

A Generative Programming Approach for Game Development

Victor T. Sarinho* Antônio L. Apolinário

State University of Feira de Santana, Brazil

Abstract

Nowadays, due to the great distance between design and implementation worlds, different skills are necessary to create a game system. To solve this problem, a lot of strategies for game development, trying to increase the abstraction level necessary for the game production, were proposed. In this way, a lot of game engines, game frameworks and others, in most cases without any compatibility or reuse criteria between them, were developed. This paper presents a new generative programming approach, able to increase the production of a digital game by the integration of different game development artifacts, following a system family strategy focused on variable and common aspects of a computer game. As result, high level abstractions of games, based on a common language, can be used to configure metaprogramming transformations during the game production, providing a great compatibility level between game domain and game implementation artifacts.

Keywords: generative game development, game feature models, game specification language, game metaprogramming.

Authors' contact:

*vsarinho@gmail.com

apolinario@ecomp.uefs.br

1. Introduction

During a game development project, according to the design and implementation diversity, the game production becomes highly complex and expensive [Sarinho and Apolinário 2008].

To change this scenario, new approaches for game production, able to organize the game development process, using available engines and frameworks, are necessary [Furtado and Santos 2006].

An interesting approach able to perform these objectives in a software production is the generative programming. According Czarnecki and Eisenecker [2000], generative programming is “the process to generate programs where automated source code creation is done through code generators to improve programmer productivity”.

The difference of generative programming by other approaches is because it aims to automate the software development process using a wide range of static and dynamic technologies, including metaprogramming, reflection, program and model analysis, for example [Czarnecki and Kim 2005]. It aims to model and implement “system families” in such a way that a given system can be automatically generated from a

specification written in one or more textual or graphical domain specific languages [Czarnecki 2004].

Nevertheless, to define a generative programming approach for game development, a lot of available development techniques and domain aspects used during the game production must be considered. Therefore, questions like: “How the generative programming can be applied on game development?”, “How many game software artifacts are necessary to achieve an automatic source code generation for a game development?”, and “How the commonality and variability can be represented and used during the game development?” are raised.

This paper presents a generative programming proposal for generic game development. It is based on game feature models, able to represent variable and common implementation aspects of computer games, and metaprogramming resources, able to represent and generate compatible source code for available game engines and game frameworks.

It is organized as follows: Section 2 presents some generative software artifacts, and their related game works, necessary for the game development. Section 3 describes the *GameSystem*, *DecisionSupport* and *SceneView* (GDS) feature model; the *Game Specification Language* (GSL); and the metaprogramming approach necessary for the game development. Section 4 presents the Pac-Man case study using the generative programming proposal. Finally, Section 5 presents some conclusions about the paper.

2. Related Work

Next subsections will present important artifacts for generative programming that will be used in this paper. In addition, some game researchs for each type of generative artifact will be described.

2.1 Feature Model Artifacts

According Czarnecki and Kim [2005], the feature modeling has several applications in generative software development, including domain analysis, product-line scoping, and feature-based product specification.

Feature modeling is a technique for managing software commonalities and variabilities. It can be used to: capture the results of domain analysis; facilitate scoping of product lines, domain-specific language families, components, platforms, and other reusable assets; and provide a basis for automated configuration of concrete products, languages, components, platforms, etc [Czarnecki and Kim 2005].

In a game development perspective, Zhang and Jarzabek [2005] proposed the application of feature models, defining similarities and differences among four different RPGs, to configure a RPG product line architecture (RPG-PLA).

To represent a generic game design, Sarinho and Apolinário [2008] proposed the **NESI** feature model, able to represent conceptual game aspects in four main views: Narrative, Entertainment, Simulation and Interaction.

Another interesting work was presented by Furtado [2006] where a game ontology to set game implementations was proposed. It was not a feature model, but some similar characteristics were presented in this research.

Unfortunately, none of them are able to represent common and variable aspects of implementation in a generic computer game.

2.2 Domain-Specific Language Artifacts

According Fowler [2005], a Domain-Specific Language (DSL) is a limited form of computer language designed for a specific class of problems. It is a small, usually declarative, language that offers expressive power focused on a particular problem domain [Furtado and Santos 2006]. In many cases, DSL programs are translated to calls to a common subroutine library, and the DSL can be viewed as a way to hide the details of that library [Furtado and Santos 2006].

An interesting research about DSL for games was presented by Moreno-Ger et al. [2005], who described a suitable Domain-Specific Language to build educational games. To support this language, an abstract syntax and its operational semantics, and a specific game engine were presented.

In this work, DSLs will be useful to realize the generic game feature models in a reusable way, providing an optimal support for application developers by the available and compatible software development environments.

2.3 Metaprogramming Artifacts

According Azanza et al. [2007], “Generative programming is about developing metaprograms that synthesize other programs”, and according Batory [2006], “metaprogramming is the concept that program synthesis is a computation”.

To exemplify, Batory [2006] asserts that Model-Driven Development (MDD) is a metaprogramming paradigm, where models are program values and transformations are program functions that map these values. In fact, “the main difference between MDD and generative software development is the focus of the latter on system families” [Czarnecki 2004].

For Trujillo et al. [2007], Scripts that transform models into executables are also metaprograms. They are “programs that manipulate values that themselves are programs”.

Looking for game researchs, Cutumisu et al. [2007] presented a three-step process allowing game authors (non-programmers) to generate the necessary procedural scripts to implement meaningful interactions between the PC and game objects.

In this work, scripts that transform the DSLs models, able to represent generic characteristics of computer games, into executables will be used.

3. The Generative Approach

The main objective of the generative approach for games presented in this paper is to integrate feature models, domain-specific languages and metaprogramming scripts for games as a unique and organized approach for game implementation.

In this way, this section will present a feature model able to represent game implementation aspects, a domain-specific language able to represent the feature model as a concrete syntax, and a collection of metaprogramming scripts able to translate the DSL for game engines, frameworks and similars.

3.1 The GDS Model

The objective of the GDS model (Figure 1) is define a game as a combination of three main standard features: *GameSystem*, *DecisionSupport* and *SceneView*, where each main feature describes generic configurations and behavioral aspects of a game. It is organized as a collection of features depicting various resources of game implementation found in several related works (see reference section).

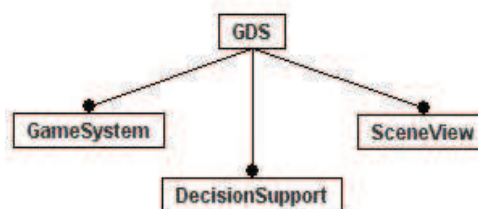


Figure 1: GDS feature model.

The *GameSystem* feature (Figure 2) is the main joint point of the game. It is responsible to control the game execution, describing available *GameBehaviors*, *GameContext* and *GameObservers* of the game. By the execution of *GameBehaviors*, activated by *GameObservers* or not, the current *Player* in the *GameSession* can trigger actions that can affect all defined data in the game, like *DecisionEntities* and *SceneNodes* for example. Some game subsystems, like *FileSystem* and *Networking* for example, can also execute some synchronous and asynchronous actions, triggering in the same way specific behaviors in the game.

The *DecisionSupport* feature (Figure 3) is an effort to integrate some AI game strategies used by different digital games. It presents *DecisionEntities*, like *Scenario* and *Agent*, the *ContextState* of each *DecisionEntity*, represented as *FiniteState* or *NeuralNet* for example, and predefined

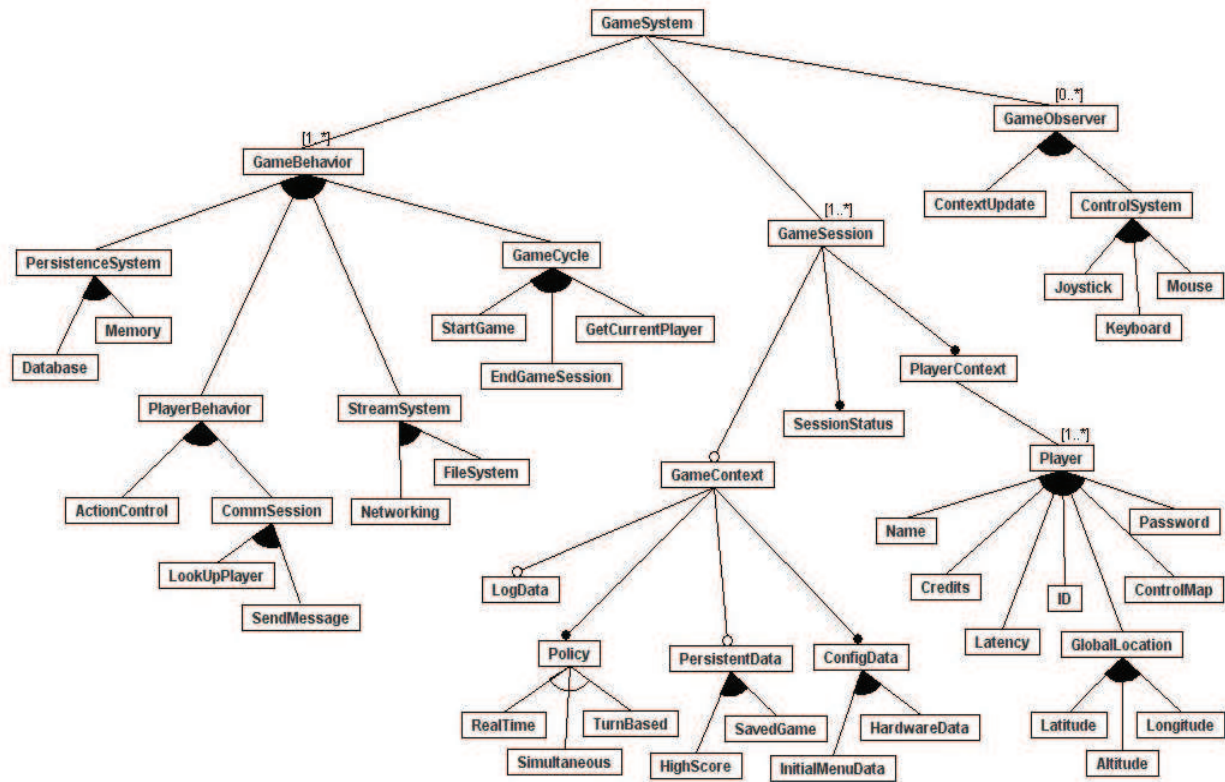


Figure 2: GameSystem feature diagram.

DecisionBehaviors, able to read and change *DecisionEntities* and *ContextStates* values during their executions.

The *SceneView* (Figure 4) feature is a collection of *SceneNodes* distributed by *Spatial* sessions with a lot of *SceneBehaviors* and *SceneObservers* to execute scene actions. Each *Spatial* session is composed by

some *SceneNodes*. A *SceneNode* represents a hierarquical information about the scene, with a specific *Location* and a *BoudingVolume* for collision detection. Each *SceneNode* can represent at the same time *AudioNode*, *GraphicNode* and *PhysicsNode* informations. *Sorting* informations about the *Spatial*, like *Portals*, *BSPs* and others can also be defined.

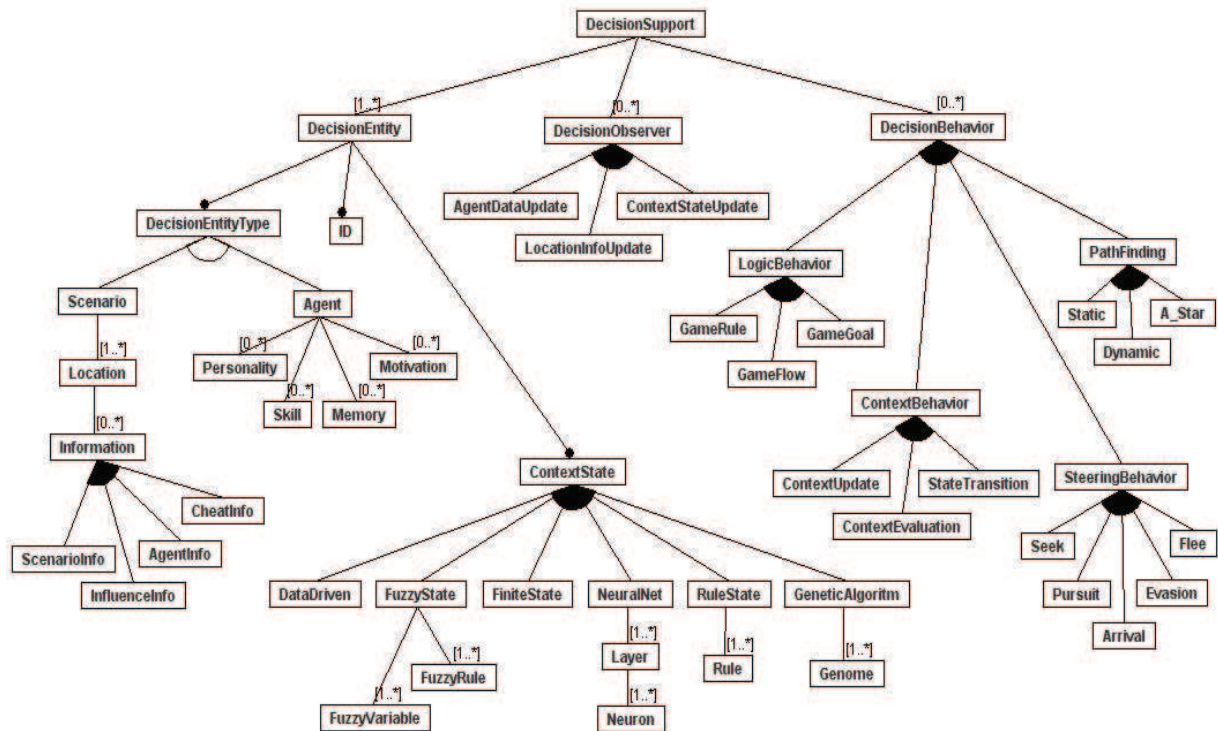


Figure 3: DecisionSupport feature diagram.

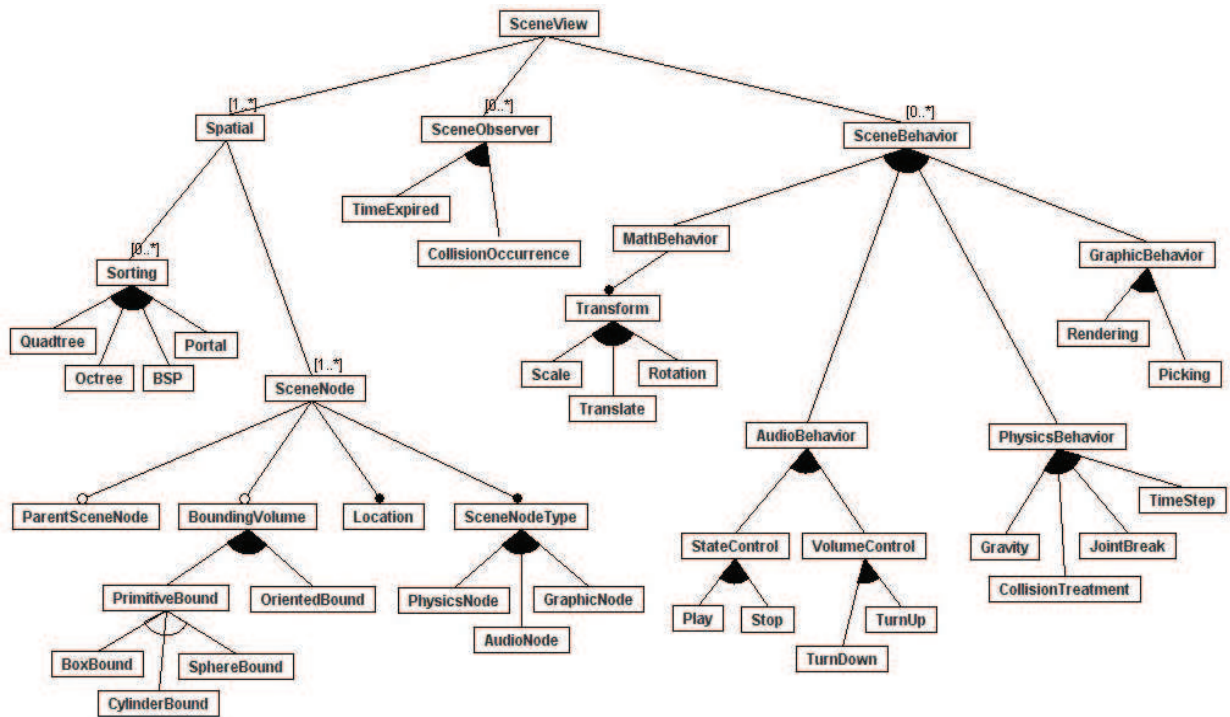


Figure 4: SceneView feature diagram.

3.2 The Game Specification Language

According [Czarnecki 2004], feature models are the starting point in the development of both DSLs and system-family architectures. DSL presents a lot of benefits: it is concise, self-documenting to a large extent, and can be reused for different purposes; it enhances productivity, reliability, maintainability and portability; it represents domain knowledge, enabling the conservation and reuse of this knowledge; and it allows validation and optimization at the domain level [Furtado and Santos 2006].

DSLs are usually declarative, offering only a restricted set of notations and abstractions. Consequently, they can be viewed as specification languages, like SQL, HTML, TeX, BNF and XML derivations, for example [Furtado and Santos 2006].

According W3C [2009], the Extensible Markup Language (XML) is a “simple, very flexible text format derived from SGML (ISO 8879)”. It was originally designed to meet the challenges of large-scale electronic publishing, allowing the exchange of a wide variety of data on the Web and elsewhere [W3C 2009].

In this way, using the GDS model as a commonality and variability guide for game implementations, and the XML as a standard text format, a Game Specification Language (GSL), which can be characterized as a textual DSL for games, is defined.

GSL presents a textual representation of the GDS model structure (taking advantage of the available XML tools), organized by the root tag *GSL* and its three main subtags: *GameSystem*, *DecisionSupport* and *SceneView*, with detailed values of attributes and subtags when necessary.

For each main subtags, a lot of tags can be defined to support the context data of a game. This context data can be represented using individual tags, like *GameContext* and *PlayerContext*, or using multiple tags, like *Players*, *DecisionEntities* and *SceneNodes*.

Subtags of *GameBehavior*, *GameObserver*, *DecisionBehavior*, *DecisionObserver*, *SceneBehavior* and *SceneObserver*, representing behavior and observer characteristics in a game, can also be defined, according the game project.

To illustrate a complete organization of these tags and subtags, section 4.1 shows an example of a GSL script, describing a simplified representation of a Pac-Man game [Iwatani 1980].

3.3 Metaprogramming Scripts for Games

Based on GDS model and XML text format, a GSL specification is a structured game description, focused on common and variable characteristics of a game implementation, where source code elements for real games can be created using a generative approach.

In fact, by the application of some translations in the GSL specification, using a specific game engine or game framework as the final target, the production of a real game based on a declarative resource (the GSL specification) is possible.

Unfortunately, different game development artifacts can be used during the game production (game engines, game makers, game frameworks, etc). As result, to support the logic and structure of a GSL specification, a specific translation process must be defined for each game development artifact used in the game production.

To solve this problem, a game metaframework, composed by a set of *core* classes able to support the

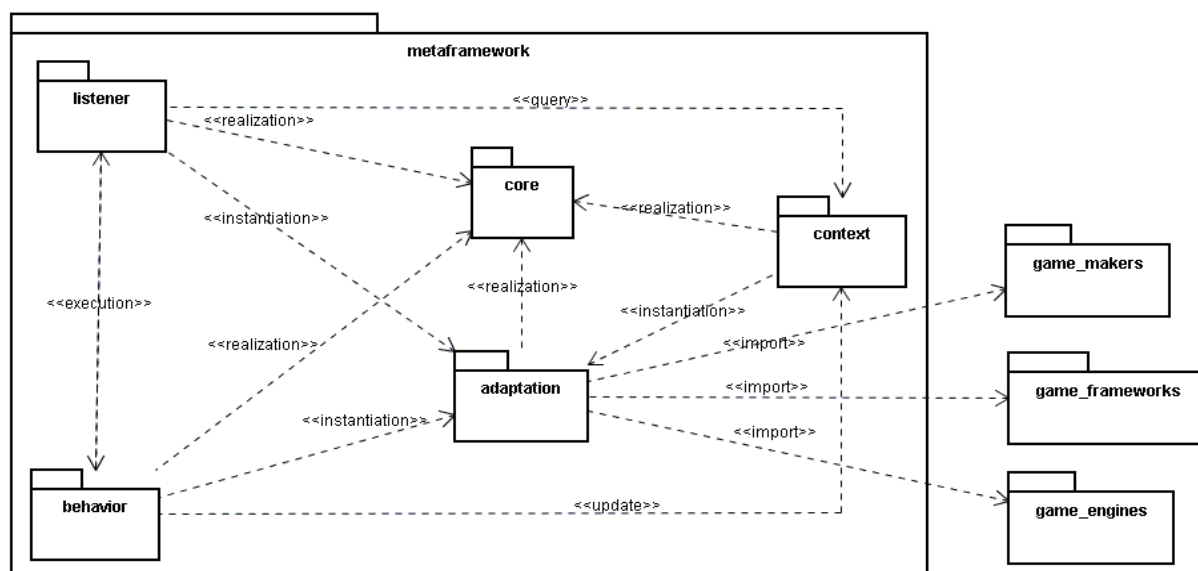


Figure 5: The metaframework package diagram.

GSL structure, that becomes the main target of the generative scripts based on GSL specifications, is proposed.

The metaframework objective is to create a communication layer between the generated code from GSL specifications with available game development artifacts. So, a direct consequence of the *adaptation* package (composed by classes that realize the *adapter* pattern [Gama et al. 1994]), necessary to maintain the compatibility between metaframework *core* classes and similars classes available in the respective game development artifact that will be reused.

Figure 5 illustrates the metaframework package diagram, showing the necessary structures to support GSL specifications: *core* (abstract structures to support GSL games), *adaptation* (collection of adapters necessary to support a new game engine or game framework), *behavior* (actions to be executed by a GSL game), *listener* (game observers that will be triggered by game subsystems or game behaviors) and *context* (game data structures that will be instantiated during the game play). External game development artifacts are also presented (*game engines*, *game frameworks* and *game makers*), showing their *import* relationships with *adaptation* structures, allowing the game portability with different game development artifacts.

Figure 6 presents a class diagram [Booch et. al. 1998] with an initial proposal for the *core* package. It describes a *GameSession* abstract class composed by three types of *GenericObservers* (*GameObserver*, *DecisionObserver* and *SceneObserver*), and some collections of game elements, like *spatials*, *decisionEntities* and *players*. A *Spatial* is composed by a collection of *sceneNodes* (which can be an *AudioNode*, a *GraphicNode* or a *PhysicsNode*). A *DecisionEntity* can be an *Agent* (with some collections of *skills*, *memories*, *motivations* and *personalities*) or a *Scenario* (with a collection of *locations* and some *informations* for each *Location*). Each *DecisionEntity*

contains informations about its *ContextState*, and each *GenericObserver* contains a collection of *GenericListener* able to execute a *GenericBehavior* when necessary.

For the metaprogramming scripts, these are described by Extensible Stylesheet Language (XSL) files [XSL 2009], able to translate GSL specifications in a compatible source code with the proposed metaframework. According Harold [1998], XSL includes both a transformation language and a formatting language. The transformation language “provides elements that define rules for how one XML document is transformed into another document”. “They are purely about moving data from one computer system or program to another”.

To illustrate the application of a XSL script to transform a GSL game to a real game, section 4.3 presents some examples of XSL rules able to produce a simplified version of the Pac-Man game (after translate the GSL specification presented in section 4.1).

4. Case Study: A Simplified Pac-Man

To exemplify an application of the proposed generative approach, a simplified version of the Pac-Man game will be designed.

This simplified project will ommit ghosts (only one ghost will be available), big pills (excluding the ghost vulnerability), difficult levels (same skills for the ghost during the game play), score, fruits, ghost house, and others.

Only the default characteristics of the Pac-Man game will be present, like: the pac-man character, one ghost character, pills, walls, collision treatment between these game elements, lifes and a treatment for the completion of the game (game over or game victory).

Initial menu, game presentation, high scores, multiplayer support and other “*GameSystems*” characteristics were also ommited.

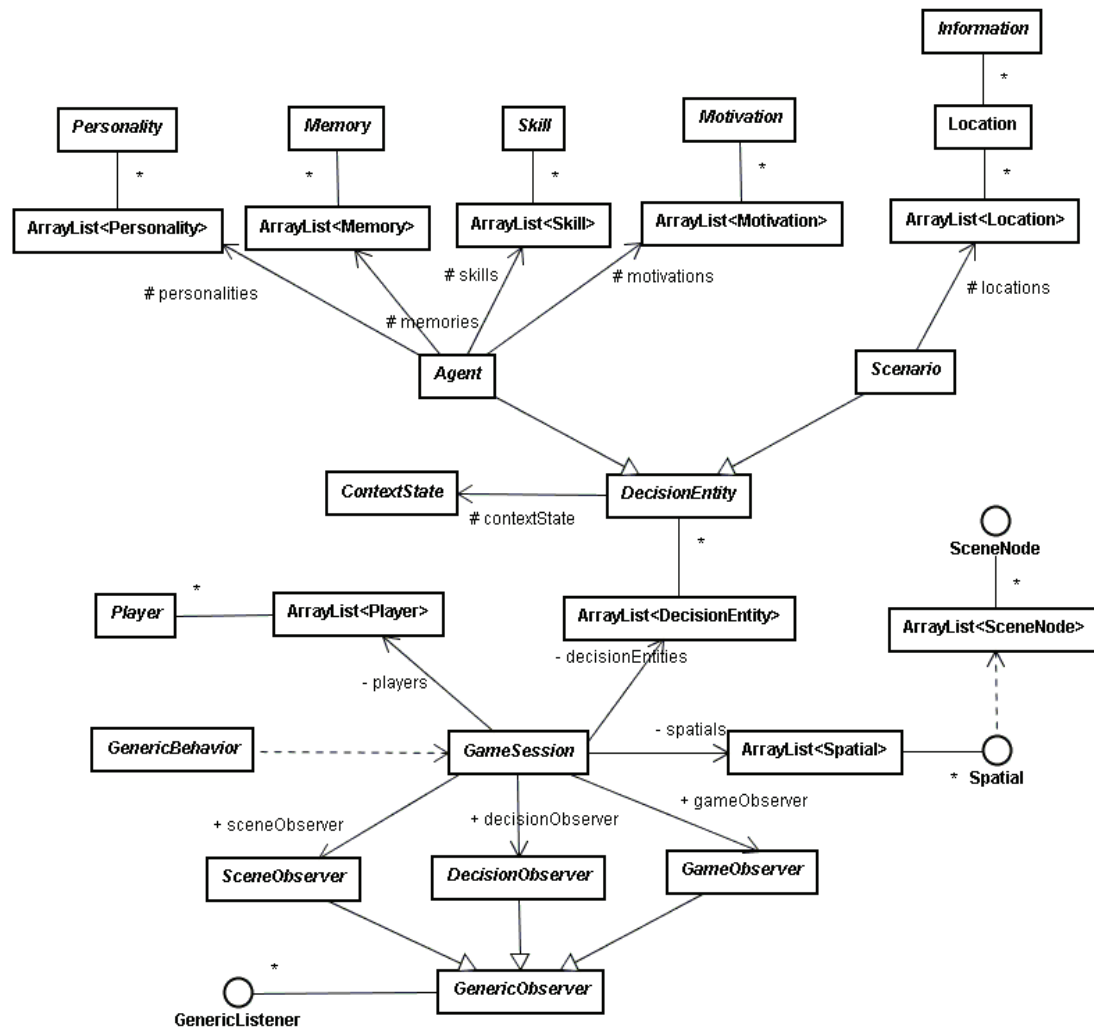


Figure 6: Class diagram for the core package.

4.1 Modeling the GSL Script

To represent the Simplified Pac-Man game, Figure 7 shows its GSL specification proposal, describing the necessary information to model and configure this game, allowing its realization according the generative approach.

For behaviors tags (*MoveGhost*, *RestoreInitialPosition*, for example) and observers tags (*UpDirection*, *LoseLifeListener*, for example), in this case, the existence of them is sufficient to represent the desired actions. Metaprogramming scripts will be responsible to decide what kind of source code will be generated by the existence of these tags.

For context tags (*DecisionEntities* and *SceneNodes*, for example), the structure and some parameters (*imageFile*, *X* and *Y* coordinates, for example) to configure the final source code, were described.

For the *Model* tag, used in all *SceneNodes*, it works with special files describing rendering and animation informations. As result, different ways to represent data file or image files are presented in the GSL specification for the *Model* tag: one image file, multiples image files, one data file, one data file and one image file, etc. Therefore, some special and

different treatments are expected in the metaprogramming scripts for each *Model* tag.

Due to space limitations, the specification of *Spatial* tags was simplified (and highlighted using different font styles) to show only the *Location* and the *Model* tags.

4.2 New Classes and Adapters

According section 3.3, the proposed metaframework is the main target for generative scripts based on GSL specifications. In this way, to allow the realization of these GSL specifications, some adaptations and new structures will be applied to the metaframework, during the execution of those generative scripts.

To illustrate these metaframework updates, Figure 8 presents a class diagram showing some new classes added to the following metaframework packages: *adaptation*, *behavior*, *listener* and *context*. These new classes and adapters were developed according the GTGE game framework [GTGE 2009], allowing the communication between them.

As result, adaptations and new classes for *Player* (*DefaultPlayer*), *GameSession* (*GameSessionGTGE* adapter), *GenericObserver* (*SceneObserverGTGE* and *GameObserverGTGE* adapters), *Spatial* (*SpatialField*

```

<GSL>
  <GameSystem>
    <GameBehavior>
      <GameCycle>
        <GameOver/> <GameVictory/>
      </GameCycle>
      <PlayerBehavior> <ActionControl/> </PlayerBehavior>
    </GameBehavior>

    <GameObserver>
      <ControlSystem>
        <Keyboard>
          <UpDirection/> <DownDirection/>
          <LeftDirection/> <RightDirection/> <ESC/>
        </Keyboard>
      </ControlSystem>
    </GameObserver>

    <GameSession>
      <SessionStatus status="GamePlay"/>
      <PlayerContext>
        <Player>
          <ID value="1"/>
          <Name value="Player1"/>
          <ControlMap>
            <LeftDirection_MoveLeft/>
            <RightDirection_MoveRight/>
            <UpDirection_MoveUp/>
            <DownDirection_MoveDown/>
            <ESC_ButtonPress/>
          </ControlMap>
        </Player>
      </PlayerContext>
    </GameSession>
  </GameSystem>

  <DecisionSupport>
    <DecisionBehavior>
      <LogicBehavior>
        <GameRule> <LoseLife/> </GameRule>
        <GameFlow>
          <RestoreInitialPosition/>
        </GameFlow>
      </LogicBehavior>
    </DecisionBehavior>

    <DecisionObserver>
      <AgentDataUpdate>
        <LoseLifeListener/>
      </AgentDataUpdate>
    </DecisionObserver>

    <DecisionEntity>
      <ID value="PacMan"/>
      <DecisionEntityType>
        <Agent>
          <PacMan>
            <Skill> <Life value="2"/> </Skill>
          </PacMan>
        </Agent>
      </DecisionEntityType>
    </DecisionEntity>

    <DecisionEntity>
      <ID value="CornerScenario"/>
      <DecisionEntityType>
        <Scenario>
          <Location file="corner.dat">
            <Information>
              <ScenarioInfo> <Ways/> </ScenarioInfo>
            </Information>
          </Location>
        </Scenario>
      </DecisionEntityType>
    </DecisionEntity>
  </DecisionSupport>

  <SceneView>
    <SceneBehavior>
      <MathBehavior>
        <Translate>
          <MoveGhost/> <MovePacMan/>
        </Translate>
      </MathBehavior>
      <GraphicBehavior>
        <Rendering> <UpdateHUD/> </Rendering>
      </GraphicBehavior>
      <PhysicsBehavior>
        <CollisionTreatment>
          <PacManPillCollisionTreatment/>
          <PacManWallCollisionTreatment/>
          <GhostWallCollisionTreatment/>
          <PacManGhostCollisionTreatment/>
        </CollisionTreatment>
      </PhysicsBehavior>
    </SceneBehavior>

    <SceneObserver>
      <CollisionOccurrence>
        <PacManPillCollision/>
        <PacManWallCollision/>
        <GhostWallCollision/>
        <PacManGhostCollision/>
      </CollisionOccurrence>
    </SceneObserver>

    <Spatial ID="GamePlay">
      <SceneNode>
        <HUDNode> ...
        <Location X="X_HUD_INIT" Y="Y_HUD_INIT"/> ...
        <Model imageFile="twoLifes.png"/> ...
      </HUDNode>
      <WallNode> ...
        <Location X="X_INIT" Y="Y_INIT"/> ...
        <Model dataFile="scenario.dat" imageFile="wall.png"/> ...
      </WallNode>
      <PillNode> ...
        <Location X="X_INIT" Y="Y_INIT"/> ...
        <Model dataFile="pills.dat" imageFile="pill.png"/> ...
      </PillNode>
      <GhostNode> ...
        <Location X="X_GHOST_INIT" Y="Y_GHOST_INIT"/> ...
        <Model>
          <Image fileName="ghost1.png"/>
          <Image fileName="ghost2.png"/>
        </Model> ...
      </GhostNode>
      <PacManNode> ...
        <Location X="X_PACMAN_INIT" Y="Y_PACMAN_INIT"/> ...
        <Model>
          <Image fileName="pacman0.png"/>
          <Image fileName="pacman1.png"/>
          <Image fileName="pacman2.png"/>
          <Image fileName="pacman3.png"/>
          <Image fileName="pacman4.png"/>
        </Model> ...
      </PacManNode>
    </SceneNode>
  </Spatial>

    <Spatial ID="GameVictory"> ...
    <Location align="center"/> ...
    <Model imageFile="victory.png"/> ...
  </Spatial>

    <Spatial ID="GameOver"> ...
    <Location align="center"/> ...
    <Model imageFile="gameover.png"/> ...
  </Spatial>
</SceneView>
</GSL>

```

Figure 7: GSL specification of the Simplified Pac-Man.

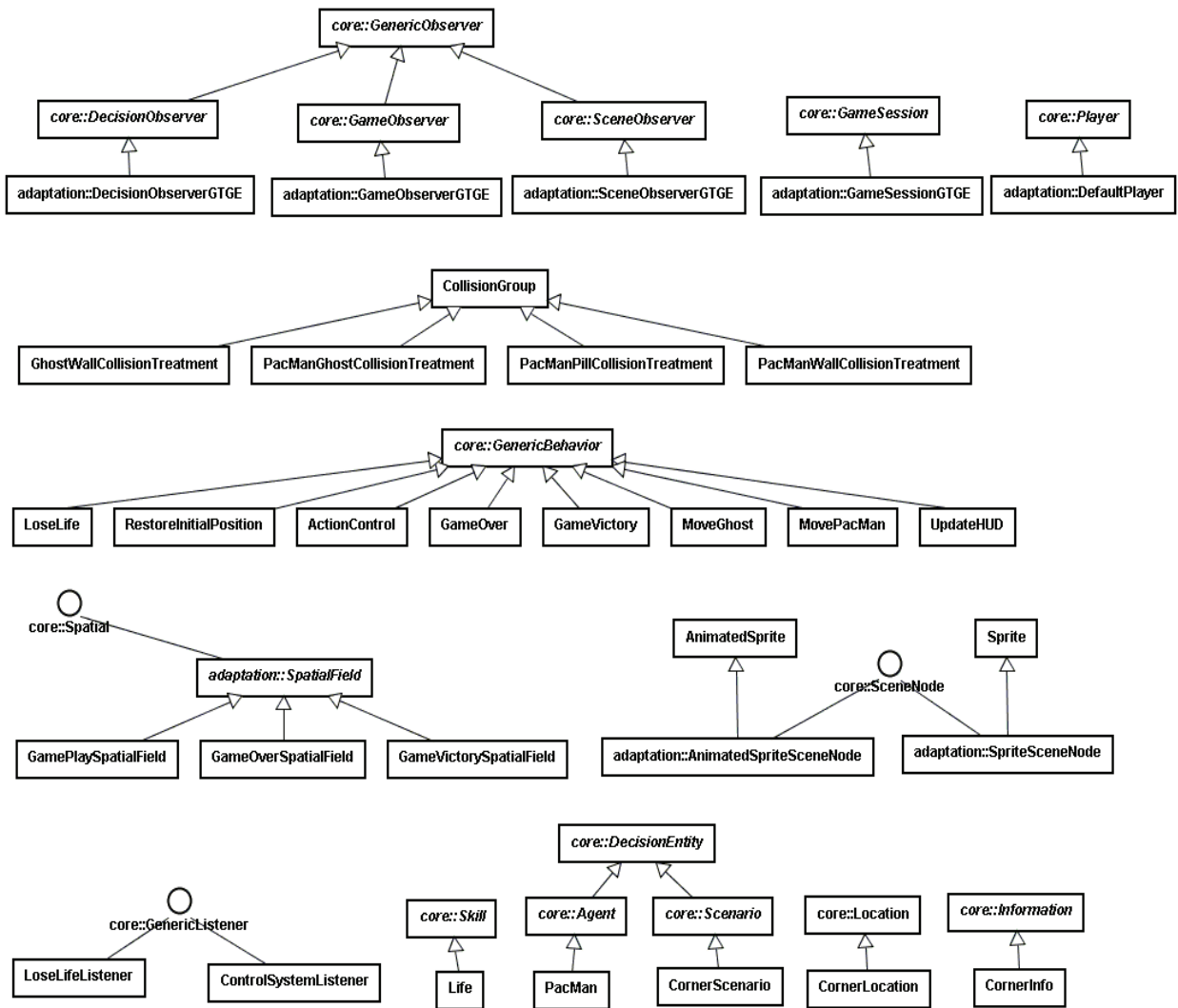


Figure 8: Class diagram with new classes for adaptation, behavior, listener and context packages.

adapter and some context classes like *GamePlaySpatialField*, *SceneNode* (*SpriteSceneNode* adapter), *DecisionEntity* (*CornerLocation*, *Life*), *GenericListener* (*ControlSystemListener*), and a lot of subclasses of *CollisionGroup* and *GenericBehavior*, were presented.

4.3 Building Games with XSL Rules

To create these game structures described above (classes and adapters) for the proposed metaframework, a set of metaprogramming scripts, able to translate the GSL specification in a compatible source code, must be created.

These metaprogramming scripts will be defined using the Extensible Stylesheet Language (XSL) format, which can define rules that transform one GSL specification to a desired source code (according section 3.3), like classes and adapters for example.

Some examples of XSL rules, able to translate the GSL specification of the Simplified Pac-Man to a real game, were illustrated in Figure 9. These rules are validated by templates, where, for each template validation, one of these transformations is expected:

direct code introduction inside a class for each identified element, internal template validation for each identified element to introduce a code, direct code introduction inside a class, direct introduction of GSL values inside a class, and a complete class introduction in the final code.

As result, after the XSL rules execution described in Figure 9, the following resources will be available to the Simplified Pac-Man game: creation of all declared *Players*, *Keyboard* event treatment, introduction to decision support code for *PacMan*, source code inclusion of the *SessionStatus* value, and the creation of a class to support the *GameOver* behavior.

4.4 Metaframework Integration

During the integration process between a game engine or a game framework with the proposed metaframework, a lot of special resources of the game development artifact can be used, instead of the metaframework structures.

For example, the GTGE framework gives a special collection of classes to work with collision between game objects. In this way, subclasses of the

```

#### Direct code introduction inside a class for each identified element:
<xsl:for-each select="/GSL/GameSystem/GameSession/PlayerContext/*">
    gameSession.addPlayer(new DefaultPlayer(<xsl:value-of select="@ID/@value"/>,<xsl:value-of select="Name/@value"/>));
</xsl:for-each>

#### Internal template validation for each identified element to introduce a code:
<xsl:for-each select="/GSL/GameSystem/GameObserver/ControlSystem/Keyboard/*">
    <xsl:choose>
        <xsl:when test="name(.)='UpDirection'">
            if (keyDown(KeyEvent.VK_UP)) {
                String[] values = {"UpDirection"};
                gameSession.gameObserver.fire(gameSession, "GameSystem&#47;GameObserver&#47;ControlSystem", values);
            }
        </xsl:when>
        <xsl:when test="name(.)='DownDirection'">
            ...
        </xsl:when>
    </xsl:choose>
</xsl:for-each>

#### Direct code introduction inside a class:
<xsl:if test="/GSL/DecisionSupport/DecisionEntity/ID/@value='PacMan'">
    gameSession.addDecisionEntity(GameDecisionData.createPacMan());
</xsl:if>

#### Direct introduction of GSL values inside a class:
gameSession.setStatus("<xsl:value-of select="/GSL/GameSystem/GameSession/SessionStatus/@status"/>");

#### Complete class introduction in the final code:
<xsl:template match="/GSL/GameSystem/GameBehavior/GameCycle/GameOver">
    class GameOver extends GenericBehavior {
        public void execute(GameSession gameSession, Object[] params){
            gameSession.setStatus("GameOver");
        }
    }
</xsl:template>

```

Figure 9: Some XSL rules to transform the GSL specification of the Simplified Pac-Man game.

CollisionGroup like *PacManGhostCollisionTreatment* are presented in the class diagram (Figure 8). But, according the GSL specification (Figure 7), *PacManGhostCollisionTreatment* is a *SceneBehavior*. So, it should be a subclass of the *GenericBehavior* class (defined in the *core* package of the metaframework - Figure 6), like *UpdateHUD* and *MoveGhost* (Figure 8), instead of a GTGE class.

In the same way, none decision information about the Pac-Man position were specified in the GSL specification (Figure 7). This information is directly extracted by available GTGE resources, avoiding the creation of *Information/AgentInfo/PacManPosition* tags for each *Location* in the *Scenario*.

These decisions about what type of game logic or game structure to be used during the generative script production are, in most cases, defined by the “game programmer”. The “game programmer” is the person who knows the available game engine or game framework resources. The “game programmer” is the best person who can define the best solution of what types of game structures must be reused or not by game engines or game frameworks. The “game programmer” is the responsible to decide what types of *GameSystem*, *DecisionSupport* and *SceneView* structures will be excluded during the generative process.

4.5 The Final Game

To show the final result of the XSL execution, transforming the proposed GSL specification (Figure

7) in a real game, Figure 10 presents a captured image of the Simplified Pac-Man during its game play.

Although different techniques had been used to develop this final game, like the GDS feature model and the proposed metaframework, some game updates can be performed in a faster way, according the generative game artifact.

For example, to change the initial quantity of lifes available to the player in the Simplified Pac-Man, two simple updates in the GSL specification (Figure 7) are necessary: a new value to the *Life* tag attribute, and a new initial image to the *HUDNode* tag.

As result, the maintainability of the Simplified Pac-Man can be described by single updates on a specific generative artifact used to produce it.

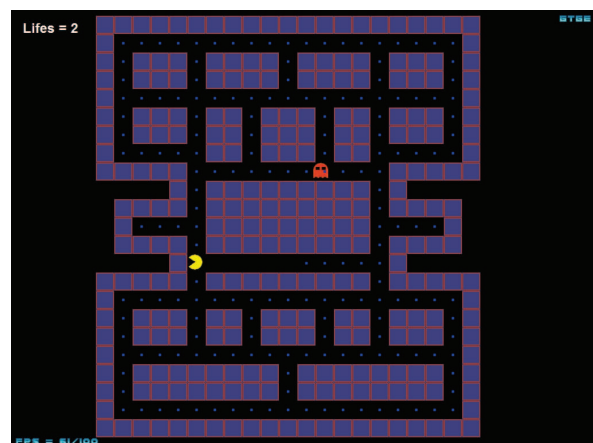


Figure 10: Simplified Pac-Man during the game play.

5. Conclusions

This paper presented a generative attempt to develop computer games in general, using different technologies to complete this task, like: the GDS feature model, the GSL format, the metaframework proposal and some metaprogramming scripts based on XSL format.

The great objective of this work is to describe a different game development experiment, able to increase the level of software reuse and maintainability, allowing the creation and the adaptation of new games by declarative specifications, metaprogramming definitions and metaframework updates.

As a result of this approach, new development strategies can be applied during the game production, according the final game environment (engines, frameworks or platforms, for example), resulting in a low coupling development strategy between game domain and game implementation software artifacts.

For future works, two main researchs will be developed. First, the creation of collections of metaframework adaptations for different game artifacts to be used in a metagenerative approach for games. Second, the definition of collections of game domain resources by the analysis of different game topics, game genres, game platforms, etc, allowing the creation of a repository of game dynamics [Hunicke et al. 2004], using feature models and GSL specifications as a common language.

References

- AZANZA, M., TRUJILLO, S. AND DIAZ, O., 2007. Towards Generative Metaprogramming. In: *2nd Summer School on Generative and Transformational Techniques in Software Engineering (GTTSE 2007)*, Braga, Portugal, 2-7 July 2007.
- BATORY, D., 2006. Multi-Level Models in Model Driven Development, Product-Lines, and Metaprogramming. *IBM Systems Journal*, 45(3).
- BOOCH, G., RUMBAUGH, J. AND JACOBSON, I., 1998. The Unified Modeling Language User Guide. Addison-Wesley. ISBN-10: 0201571684.
- CUTUMISU, M., ONUCZKO, C., CRAWFORD, C., McNAUGHTON, M., ROY, T., SCHAEFFER, J., SCHUMACHER, A., SIEGEL, J., SZAFRON, D., WAUGH, K., CARONARO, M., DUFF, H., AND GILLIS, S., 2007. ScriptEase: A generative/adaptive programming paradigm for game scripting. In: *Science of Computer Programming, Volume 67, Issue 1 (June 2007)*, pp. 32-58. ISSN:0167-6423, Elsevier North-Holland, Inc. Amsterdam, The Netherlands.
- CZARNECKI, K., 2004. Overview of generative software development. In *UPP 2004, volume 3566 of LNCS*, pages 326--341. Springer.
- CZARNECKI, K. AND EISENECKER, U., 2000. Generative Programming. Addison-Wesley, ISBN: 0201309777.
- CZARNECKI, K. AND KIM, C., 2005. Cardinality-Based Feature Modeling and Constraints: A Progress Report. In *OOPSLA'05 International Workshop on Software Factories (online proceedings)*.
- FOWLER, M., 2005. Language Workbenches: The Killer-App for Domain Specific Languages?. Available from: <http://www.martinfowler.com/articles/languageWorkbench.htm> 1 [Accessed 07 July 2009].
- FURTADO, A AND SANTOS, A., 2006. Applying Domain-Specific Modeling to Game Development with the Microsoft DSL Tools. Tutorial (in English). In: *3rd Brazilian Symposium on Computer Games and Digital Entertainment (SBGames2006)*.
- FURTADO, A., 2006. Defining and Using Ontologies as Input for Game Software Factories. In: *Proceedings of the 3rd Brazilian Symposium on Computer Games and Digital Entertainment*.
- GAMMA, E., HELM, R., JOHNSON, R. AND VLISSIDES, J., 1994. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley Professional. ISBN 0-201-63361-2.
- GTGE, 2009. *Golden T Game Engine (GTGE)*. Golden Studios. Available from: <http://www.goldenstudios.or.id/products/GTGE/> [Accessed 07 July 2009].
- HAROLD, E., 1999. XML Bible. Wiley. ISBN-10: 0764532367
- HUNICKE, R., LeBLANC, M., AND ZUBECK, R., 2004. MDA: A formal approach to game design and game research. In: *Proceedings of the AAAI-04 Workshop on Challenges in Game AI, July 2004.*, pp. 1-5.
- IWATANI, T. 1980. Pac-Man. Namco.
- MORENO-GER, P., MARTÍNEZ-ORTIZ, I., SIERRA, J. AND FERNÁNDEZ-MANJÓN, B., 2005. Language-Driven Development of Videogames: the <e-Game> Experience. In: *Proceedings of the 5th International Conference on Entertainment Computing (ICEC 2005)*, Cambridge, UK. *Lecture Notes in Computer Science* 4161, 153-164.
- SARINHO, V. AND APOLINÁRIO, A., 2008. A Feature Model Proposal for Computer Games Design. In: *Proceedings of the VII Brazilian Symposium on Computer Games and Digital Entertainment, Belo Horizonte*, p. 54-63.
- TRUJILLO, S., AZANZA, M. AND DIAZ, O., 2007. Generative Metaprogramming. In: *6th International Conference on Generative Programming and Component Engineering (GPCE'07)*, October 1-3, Salzburg, Austria.
- W3C, 2009. *World Wide Web Consortium*. Available from: <http://www.w3.org/> [Accessed 07 July 2009].
- XSL, 2009. *The Extensible Stylesheet Language Family*. Available from: <http://www.w3.org/Style/XSL/> [Accessed 07 July 2009].
- ZHANG, W. AND JARZABEK, S., 2005. Reuse without Compromising Performance: Experience from RPG Software Product Line for Mobile Devices. In: *Proceedings of the 9th Int. Software Product Line Conf., SPLC'05, Sept. 2005, Rennes, France*, pp. 57-69.