Stereotypes as Design Patterns for Serious Games to Enhance Software Comprehension

René Gökmen*, David Heidrich†, Andreas Schreiber‡, and Christoph Bichlmeier*
*Faculty of Informatics—Game Engineering, University of Applied Sciences, Kempten, Germany
goekmen.rene@web.de, christoph.bichlmeier@hs-kempten.de
† Institute for Software Technology, German Aerospace Center (DLR), Weßling, Germany
david.heidrich@dlr.de
‡ Institute for Software Technology, German Aerospace Center (DLR), Cologne, Germany
andreas.schreiber@dlr.de

***Abstract*—Stereotypes support a high-level software comprehension by implying roles and responsibilities of classes in software systems. We propose the idea of using stereotypes as design patterns for serious games to enhance object oriented software comprehension. These design patterns can reduce the complexity of software systems and encode software knowledge into game mechanics. We provide examples of possible game mechanics and discuss the application of the proposed design patterns.**

***Index Terms*—Serious Games, Design Patterns, Software Engineering, Software Comprehension, Stereotypes**

I. Introduction

In object-oriented programming, *classes* contain the behavior and data of a software system. Hence, understanding the role and responsibilities of classes is necessary for many software engineering tasks [1]. Due to the abstract and complex nature of software systems, this can quickly evolve into a mentally demanding and time consuming activity. For example, professional developers invest more than 50% of their working time on software comprehension instead of writing source code [2].

Serious games can be an effective tool for acquiring expert knowledge in basically every area [3]. By encoding expert knowledge in game mechanics, the player repetitively applies knowledge [4]. This repetitive knowledge application ultimately results in a learning effect [5]. Additionally, well-designed video games are highly engaging and produce flow [6], [7]. Flow is also called the optimal experience or being in *the zone* and it ultimately increases a player's intrinsic motivation for knowledge acquisition [8].

The design of serious games in software comprehension is generally considered very challenging, due to the high number of different concepts [9]. Still, a few serious games in the broader area of software comprehension exist [10]–[12]. However, designing serious games specifically for software comprehension features some challenges. Depending on the size of the software system, the amount of knowledge that must be taught can be very high. Software comprehension requires knowledge of—for example—class properties (e.g., names or lines of code), dependencies between classes, or dependencies between all other software elements [13]. Even in medium-sized software systems, the number of different classes can easily reach into the thousands. Hence, it would be far too time-consuming to integrate all these properties into the game design by hand. Additionally, no two software systems are alike and the distribution of class roles can differ strongly [14], which amplifies this design challenge even more. Thus, there is a need for a general reusable solution to design serious games for software comprehension, i.e., game design patterns [9], [15].

We propose the use of *stereotypes* as game design patterns for serious games. Stereotypes are a high-level implication of a class' role and responsibility in a software system [14], [16]. Class roles contain implications on—for example—activities and dependencies, which allow for interesting and diverse game design patterns. Stereotypes help in comprehending high-level software systems—independent from the source code. Hence, stereotypes can also enable non-expert users, who cannot understand source code, to gain insight of a software system.

In the following, we propose the idea of using game design patterns based on stereotypes (Section II) and discuss their application (Section III).

II. Stereotypes as Game Design Patterns

Stereotypes are classified roles of software objects [16] that can help developers to gain and maintain a better understanding of a software system [18], which is crucial for understanding its functionality and for modifying source code [19]. For example, a stereotype indicates how a class collaborates with other classes or how important a class might be to the software system (Table I). Since every software system is different, the distribution of stereotypes can differ strongly [14], [16]. Hence, before designing a serious game for a specific software system, the stereotype distribution of that software system should be studied by the game designer.

978-1-6654-3886-5/21/$31.00

TABLE I
OVERVIEW OF THE CLASS ROLE STEREOTYPES [16] WITH GAME DESIGN PATTERNS AND AN EXAMPLE OF THEIR PROBABILITIES [17].

Stereotype	Description	Game Design Pattern	Prob.
Controller	Make decisions and control others actions.	*Controllers* have the power to tell all other stereotypes what to do and how to do it.	2.5%
Coordinator	Delegate work to other objects when triggered by events.	*Coordinators* do not like to make own decisions. They rather delegate work to *Service Providers* when being told by *Controllers*, *Interfacers*, or events. *Coordinators* may report back to the *Controller*.	10.1%
Information Holder	Knows certain information and provides information to others	*Information Holders* provide *Controllers*, *Coordinators*, and *Service Providers* with information on demand.	29.7%
Interfacer	Transforms information and requests between distinct parts of a system.	*Interfacers* are often controlled by a *Controller*. *Interfacers* might collaborate with *Coordinators* and *Service Providers* to move certain information across different system layers.	9.9%
Service Provider	Performs work and offer services to others.	*Service Providers* do work and can store information by collaborating with *Imformation Providers* and *Structurers*.	41.5%
Structurer	Maintains relationships between objects and provides information about those relationships.	*Structurers* organize and store informations in *Information Holders*. *Structurers* may collect information and link it to several information holders.	6.3%

A stereotype's role, responsibility, and collaboration can act as design patterns to support the idea generation and help create a game design. Initially, a game designer should not have to dive into the potentially overwhelming amount of data of a software system. Rather, they start by gaining a high-level insight of the software system, by classifying classes into stereotypes. The design patterns of the stereotypes add game characteristics and game behaviours to the classes. Additionally, the game designer gains insights of the stereotype distribution of the software system, which can further support the idea generation. The game characteristics of the classes become even more diverse, when the game designer adds additional software metrics that further enhance the software comprehension — such as dependencies or lines of code. The results are game design patterns that provide insights into the software system by using stereotypes and software metrics. As these design patterns are not based on specific source code, the game designer can also apply them to other comparable software systems. The final step is to convert the design patterns into a concrete game design that respects the identified game characteristics of the classes.

In the following, we describe the stereotypes design patterns and possible example game mechanics in detail—including an overview of all stereotypes and their game design implications (Table I).

- *Controller* classes are designed to make decisions and to control others actions. They make complex decisions on basis of their provided information and then tell all other stereotypes what to do and how to do it. This could be implemented into the game design as a general that makes decisions and orders its subordinates.
- *Coordinator* classes delegate work to other stereotypes when being triggered by events. Coordinators do not like to make own decisions. They rather delegate work to Service Providers when being told by a Controller, an Interfacer, or on a specific event. Coordinators may report back to the Controller. This could be implemented into the game design as a unit leader that receives orders from the general and then delegates the fighters to complete the order. If the fight was waged the unit leader may report back to the Controller.
- *Information Holder* classes know certain information and provide information to others. They provide Controllers, Coordinators, and Service Providers with information on demand. This could be implemented into the game design as a storage container or a prison room that is organized by an overseer. The storage container shares its content with the general, unit leader and fighter on demand.
- *Interfacer* classes transform information and request between distinct parts of a system. Interfacers are often controlled by a Controller. Interfacers might collaborate with Coordinators and Service Providers on different system layers to conduct a cross-layer task. This could be implemented into the game design as a medium between different stereotypes like a radio or telephone.
- *Service Provider* classes do all kinds of work and offer services to Coordinators, Interfacers, and Controllers. For example, they can store information by collaborating with Information Holders and Structurers. This could be implemented into the game design as a fighter in the front line, awaiting orders. The fighter concentrates only on fulfilling its task and may use the storage container and overseer to access or store information.
- *Structurer* classes maintain relationships between objects and expose information about those relationships. Structurers organize and store information in Information Holders. This could be implemented into the game design as an overseer, which makes sure that the content of the assigned storage container are organized. An overseer may receive content from fighters and generals.

III. Discussion

Using serious games for software comprehension is challenging. In this domain, game designers must be experts in software engineering to understand complex software systems and to translate this knowledge into game design. Additionally, as each software system differs, the game design for one software system might not work with other software systems. Hence, every software system may require the game designer to fundamentally rethink their game design.

Using automatic identification of stereotypes [14], [17] can help and speed up the software comprehension process required by the game designer. That way, instead of inspecting and interpreting each class' source code, classes' roles and responsibilities are instantly apparent to the designer. Additionally, software systems can be compared based on their stereotype distribution. Similar distributions should be able to share similar game design approaches. Since all object-oriented software systems are based on the same basic stereotypes, well-designed games might even be able to address different stereotype distributions and different sizes of software systems (i.e., the number of classes).

All these considerations, however, require an automatic analysis of stereotypes that does not misidentify stereotypes. When classes cannot be assigned to a specific stereotype or when a wrong stereotype is assigned, the educational aspect of the serious game will be undermined. Especially in big software systems, it becomes unreasonable to verify every single class stereotype. For this reason clear classification rules (e.g., based on a classes method properties [14]) must be used.

In addition to the proposed design patterns, we encourage game designers to use additional software metrics that further enhance the software comprehension. Possible software metrics are—for example—number of methods or code complexity [20].

IV. Conclusion and Future Work

Stereotypes contain characteristics and behaviors that can be translated into design patterns. These design patterns support the game designer by reducing the overall complexity of the software system and by supporting the idea generation. Resulting game designs can be applied to varying software systems, because stereotypes are dissociated from specific software systems or source code. Most importantly, our design patterns encode the knowledge of the classes' roles and responsibilities into game mechanics. Hence, the proposed design patterns should be suited for serious games. Therefore, future work includes the application of the proposed game design patterns to exemplary serious games with different types of software systems to evaluate learning effects.

References

[1] N. Dragan, M. L. Collard, and J. I. Maletic, "Reverse engineering method stereotypes," in *2006 22nd IEEE International Conference on Software Maintenance*. IEEE, 2006, pp. 24–34.

[2] X. Xia, L. Bao, D. Lo, Z. Xing, A. E. Hassan, and S. Li, "Measuring program comprehension: A large-scale field study with professionals," *IEEE Transactions on Software Engineering*, vol. 44, no. 10, pp. 951–976, 2017.

[3] K. A. Ericsson, R. T. Krampe, and C. Tesch-Römer, "The role of deliberate practice in the acquisition of expert performance," *Psychological review*, vol. 100, no. 3, p. 363, 1993.

[4] S. Oberdörfer and M. E. Latoschik, "Knowledge encoding in game mechanics: Transfer-oriented knowledge learning in desktop-3d and vr," *International Journal of Computer Games Technology*, vol. 2019, 2019.

[5] J. P. Gee, "What video games have to teach us about learning and literacy," *Computers in Entertainment (CIE)*, vol. 1, no. 1, pp. 20–20, 2003.

[6] M. Csikszentmihalyi and M. Csikzentmihaly, *Flow: The psychology of optimal experience*. Harper & Row New York, 1990, vol. 1990.

[7] J. Chen, "Flow in games (and everything else)," *Communications of the ACM*, vol. 50, no. 4, pp. 31–34, 2007.

[8] C. J. Dede, J. Jacobson, and J. Richards, "Introduction: Virtual, augmented, and mixed realities in education," in *Virtual, augmented, and mixed realities in education*. Springer, 2017, pp. 1–16.

[9] G. Zavcer, S. Mayr, and P. Petta, "Design pattern canvas: towards co-creation of unified serious game design patterns," in *2014 6th International Conference on Games and Virtual Worlds for Serious Applications (VS-GAMES)*. IEEE, 2014, pp. 1–2.

[10] M. A. Miljanovic and J. S. Bradbury, "Robot on! a serious game for improving programming comprehension," in *Proceedings of the 5th International Workshop on Games and Software Engineering*, 2016, pp. 33–36.

[11] R. Bryce, "Bug wars: a competitive exercise to find bugs in code," *Journal of Computing Sciences in Colleges*, vol. 27, no. 2, pp. 43–50, 2011.

[12] N. Tillmann, J. Bishop, N. Horspool, D. Perelman, and T. Xie, "Code hunt: Searching for secret code for fun," in *Proceedings of the 7th International Workshop on Search-Based Software Testing*, 2014, pp. 23–26.

[13] L. Merino, M. Ghafari, C. Anslow, and O. Nierstrasz, "A systematic literature review of software visualization evaluation," *Journal of Systems and Software*, vol. 144, pp. 165–180, 2018.

[14] N. Dragan, M. L. Collard, and J. I. Maletic, "Automatic identification of class stereotypes," in *2010 IEEE International Conference on Software Maintenance*. IEEE, 2010, pp. 1–10.

[15] J. Holopainen and S. Björk, "Game design patterns," *Lecture Notes for GDC*, 2003.

[16] R. Wirfs-Brock and A. McKean, *Object design: roles, responsibilities, and collaborations*. Addison-Wesley Professional, 2003.

[17] A. Nurwidyantoro, T. Ho-Quang, and M. R. Chaudron, "Automated classification of class role-stereotypes via machine learning," in *Proceedings of the Evaluation and Assessment on Software Engineering*, 2019, pp. 79–88.

[18] S. Yusuf, H. Kagdi, and J. I. Maletic, "Assessing the comprehension of uml class diagrams via eye tracking," in *15th IEEE International Conference on Program Comprehension (ICPC'07)*. IEEE, 2007, pp. 113–122.

[19] T. Ball and S. G. Eick, "Software visualization in the large," *Computer*, vol. 29, no. 4, pp. 33–43, 1996.

[20] M. Lanza and R. Marinescu, *Object-oriented metrics in practice: using software metrics to characterize, evaluate, and improve the design of object-oriented systems*. Springer Science & Business Media, 2007.