Using Game Design Mechanics as Metaphors to Enhance Learning of Introductory Programming Concepts

Chaima Jemmali*, Erica Kleinman*, Sara Bunian*, Mia Victoria Almeda†, Elizabeth Rowe†, Magy Seif El-Nasr*
∗ Northeastern University, † EdGE at TERC
{jemmali.c,kleinman.e,banian.s}@husky.neu.edu,{mia_almeda,elizabeth_rowe}@terc.edu,magy@northeastern.edu

ABSTRACT

There are several educational games and tools that teach programming. However, very few offer a deep understanding of Computer Science concepts such as Abstraction, Modularity, Semantics, and Debugging. We present *May's Journey*, an educational game that supports learning of basic programming concepts, where players solve puzzles and interact with the environment by typing in a custom programming language. The game design seamlessly integrates learning goals, core mechanics, and narrative elements. We discuss how we integrate the CS concepts mentioned above using game mechanic metaphors.

CCS CONCEPTS

• **Human-centered computing** → *Scenario-based design*; • **Applied computing** → *Interactive learning environments*;

KEYWORDS

Computer Science Education, Learning, Game Design, Metaphors

ACM Reference Format:
Chaima Jemmali*, Erica Kleinman*, Sara Bunian*, Mia Victoria Almeda†, Elizabeth Rowe†, Magy Seif El-Nasr*. 2019. Using Game Design Mechanics as Metaphors to Enhance Learning of Introductory Programming Concepts. In *The Fourteenth International Conference on the Foundations of Digital Games (FDG '19), August 26–30, 2019, San Luis Obispo, CA, USA.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3337722.3341825

1 INTRODUCTION

Making programming education more accessible to beginners is an active field of research. Constructionist approaches to programming have become popular due to their ease of use. Examples include Scratch [20], Alice [2] or Logo Programming [17], where students are invited to freely explore and construct their programs. However, while increasing interest and engagement, previous empirical studies have shown that students who used these tools had either not discovered important programming concepts, like booleans, variables, and control flow, or held misconceptions about their functions [6, 9, 11]. Furthermore, the use of these environments to teach programming depends largely on a tutor providing structure to allow students to learn specific concepts. Rather than going with a constructionist approach, we decided to use a structured approach to target informal settings. Our idea is to deliver a gaming environment that can teach specific programming concepts.

Game-based approaches provide structure while also offering agency and freedom to play. However, in most existing games, such as *CodeCombat*[1] or *Human resource Machine*[2], problems are presented as a series of tasks. While players do use knowledge acquired in previous levels to solve the current, and future, one, the code, functions, and constructs previously learned cannot be reused. The tasks are seen as independent, and, therefore, the learning objectives are mostly constrained to scripting or coding, and do not extend to modularity, abstraction, or reusability, which we target in our design.

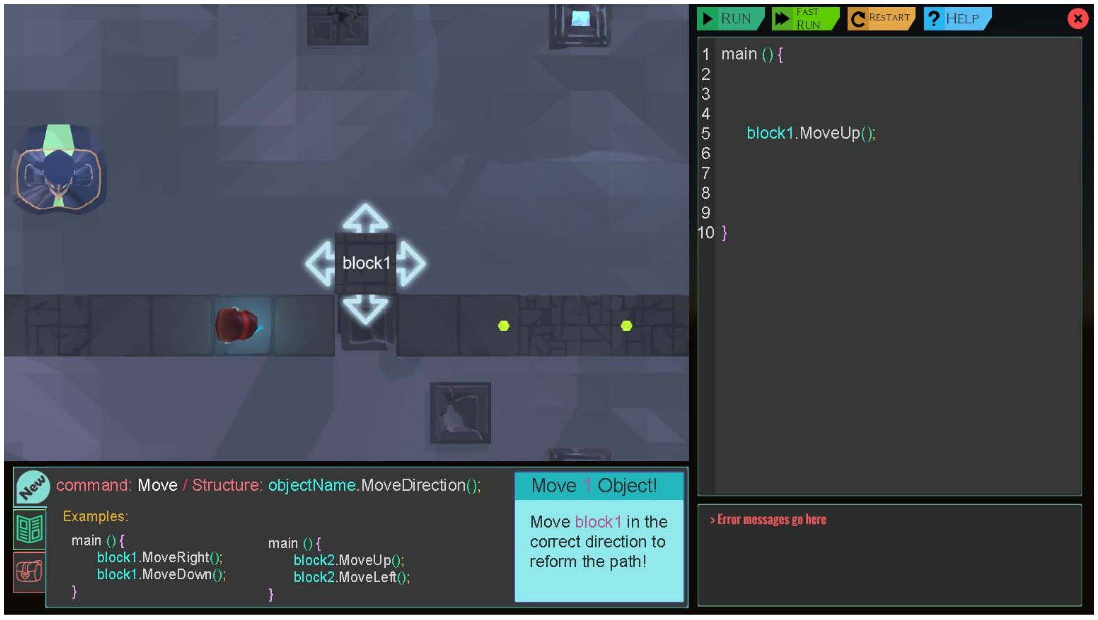
In our game, a story connects the programming problems in a way that supports the idea of decomposition of a program into smaller modular reusable units that players build and use throughout the game. Additionally, our game endorses the idea of a game world made of Objects interacting with each other, introducing Object Oriented Design in an interactive and applied way.

2 RELATED WORK

Programming and puzzle games are both environments that share requirements for inquisitive thinking and efficient problem solving. Considering programming as an application of problem-solving skills has been adopted by a strong movement in computing and informatics [3, 10, 12, 18, 22]. Problem based learning increases knowledge and understanding through complex problem solving. The challenge is that the success of the approach largely depends on the quality of the scenarios [25], which makes designing the problems, and in our case the puzzles, a critical process. There is a large number of games that are designed to focus on programming or computer science education [13, 23]. However, many of these lack scientific validation regarding their learning outcomes [16]. Moreover, in a recent review of Serious Games for programming [13], most games did not cover important concepts, such as abstract data types, or the use of software libraries. The general trend seemed to focus on problem solving with limited, or no, emphasis on formality. These concepts can be challenging for beginners and introducing them through metaphors can be an interesting approach.

In fact, metaphors have proven to be effective tools for learning [14, 19, 21, 24], exercising the imagination, promoting thought, and allowing students to conceptualize new information in a more concrete way [4, 5]. Metaphors help students build imaginative conceptions and bridge the gap between those and what is being

[1] codecombat.com
[2] tomorrowcorporation.com/humanresourcemachine



```
main () {



    block1.MoveUp();




}
```

command: Move / Structure: objectName.MoveDirection();

Examples:

```
main () {
    block1.MoveRight();
    block1.MoveDown();
}
```

```
main () {
    block2.MoveUp();
    block2.MoveLeft();
}
```

Move 1 Object!

Move block1 in the correct direction to reform the path!

> Error messages go here

Figure 1: Coding phase of the game: In the first level, the player has to change the code provided to move block1 down and rebuild the path. Top right is where they can type their code. Bottom right is the console. Bottom Left is where new commands are introduced and previous ones can be viewed.

taught [4, 15]. In subject areas where direct experience or interaction with the subject matter is impossible, metaphors can help students build an understanding by comparing it to familiar domains [4, 15, 21].

In this work, we explore the use of metaphors as game mechanics to design a game that not only teaches programming but also focuses on important concepts that have not been addressed by previous designs. These concepts include: Abstraction, Modularity, Semantics, and Debugging.

3 DESIGN

May's Journey is a 3D puzzle game in which players type code in a custom programming language to manipulate the environment and solve increasingly complex puzzles [7, 8]. The narrative tells the story of a game world, created with code, that is breaking. The player is asked to help *May*, the protagonist, solve the mystery behind the broken world in order to fix it. The gameplay alternates between two phases; an exploration phase where players can walk around, talk to NPCs, collect objects, and discover new areas, and a coding phase in which players can type programs to interact with the environment as seen in Figure 1.

The game is designed to teach the basics of programming, focusing on four essential constructs:

- **Abstraction**: understanding Object Oriented Design principles demonstrated through the ability to construct simple abstractions of classes and instances of classes.
- **Modularity and Reusability**: creating functions and routines that can serve more than one purpose and therefore can be reused.
- **Semantics**: differentiating between semantically correct and incorrect programs and identifying expected output.
- **Debugging**: identifying, locating, and solving programming logical and non-logical errors when they arise.

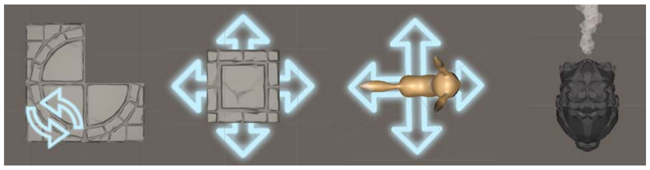
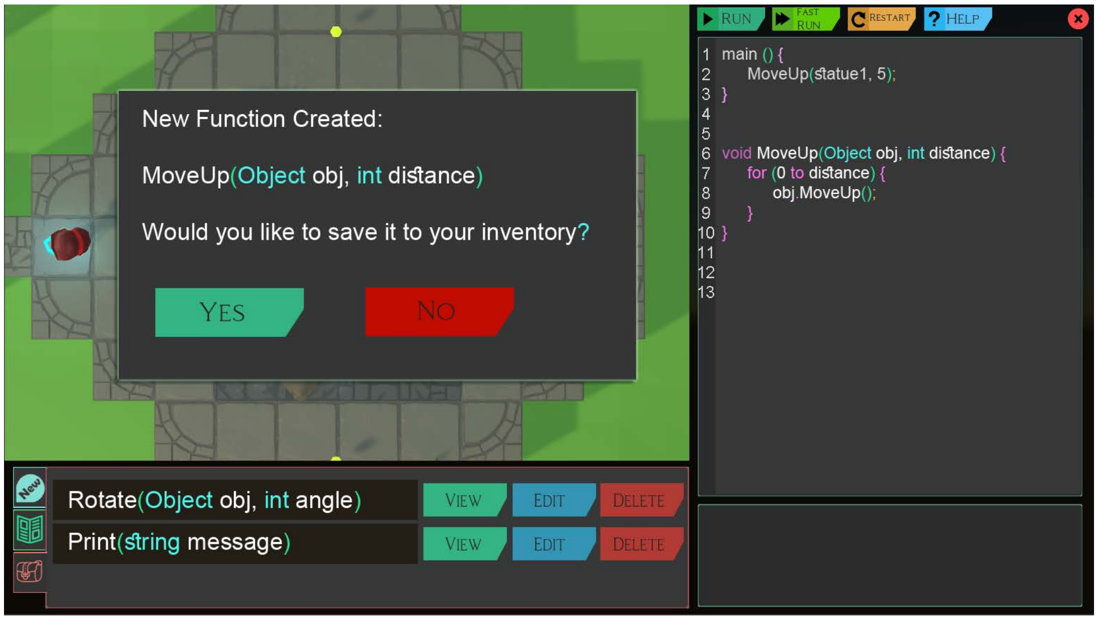


Figure 2: Interactable objects in the game: From left to right, rotatable block, block, cat, and windblock.

3.1 Abstraction

We follow Zhu and Zhou's methodology for teaching abstraction through Object Oriented paradigms starting from real world observations [26]. Their approach consists of emphasizing good practices rather than teaching the specificities of a particular language.

The use of a metaphor to teach abstraction lies in the storyline of how the game world is constructed. The hero *May* invites the player into her world. The latter discovers that her world is made of code and that they need to edit it to restore and fix the world. From observation, every element in the game is an Object from a specific class. In the first levels of the game, the objects that can be coded are already available in the scene. In later levels, players have to create new objects from classes they have already seen, and new classes for new objects.


New Function Created:

MoveUp(Object obj, int distance)

Would you like to save it to your inventory?

```
main () {
    MoveUp(statue1, 5);
}


void MoveUp(Object obj, int distance) {
    for (0 to distance) {
        obj.MoveUp();
    }
}
```

Rotate(Object obj, int angle)

Print(string message)

Figure 3: Design of the library/inventory system: After running a correct program, the player is prompted whether they would like to save new functions they created into their inventory. The bottom left part, shows previous functions that players have saved, they can be viewed, edited, or deleted.

For example, the Object of class "Block" is any block that a player can interact with in the game. There exists another class, "WindBlock", which is the same as Block, but allows for more functions. For example, the wind from a WindBlock can obstruct the player's way, but it can also propagate fire to light pits and reveal new items or unlock additional locations. The WindBlock class can be thought of a as class that extends Block. They share similar properties with an added functionality for the child class. Figure 2 shows some of the available interactable objects in the current version of the game. The player must write code to move or rotate these objects such that they are able to fulfill their roles within the gameplay.

This kind of metaphor is very important for beginners. Seeing the pieces of the world as objects, and seeing the whole game world as a multitude of programmable objects interacting with each other helps in presenting programming problems as "real life" situations, which makes problem solving more efficient[1].

3.2 Modularity

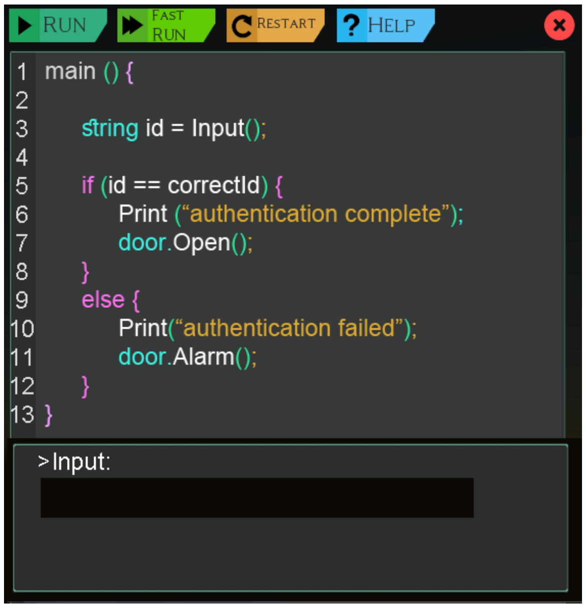
To encourage students to reuse functions when solving large problems, we first introduce smaller problems, in which they use simple functions. Whenever the player creates a function for the first time, they are given the choice to save that function in their inventory. The inventory acts as our metaphor for modularity, emulating the inventory system in games, where players store tools that help them progress. In this case, their tools are the programming functions and classes they create. Figure 3 shows an example of a player writing the MoveUp function. After they run their code, if the function does not contain errors, they are able to save it into their inventory and reuse it in the following levels.

Players can also view, edit, or remove functions they had already created. Later problems in the game will require players to use a multitude of functions from the inventory. This is a design choice that allows us to emphasize the importance of considering the bigger picture when solving a problem or writing a function. Players may need to revisit their functions and modify them to facilitate their work. For example, the game provides built-in functions such as Object.MoveUp(); as shown in 1. The player can augment this function by adding a distance argument. The function becomes MoveUp(GameObject obj, int distance) and it moves a game object "obj" Up by "distance" units. This function can also be modified to include a "direction" argument. MoveObject(GameObject obj, string direction, int distance) will move "obj" by "distance" units in a specific "direction". At the beginning, through exploration, feedback, NPCs, and manuscripts, the game will provide hints and advice. Eventually, players will learn to recognize patterns and create reusable functions without help from the game.

3.3 Semantics

In certain levels, players discover a puzzle where a piece of code has already been written. Depending on the context, it could have been left there by the antagonist, a companion, etc. The code will be syntactically correct, but semantically wrong (gives a runtime error). The player needs to understand why the error is happening and fix it.

In other levels, players will find a programming interface that has correct code, but the run button is compromised and they cannot click it to run the program. The output of the code given provides



```
main () {

    string id = Input();

    if (id == correctId) {
        Print ("authentication complete");
        door.Open();
    }
    else {
        Print("authentication failed");
        door.Alarm();
    }
}
```

```
>Input:
```

Figure 4: Example of a semantics puzzle: Players have to enter the correct password in the input field.

important information - a secret code to a door or an answer to a riddle they need to say to an NPC, etc. Players should be able to fully understand how their programs work and how variables change in each step.

In some other levels, players will be given an existing program they need to decode in order to progress. The example in Figure 4 shows a problem where a locked door needs a password. The players need to type in the correct password in the input field, but cannot modify the existing code. If they enter the wrong password, an alarm is activated and they have to restart the level. The correct password is hidden in another program within the level. This example might be simple, but later levels will include loops, and variables inside loops. Misconceptions about variables in loops have been observed with beginners [11], hence the importance of addressing this point through increasingly difficult levels.

These types of levels also present a variation from writing their own code. It enforces the narrative where programming is prevalent in this world, other characters can use it, and players need to uncover secrets, passwords, and riddles as a metaphor for semantics puzzles.

3.4 Debugging

In some levels, a broken program is presented to the player. The program is not behaving in the correct way and the player must identify the problem, the expected output, and correct the program to produce the correct behavior. This task helps with semantics as well as debugging. Understanding a given program, debugging, and correcting it are essential skills for a programmer. To make the debugging process smoother, error messages are designed to be simple and easy to understand. The messages are also designed in the form of a conversation with the machine. If there is an error, it is because the editor did not understand the player's code, so they need to change it. The messages will get easier if the player encounters the same error multiple times. For example, instead of writing "'blockk1' does not exist in the current context, line 3", the message will read "I don't understand the word 'blockk1' in line 3". If the same error is encountered again, the message becomes "I don't understand the word 'blockk1' in line 3, did you mean 'block1'? ". Debugging becomes a type of conversation with the code editor. It is also analogous to the interaction with the *Google* search bar when there is a typo. The metaphor of a discussion with the console as a way of debugging can make fixing mistakes more natural and less intimidating.

Players can also rely on a debugging button to further understand the program. During the first levels of the game, step by step code running and highlight of which line is being run happens automatically. As the player progresses, this becomes optional since it is time consuming and might take away from playtime. This is when the debugging button appears, it will still provide step by step running but will add the functionality of looking into variables content after each line of code.

4 DISCUSSION AND LIMITATIONS

There are multiple challenges that arise from designing around these specific constructs. The first is how to assure a smooth learning curve. In each level, we introduce either one new CS construct or one new game mechanic. This becomes more and more difficult when constructs get more complicated and more dependent on each other. For instance, how does one introduce boolean logic in a playful way? This would most likely include introducing conditionals or while loops which may be overwhelming, but at the same time introducing booleans by themselves could seem boring or non-playful.

Another challenge is how to make sure that students are learning the proper constructs. For example, how to encourage them to use the inventory without forcing them to do so. There should be an intrinsic motivation and an awareness about its utility rather than extra points or a penalty.

Finally, assessing all these constructs will require a longitudinal study in which students play parts of the game over a long period of time so they have time to assimilate and build on their previous knowledge.

5 CONCLUSION AND FUTURE WORK

We presented *May's Journey*, an educational puzzle game teaching programming. The main distinguishing features of the game are the use of problem solving and real world analogy to teach object oriented paradigms, the integration of a story to increase intrinsic motivation, and the introduction to a text-based custom language to emphasize debugging. Most importantly, the game highlights the importance of good practices for programmers, focusing on aspects such as Abstraction, Modularity, Semantics, and Debugging. In future work, we plan to build several puzzles for each construct and conduct studies to assess how does the game affect students' learning in each of these constructs.

6 ACKNOWLEDGEMENTS

This research is supported by NSF AISL (Advancing Informal STEM Learning) Award Id : 1810972

REFERENCES

[1] David Boud and Grahame Feletti. 2013. *The challenge of problem-based learning*. Routledge.

[2] Stephen Cooper, Wanda Dann, and Randy Pausch. 2000. Alice: a 3-D tool for introductory programming concepts. In *Journal of Computing Sciences in Colleges*, Vol. 15. Consortium for Computing Sciences in Colleges, 107–116.

[3] Giuliana Dettori and Ana Paiva. 2009. Narrative learning in technology-enhanced environments. *Technology-Enhanced Learning* (2009), 55–69.

[4] Reinders Duit. 1991. On the role of analogies and metaphors in learning science. *Science education* 75, 6 (1991), 649–672.

[5] Shaun Gallagher and Robb Lindgren. 2015. Enactive metaphors: Learning through full-body engagement. *Educational Psychology Review* 27, 3 (2015), 391–404.

[6] Shuchi Grover and Satabdi Basu. 2017. Measuring student learning in introductory block-based programming: Examining misconceptions of loops, variables, and boolean logic. In *Proceedings of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education*. ACM, 267–272.

[7] Chaima Jemmali, Sara Bunian, Andrea Mambretti, and Magy Seif El-Nasr. 2018. Educational Game Design: An Empirical Study of the Effects of Narrative. *learning* 66 (2018), 68.

[8] Chaima Jemmali and Zijian Yang. 2016. *May's Journey: A serious game to teach middle and high school girls programming*. Master's thesis. Worcester Polytechnic Institute.

[9] D Midian Kurland, Catherine A Clement, Ronald Mawby, and Roy D Pea. 1987. Mapping the cognitive demands of learning to program. In *Mirrors of Minds: Patterns of experience in educational computing*. Ablex Publishing Corp., 103–127.

[10] Essi Lahtinen, Kirsti Ala-Mutka, and Hannu-Matti Järvinen. 2005. A study of the difficulties of novice programmers. In *Acm Sigcse Bulletin*, Vol. 37. ACM, 14–18.

[11] John H Maloney, Kylie Peppler, Yasmin Kafai, Mitchel Resnick, and Natalie Rusk. 2008. *Programming by choice: urban youth learning programming with scratch*. Vol. 40. ACM.

[12] Richard E Mayer. 1992. Teaching for transfer of problem-solving skills to computer programming. In *Computer-based learning environments and problem solving*. Springer, 193–206.

[13] Michael A Miljanovic and Jeremy S Bradbury. 2018. A Review of Serious Games for Programming. In *Joint International Conference on Serious Games*. Springer, 204–216.

[14] Kai Niebert and Harald Gropengießer. 2011. âĂIJCO 2 Causes a Hole in the AtmosphereâĂİ: Using LaypeopleâĂŹs Conceptions as a Starting Point to Communicate Climate Change. In *The economic, social and political elements of climate change*. Springer, 603–622.

[15] Kai Niebert, Sabine Marsch, and David F Treagust. 2012. Understanding needs embodiment: A theory-guided reanalysis of the role of metaphors and analogies in understanding science. *Science Education* 96, 5 (2012), 849–877.

[16] Velian T Pandeliev and Ronald M Baecker. 2010. A framework for the online evaluation of serious games. In *Proceedings of the International Academic Conference on the Future of Game Design and Technology*. ACM, 239–242.

[17] Roy D Pea. 1987. Logo programming and problem solving. (1987).

[18] Arnold Pears, Stephen Seidman, Lauri Malmi, Linda Mannila, Elizabeth Adams, Jens Bennedsen, Marie Devlin, and James Paterson. 2007. A survey of literature on the teaching of introductory programming. *ACM SIGCSE Bulletin* 39, 4 (2007), 204–223.

[19] Miriam Reiner, James D Slotta, Michelene TH Chi, and Lauren B Resnick. 2000. Naive physics reasoning: A commitment to substance-based conceptions. *Cognition and instruction* 18, 1 (2000), 1–34.

[20] Mitchel Resnick, John Maloney, Andrés Monroy-Hernández, Natalie Rusk, Evelyn Eastmond, Karen Brennan, Amon Millner, Eric Rosenbaum, Jay Silver, Brian Silverman, et al. 2009. Scratch: programming for all. *Commun. ACM* 52, 11 (2009), 60–67.

[21] Tanja Riemeier and Harald Gropengießer. 2008. On the roots of difficulties in learning about cell division: process-based analysis of studentsâĂŹ conceptual development in teaching experiments. *International Journal of Science Education* 30, 7 (2008), 923–939.

[22] Ralf Romeike. 2008. What's my challenge? The forgotten part of problem solving in computer science education. *Informatics Education-Supporting Computational Thinking* (2008), 122–133.

[23] Adilson Vahldick, António José Mendes, and Maria José Marcelino. 2014. A review of games designed to improve introductory computer programming competencies. In *2014 IEEE frontiers in education conference (FIE) proceedings*. IEEE, 1–7.

[24] Marianne Wiser and Tamer Amin. 2001. âĂIJIs heat hot?âĂİ Inducing conceptual change by integrating everyday and scientific perspectives on thermal phenomena. *Learning and Instruction* 11, 4-5 (2001), 331–355.

[25] Diana F Wood. 2003. ABC of learning and teaching in medicine: Problem based learning. *BMJ: British Medical Journal* 326, 7384 (2003), 328.

[26] Haibin Zhu and MengChu Zhou. 2003. Methodology first and language second: A way to teach object-oriented programming. In *Companion of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*. ACM, 140–147.