

# Analogy of Incremental Program Development and Constructivist Learning

Václav Rajlich, Shaochun Xu  
Department of Computer Science  
Wayne State University  
Detroit, MI 48202  
{vtr, w20043}@cs.wayne.edu

## Abstract

*During software evolution, programmers add new functionalities and release new versions of software. This complicated work involves not only program development but also learning new knowledge. This paper explores an analogy between incremental program development and constructivist learning, and presents a case study that investigates this analogy. Four types of cognitive processes have been identified. They parallel analogous software engineering activities*

## 1. Introduction

One of the puzzling issues of software engineering is the nature of the knowledge that is needed in order to comprehend a program. The program itself is a repository of knowledge about the program domain and may contain knowledge that is not available elsewhere, as documented in [5]. It also contains knowledge of all design decisions that were made during the program development and consequent program evolution [11]. The work of programmers during the program development and evolution is to process the domain knowledge and turn it into design decisions that will be reflected in the source code.

When evolving or maintaining the program, it is necessary to recover that knowledge; otherwise maintenance or evolution will be impossible. It is also necessary to communicate this knowledge to all new programmers who are joining an existing software project.

Although the knowledge is embedded in the program, it cannot be easily recovered since the bits of knowledge are delocalized. Another hurdle is that the consequences

of the decisions, rather than the decisions themselves, appear in the code. In many ways, the recovery of knowledge from the code is an activity that is similar to solving a puzzle, and is laborious and error prone.

In order to understand the nature of programming knowledge, we analyzed the discussion of a pair of programmers [6] during the incremental software process. In pair programming, two programmers work side-by-side at one machine. They collaborate on design, algorithm selection, implementation and testing [13]. The programming pair has to communicate and share the knowledge. This gives an opportunity to analyze what that knowledge consists of. By analyzing their discussion, we traced a continuing and parallel growth of both the program and the programmers' knowledge. We claim that the knowledge growth is analogous to the program growth and is explained by constructivist learning theory. The analogy is captured by analogy between four fundamental software engineering activities and four basic cognitive processes.

Section 2 describes our terminology. Section 3 describes the case study. Section 4 contains an overview of the related literature. Section 5 contains conclusions and the future work.

## 2. Incremental program development and constructivist learning

Incremental program development and subsequent program evolution is a process where the programmers add one program property at a time. Very rarely does the programming start from scratch. Often, programmers start with an existing program and add and delete properties from it. Sometimes they start with a reusable program library or a program framework that covers fundamental functionality.

The programmers construct the program based on the sequence of change requests, called “user stories” in XP [1]. An example of a change request is “add processing of credit cards to a point of sale system.”

During the change the programmer deals with two systems: One is the backlog of change requests and the other one is the program. The programmers take change requests from the backlog one at a time and make corresponding program changes. Each step is called an iteration.

However as programmers do iterations, they reject some change requests because they are too difficult or even impossible to accomplish. An example of an impossible change request is to increase the processing speed of a program that is already fully optimized. There is also a continuum between complete acceptance of a change request and outright rejection: Programmers may accept some change requests after they have been suitably modified. However for the clarity of the discussion, we will emphasize either complete acceptance or complete rejection of the change requests.

The changes that the programmers perform on the programs fall into three categories: additions of the new functionality called incremental changes, deletions of functionality called retractions, and restructurings of the software called refactorings. The actual changes done to the program can be combinations of these three basic changes, but again for the clarity of the discussion, we will emphasize these three “pure” changes. Of them, the incremental change is done in direct response to a change request, while retraction and refactoring are done in response to the whole history of accumulated incremental changes and resulting program structure.

As the programmers develop the program incrementally, they also obtain new knowledge. We use the constructivist theory of learning as the foundation for our study of how programmers handle this new knowledge.

Constructivist learning is a theory that explains the process through which people learn new facts and add them to their knowledge. It is based on the work of Piaget [8]. The original aim of the Piaget’s work was to explain learning in children, but the constructivist theory extends to adult learning and to epistemology [12]. The basic assumption of the theory is the hypothesis that the learners actively and incrementally construct their knowledge. They start from some pre-existing knowledge and they extend it by connecting new facts to previously learned knowledge. They may go through stages in which they may accept ideas they will later discard as wrong. According to Piaget, he two main actions are assimilation and accommodation.

Assimilation describes how learners deal with new knowledge. In order to describe assimilation better and emphasize the analogy with incremental program construction, we subdivide it into two processes. “Positive assimilation” (assimilation +) means that the learners add new facts to their knowledge. However in situations where the new facts do not fit in, learners reject them. We call this second process “negative assimilation” (assimilation -).

Accommodation describes how learners reorganize their existing knowledge. Again we subdivide it into two processes. “Positive accommodation” (accommodation +) means that the learners reorganize their knowledge to aid absorption of new facts. “Negative accommodation” (accommodation -) is the process where part of the knowledge becomes obsolete or provably incorrect and the learners reject it.

The analogy between incremental program development and constructivist learning is summarized in Table 1 where analogous terms are in the same row.

**Table 1. Analogy of incremental development and constructivism**

<b>Incremental development</b>	<b>Constructivist learning</b>
Program	Knowledge
Change request	New fact
Incremental change	Assimilation +
Rejection of change request	Assimilation -
Refactoring	Accommodation +
Retraction	Accommodation -

The process that we study in this paper is the process of incremental development of both program and the knowledge that programmers possess about the program. As the program grows during the incremental development, so does programmers’ knowledge. Each incremental addition to the program means also an addition to the programmers’ knowledge.

Pair programming offers a unique opportunity to study this parallel construction of the program and the knowledge. In pair programming, the programmers work in pairs and communicate with each other about the evolving program. By recording their dialog, we can study how the knowledge and program grow together. In particular, it allows us to study all basic processes of both program and knowledge construction. The case study of the next section examines these processes.

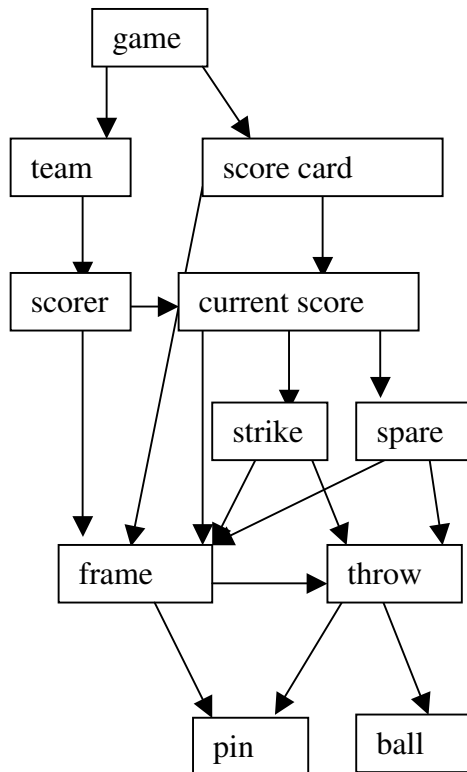


Figure 1. Domain knowledge

### 3. Case study

We analyzed a case study of pair programming applied to development of the program that keeps bowling scores [6]. The programming pair starts the process with the knowledge of the domain (bowling rules) and the knowledge of the software development techniques. The case study of [6] records both the growth of the program and the dialog. We analyzed the dialog as the evidence of the way the knowledge of both programmers grows.

The original knowledge of the program domain is in Figure 1. Concepts are represented by rectangles and the arrows represent the dependencies. The dependencies stand for the order in which the concepts have to be explained to somebody who is not familiar with the domain. A concept can be explained only if all previous concepts it is dependent on have been explained and understood.

This original domain knowledge serves also as a backlog of things to do. The programmers want to

implement a program with functionality described by this domain knowledge.

Pre-existing programming knowledge includes knowledge of the programming languages, techniques, principles and patterns, computing environment, algorithms, testing, and so forth.

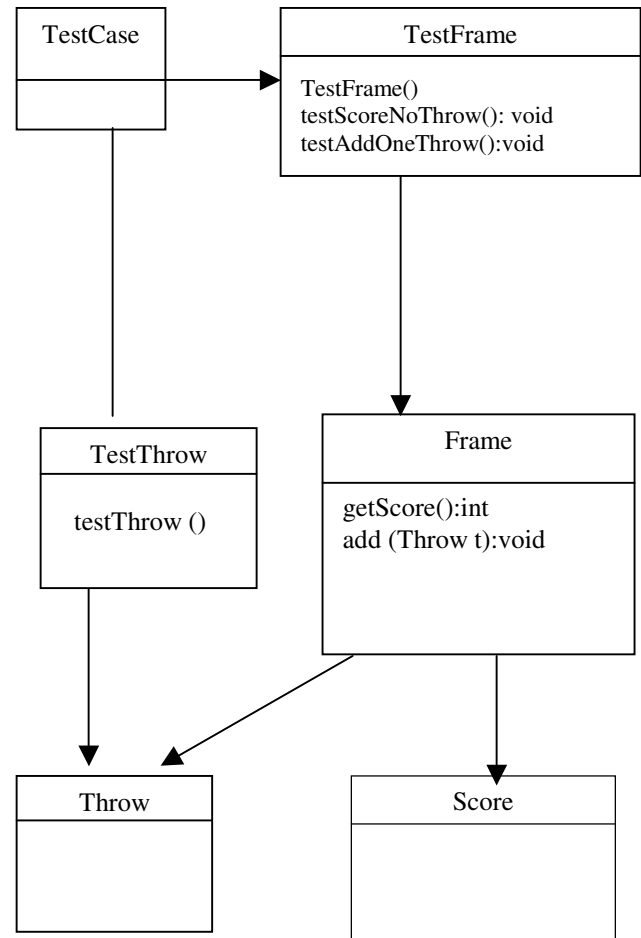
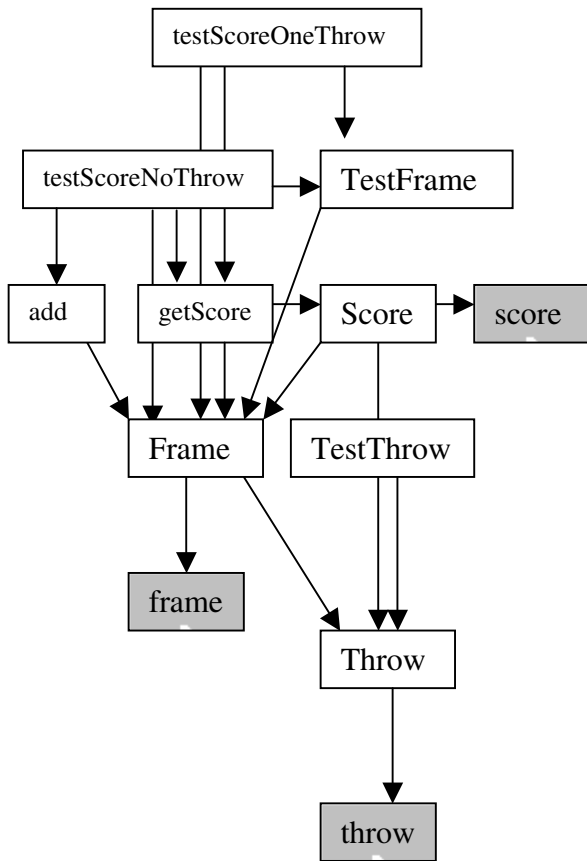


Figure 2. UML diagram of the first version

Equipped with this knowledge, the programmers implemented the first version of the program. The UML diagram of the first version is in Figure 2. The design decisions that led to this diagram were discussed in the accompanying dialog and appear in Figure 3. The rectangles represent design decisions while arrows represent the order in which the design decisions were made. Dark rectangles represent domain concepts that

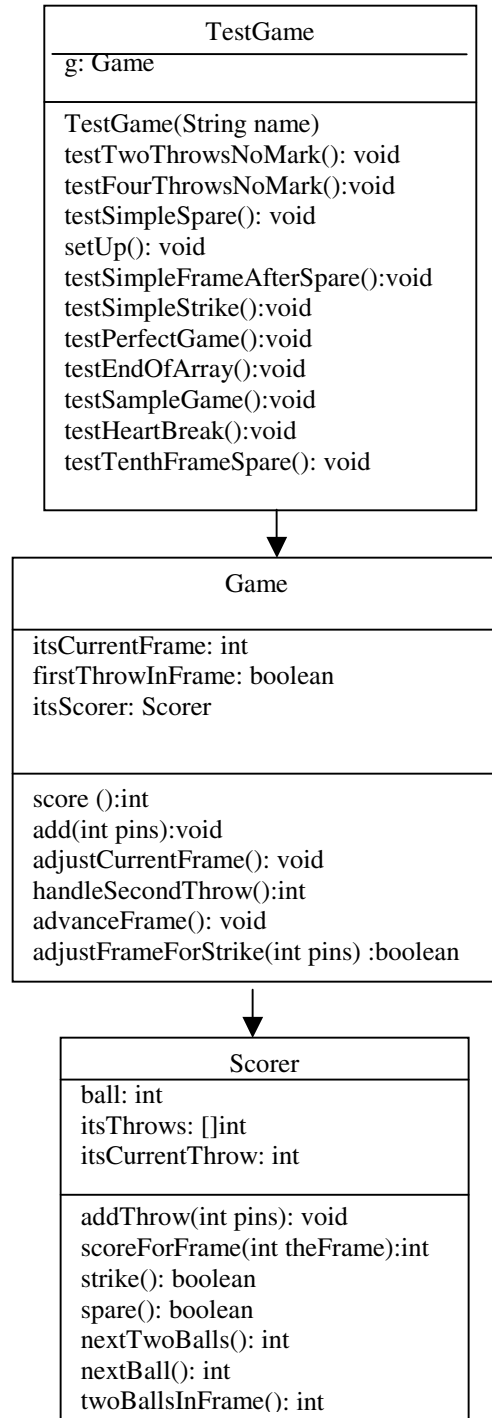
serve as the basis of some of the design decisions. Please note that the order of the design decisions does not correspond to the order of the dependencies in the UML diagrams. If the two orders were identical, that would mean that all design decisions were done in the bottom-up order.



**Figure 3. Design decisions for the first version**

During the pair programming process, the knowledge of the programmers about the application domain grew. At the same time, the code was expanding. The new knowledge therefore also included knowledge of the program and the design decisions in it, as the domain knowledge was increasingly reflected in the growing program.

Appendix 1 contains design decisions accumulated in the first part of the dialog, mostly through the repeated positive assimilation. It contains several homonyms, as different design decisions are denoted by the same name, similarly as with the name overloading in object oriented programs.



**Figure 4. Final program**

However during this first part of the dialog, there were also instances of negative assimilation, as programmers discussed and rejected implementation of concepts of Score Card and Team.

Appendix 2 shows the situation after the programmers realized that the class “Frame” and the related classes do not contribute towards the functionality of the program and deleted them. In parallel with that, they changed their knowledge and through negative accommodation they abandoned the design decisions related to these classes.

The class diagram of the final program, including the testing class, is in Figure 4.

In the case study, we observed that the knowledge required by even a small program is quite extensive, as demonstrated in figures in Appendix 1 and 2. It should be remembered that these figures are only the “tip of the iceberg”, as there is a large preliminary knowledge of the domain and programming that these figures do not capture. Yet all that knowledge is necessary to evolve the program.

During the iterative program development, the most common change in the program was incremental change and the most common cognitive process was positive assimilation. These two processes were the driving force behind the program development. However there was one large episode of code retraction accompanied with the corresponding negative accommodation. There were also frequent episodes of refactoring and corresponding positive accommodation. The rejection of concepts Score Card and Team in the first part of the dialogs are examples of negative assimilation.

#### 4. Related work

There have been numerous publications on constructivist learning, including the classical work of Piaget [8]. A more recent summary of the constructivism appeared in [12]. Piaget dedicated his work to observe and hypothesize how children make their discoveries and defined assimilation and accommodation as two complementary processes [8]. In [12], von Glaserfeld described constructivism as a theory of knowledge. He defined radical constructivism as a theory of learning, and saw facts as being actively received either through the senses or through communication. In [7], Novak addressed theories of learning, knowledge and instruction. Novak used concept mapping and V diagrams as tools for facilitating learning, understanding and knowledge creation.

Incremental software development has been described in numerous publications, for example [1], [6], [13].

Beck [1] introduced a new approach to software development, called eXtreme Programming (XP). XP is based on incremental development, oral communication, continuous testing and several other practices. Pair programming is one of XP practices.

Several researchers have studied knowledge contained in a program. According to [2], a programmer understands a program through construction of a mental model that consists of successive knowledge domains. Fisher et al. [3], proposed a support for the incremental development based on a specific knowledge model, where seeding, evolution and reseeding are the three stages of knowledge capture and transformation. Henninger [4] recognized that software development is a process involving various knowledge resources, which keep changing during the process. Robillard [10] identified two types of knowledge: topical and episodic. Topical knowledge refers to the meaning of words and episodic knowledge consists of people’s experience with knowledge.

The role of evolution in software lifecycle was described in [9]. The similarity between program comprehension and human learning was also mentioned in this paper.

#### 5. Conclusions and Future Work

In this paper, we studied the process of incremental program development from the perspective of parallel construction of both the program and the programmers’ knowledge. We have demonstrated an analogy between software development process and constructivist learning by a case study. The case study is based on the analysis of the dialog in a programming pair. We identified four fundamental programming and cognitive processes that lead to growth of both the program and the corresponding knowledge.

Future work includes additional studies of the pair programmer dialogs during the development of larger programs. The studies will help us to better understand the structure of the programming knowledge as reflected in programmer dialog. This may lead towards a new generation of documentation systems that would be better capture the program knowledge that is necessary for program evolution.

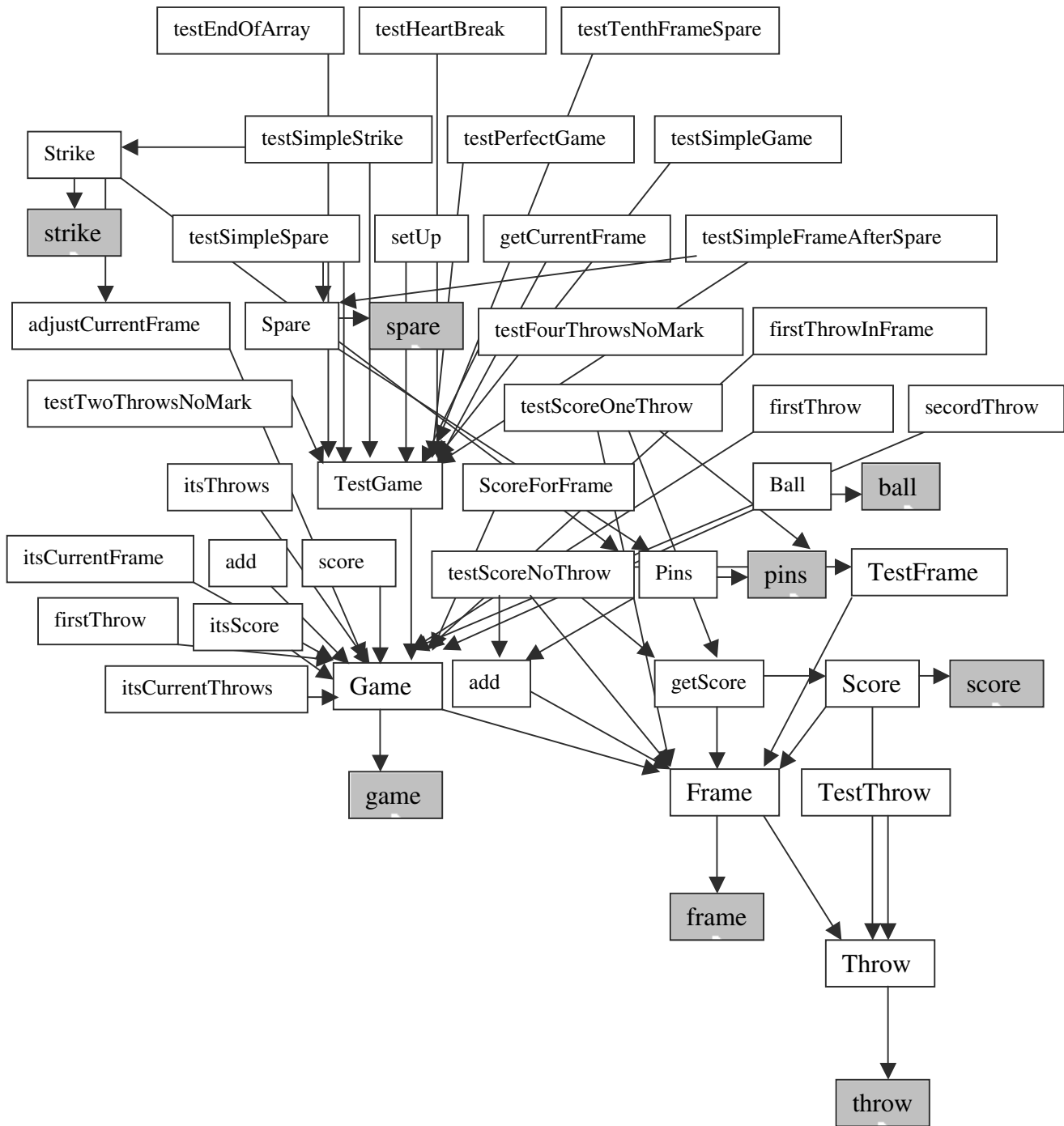
#### Acknowledgements

The authors would like to acknowledge Jay Rajnovich and his help with writing this paper. We also thank the

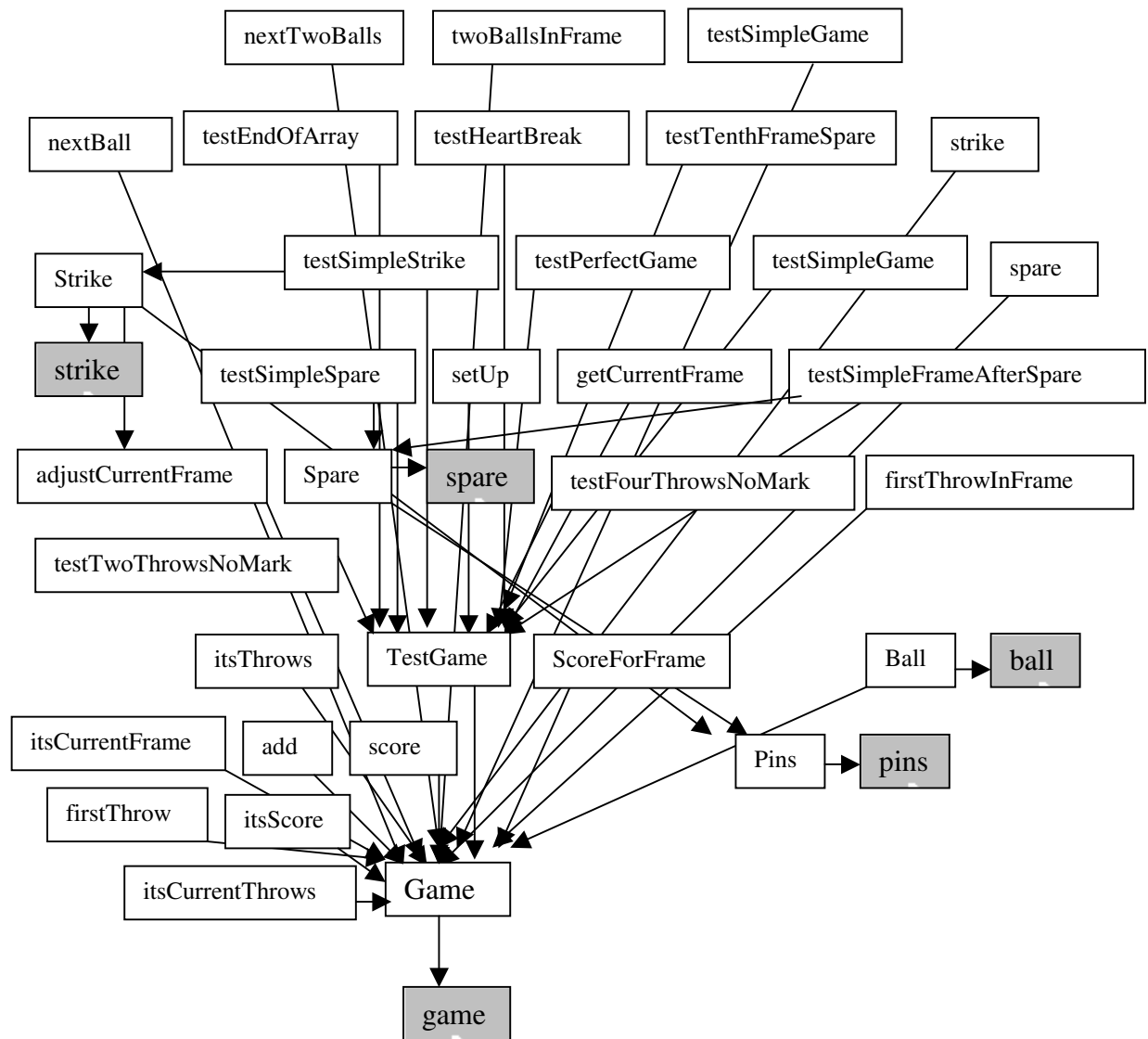
anonymous ICCI'03 reviewers for their comments that significantly improved this paper.

## References

- [1] Beck, K., *Extreme programming explained*, Addison-Wesley, Massachusetts, 2000.
- [2] Brook, R., "Towards a Theory of Comprehension of Computer Programs", *International Journal of Man-Machine Studies*, 1983 (Vol. 18), pp. 86-98.
- [3] Fischer, G., McCall, R., Ostwald, J. Reeves, B., and Shipman, F., "Seeding, Evolutionary Growth and Reseeding: Supporting the Incremental Development of Design Environments", *Proceeding of the Conference on Computer-Human Interaction (CHI'94)*, Boston, MA, 1994, pp. 292-298.
- [4] Henninger, S., "Tools Supporting the Creation and Evolution of Software Development Knowledge", *Proceeding of the International Conference on Automated Software Engineering (ASE'97)*, 1997, pp. 46-53.
- [5] Kozaczynski, W. and Wilde, N., "On the Re-Engineering of Transaction Systems", *J. Software Maintenance*, Vol. 4, 1992, pp.143-162.
- [6] Martin, R.C., *Agile Software Development, Principles, Patterns, and Practices*, Addison Wesley, Massachusetts, 2002.
- [7] Novak, J.D., *Learning, Creating, and Using Knowledge*, Lawrence Erlbaum Associates, Mahwah, NJ, 1998.
- [8] Piaget, J. (1954), *The construction of reality in the child*, Basic Books, New York, 1959.
- [9] Rajlich, V.T., Bennett, K.H., "The staged model of the software lifecycle", *IEEE Computer*, July 2000, pp.66-71.
- [10] Robillard, P. N., "The Role of Knowledge in Software Development", *Communications of ACM*, January 1999 (Vol 42, No. 1), pp. 87-92.
- [11] Rugaber, S., Stephen B. Ornburn, Richard J. LeBlanc, Jr., "Recognizing Design Decisions in Programs", *IEEE Software*, January/February 1990 (Vol. 7, No. 1), pp. 46-54 .
- [12] von Glasersfeld, E., *Radical Constructivism*, The Falmer Press, London, 1995.
- [13] Williams, L., Kessler, R., Cuningham, W., and Jeffries, R., "Strengthening the Case for Pair-Programming", *IEEE Software*, July/August 2000, pp. 19-25.



Appendix 1. Design decisions in the middle of the development



**Appendix 2. Design decisions after class Frame was abandoned**