Domain-Specific Languages

Oriol Camps Pérez

Abstract— **Domain-specific languages have been in use since many decades ago. They allow software developers to rapidly write correct programs in a specific application domain, and non-programers to develop software in its domain of expertise. Domain-specific languages have many advantages, but at the cost of generality. Furthermore, its high cost and difficulty of creation does not favour its development and, therefore, its use. These advantages and disadvantages, the motivations and the phases of development, and the different implementation approaches are included and discussed in this paper. The aim of this paper is to give an overview of the topic, as well as to link it with the games engineering field. Thereby, three examples of domain-specific languages tailored towards game development are also presented in the second half of the paper.**

I. INTRODUCTION

In Computer Science, software developers tend to simplify problems. They do so by adding new levels of abstraction, stacking layer over layer to, finally, manipulate certain operations from the comfort provided by a high-level abstraction. This high-level abstractions, constructed from the encapsulation of low-level operation knowledge, let developers spare the effort of developing all the low-level operations, while allowing them to think in terms of the higher-level concepts [1]. As E. Visser has noted: "By stacking layers of abstraction, developers can avoid reinventing the wheel in each and every project" [1].

Knowing that, one could deduct that higher or lower levels are just a matter of perspective. As a matter of fact, a former high-level abstraction could become a lower-level one, when new technologies use the old-high-level abstractions as a base for even higher-level abstractions. This is what happened when *algorithmic languages*, specifically Algol, were considered high-level in relation to assembly language, but nowadays, Algol-like languages like C or Java are considered low-level languages [1], [2].

And so, conventional *general-purpose languages* (GPLs) and their mechanisms could not be enough to create new abstractions, specially when required in a specific domain. Libraries and frameworks can be a good solution by encapsulating functionalities, but frequently the language utilized for calling or using that functionality, i.e. the *application programming interface* (API), could be cumbersome [1].

In some situations, a specific language specially designed to offer a notation tailored towards an application domain could be notably convenient, and that is what *domain-specific languages* (DSLs) aim for [3], [4], [5]. For instance, if someone needs to interact with a database, the use of SQL might be a good solution. Likewise, when someone needs to generate easily-maintainable PDFs with scientific notations and specific formats, one can use LaTeX. In both examples, knowledge in general programming is not required, even if it might be helpful.

Definition of DSL

A DSL is high-level programming language specially conceived and oriented towards a specific application or problem domain, supporting the appropriate notations and abstractions in order to focus on expressivity and development performance [1], [3], [4].

As explained by M. Mernik et al. [3], DSLs are also known as *application-oriented*, *special purpose*, *specialized*, *task-specific*, or *application* languages. This various names could also help to understand the DSL concept. Nevertheless, there is not an obvious or commonly-agreed boundary on the DSL definition [3], but we will not dive deeply into this particular subject in this paper.

It is also important to remark, that the concept and the use of DSLs are not a new thing. For instance, in February 1959 the development of APT (*Automatically Programmed Tools*), a DSL for programming numerically controlled machine tools, was officially presented in a Press Conference at MIT [6].

Executability of DSLs

This aspect of the DSLs is controversial and there is no agreement in the DSL literature [3]. But in practice, there are literally hundreds of existing languages considered DSL [7] and many have different levels of *executability*, some of them even not being executable at all [3]. While for some authors, like in van Deursen and Klint [4] and in van Deursen et al. [7], DSLs are directly considered "programming languages or executable specification languages" and so, their final goal should be to be in some way executed (not only restricted in an algorithmic sense, but, for instance, generating a document (TeX), pictures (PIC), etc. [7]); for others, such as in D. S. Wile [8], the DSL definition is much more wide.

D. S. Wile [8] expounds a definition of DSL that is not bounded on the computer science domain, considering for example, as a proper DSL, the language of music and specially its notation. In that sense, DSL would be defined as a language to express something in a specific domain in order to make it more comprehensible and effective for the users [8].

There are four different points on a DSL *executability scale* which could be differenced:

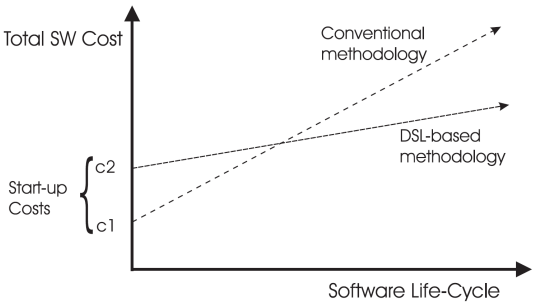
- DSL with execution semantics which are precisely defined, such as Excel macro language or HTML [3].
- Input language of an *application generator*. They typically have a declarative approach with less well-defined

execution semantics in regard to the details of the subsequent generated application, but they still are executable. The compiler of the DSL would be the application generator for which it was created in the first place [3].
– DSL not originally designed to be executable but yet, it is indeed useful for application generation. To give an example of a DSL with an essentially declarative character, which is also able to serve as an input language for a parser generator, we could look at the syntax specification formalism BNF [3].
– DSL not meant to be executable. They may still have many of the benefits of a non-executable DSL in terms of tool support, such as checkers, analysers and visualizers, prettyprinters or specialized editors. Representations of domain-specific data structures could be classified as this type of DSL [3], [8].

II. COMPARISON BETWEEN DSLs, GPLs AND LIBRARIES

Figure 1 shows a simplified graph of the use of DSLs for software development. As the figure illustrates, the start-up cost of the development with DSLs can be higher than the one of traditional methodologies, using GPLs. Nevertheless, the cost increase is lower and thus, the usage and maintenance of DSLs is more economical on later stages [9].

Fig. 1. A simplified graph of the Payoff of DSLs, taken from [9].



The development of a DSL has a big cost, because it is essential for the developers to master both the language and, of course, the tailored domain itself. This, in part, explains why it is not the most common practice to use DSLs when solving software engineering problems [5], specially when, and as explained before, a GPL combined with an *application library* could behave as a DSL [3]. In fact, it is likely that most DSLs don't even overcome the *application library* stage [3]. In that case, and specially when they are built in this way on purpose, they are also known as *domain-specific embedded languages* (DSELs) [10].

Nevertheless, in particular domains or situations, DSLs are widely used and could be seen as a better solution for several reasons, in comparison to GPLs [3].

The main advantages of a DSL are the high expressiveness and ease of use at expenses of generality, compared with a GPL. This translates into improvements in productivity (see Table I) and reductions in the maintenance costs, while at the same time, the number of possible developers increases. This is mainly because someone who is specialized in that particular domain but not in general programming, will still be capable of easily develop software for said domain. This is thanks to the provided notations, abstractions, and constructs included in the DSL that are also present on the domain itself, so after all, concepts that the person would already know [3].

Nevertheless, one could argue that libraries and a GPL could be as good as a proper DSL (and it could be in certain cases, specially when searching cost-effective solutions [3]), but there are some nuances that could show the benefits of the DSL, as well as the measured gains in productivity (see Table I):

– Appropriate or established *domain-specific notations* offered by a DSL are generally more advanced than the default notation or any user-defined notation offered by a GPL. That is because from the start a DSL can offer the pertinent related domain-specific notation characterizing the intention of the domain [3], [11].
– DSLs can represent appropriate domain-specific constructs and abstractions from the start. In contrast, it is cumbersome to express these in a GPL by the means of functions or objects from an *application library* . This is specially noticeable for error handling or model traversals [3], [11], [12].
– Further possibilities for *analysis*, *verification*, *optimization*, *parallelization*, and *transformation* that would be very difficult or even impossible to employ using a GPL [3].
– Solutions that can be quickly constructed, as well as avoid human-errors due to repetitive code sequences because DSLs hide the low-level details that could be present in a GPL while at the same time allow the programmer to focus on the key abstractions [11]. Anyway, it is fair to say that there could also be problems while reporting and handling errors in DLS [4], [5].

Giving some examples of DSLs: in Table I [3], [13] there are shown two short lists of known DSLs and GPLs, respectively, and the domain of application of each one. In it, one can compare the Level of the selected DSLs and the GPLs, as well as to observe the performance difference between them (see Table II). One might notice that the only DSL that has a worst performance and a lower level than any of the listed GPL is APT, the already mentioned in this paper DSL that dates from the late 50s [6]. All the others DSLs seem to be superior in both terms of level and performance, having less *average source statements per Function Point*. See the complete list in C. Jones [13].

Table II [3], [13] shows the relation between the *Language Level* and the productivity that the language offers, measured as *Function Points*. The statements needed in order to code

TABLE I

COMPARISON BETWEEN DIFFERENT KNOWN DSLS AND GPLS IN TERMS OF *level* AND *average source statements per function point*

DSL	Application Domain	Level	Average Source statements per function point
APT	Machine tools	4.5	71
BNF	Syntax specification	n.a.	n.a.
Excel *(version 5)*	Spreadsheets	57.0	6
HTML *(version 2)*	Hypertext web pages	20.0	16
LaTeX	Typesetting	n.a.	n.a.
Make	Software building	15.0	21
MATLAB	Technical computing	n.a.	n.a.
SQL	Database queries	25.0	13
VHDL	Hardware design	17.0	19

GPL	Level	Average Source statements per function point
Assembly (Basic)	1.5	320
C	2.5	128
C++	6.5	53
JAVA	6.5	53

Based on [3], [13].

one *Function Point* decrease as language levels increase [3], [13]. That means that the higher the level is, the higher are the abstractions which could be directly written without focusing on low-level details, and so, for the same application, fewer statements would be required.

Accurate productivity rates can be obtained by looking at the *average monthly Function Point production* related to each *language level*, instead of looking just to the effort. This is due to the fact that the correspondence between the *level* of a language and the productivity is not linear, being the coding effort approximately only a 30 percent of the total effort for most of sizeable software projects [13]. See C. Jones [13] for further information about that particular topic.

TABLE II

CORRELATION BETWEEN *Language Level* AND *Productivity*

Language Level	Productivity Average per Staff Month
1 – 3	5 to 10 Function Points
4 – 8	10 to 20 Function Points
9 – 15	16 to 23 Function Points
16 – 23	15 to 30 Function Points
24 – 55	30 to 50 Function Points
Above 55	40 to 100 Function Points

Taken from [13].

III. DEVELOPMENT OF A DSL

A. Motivations to develop a DSL

Before anything else, a decision to develop a DSL has to be made. A good alternative for the development of DSL are libraries and frameworks, since libraries can effectively capture many aspects of the development of applications.

It is not reasonable to start developing a DSL whenever there is little knowledge of the domain due to its novelty. When this is the case, the conventional software engineering process should be applied in the first place, as a means to establish the fundamental concepts of the domain and to develop a code base with the use of libraries. Once there is enough information of the field, and the traditional programming techniques cannot supply the right abstractions, the development of a DSL can be considered [1].

B. Development phases

The development phases of DSL are divided into: (1) *decision*, (2) *analysis*, (3) *design*, (4) *implementation* (covered in Section IV), and finally its deployment and usage [3], [7]. Nevertheless, in practice, the development of DSL is not as straightforward. The preliminary analysis can influence the decision process, and likewise, the preliminary analysis may have to provide answers to unexpected questions that might arise during the design phase, which, in turn, is frequently influenced by implementation considerations. So, all in all, the development process is usually not a simple sequential process. [3].

1) Decision: It is usually difficult to decide to go for a new DSL. Its initial cost is high and thus, it has to be seen as an investment that can be compensated later on by a cheaper software development and / or maintenance. There is no doubt that to adpot an already existing DSL is considerably more economical and does not require as much knowledge as creating a new one. Nevertheless, information about DSLs is often disperse and/or disguised, and therefore, it can be hard to learn about accessible DSLs. Moreover, the use of poorly documented DLS can be quite reckless [3].

2) Analysis: The analysis phase of DSL development consists in identifying the problem domain and in gathering knowledge about it [3], [7].

For instance, as E. Visser [1] pointed out, a first analysis of the domain of developing web applications would show that it involves a data model, an object-relational mapping, a user interface, data input and output methods, data validation, page flow, and access control. Furthermore, it could also involve file upload, sending and receiving email, versioning of data, internationalization, and higher-level aspects like work-flow. A more exhaustive analysis would explore in more detail each of the concepts of a domain and set requirements and terminology. Later, these would be the input for the design of a DSL [1].

The input of the analysis phase, consists of multiple sources of explicit or implicit domain knowledge. For example, technical documents, knowledge from domain experts, existing GPL code, and surveys from customers.

There is a wide variety of output from domain analysis, but it primarily consists of domain-specific terminology and semantics in a higher or lower form of abstraction [3]. The

output of formal domain analysis is called *domain model*. It consists of [3]:

- A domain definition which defines the extent of the domain.
- The terminology of the domain (vocabulary, ontology).
- The descriptions of domain concepts.
- Feature models that describe the variabilities and commonalities of domain concepts and their interconnections.

Variabilities disclose what exact information is needed to specify an instance of a system. This information has to be directly specified in, or be resultant from a DSL program. Terminology and concepts help to guide the actual DSL constructs' development, matching the variabilities. Commonalities are employed to define the execution model and primitives of the language. The execution model is defined by a series of common operations [3].

ODM (Organization domain modelling) [14], [15] or FODA (Feature-Oriented Domain Analysis) [16] are examples of these formal domain analysis methods. Both methods, among other formal methodologies, can be applied in this phase [7].

3) Design: Basing a DSL on an already existing language is the easier way to design DSLs. There are different possible advantages of doing so, such as an easier implementation. One might think that using a pre-existing language will also entail familiarity for users, but this is only the case when the user is acquainted with the pre-existing language, which may not always be the case [3].

On the opposite side, there are DSLs that are designed with no connection to any pre-existing language. Practically, it can be really difficult to develop this sort of DSLs. It is important that the designer of a DSL keeps in mind the goal of DSLs, and the fact that there is no need for the users to be programmers. However, there are some design criteria for GPL that are also valid for DSLs, like for example, readability and simplicity.

Nevertheless, given that DSLs are using already established notations for its particular domain, the designer should avoid improving or even modifying too much the DSL's notation, for example by over-simplifying it. Otherwise, the notation would vary more than necessary and non-programmers would need more effort to learn the resulting DSL [3].

IV. IMPLEMENTATION APPROACHES

There are several approaches used to create different DSLs. Most of the existing articles about DSLs are about a particular DSL that has been developed by the author or authors of the paper, or about the analysis of a specific one. This makes the comparison between the different approaches a tough work. Fortunately, M. Mernik et al. [3] and T. Kosar et al. [5] have studied this issue, shedding light onto the matter.

T. Kosar et al. [5] used the previous guidelines about DSLs implementation given by M. Mernik et al. [3] to compare the different implementations by writing programs using the same DSL with the selected approaches and then comparing the obtained results. Next, some of these implementation patters will be presented:

A. Preprocessing

In this approach, the constructs of the DSL are directly translated into a base language. That mainly causes that the static analysis from the base language processor limits the DSL static analysis. This is the main disadvantage. It happens because the absence of semantic analysis means that optimizations at the domain level can not be done since there is no static checking.

However, that could simultaneously be the main advantage of this implementation approach: since almost all semantic analysis is delayed and conducted by the base language processor, the DSL is very easy to implement.

Four principal sub-patterns could be differenced [5].

- *Macro processing*. The meaning of new constructs in this implementation approach, is defined in the matter of other constructs in the base language.
- *Source-to-source transformation*. It is typically defined through syntax-directed patterns. The DSL source code is translated into the chosen pre-existing language.
- *Pipeline*. Processors are constantly operating the sub languages of a DSL in order to translate them to the next stage input's language.
- *Lexical processing*. Just an uncomplicated lexical scanning is needed, without any sort of complex tree-based syntax analysis.

It is important not to forget another disadvantage, also derived from that translation: the error reporting messages will always be in terms of the base language concepts and not of the DSL [5].

B. Embedding (DSELs)

This implementation approach, which has been already mentioned before in this paper, is also known as DSEL [10], [9], as well as *domain-specific libraries* [7]. It could probably be considered as one of the most recommended approaches because of its high efficiency and fastness in its design when compared with other implementations [5], [10], [9], and may be the reason why it has a specific term for it. Based on the idea of reuse [5], [9], this approach consists on building a library of domain-specific operations on a host language and then, utilize its compiler or interpreter. This way, it becomes significantly easier and straightforward in terms of design since the developer does not need to build a complete programming language from scratch. Instead, the DSEL inherits the host's general language constructs and then adds the domain-specific primitives that are closer to the DSL user, even being possible for non-programmers specialized in the domain to understand much of the code [5], [10]. But as with the preceding approach, error reporting still being, for the same reasons, awkward. Moreover, some expressiveness restrictions in terms of domain-specific notation will be

present because of the limitations of the host language itself [5].

Another interesting characteristic of the DSEL is that developing different DSELs for different domains, but derived from the same host language and with a similar look-and-feel, can offer a common shared core language and tools, which can be a clear advantage for developing sizeable multi-domain applications with multiple DSELs [9].

C. Compiler/interpreter

In this approach, a DSL is implemented by using standard compiler/ interpreter techniques. Concerning the compiler, base language constructs and library calls are translated from the DSL constructs and a complete static analysis on the DSL program/specification is executed. As for the interpreter, a Fetch-decode-execute cycle is used so as to recognize and interpret the DSL constructs. It is important to take into account that the cost of building from the ground up a complier/interpreter is a considerable disadvantage. Nevertheless, a closer syntax to the notation used by domain experts and a proper error reporting are some advantages that should not be forgotten [5].

D. Compiler generator

The compiler generator approach is resemblant to the Compiler/ interpreter one besides the use of compiler writing tools, or, in other words, of language development systems for the implementation of some of the Compiler/interpreter phases. This allows to reduce the implementation effort and, as a result, to minimize the last approach's disadvantage [5].

E. Extensible compiler/interpreter

The extensible compiler/interpreter approach uses either or both domain-specific optimization rules and domain-specific code generation, to extend the GPL compiler/ interpreter. When applicable, facilities such as reflection and introspection can be fairly useful for this approach. Another thing to consider is the fact that the implementation effort is considerably reduced in comparison to the two previous approaches, as a consequence of reusing a complete compiler infrastructure which already exists. Nonetheless, it should be taken into account that it is arduous to extend a compiler. Therefore, to prevent interferences between domain-specific notation and an existing one, it is important to be extremely cautious [5].

F. Commercial Off-TheShelf (COTS)

The Commercial Off-The-Shelf (COTS) approach creates conceivable solutions to specific domain problems by using existing tools and/ or notations to a specific domain. A clear example can be found on the XML-based DSLs. Even though writing and reading XML is usually complex for humans, it provides reassuring solutions in processing and querying documents [5].

V. CONCRETE EXAMPLES OF DSLs FOR GAME DEVELOPMENT

As explained throughout the paper, the use of DSLs can be certainly beneficial when developing software in specific domains [1], [3], [9]. This also includes the development of video games, specially to transition from the game design to the implementation phase [17].

In this section, three different DSLs related to the game development domain are presented superficially. The goal is to see in real examples how there can be different scopes even for DSLs that belong to the same domain. Each one of these, has different objectives and were conceived to solve distinct problems. The section focuses on showing and explaining some examples in order to highlight some interesting features of each DSL.

A. Hagl (Haskell game language)

Hagl is an embedded DSL (see Section IV-B) into Haskell, created by E. Walkingshaw and M. Erwig [18], and it is based on representing or modelling a game using tree structures.

Its purpose does not consist of providing a programming language which can be used to develop an entire video game, but rather a DSEL for experimental game theory, allowing to define games and strategies and evaluate them. In order to do that, these games and strategies can be executed multiple times to observe the results [18], [19].

For instance, E. Walkingshaw and M. Erwig [18] modelled different games in its paper, such as the iterative prisoner's dilemma.

Fig. 2. Representation of the prisoner's dilemma, taken from [18].

		Player 2	
		C	**D**
Player 1	**C**	2, 2	0, 3
	D	3, 0	1, 1

In this prisoner's dilemma game, there are two players and each one must choose to either "cooperate" (**C**) or "defect" (**D**). Choosing "defect" will ensure a payoff for that player and the highest payoff if the other player chose to "cooperate". Nevertheless, both players will benefit moderately if both them decide to "cooperate". This is often represented by a payoff matrix, like in Figure 2 [18].

Following there are some selected Hagl code extracts corresponding to that game experiment, taken from [18].

Extract from the Prisoner's Dilemma definition

```
1  data Dilemma = Cooperate | Defect
2
3  pd :: Game Dilemma
4  pd = matrix [Cooperate, Defect] [[2,2],[0,3],
5                                   [3,0],[1,1]]
```

The matrix is actually a syntactic sugar to produce a decision tree (which can be found in [18, p. 4]).

Definition of two simple constant players

```
6  mum = "Mum" ::: pure Cooperate
7  fink = "Fink" ::: pure Defect
```

There are different functions in Hagl to print information about the current state of a game execution. To print the transcripts of all completed games, one might use `printTranscript`. To print the current score of each player, one might use `printScore` [18].

In the following code extract, the result of one game is printed, then the game runs 99 more times, and the total score, including the first game, is finally printed.

Running and printing the experiment

```
> runGame pd [mum, fink](once >> printTranscript
                         >> times 99 >> printScore)
Game 1:
        Mum's move: Cooperate
        Fink's move: Defect
        Payoff: [0.0,3.0]
Score:
        Mum: 0.0
        Fink: 300.0
```

The authors also modelled other games, such as the rock-paper-scissors game and even a Cold War simulator.

B. Adventure DSL

R. Walter and M. Masuch [17] envisioned Adventure DSL. It would be an expressive DSL specifically tailored towards the development of 2D Point-and-Click adventure games. The authors conceived the DSL to be implemented with preprocessing techniques (see Section IV-A) and to generate code in C#.

The language differentiates between two categories of entities, (1) *basic* and (2) *gameplay entities*. The *basic entities* are used to describe the visual structure of a 2D Point-and-Click adventure game [17]. Following, there is a non-exhaustive list of the most important ones:

- **Room**: an area the player's character can enter and leave through doors. It is represented with a background image and can optionally include background music and different objects and/or other characters.
- **Door**: a gate the player can use to change rooms. A door always has an assigned position, it always belongs to two rooms and has at least two sprites to define its state as "open" or "closed".
- **Character**: either playable or non-playable, characters are always designated to precisely one room and one position at a time. They have at least one sprite.
- **Object**: describes usable and non-usable objects. They have at least one sprite and are designated to a particular room and position.
- **Inventory**: the virtual place where the player interacts with the collected objects.

The *gameplay entities* consist of abstractions or intangible entities [17]:

- **Task**: the different challenges encountered by the player in the game.
- **Goal**: sets a condition the player must reach to progress.
- **Constraint**: describes causal relationships between tasks by checking if goals have been achieved.
- **Action**: an action that the game has to execute. It is triggered by either the player or the game itself.
- **Dialogue**: models and represents the verbal interactions between non-player and player characters.

The high-level abstractions that this DSL provides, allows game designers to not have to care about internal structures and focus on the design of the game. This can be seen with the following code extracts taken from [17], where three different `room` entities are created.

Definition of three room entities

```
The cafeteria is a room.
Its background is "Sprites/cafeteria.png".
The hallway is a room.
Its background is "Sprites/hallway.png".
The atrium is a room.
Its background is "Sprites/atrium.png".
```

If the preceding code was processed, it would be transformed into the following C# code extract. One can see that for each defined room, an instance of `Room` is added to the `roomList`-object in C# with its respective sprite.

Generated C# code extract

```
// Add Rooms to roomList
roomList.Add(new Room("Sprites/cafeteria.png"));
roomList.Add(new Room ("Sprites/hallway.png"));
roomList.Add(new Room("Sprites/atrium.png"));
```

In Figure 3 there is a code example written in Adventure DSL, where one can see the high expressivity and similarity to natural language. It is also shown how decision dialogues work.

Fig. 3. Example written in Adventure DSL, taken from [17].

example.pca

```
The cafeteria is a room. Its background is "Sprites/cafeteria.png".
Its background music is "Sounds/cafeteria.wav".

The fork is an object. Its image is "Sprites/fork.png".
Its position is (400, 300). Its display name is "A dirty fork".
It is in the cafeteria. It cannot be picked up.

Einstein is a character. His image is "Sprites/einstein.png".
His position is (200, 300). His display name is "Prof. Einstein".
He is in the cafeteria. His dialogue is FirstDialogueWithEinstein.

EinsteinIsHungry is a goal.
Forkgoal is a goal.

FirstDialogueWithEinstein is a dialogue:
1 "Hello, how are you?" - "I'm hungry!" - EinsteinIsHungry is achieved.
1.1 "Me too." - "Let's find something to eat."
1.2 "Sorry, I have nothing to eat." - "Bye bye."
2 "Sorry, I have to go."
The dialogue ends here.

Pick up fork ( EinsteinIsHungry is achieved. ):
Say "I won't take that fork, it's dirty.".
Einstein says "Hang on, I'll clean it for you.".
```

Every line of the dialogue begins with a numeric identifier, and it is followed by an option–answer–information tuple. The option the player chooses in the game triggers the corresponding NPC answer. Then, if available, the information is received. Afterwards, corresponding lower-level dialogue lines are triggered. This allows the game developer to define the dialogues as decision trees.

The authors explained that this tree structure for writing dialogues was not conceived from the start. Nonetheless, while the DSL was being designed, they thought that this system "noticeably enriches the gameplay options", so they

finally added it. This shows again that the development of a DSL is not a strict sequential process, and that iterative methodologies can be adopted in order to create better or richer languages [17].

C. Casanova 2

Casanova 2 is a declarative programming language for games derived from F#, originally developed by M. Abbadi [20]. Casanova 2 aims to help game developers to reach their objectives by considerably reducing the development efforts. This is particularly favourable for relatively small game development teams. [20], [21].

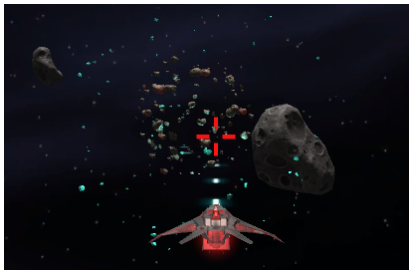
In contrast to Adventure DSL, the design of the syntax and semantics of Casanova 2 allows supporting the definition of games regardless of their genre or structure. A series of distinct games were developed in Casanova 2 to evaluate the generality of the language:

- *Galaxy wars*, a real time strategy game.
- *Zombie shooter*, a virtual-reality multiplayer shooter, where a group of players tries to escape by car from a city infested by zombies.
- *Dyslexia*, a game conceived to detect dyslexia in children (see Figure 4).
- *3D asteroid shooter*, a 3D asteroids shooter game (see Figure 5).
- *Contact*, a multiplayer game to study the evolution of language.

Fig. 4. Screenshot of *Dyslexia*, taken from [20].

Anonymous bachelor students of computer sciences specifically learned Casanova 2 to be able to develop most part of these games. This shows that Casanova 2 can be successfully used by junior developers.

Fig. 5. Screenshot of *3D asteroid shooter*, taken from [20].



Casanova 2 uses a seven-layer source-to-source (see Section IV-A) code generator compiler to generate new code in a target language, in C# by default. Thus, every Casanova 2 program is translated to an equivalent one in C#. Despite that, the layered Casanova 2 compiler was conceived with the idea to allow developers to extend it when needed. For instance, M. Abbadi [20] suggested that it could support other languages, like JavaScript, to allow Casanova 2 to directly run in a browser. A structure diagram of the compiler can be found in [20, p. 54].

This flexible compiler architecture also allows interoperability. For example, Casanova 2 includes full Unity integration, but other engines and libraries can also be used [21]. It needs to support interoperability because Casanova 2 does not address all the aspects of game development, like rendering and content management, among others. The goal is to be able to delegate these non-included tasks to already existing tools. Since Unity3D has its own game engine, the Casanova 2 code is processed by the Casanova 2 compiler, generating the C# code, which is then run by the Unity3D engine [20].

M. Abbadi [20] evaluated quantitatively the compactness and readability, and thus, the development performance of Casanova 2 compared to other four programming languages. In order to do it, the author compared, for the same program, the size of the code required for each language in terms of number of syntagms (defined by the author as "the number of distinct keywords and operators" [20]), lines of code and total number of words.

As one can see in Table III, in all cases Casanova 2 is equally or more efficient. This is specially pronounced for the total number of words.

TABLE III
COMPARISON OF CODE SIZES FOR THE SAME PROGRAM BETWEEN *Casanova 2* AND THE OTHER TESTED LANGUAGES

Language	Syntagms	Lines of code	Total words
Casanova 2	47	31	104
C#	61	69	269
JavaScript	52	41	257
Lua	47	45	249
Python	50	34	214

Taken from [20].

VI. CONCLUSIONS

This paper has provided an overview on the development of DSLs. As has been seen, before taking a decision to start developing a DSL, wide knowledge about the application domain is needed. Having a considerable amount of code base in that domain with insufficient specific abstraction is also a good indicator that a DSL would be useful. On the contrary, as E. Visser [1] suggested, developing DSLs for fairly new domains where there is not enough domain application knowledge or expertise might not be a good nor a cost-effective solution.

From the perspective of different authors, such as T. Kosar [5] or P. Hudak [9], the embedding implementation approach seems to be the most practical one among all the others. Even if the final result can be less expressive when compared to other approaches, the cheap development of DSELs makes it more cost-effective. This is specially true when the language does not seek to be used by a large group of people, or for example, when a good error reporting system is not required. However, in other cases where the end-user effort is more important than the effort required by a programmer to implement a DSL, it will be better to implement a full DSL compiler using compiler generators.

It has been seen that the use of DSLs brings more benefits than inconveniences. It is true that the beginning of a project using a DSL can be slower, because a training period is needed. However, it is proven that later on, the development speed and quality increase, due to its easier maintainability among other reasons [11], [9], [17]. Not using repeated manual implementation, and using code generators instead, guarantees that concepts described with a DSL will always be equally implemented. This, in turns, guarantees a stable quality-level of the code. In the case of the game developing domain, an expressive DSL lets developers write code which is closer to the language used by the designers. By doing so, even designers without programming knowledge could program. This example is generalizable to any specific application domain, and that is also what makes DSLs specially useful.

One can also conclude, specially after looking at the game development DSL examples, that the specificity of a DSL can be relativized. For instance, from the perspective of Adventure DSL, Casanova 2 is a really general game developing language, while Hagl is too abstract. Nevertheless, they cannot actually be compared since their purposes are totally different, even if apparently they belong to the same domain.

In any case, as authors of Adventure DSL [17], pointed out, creating an overly-specific DSL may not be a good practice, and neither would it be to create an overly-general one.

REFERENCES

[1] E. Visser, "WebDSL: A Case Study in Domain-Specific Language Engineering," in *Generative and Transformational Techniques in Software Engineering II: International Summer School, GTTSE 2007, Braga, Portugal, July 2-7, 2007. Revised Papers*, R. Lämmel, J. Visser ans J. Saraiva, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 291-373.

[2] J. W. Backus et al., "Report on the algorithmic language ALGOL 60," *Communications of the ACM*, vol. 3, no. 5, pp. 299–314, May 1960.

[3] M. Mernik, J. Heering, and A. M. Sloane, "When and how to develop domain-specific languages," *ACM Computing Surveys*, vol. 37, no. 4, pp. 316–344, Dec. 2005.

[4] A. Van Deursen and P. Klint, "Domain-Specific Language Design Requires Feature Descriptions," *Journal of Computing and Information Technology*, vol. 10, no. 1, pp. 1-17, 2002.

[5] T. Kosar, P. E. Martínez López, P. A. Barrientos and M. Mernik, "A preliminary study on various implementation approaches of domain-specific language," *Information and Software Technology*, vol 50, no. 5, pp. 390-405, April 2008.

[6] D. T. Ross, "Origins of the APT language for automatically programmed tools", in *History of programming languages*, Eds. New York, NY, USA: Association for Computing Machinery, June 1978, pp. 279–338.

[7] A. van Deursen, P. Klint, and J. Visser, "Domain-specific languages: An annotated bibliography," *ACM SIGPLAN Notices*, vol. 35, no. 6, pp. 26–36, June 2000.

[8] D. S. Wile, "Supporting the DSL Spectrum," *Journal of Computing and Information Technology*, vol. 9, no. 4, p. 263, 2001.

[9] P. Hudak, "Modular domain specific languages and tools," *Proceedings. Fifth International Conference on Software Reuse (Cat. No.98TB100203)*, 1998, pp. 134-142.

[10] P. Hudak, "Building domain-specific embedded languages," *ACM Computing Surveys*, vol. 28, no. 4es, pp. 196-es, Dec. 1996.

[11] J. Gray and G. Karsai, "An examination of DSLs for concisely representing model traversals and transformations," *36th Annual Hawaii International Conference on System Sciences, 2003. Proceedings of the*, 2003, pp. 10 pp.-.

[12] D. Bonachea, K. Fisher, A. Rogers, and F. Smith. 2000. "Hancock: a language for processing very large-scale data," in *Proceedings of the 2nd conference on Domain-specific languages (DSL '99)*, Eds. New York, NY, USA: Association for Computing Machinery, 2000, pp. 163–176, ISBN: 978-1-58113-255-7.

[13] C. Jones, "Programming Languages Table, Release 8.2," *Software Productivity Research, Inc*, Mar. 1996, `cs.bsu.edu`. [Online].

[14] M. Simos. "Organization domain modeling (ODM): Formalizing the core domain modeling life cycle", in *Proceedings of the Symposium on Software Reusability SSR'95*. M. Samadzeh and M. Zand, Eds. ACM Software Engineering Notes. August 1995. pp 196-205.

[15] M. Simos, D. Creps, C. Klinger, L. Levine, and D. Allemang. "Organization Domain Modeling (ODM) Guidebook, Version 1.0". *Technical Report STARS-VC-A023/011/00*, Unisys Corporation, March 1995.

[16] K.C. Kang, S. Cohen, J.A. Hess, W.E. Novak, and A.S. Peterson. "Feature-Oriented Domain Analysis (FODA) Feasibility Study", *Technical Report CMU/SEI-90-TR-21*. Carnegie Mellon University Pittsburgh, Pennsylvania, USA, November 1990.

[17] R. Walter and M. Masuch. "How to integrate domain-specific languages into the game development process", in *Proceedings of the 8th International Conference on Advances in Computer Entertainment Technology (ACE '11)*. Association for Computing Machinery, New York, NY, USA, Article 42, 1–8. 2011.

[18] E. Walkingshaw and M. Erwig. "A domain-specific language for experimental game theory". *J. Funct. Program.*pp 645-661. November 2009.

[19] E. Walkingshaw. "Github Hagl page", First commit, April 2009. Available: `https://github.com/walkie/Hagl`

[20] M. Abbadi. "Casanova 2: a domain specific language for general game development", Department of Environmental Sciences, Informatics and Statistics, Ca' Foscari University of Venice, Veneto, Italy, September 2017.

[21] M. Abbadi et al. "Github Casanova 2 page". First commit, August 2015. Available: `https://github.com/vs-team/casanova-mk2`