Conceptual Object Exchanges among Software Games by Non-programmers

Iaakov Exman and Guy Shimoni
Software Engineering Dept., The Jerusalem College of Engineering – JCE - Azrieli, POB 3566, Jerusalem, Israel

Keywords: Conceptual Objects, Software UFOs, Unidentified Flying Objects, Software Games, On-the-Fly, Conceptual Exchanges, Embedding, Non-programmers.

Abstract: A non-programmer, who is an expert game player, could easily grasp the idea of improving his *jumping* avatar by importing a *flying* property found in another game. But in order to actually exchange conceptual objects between games one would need a suitably transparent software infra-structure. We propose the usage of UFOs – Unidentified Flying Objects – that can be smoothly transmitted and embedded into the target game, and except for their conceptual label, are so-to-speak unidentified by the non-programmer. This approach has been designed and implemented into a distributed system of multiple software games in different mobile computers. Beyond games, this system serves as a feasibility proof for on-the-fly exchange of Conceptual Objects among any kinds of software applications.

1 INTRODUCTION

Software reuse ideas led to the proposal of software design patterns, of which GoF patterns (Gamma, 1995) are a widely known example. The latter kind of patterns is indeed composed of small groups of classes, but their reuse is not supposed to be done in terms of whole untouchable classes. Software engineers usually modify internal attributes and functions of some of the pattern classes, essentially disrupting class encapsulation.

On the one hand, there is widespread manipulation of software in daily life by human end-users which are non-programmers. In the specific context of software games, people may be expert players – perfectly understanding the games' conceptual world – without being aware of the intricacies of the underlying software. Thus, for such users, keeping conceptual neatness is very desirable, in order to preserve world view consistency.

On the other hand, given the trend of increasing level of abstraction (Exman, 2015) in software design and implementation, it is suitable from software development and maintenance considerations, to work with conceptual classes – which besides faithfully representing natural language concepts of the end-users' world, do not disrupt encapsulation.

This work demonstrates, in the context of software games, a system designed, implemented and run as a feasibility proof of conceptual objects exchange between different programs located in diverse mobile computers. The system is usable by non-programmers, since except for the conceptual labels natural for the human end-user, the conceptual exchange software infra-structure is transparent to the end-user.

In this paper we thoroughly examine the meaning of "conceptual objects", describe the software system architecture, and employ case studies to demonstrate the approach.

1.1 Related Work

Since we deal with conceptual objects exchanged among games, a primary question to be asked is about the scope of these concepts. A generic answer may be provided by a game ontology. Calleja (Calleja, 2007) in his course on Computer Game Theory provides a good introduction to Game Ontology as a development of a critical vocabulary for computer games. Zagal and Bruckman (2008) describe a game ontology project in the context of learning; a more general description of the same project is found in Zagal et al., (2005). Chan and Yuen (2008) refer to a Digital Game Ontology for developing web game applications. Aarseth

Exman, I. and Shimoni, G..
Conceptual Object Exchanges among Software Games by Non-programmers.
In *Proceedings of the 6th International Workshop on Software Knowledge (SKY 2015)*, pages 18-25
ISBN: 978-989-758-162-5

(Aarseth, 2010) discusses the purpose of game ontologies stressing the gap between lay language and academic language in this domain.

A less formal approach with the same purpose – to define a vocabulary of games – is taken by Costikyan (Costikyan, 2002). The issue at stake is the relevant vocabulary to design games.

Another issue is the characterization of players as non-programmers. Non-programmers should face a simple, almost self-explanatory, user interface to use and compose games. A particular example is the case of youth which learn "programming" visually by manipulating blocks, with "Scratch", as in Maloney et al., (2008). Poremba's Master's thesis deals with the player as author of digital games (Poremba, 2003).

Modification of existing games is coined "modding". Specific references on this sub-topic include Sotamaa (Sotamaa, 2005), and Mactavish (Mactavish, 2005).

There have been systematic efforts for generic end-user (non-programmers) development or tailoring of computer programs. Lieberman et al., (2006) characterize development by non-professional end-users and refer to it as an emerging paradigm. A more recent and more generic review is found in (Gulwani, 2010), which analyses different dimensions of program synthesis.

1.2 Paper Organization

In the remaining of the paper we define Conceptual Objects Exchange (section 2), shortly introduce the Software Games context (section 3), overview the UFOs software architecture (section 4), discuss case studies as a demonstration of the approach (section 5), deal with implementation issues (section 6) and conclude with a discussion (section 7).

2 CONCEPTUAL OBJECTS EXCHANGE

In this section we try to specify our understanding of conceptual objects and their exchange by non-programmers, e.g. to modify software games.

2.1 Non-programmers

Before we specify the meaning of conceptual objects we need to characterize non-programmers.

A non-programmer (Exman, 2013) is a person – a layman – which is neither a professional software engineer nor a programmer, but it is here assumed to have two definite characteristics:

- *Software Usage Proficiency* – the person skilfully manipulates a common activity involving software – e.g. playing a software game;
- *Conceptual Comprehension* – the person perfectly grasps the notions of the software activity, i.e. has a good understanding of the activity concepts.

2.2 Conceptual Objects

Nowadays it is common to design software systems using UML – Unified Modelling Language – see e.g. (Booch, 2005). Within UML there are a set of diagrams to describe the structure and behavior of the system. A typical description of the structure of a software system is provided by a class diagram.

Classes should be designed to represent the main concepts of a system. In our view of the highest abstraction levels of the software system, a class diagram is equivalent to an *application* ontology (Exman, 2014b) which represents the system concepts. In other words, if we abstract ourselves of the internal details of a class, each class is equivalent to a single concept in the particular application ontology linked to the system. In this general sense, every class is a concept, and its instances – the objects – are necessarily conceptual objects.

But in this work we use "conceptual objects" to denote more specific entities. We shall refer to the objects related to the activity concepts grasped by non-programmers as "conceptual objects".

What are conceptual objects?

Conceptual objects technically are instances of software classes labelled by terms belonging to the vocabulary of non-programmers. As such their labels are well-understood by non-programmers. For instance, "jumping" is a well-understood concept within games.

What are not conceptual objects?

Labels of conceptual objects do not necessarily belong to ontologies of the respective domain. They could be naturally added to the respective ontologies.

Conceptual objects and their labels are usually not given a dictionary definition or ontological definition by the software classes. The non-programmer is aware of say the "jumping" concept through his previous knowledge of games or by receiving an informal explanation on how to play the game.

2.3 Conceptual Objects Exchange by Non-programmers

The goal of this work is to allow reasoned decision of software systems modifications by non-programmers at their will. This can be achieved, among other possible ways, by exchanging conceptual objects between existing systems. In this way one avoids the need to build from scratch an object. It already exists in another system.

In order to enable non-programmers to efficiently exchange conceptual objects between different programs, one needs to build a transparent software infra-structure, which externally is easy to manipulate. The non-programmer does not need to understand the internal mechanisms of the infra-structure.

The conceptual object exchange should be applicable to any software system whatsoever in any domain. For the purposes of proof of feasibility and practical demonstration, in this work we apply it to software games. But the idea is general.

2.4 Conceptual Framework for Objects Exchange by Non-programmers

There are requirements on a few conditions to allow the envisioned conceptual exchange by non-programmers:

- *Ontological Super-type* – in the relevant ontological hierarchy, the ontological super-type should be internally recognized by the software system; for instance, if one is dealing with “jumping”, “flying” or “running”, which are kinds of motion, either a “*motion*” super-type or an even higher “*player*” super-type should be present in the software game;
- *User-interface* – existence of some available GUI (Graphical User Interface – such as “buttons” or “menus”) to receive the activity commands and communicate them to the internal functions within the conceptual objects.

3 THE SOFTWARE GAMES CONTEXT

The software games context is used in this work just to convey a feasibility proof for the more general claim of “conceptual objects exchange” by non-programmers. This context is convenient, since it is easily understood due to its concrete visualization of the demonstration.

Within the software games context we focus for simplicity on 2-D Platform games. This should minimize development efforts on inessential issues, enabling one to concentrate on the essentials of conceptual objects exchange.

3.1 2-D Platform Games

2-D platform games (Reyno, 2008) are characterized by a guided avatar which moves and jumps into suspended platforms, while performing activities such as overcoming obstacles and collecting prizes. The player should guide his avatar through the several stages of the game.

In this work we consider two kinds of concepts involved in these games:

a. *Avatar Appearance* – this is seen by the graphic representation of the avatar, e.g. a ball (smiley) or a humanoid (astronaut);
b. *Kind of Motion* – e.g. smooth lateral motion, jumping or flying.

3.2 Fast Programming by Non-programmers

The idea of fast programming by non-programmers is the existence of an infra-structure to exchange whole concepts from a software system in a given computer to another software system in a different computer. Thus software systems are modified by moving around conceptual objects. It is fast as it is simple to do it. It is a sort of composition of up-to-now unknown objects into a different existing system.

4 SOFTWARE ARCHITECTURE

In this section we shortly describe the system software architecture principles and the essential UFOs exchangeable classes within the software system architecture.

4.1 Software Architecture Principles

There were two main principles guiding the software architecture:

1. *Game Engine Separation* – in order to avoid detailed development of games, we decided to base the development upon a ready-made *game engine* providing the basic library of game functions;
2. *UFO Uniformity* – all the exchangeable classes

should inherit from the same basic class, and be composable into a "super-type" referred above.

4.2 UFOs Software Architecture

The system software architecture is schematically seen in Fig. 1. It has four modules:

1. *Game Engine Class* – from which all the active classes of the games inherit (named *Behavior Class* – see the implementation section 6 in this paper for details) ;
2. *Player Class* – this is the referred super-type which is composed of the exchangeable classes;

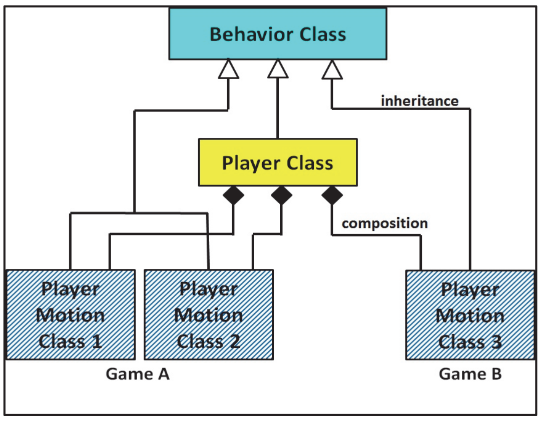

Figure 1: SYSTEM SOFTWARE ARCHITECTURE – It has four main class types: 1. *Game Engine Class* (in light blue background, named Behavior Class), from which all active classes inherit; 2. *Player Class* (in yellow background) is the "super-type" for exchangeable uniformity among different games; 3. *Player Motion classes* (the exchangeable UFOs: dark blue hatched background) from which the Player Class is composed; 4. *Game Classes* – are additional non-exchangeable classes (not shown in this figure). In this scheme Game A has two kinds of PlayerMotion (Class 1 and Class 2), while Game B has just one PlayerMotion (Class 3).

3. *PlayerMotion* – the exchangeable motion classes (the UFOs) existing in different games;
4. *Game Classes* – additional classes that are not exchangeable; for instance the graphical background of the games (not shown in the scheme of Fig. 1)

One can see that this architecture obeys the above guiding principles (in sub-section 4.1): it separates the Behavior Class obtained from the Game Engine from other classes; it displays a super-type – the Player Class – composed of exchangeable classes.

The diagram in Fig. 1 clarifies the infra-structure for conceptual object exchange. For instance, one could send *PlayerMotion* Class 2 from Game A to Game B, since both games have the same "super-type" *PlayerClass*. This is analogous to extensible design patterns such as Strategy (Gamma, 1995).

5 CASE STUDIES

The case studies in this section – two games with different avatars and distinct player motions developed with the special purpose to exchange conceptual objects – illustrate what happens when appearances and motion objects are exchanged.

The purpose of these case studies is to be a feasibility proof of conceptual object transfer. They are not intended to be either an extensive exploration of the of the games' space, or a detailed examination of performance and network bandwidth issues.

A series of experiments was performed: exchanging avatars with their original properties; exchanging avatars keeping the properties of the current game; adding motions to the original avatar. Next we describe the simplest experiment.

5.1 The Jumping Smiley Game as a Source

The Jumping Smiley game has a Smiley as an avatar. The Smiley has two motions:

a. Left-Right motion – on top of a platform;
b. Jumping – continuous jumping up and down on top of a platform;

The Left-Right motions are performed by means of keyboard arrows. The Jumping motion is continuous from the beginning of the game. Its GUI is seen in Fig. 2. This game is the source of the conceptual object to be sent to the target game.

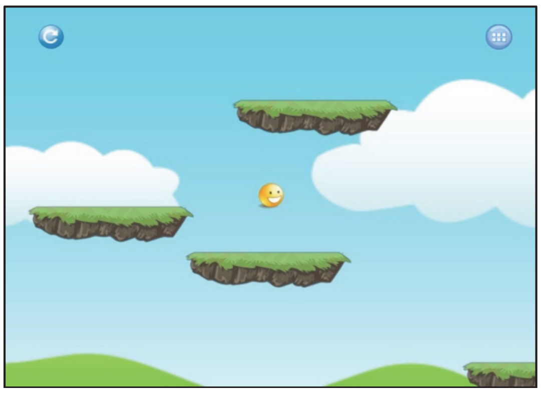

Figure 2: JUMPING SMILEY GAME GUI – It displays the Smiley above a platform and below another one, a series of platforms, a menu button on top-right and a restart button on top-left.

5.2 The Space Adventure Game as a Target

The Space Adventure game has an Astronaut as an avatar. The astronaut has two kinds of motion:

a. Left-Right motion – on top of a platform;
b. Flight – to achieve another platform from the present one.

The motions are performed by means of keyboard arrows. Its GUI is seen in Fig. 3. This game in a second computer is a target of the conceptual object to be sent from the source game.

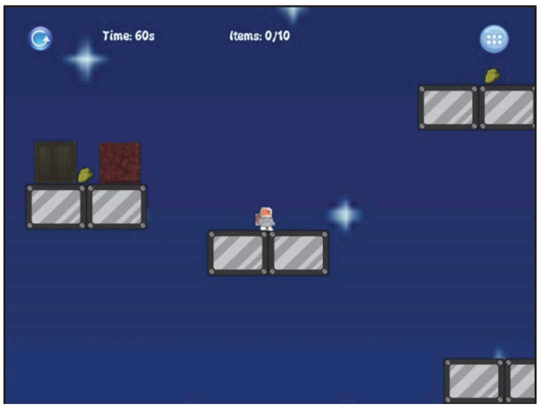

Figure 3: SPACE ADVENTURE GAME GUI – It displays the astronaut above the middle platform, a left platform with two square obstacles and a yellow “prize” in between to be collected, an upper-right platform with another prize on top of it, a menu button on top-right and a restart button on top-left.

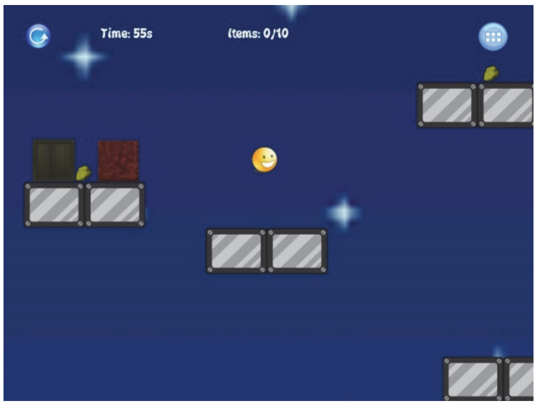

Figure 4: SPACE ADVENTURE GAME GUI WITH JUMPING SMILEY – It displays the Smiley, instead of the astronaut jumping above the middle platform, a left platform with two square obstacles and a yellow “prize” to be collected, a right platform with another prize on top of it, a menu button on top-right and a restart button on top-left.

After the Smiley is sent from the source game to the target game, the GUI is as seen in Fig. 4.

The outcomes of the exchange of the astronaut by the Smiley are:

a. The background has not changed: it is still the outer Space;
b. The avatar appearance has changed;
c. The motions have changed: the avatar now is able to perform Left-Right motions and jumping motions.

6 UFOs SOFTWARE IMPLEMENTATION

In this section we give details of the UFOs software system implementation: programming using a game engine, the tasks performed by the system modules and characteristics of the main GUI.

6.1 Programming of 2-D Platform Games

In order to avoid dealing with detailed problems of Game development – which in this work are only used for a feasibility proof – we used a ready-made game engine and its libraries.

The game engine is Unity (Goldstone, 2011), which has been applied to various devices in several projects. Unity supports the programming languages C# and JavaScript. The games were written in C#.

6.2 Implementation of the UFOs Exchange Software System

The whole UFOs Exchange Software system involves a series of modules performing the tasks listed in the next text list:

<u>TASKS in MODULES</u>

a. *Transparent Connection* – Connect other game by TCP/IP wireless network;
b. *Choice of Classes to Transmit* – select objects (avatar appearance as a jpg file and motion types as C# classes) in the source game;
c. *Classes Packing* – serialize (Carpenter, 1999) and pack the chosen classes into an XML file to be transmitted through the network;
d. *Transmission* – transmit the XML file;
e. *Classes Unpacking* – unpack the transmitted classes using an XML parser;

f. *Chose Classes to Apply* – in order to apply classes in the new game, they must be compiled at run time;
g. *Compile the Chosen Classes* – using the Mono compiler (Mono, 2015) from the Xamarin company; the compiler is installed in a Windows operating system;
h. *Add the Compiled feaTures to Avatar* – in order to be usable;
i. *Final Choice of Game Features* – using a suitable menu (see sub-section 6.3);
j. *Play the New Choice*.

6.3 GUI for Exchangeable UFOs

We developed a series of friendly menus enabling the choices referred above. In Fig. 5 one sees a menu to choose the desired features.

The possible choices are:

a. *Original Avatar* – with its local properties in the target game;
b. *Newly Added Avatar* – with the original properties in the source game;
c. *Newly Composed Avatar* – with any combination of local and original properties.

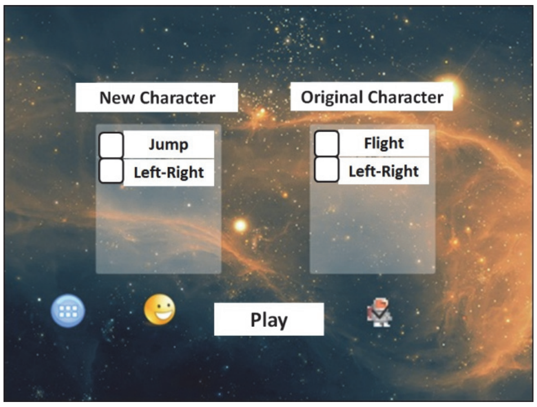

Figure 5: MENU IN SPACE ADVENTURE GAME WITH ADDED JUMPING SMILEY – A player may choose to play with: a- the original astronaut (the current avatar, marked by a black V sign); b- the newly added Smiley (which can be chosen by clicking it); c- any of the two avatars as a newly composed character with any desired combination of chosen motions.

7 DISCUSSION

This discussion is divided into a few different topics: fundamental issues, some practical considerations, applications and future work. It is concluded with a short statement of the main contribution.

7.1 Fundamental Issues

There have been previous efforts whose purpose is end-user development and program tailoring, as referred to in the Related Work sub-section 1.1 in this paper. The main differences between the current work and preceding efforts are our focus on two aspects:

a. *Conceptual Objects* – The internal modularity (Exman, 2014a) of our software system corresponds to the well grasped concepts of the non-programmer end-user. This characteristic keeps the software architectural neatness.
b. *Interplay between Existing Programs* – We are using two or more programs to move the conceptual objects among them. In our demonstration case studies, we move avatars and their properties from a source game to a target game.

The next issue to be considered is conceptual consistency. In the game's world, probably astronauts should fly and smileys should jump. But there are no hard rules, at least for these cases. One can easily think about inconsistencies in different worlds, in need of ways to constraint them.

A deeper issue is the eventual and possibly unpredictable relationships between different software systems with different goals and strategies. Keeping control of the different actors in such systems is a significant problem, beyond the scope of this work.

7.2 Practical Considerations

Among the practical considerations, we refer to motion combinations. Any combined motions, such as both jumping and flying, are added in a "vectorial" way. In our experiments we observed that one may add twice the same kind of motion. Note that both the astronaut and the Smiley may perform Left-Right motions (as seen in the menu in Fig. 5). If one adds both motions of the same kind, the actual outcome is performing twice the same motion, which means moving a double distance.

In a more sophisticated system, one should add a few specific functionalities:

- *Internal limits* – to regulate the number of times one performs the same motion consecutively;
- *Selected Property Transfer* – to enable communication efficiency, e.g. to avoid transferring motions that one does not intend to

apply; in other words separating properties from a "character" (or avatar).

7.3 Applications

One should consider real applications with critical demands. These may include robotic applications in medical domains or in dangerous environments.

A possible example, of such a complex situation in a medical domain, is the combination of motions in a robotic system for remote surgery by a non-programmer surgeon. In the middle of a surgical operation, the surgeon could decide as needed to send a different set of scalpel motions to a remote system.

Another example, in a dangerous environment, is to send robots to deal with radioactive materials in case of disaster such as an earthquake. The robot may send back to the human controllers photos of unexpected obstacles, implying additional motions to overcome the referred obstacles.

7.4 Future Work

A most interesting set of open problems is to extend the infra-structure to other worlds, besides the very limited 2-D platform games.

It is an open question, to be investigated in depth, whether the software architecture principles formulated in sub-section 4.1 are necessary and sufficient to apply the approach to more sophisticated worlds.

In particular we refer to the "super-type" condition: is it enough for any kind of application? Or on the contrary, it is too restrictive concerning the system flexibility: what are the requirements in order to allow combinations of two or more kinds of applications?

In straightforward practical terms, the feasibility proof of this work was done with computers running Windows operating systems. It would be interesting to extend these capabilities to diverse platforms, as a demonstration of the robustness of the core infra-structure.

7.5 Main Contribution

The main contribution of this work is the feasibility proof that a non-programmer may modify and indeed "*program*" anew existing software systems, based upon well-understood *conceptual objects*.

REFERENCES

Aarseth, E., "Define Real, 2010. Moron! Some Remarks on Game Ontologies", in Gunzel, S., Liebe, M. and Marsch, D., (eds.) DIGAREC Keynote-Lectures 2009/10, Potsdam University Press pp. 050-069. Web site: http://pub.ub.unipotsdam.de/volltexte/2011/4981/[urn:nbn:de:kobv:517-opus-49810].

Booch, G., Rumbaugh, J. and Jacobson, I., 2005. *The Unified Modeling Language User Guide*, Addison-Wesley, Cambridge, MA, USA, 2nd ed.

Calleja, G., 2007. "Game Ontology", Session 2 in the "Computer Game Theory". Web site: http://www.gordoncalleja.com.

Carpenter, B., Fox, G., Ko, S. H. and Lim, S., 1999. "Object Serialization for Marshalling Data in a Java Interface to MPI", Syracuse University, 18 pages. Web site:http://www.newsubmit.pdfnpac.org/users/fox/documents/JG99mpi/c427resubmit.pdf.

Chan, J. T. C. and Yuen, W. Y. F., 2008. "Digital Game Ontology: Semantic Web Approach on Enhancing Game Studies", in Proc. 9th Int. Conf. on Computer-Aided Industrial Design and Conceptual Design, IEEE.

Costikyan, G., 2002. "I Have no Words & I Must Design: Toward a Critical Vocabulary for Games", in Proc. of Computer Games and Digital Cultures Conf., F. Mayra (ed.), Tampere University Press, pp. 9-33.

Exman, I. and Alfia, A., 2013. "Knowledge-Driven Game Design by Non-Programmers", in Proc. SKY'2013 Int. Workshop on Software Knowledge, pp. 47-54, ScitePress, Portugal.

Exman, I. 2014a. "Linear Software Models: Standard Modularity Highlights Residual Coupling", Int. Journal of Software Engineering and Knowledge Engineering, Vol. 24, pp. 183-210, DOI: 10.1142/S0218194014500089.

Exman, I. and Iskusnov, D., 2014b. "Apogee: Application Ontology Generation from Domain Ontologies", in Proc. SKY'2014 Int. Workshop on Software Knowledge, pp. 31-42, ScitePress, Portugal.

Exman, I., Llorens, J., Fraga, A. and Alvarez-Rodriguez, J.M., 2015. "SKYWare: The unavoidable convergence of Software towards Runnable Knowledge", Accepted for publication.

Gamma, E., Helm, R., Johnson, R. and Vlissides, J., 1995. *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, Boston, MA, USA.

Goldstone, W., 2011. *Unity 3.x Game Development Essentials,* 2nd edition, Packt Publishing, Birmingham, UK.

Gulwani, S., 2010. "Dimensions in Program Synthesis", in Proc. of PPDP'10 12th Int. ACM SIGPLAN Symp. Principles and Practice of Declarative Programming, pp. 13-24. DOI: 10.1145/1836089.1836091.

Lieberman, H., Paterno, F., Klann, M. and Wulf, V., 2006. "End-User Development: An Emerging Paradigm", in Human-Computer Interaction Series, Vol. 9, pp. 1-8, Springer-Verlag, Berlin, Germany. DOI: 10.1007/1-4020-5386-X_1

Mactavish, A., 2003. "Game Mod(ifying) Theory: The Cultural Contradictions of Computer Game Modding", in Power Up: Computer Games, Ideology, and Play, Bristol, UK.

Maloney, J., Peppler, K., Kafai, Y. B., Resnick, M. and Rusk, N., 2008. "Programming by Choice: Urban Youth Learning Programming with Scratch", pp. , in Proc. SIGCSE'08, 39th SIGCSE Technical Symposium on Computer Science Education, pp. 367-371.

Mono Compiler, 2015. Web site: http://www.mono-project.com/docs/about-mono/languages/csharp.

Poremba, C., 2003. "Player as Author: Digital Games and Agency", Master Thesis, Dept. Computing Arts and Design Sciences, Simon Fraser University.

Reyno, E. M. and Cubel, J. A. C., 2008. "Model-Driven Game Development: 2D Platform Game Prototyping", Polytechnic University of Valencia, 3 pages.

Sotamaa, O., 2005. "Have Fun Working with our Product! Critical Perspectives on Computer Game Mod Competitions", Proc. DiGRA 2005 Conference: Changing Views – Worlds in Play, Vol., pp. 1-13.

Zagal, J. P., Mateas, M., Fernandez-Vara, C., Hochhalter, B. and Lichti, N., 2005. "Towards an Ontological Language for Game Analysis", in Proc. DiGRA 2005 Conference: Changing Views – Worlds in Play, Vol. 3, pp. 1-13.

Zagal, J. P. and Bruckman, A., 2008. "The game ontology project: supporting learning while contributing authentically to game studies", in Proc. ICLS'08 8th Int. Conf. for the Learning Sciences, pp. 499-506.