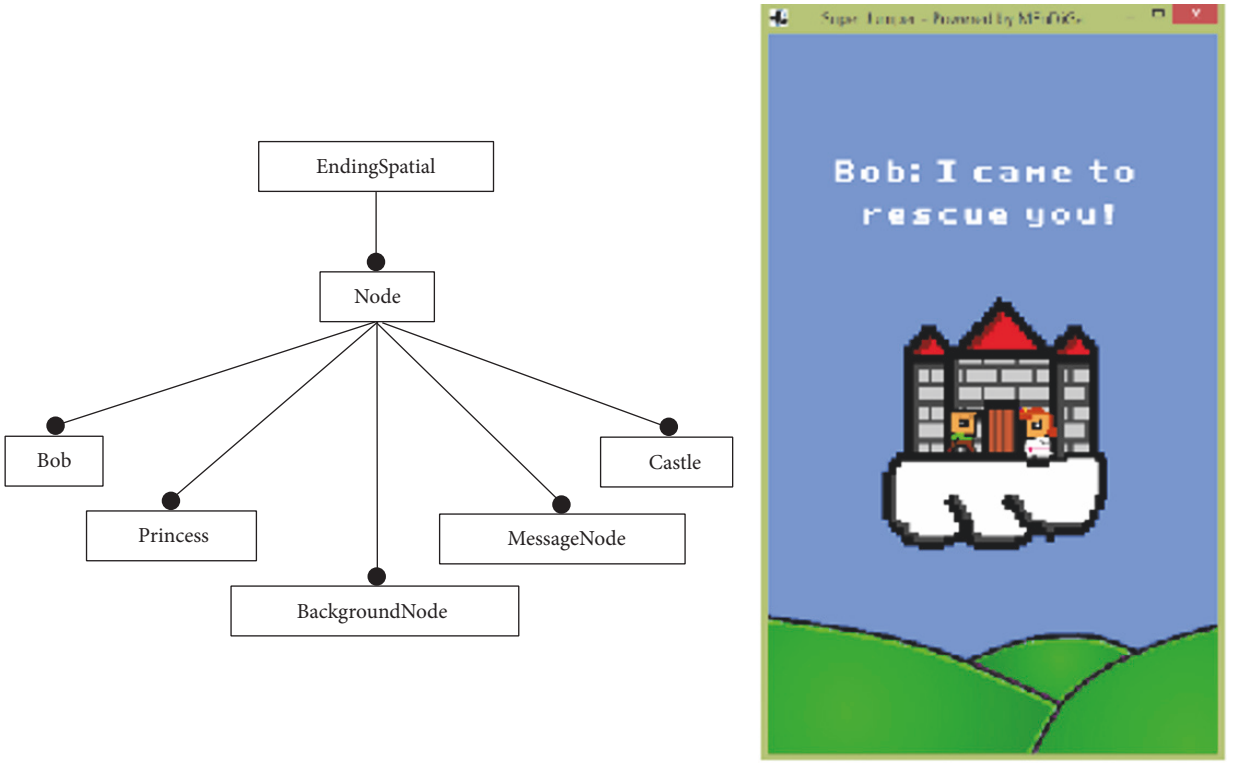

Figure 7: Ending screen and respective *Spatial* configuration.

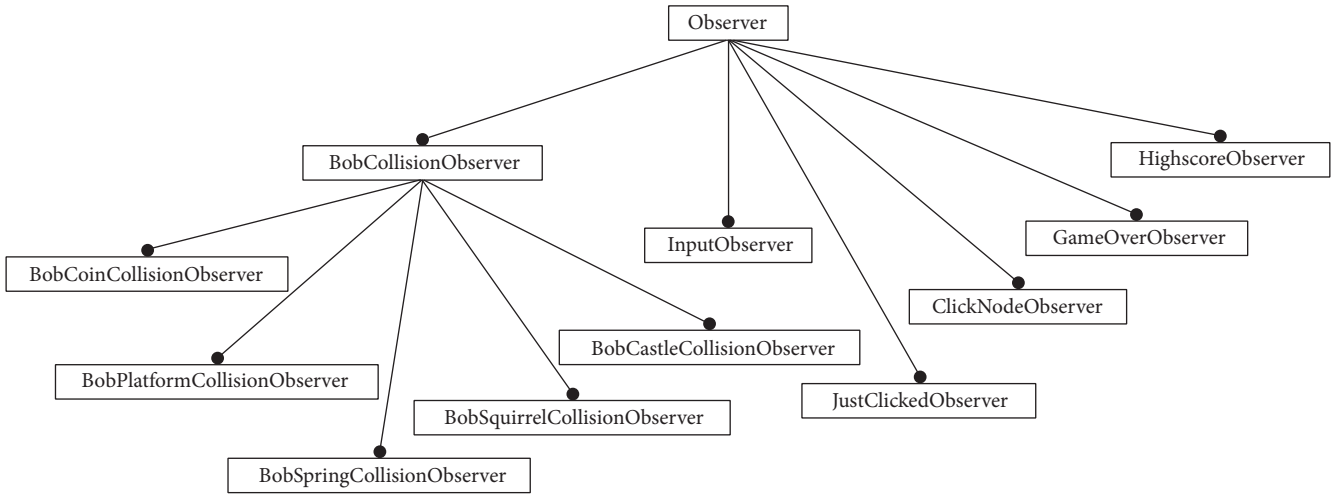

Figure 8: *Observer* subfeatures for the Super Jumper game.

index number (Figure 10). Animations are composed using selected frames to be rendered and their respective *frame delay* value (the amount of time that a frame will remain on the screen).

Regarding Super Jumper game dynamics, the *SuperJumperGame* class extends the *GameSession* class to "configure" the game (Algorithm 2). This class is programmed to instantiate *AudioNode* feature values, initial game observers to start the game, and respective game behaviors to perform the game initialization. After this, the *SuperJumperGame* executes the *ChangeToMainMenuScreen* behavior (Algorithm 2) to prepare the *MainMenuSpatial* (Figure 5) to be displayed and start up the game.

When the player selects the "*Play*" option on the created main menu, the *ClickNodeObserver* instance executes *ReadyGame* behaviors (Figure 9). *ReadyGame*, in turn, executes the *GenerateLevel* behavior, which is responsible for creating and randomly placing the platforms, springs, coins, and

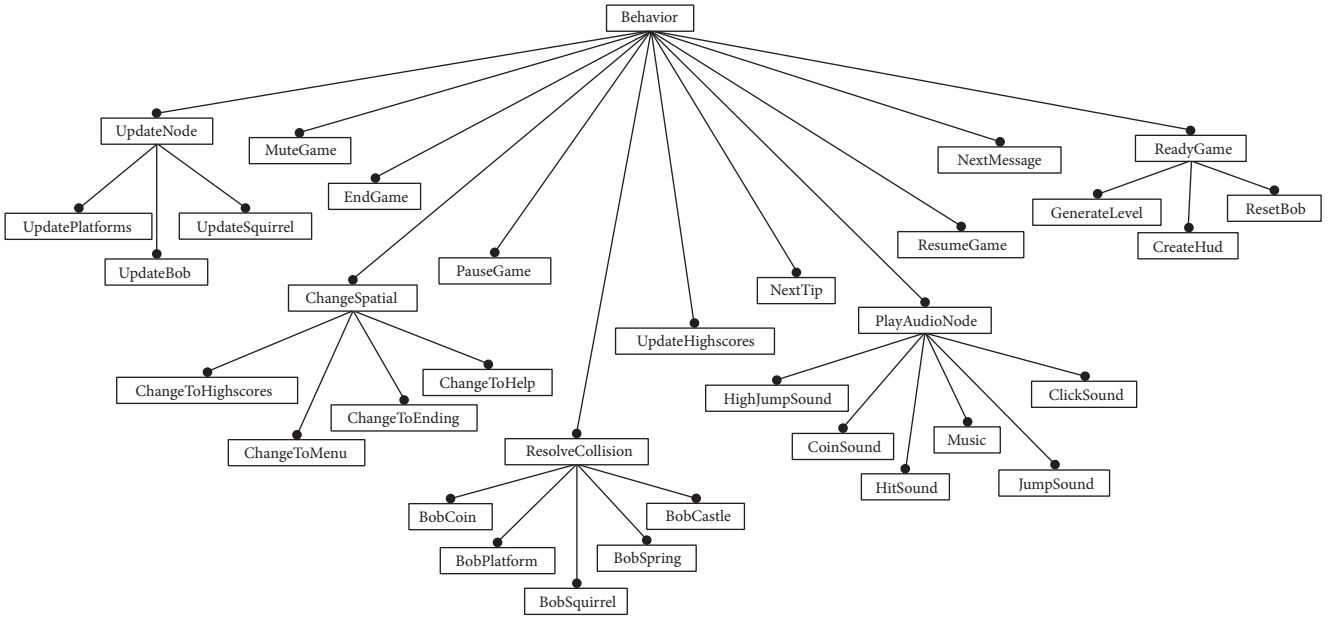

FIGURE 9: *Behavior* subfeatures for the Super Jumper game.

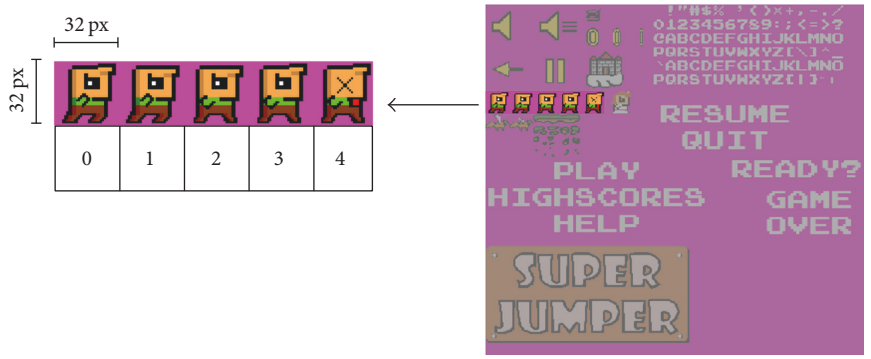

FIGURE 10: The highlighted region is extracted from the texture, according to the *GraphicNode* configuration.

```
// Create the graphic node
SpriteNode graphicNode = new SpriteNode("bob", // sprite name
        "res/superjumper/items.png", // source file path
        32, 32); // width and height of a single frame within the sheet
// Select a rectangular region of the texture
graphicNode.setRegion(0, 128, // Upper left corner position
        160, 32); // region width and height
// Defining animations
graphicNode.defineAnimation("jump", // Animation name
        new int[]{0,1},0.2f,          // Animations frame and frame delay
        PlayMode.LOOP_NORMAL);      // Play mode
graphicNode.setCurrentAnimation("jump"); // Set "jump" as the current animation
graphicNode.setScale(1/32f); // Set the scale as 32 pixels per game unit
this.addGraphicNode(graphicNode); // Add the sprite to the node
```

ALGORITHM 1: Setting up the *GraphicNode* of the playable character.

```
public class SuperJumperGame extends GameSession {
     public static enum GameState {RUNNING, PAUSED, READY, WIN}
     private GameState gameState;
     private AudioNode[] audioNodes;
     public SuperJumperGame() {
          // Configuring AudioNodes
          this.audioNodes = new AudioNode[] {
                  new AudioNode("coinSound", "res/superjumper/coin.wav"), // ···
                  new AudioNode("gameMusic", "res/superjumper/music.mp3") };
          // Configuring Observers
          this.addObserver(new CheckForNewHighScore(this));
          this.addObserver(new CheckGameOver(this));
          // Configuring Behaviors
          this.addBehavior("changeToMenu", new ChangeToMainMenuScreen(this));
          this.addBehavior("generateLevel", new GenerateLevel(this));
          this.addBehavior("createHud", new CreateHud(this));
          this.addBehavior("resetBob", new ResetBob());
          this.addBehavior(GameState.RUNNING.toString(), new UpdateRunning(this));
          this.addBehavior("endGame", new EndGame(this));
          this.addBehavior("playClickSound", new PlayClickSound(this));
     }
     public void initialize() {
          this.getBehavior("changeToMenu").execute(); // load the game menu
          this.getAudioNode(MUSIC).setState(State.PLAYING);
     }
     public void update(float deltaTime) {
          // Perform the current Running Behavior if the game still running
          if (this.gameState.equals(GameState.RUNNING)) {
             this.getBehavior(this.gameState.toString()).execute(deltaTime);
          }
     } // ···
```

Algorithm 2: Partial description of the *SuperJumperGame* class.

squirrels on the *WorldSpatial* instance. With the *WorldSpatial* populated, *ReadyGame* behavior calls the *ResetBob* behavior, putting the *Bob* object in the initial state before inserting it into the *WorldSpatial* instance. The *CreateHud* behavior is executed next, creating the *HUDSpatial* instance that holds the necessary *Node* instances for the *Head-Up Display*. *ReadyGame* behavior also sets the observers and behaviors required for the gameplay, such as the *InputObserver* instance to control *Bob*, the game collision observers and related behaviors, the behaviors to update node values during the gameplay, and the game observers to check for high score updates and game over events.

UpdateNode behaviors (Figure 9) are also executed for each generated frame in the game. They update the state of dynamic game elements based on defined game configurations and player inputs. The *UpdateSquirrel* and the *UpdatePlatform* behaviors are set to change the *Position* of *Squirrel* and *Platform* instances according to their *speed Property* value. If a *Platform* instance is set to crumble, the *UpdatePlatform* behavior is responsible for updating the countdown timer for the platform destruction. *UpdateBob* behavior changes the player *Position* using *gravity* and *horizontal speed* values. *UpdateHighscore* behavior is activated by the *HighscoresObserver* when the game ends after falling down the screen or hitting an enemy/castle, changing the score ranking if the player has enough points.

BobCollisionObserver instances are responsible for evaluating collision events among the player character and each *Node* instance in *WorldSpatial*. They fire appropriate *ResolveCollision* behaviors (Figure 9) if the player's *BoundingVolume* intersects another *Node* instance. Some of these observers only trigger collisions on specific situations. For example, *Bob* instance can pass through *Spring* and *Platform* objects from below, but *ResolveCollision* behaviors must be performed if *Bob* hits them from above.

Bob/Coin collisions destroy the collided *Coin* instance and increase the *score Property* value. Collisions between *Bob* instance and the *Platform* or *Spring* instances increase its jumping *speed*. However, hitting a *Platform* instance may cause it to crumble, destroying it after a crumbling animation is complete. Hitting a *Squirrel* will trigger the *EndGame* behavior, showing a "*Game Over*" message on the *HUDSpatial* instance as *Bob* goes down through the screen. After clicking anywhere on the screen when the game is over, it will notify the *JustClickedObserver* instance to trigger the *ChangeToMainMenuScreen* behavior. Hitting the *Castle* instance will trigger the *ChangeToEndingScreen* behavior, completing the play through.

4.3. Super Jumper Adaptation. In a similar strategy to FEnDiGa [7], the next and final step consists of applying Super Jumper instances to use MEnDiGa adapters. Such adaptation classes establish an integration pattern between game configurations with implementation resources of a chosen game platform.

LibGDX has been used in this project as a target game engine to provide MEnDiGa adapter classes. It is an open-source framework which presents good market acceptance to produce multiplatform Java games. Among LibGDX available adapters, *SpatialRenderer, GraphicNodeRenderer*, and *InputAdapter* should be highlighted.

For each *Spatial* instance on the *SuperJumperGame*, a *SpatialRenderer* needs to be created. It is responsible for loading and rendering specified settings of each *GraphicNode* object available in a *Spatial*. It also follows the rendering order of the *GraphicNode* instances according to the informed depth (z-axis of the *Position*) on each *Node* instance.

For each *GraphicNode* instance to be rendered, a specific *GraphicNodeRenderer* object is created based on the *GraphicNode* type. *TextNodeRenderer* uses the information on a *Text GraphicNode* instance to set up LibGDX's Bitmap Font utilities, loading an image file containing the glyphs and a font file to provide image characters. *SpriteNodeRenderer* uses the information stored on a *Sprite GraphicNode* instance to load the desired image file as a LibGDX *Texture* object. This texture is split and its frames are stored in an array, allowing the image usage as shown in Figure 10. In the adaptation process, the animation information defined in the *GraphicNode* instance is used to create *Animations*, LibGDX objects capable of managing *Sprite* animations.

The *InputAdapter* is the class responsible for notifying the *InputObserver* about input events. It implements the *InputProcessor*, an interface provided by LibGDX to receive input events from the keyboard, touchscreen, or mouse. The *InputAdapter* has an *adaptedControlMap* structure that makes the correspondence among *Player* commands (MOVE_LEFT, MOVE_RIGHT, etc.) and the LibGDX constants for input keys. If the pressed/released key corresponds with some *Player* command, the *InputAdapter* notifies the fired/released command to the *InputObserver* that performs the appropriate *playerBehavior* according to the configured game dynamics.

SuperJumperLibGDX performs the final adaptation among MEnDiGa assets and the LibGDX engine. It implements the *ApplicationListener* interface of the LibGDX, which contains the methods called during the game lifecycle such as *create* and *render*. *SuperJumperLibGDX* overrides the *create* method to instantiate *SuperJumperGame* and the other needed *adapters* during the application launching. Associations among *Observer* instances of the *SuperJumperGame* and the package *adapter* (*InputAdapter*, for instance) are also configured during the creation of the *SuperJumperLibGDX*. The *render* method is also being overridden to continually *draw* the respective *SpatialRenderer* and evaluate monitoring aspects of *Observer* instances.

In the end, after the *SuperJumperLibGDX* execution, LibGDX starts the initialization, rendering and updating processes of adapted MEnDiGa assets, displaying as a result the configured Super Jump game to be played. As MEnDiGa is an open-source project, more details about the LibGDX adaptation process can be found at https://bitbucket.org/fmbboaventura/mendiga (master branch).

5. Super Jumper Metrics Analysis

According to Pressman [22], product metrics help software engineers to visualize the design of the software, focusing on specific and measurable attributes of software engineering artifacts. In order to perform a comparative analysis and evaluate the quality of the produced digital game, a set of software metrics was collected from the original Super Jumper version and the respective cloned MEnDiGa version.

Table 1 presents the collected OO metrics of both games using the Eclipse Metrics plugin [23]. It provides metrics calculation and dependency analysis according to *Number of Classes, Number of Attributes, Number of Methods, Number of Static Attributes*, *Number of Static Methods*, *Number of Packages*, *Number of Overridden Methods*, *Average Number of Parameters*, *Total Lines of Code*, *Average Lines of Code per Method*, *Abstractness (A)*, *Afferent Coupling (Ca)*, *Efferent Coupling (Ce)*, *Instability (I)*, *Normalized Distance from Main Sequence (Dn)*, *Lack of Cohesion of Methods (LOCOM)*, *Average Nested Block Depth*, *Cyclomatic Complexity*, *Weighted Methods per Class (WMC)*, and *Depth of Inheritance Tree (DIT)*.

Number of Classes, Number of Attributes, Number of Methods, Number of Static Attributes, Number of Static Methods, Number of Packages, Number of Overridden Methods, Average Number of Parameters, Total Lines of Code, and *Average Lines of Code per Method* are simple and straightforward metrics, with a detailed explanation about them being unnecessary. *Abstractness (A)* represents the ratio of the number of abstract classes and interfaces to the total number of classes in the selected scope. It varies from zero to one, with $A = 0$ indicating a completely concrete solution and $A = 1$ indicating a completely abstract solution. *Afferent Coupling (Ca)* indicates the number of classes outside of the selected package which depend on the classes inside the package [24]. It also indicates the level of responsibility of the given package. *Efferent Coupling (Ce)* indicates the number of classes inside a package which depend on classes from other packages [24]. *Instability (I)* is obtained using $Ce/(Ca + Ce)$. It indicates the level of instability of a package, where $I = 0$ indicates a completely stable package and $I = 1$ a completely unstable package [24]. *Normalized Distance from Main Sequence (Dn)* is calculated using $|(A + I) - 1|$. It measures how far away a package is from the idealized line $A + I = 1$, called the Main Distance [24]. The distance should be close to zero for packages with a good balance between stability and abstractness. *Lack of Cohesion of Methods (LOCOM)* is calculated using $(M - \text{sum}(MF)/F)(M - 1)$, where M is the number of methods in the class, F is the number of fields, MF is the number of methods accessing a particular field of the class, and $\text{sum}(MF)$ is the sum of MF over all fields of the class. As a measure of the cohesiveness of a class, this metric is calculated using the Henderson-Sellers method [25]. A low LOCOM indicates a cohesive class, while a LOCOM close to 1 indicates lack of

Table 1: OO metrics from original Super Jumper and cloned MEnDiGa version.

Metric property	Original Super Jumper	Cloned MEnDiGa version
Abstractness (A)	0,107	0,042
Afferent Coupling (Ca)	14,8	2
Depth of Inheritance Tree (DIT)	2,231	1,87
Efferent Coupling (Ce)	13	17
Instability (I)	0,41	0,895
Lack of Cohesion of Methods (LOCOM)	0,023	0,267
Cyclomatic Complexity	1,576	2,18
Average Lines of Code per Method	7,129	7,57
Average Nested Block Depth	1,242	1,6
Normalized Distance from Main Sequence (Dn)	0,483	0,064
Number of Attributes	44	90
Number of Classes	65	23
Number of Methods	131	94
Number of Overridden Methods	43	18
Number of Packages	5	1
Average Number of Parameters	0,856	0,64
Number of Static Attributes	41	72
Number of Static Methods	1	6
Total Lines of Code	1912	1310
Weighted Methods per Class (WMC)	208	218

cohesion. *Average Nested Block Depth* indicates the average number of nested code blocks in the selected scope. Too many nested blocks lead to a more complex and less readable solution.

Cyclomatic Complexity is used to identify the complexity of a piece of code based on the amount of execution flows [26]. It is the number of independent paths in the source code and it is calculated for methods only. High values of this metric imply a complex solution that can be difficult to understand. *Weighted Methods per Class (WMC)* is the sum of the complexity of all methods in the selected class [27] (for this work, the measure of complexity of methods is Cyclomatic Complexity). Classes with high WCM tend to be complex and hard to reuse. *Depth of Inheritance Tree (DIT)* is the maximum length from the class to the root of the inheritance tree [27]. Hierarchies of classes with high DIT can contribute to reuse because the deeper classes in the hierarchy inherit more methods. However, very deep trees tend to be more complex because there are more classes and methods involved [27].

Analyzing the obtained metric results, it is possible to verify the following:

(i) *Normalized Distance from Main Sequence (Dn)*, *Afferent Coupling (Ca)*, *Efferent Coupling (Ce)*, and *Instability (I)* show strong game dependency of the cloned version to MEnDiGa structures in contrast with original game version.

(ii) *Abstractness (A), Depth of Inheritance Tree, Total Lines of Code, Number of Methods, Number of Static Methods, Average Number of Parameters, Number of Packages, Number of Classes*, and *Number of Overridden Methods* metrics confirm the structural simplification of the cloned version in comparison with the original version, despite a higher *Number of Attributes* and *Number of Static Attributes*.

(iii) *Average Lines of Code per Method, Average Nested Block Depth, Cyclomatic Complexity*, and *Weighted Methods per Class (WMC)* indicate a small increment in the complexity of game production with MEnDiGa.

(iv) *Lack of Cohesion of Methods (LOCOM)* presents a higher result in MEnDiGa version, something that can be explained due to *Behavior* and *Observer* classes created to configure and monitore other classes instead of itself.

6. Conclusions and Future Work

This paper presented MEnDiGa, a game engine proposal based on the simplification of the NESI and GDS feature models. For this, MEnDiGa provides a minimal collection of necessary features capable of designing small and casual games. MEnDiGa also provides an implementation framework that can be configured and performed according to

game designer intentions. Together, these MEnDiGa artifacts are able to realize the *G-factor* portability followed by NESI and GDS models and provide a product line solution capable of building *G-factor* based games in large scale.

Regarding the game platform portability, MEnDiGa assets were implemented and adapted to be interpreted using LibGDX game engine. Per FEnDiGa results [7] and by the production of respective adapter classes, it is possible to affirm that MEnDiGa structure is capable of being extended to additional Java game engines, such as jMonkeyEngine [28], JGame [29], and GTGE [30]. For other game platforms based on different types of programming languages, such as Unreal Engine 4 [31] that uses C++, it is necessary to reimplement MEnDiGa classes to the respective support language. To facilitate this conversion process among game engines based on distinct programming languages, a common XML specification of MEnDiGa games will be defined in the future for generative [5] and interpretive [7] game development approaches.

Regarding the production of games from interactive GUIs, game platforms with graphical support environment such as Unity [32], Godot [33], and Scratch [34] have been widely used to produce digital games. However, it is important to reinforce the fact that important game engines still use the API programming approach to implement the game logic for designed games, such as PixiJS [35] and Panda3D [36]. As described in this paper, MEnDiGa follows the traditional API programming approach to implement desired behaviors, define observer criteria, and "configure" classes of the proposed MEnDiGa framework. In the future, a graphical support environment for MEnDiGa will be developed to allow the visual configuration of a future XML representation of MEnDiGa games.

Moreover, regarding the complexity and variability in the production of digital games, dedicated game platforms for specific game categories [6], such as RPG Maker [37] and Adventure Game Studio [38], have been well accepted in digital game productions. MEnDiGa in its current modeling does not include specific game domains resources, being focused on providing generic elements available in casual and small games. To improve MEnDiGa as a dedicated game platform, it is necessary to define features and classes able to represent game structures of specific game categories, such as menus, HUDs, user interfaces, game rules, and game elements. The provided collection of dedicated features and classes will be able to produce casual games for specific game categories in a highly reusable way, evolving MEnDiGa consequently to the status of product line for specific game domains in the future.

Finally, clone implementation of the Doodle Jump game using MEnDiGa assets was also demonstrated in this paper. As an equivalent example of casual games available today, the developed clone game has similar mechanics, dynamics, and aesthetics characteristics [39] in comparison to the original Super Jumper version. It is an important verification/validation step of this project as an attempt to show the feasibility of MEnDiGa assets in the generation of concrete digital games. Some OO metrics were also collected from the original Super Jumper and cloned MEnDiGa version. By comparison, they confirm that, with a simpler structure and a small increase in complexity, MEnDiGa allows the configuration of digital games from a core structure that follows the *G-factor* concept of game portability across distinct game platforms.

Conflicts of Interest

The authors declare that they have no conflicts of interest regarding any product or concept discussed in this article.

Acknowledgments

The authors acknowledge the Foundation for Research Support of the State of Bahia (FAPESB) for granting a scholarship to the graduate and coauthor Filipe M. B. Boaventura during the development of a preliminary proposal of this work [40].

References

[1] G. Jason, *Game Engine Architecture*, CRC Press, 2009.

[2] M. Lewis and J. Jacobson, "Games engines in scientific research," *Communications of the ACM*, vol. 45, no. 1, p. 21, 2002.

[3] A. BinSubaih and S. Maddock, "Game Portability Using a Service-Oriented Approach," *International Journal of Computer Games Technology*, vol. 2008, Article ID 378485, 7 pages, 2008.

[4] V. Sarinho and A. Apolinário, "Feature Model Proposal for Computer Games Design," in *Proceedings of the VII Brazilian Symposium on Computer Games and Digital Entertainment*, pp. 54–63, 2008.

[5] V. T. Sarinho and A. L. Apolinário, "A generative programming approach for game development," in *Proceedings of the 8th Brazilian Symposium on Games and Digital Entertainment (SBGAMES' 09)*, pp. 83–92, Rio de Janeiro, Brazil, October 2009.

[6] M. Wolf, *The Medium of the Video Game*, University of Texas Press, Tex, USA, 2002.

[7] V. T. Sarinho, A. L. Apolinário Jr., and E. S. Almeida, "A feature-based environment for digital games," in *Proceedings of the 10th International Conference on Entertainment Computing (ICEC' 12)*, vol. 7522, pp. 518–523, Springer, Berlin, Germany, 2012.

[8] V. Sarinho and A. Apolinário, "Detailing the UML Profile of the OOFM Technique," in *Proceedings of the 3rd Brazilian Workshop on Model Driven Development (WB-DSDM' 12)*, vol. 8, pp. 25–32, 2012.

[9] E. Fayad, C. Schmidt, and R. Johnson, *Building Application Frameworks Object-Oriented Foundations of Framework Design*, John Wiley Sons, 1999.

[10] E. Folmer, "Component based game development: a solution to escalating costs and expanding deadlines?" in *Proceedings of the 10th International ACM SIGSOFT Symposium Component-Based Software Engineering (CBSE' 07)*, vol. 4608, Springer, Berlin, Germany, 2007.

[11] W. Zhang and S. Jarzabek, "Reuse without Compromising Performance: Industrial Experience from RPG Software Product Line for Mobile Devices," in *Proceedings of the 9th International Conference on Software Product Lines (SPLC' 05)*, vol. 3714 of *Lecture Notes in Computer Science*, pp. 57–69, Springer, Berlin, Gemany, 2005.

[12] E. Albassam and H. Gomaa, "Applying software product lines to multiplatform video games," in *Proceedings of the 2013 3rd International Workshop on Games and Software Engineering: Engineering Computer Games to Enable Positive, Progressive Change (GAS' 13)*, pp. 1–7, San Francisco, CA, USA, May 2013.

[13] A. W. B. Furtado, A. L. M. Santos, and G. L. Ramalho, "SharpLudus revisited: From ad hoc and monolithic digital game DSLs to effectively customized DSM approaches," in *Proceedings of the Compilation of The Co-Located Workshops on DSM'11, TMC'11, AGERE! 2011, AOOPES'11, NEAT'11, & VMIL'11*, pp. 57–62, ACM, Portland, Oregon, USA, October 2011.

[14] A. W. B. Furtado and A. L. M. Santos, "Using domain-specific modeling towards computer games development industrialization," in *Proceedings of the 6th OOPSLA Workshop on Domain-Specific Modeling (DSM' 06)*, October 2006.

[15] V. M. Müller, "An open source architecture for building interactive dramas," in *Proceedings of the 10th Brazilian Symposium on Computer Games and Digital Entertainment (SBGames' 11)*, pp. 89–100, Salvador, Brazil, November 2011.

[16] M. C. Machado, G. L. Pappa, and L. Chaimowicz, "Characterizing and modeling agents in digital games," in *Proceedings of the XI Brazilian Symposium on Computer Games and Digital Entertainment*, pp. 26–33, 2012.

[17] K. Kang, S. Cohen, J. Hess, W. Novak, and S. Peterson, "Feature-oriented domain analysis (FODA): feasibility study," Tech. Rep., Software Engineering Institute, Pa, USA, CMU/SEI-90-TR-21, 1990.

[18] M. Antkiewicz and K. Czarnecki, "FeaturePlugin: feature modeling plug-in for eclipse," in *Proceedings of the 004 OOPSLA Workshop on Eclipse Technology eXchange*, pp. 67–72, Vancouver, British Columbia, Canada, October 2004.

[19] K. Czarnecki, "Overview of generative software development," in *Proceedings of Unconventional Programming Paradigms (UPP)*, vol. 3566 of *Lecture Notes in Computer Science*, pp. 326–341, Springer, Berlin, Germany, 2004.

[20] D. A. Beuche and M. A. Dalgarno, "Software product line engineering with feature models," *Methods & Tools*, vol. 14, no. 4, pp. 9–17, 2006.

[21] LibGDX, "Desktop/Android/BlackBerry/iOS/HTML5 Java game development framework," http://libgdx.badlogicgames.com.

[22] R. Pressman, *Engenharia de Software: Uma Abordagem Profissional*, McGraw-Hill, 7th edition, 2011.

[23] Eclipse Metrics Plugin, http://metrics.sourceforge.net/.

[24] R. Martin, "OO Design Quality Metrics An Analysis of Dependencies," in *Proceedings of the in Proceedings of the Workshop Pragmatic and Theoretical Directions in Object-Oriented Software Metrics (OOPSLA '94)*, 1994.

[25] B. Henderson-Sellers, *Object-Oriented Metrics: Measures of Complexity*, Prentice Hall, 1996.

[26] T. J. McCabe, "A complexity measure," *IEEE Transactions on Software Engineering*, vol. SE-2, no. 4, pp. 308–320, 1976.

[27] S. R. Chidamber and C. F. Kemerer, "A Metrics Suite for Object Oriented Design," *IEEE Transactions on Software Engineering*, vol. 20, no. 6, pp. 476–493, 1994.

[28] jMonkeyEngine, A cross-platform game engine for adventurous Java developers, http://jmonkeyengine.org/.

[29] JGame - a Java/Flash game engine for 2D games, http://www.13thmonkey.org/~boris/jgame/.

[30] GTGE, Golden T Game Engine - Game Programming for Java Programmer, http://goldenstudios.or.id/products/GTGE/.

[31] Unreal, Unreal Engine 4, https://www.unrealengine.com/what-is-unreal-engine-4.

[32] Unity, Unity 3D Game Engine, https://unity3d.com.

[33] Godot, https://godotengine.org/.

[34] Scratch, "Imagine, Program, Share", https://scratch.mit.edu/.

[35] PixiJS, PixiJS v4 - The HTML5 Creation Engine, http://www.pixijs.com/.

[36] Panda3D, Free 3D Game Engine, https://www.panda3d.org/.

[37] RPG Maker, https://en.wikipedia.org/wiki/RPG_Maker.

[38] Adventure Game Studio - AGS, https://www.adventuregamestudio.co.uk/.

[39] R. Hunicke, M. Leblanc, and R. Zubek, "MDA: A formal approach to game design and game research," in *Proceedings of the AAAI-04 Workshop on Challenges in Game AI*, pp. 1–5, July 2004.

[40] V. T. Sarinho and F. M. B. Boaventura, "Uma Proposta de Motor de Jogos Baseado em um Conjunto Simplificado de Features de Jogos Digitais," in *Anais da ERBASE - Escola de Computação Bahia-Alagoas-Sergipe*, pp. 1–10, 2014.