# Maintenance-Oriented Design and Development: A Case Study

**José Pablo Zagal,** *Virtualia S.A.*
**Raúl Santelices Ahués,** *Blue Planet Software*
**Miguel Nussbaum Voehl,** *Pontificia Universidad Católica de Chile*

**The authors use a case study to compare traditional and maintenance-oriented processes. Accepting the importance of maintenance, and focusing the development process toward it, produces software projects that are successful, on time, and on budget.**

General consensus says that maintenance efforts are the most time- and resource-consuming part of the entire software development process.[1–3] However, the importance of software maintenance is greater now than ever before due to software's increased complexity, size, and lifespan.[4] The way we think of maintenance is the source of our neglect. The software development process basically covers three phases: design, implementation, and maintenance (see Figure 1a).[5,6] By default, maintenance is regarded as the last and least rewarding step of software development, with most of the effort focused on the design and implementation. Thus, software engineering tends to concentrate on a small part of the software life cycle.[7]

What happens when we shift our perspective and embed the implementation stage inside the maintenance step? Software development becomes an initial base design followed by maintenance because the initial design stage now considers maintenance as the next step (see Figure 1b). The base design specifies the design schemes and how they will handle quality assurance and requirement changes.[8] This focus ensures that the software to be designed is maintainable, because once the base design is finished, maintenance starts, even if there is nothing implemented yet.

When we place maintenance at the same level as design and implementation, the entire software development process focuses on the area that demands the most time, money, and effort.[3] This focus allows not only the development of better-quality software but also software that is more adaptable to subsequent changes. The software is easy to maintain simply because of the way it was designed and developed. This is especially important when we consider that maintainability is obviously related to the skills of the maintenance team, the tools available, the process's maturity, and so on.[9] Considering software maintenance before it is even implemented is akin to asking who will use my software and how five years after I develop it. In other words, maintenance begins where development begins.[10]

The case study project we present here is an excellent opportunity to see if this kind of approach reduces costs, shortens devel-


0740-7459/02/$17.00 © 2002 IEEE

opment time, and increases software stability. Our objective is to compare and weigh this paradigm against the traditional method of software development in the video game domain.

## Starting the project

Between 1996 and 1999, a multidisciplinary team from the Schools of Engineering and Psychology at the Pontificia Universidad Católica de Chile undertook a project for developing handheld video games with educational content[11] that used maintenance-oriented design and development. The project had an experimental focus and was initially based on Nintendo's handheld GameBoy device. Seven different games with up to eight distinct content sets were developed with maintenance in mind.

From the software development viewpoint, this project posed a series of challenges:

- Because the project itself was experimental, our initial specifications were highly unstable. In fact, we derived most of them in tandem with the project as we analyzed our results. This was because we had to derive the specifications for the user interface and game playability from the users (children) and the specifications for the experiment as well as the educational contents from the experimenters (psychologists and educators).
- The hardware platform for which we had to develop the software was unknown to us and not definitive. It also differed from traditional software development environments in its lack of tools, the nature of its applications, and the programming languages it used.
- Because the software was to be used experimentally in a wide variety of scenarios, it was critical that it be easily modifiable to rapidly adapt to a variety of different specifications and situations.
- Software stability was also an issue because failures could seriously affect the credibility of our results as well as the games' classroom usage. The software's embedded nature (stored in flash memory) also prevented on-the-fly corrections or modifications.

Most of these challenges are probably quite typical of most software development projects. What is essentially different in this case is not how we solved these problems, but how we viewed the software development process. Our different perspective minimized the impact that these problems would have in later stages.

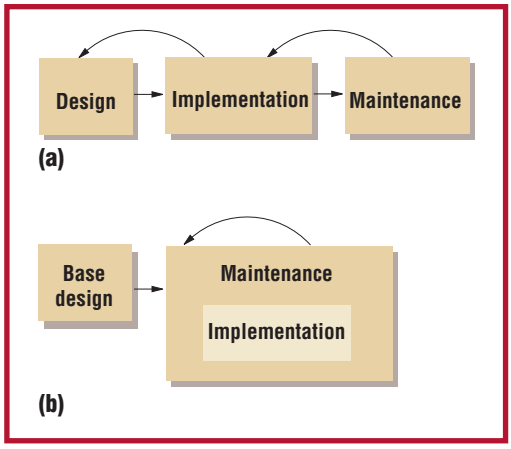


**Figure 1. The (a) traditional approach to software development and (b) a shift in that perspective.**

The four challenges the project faced concerned changes to the software. Accounting for the fact that maintenance is a process of change management,[12] it made sense to consider the software's development from a maintenance viewpoint from the beginning.

## The project's evolution

Initially, the project concentrated on developing a prototype game called Selva, which would serve as an initial experiment for playability and resource stress and provide immediate feedback to the development team. This prototype's construction followed the traditional approach to software development. We use this experience in this article as the control case, whose results we will compare with our proposed maintenance-oriented method.

Once we completed this prototype game, we had enough knowledge of the requirements and development issues involved so as to start the formal production of all the games needed (eventually seven games on different topics). Because we would have to carry out a maintenance process later, the part of the team in charge of the games' design and implementation analyzed the areas that would most commonly need maintaining:

- Modifications to the game levels (increase challenge, update artwork, accommodate technological limitations)
- Modifications in each game's dynamics (make games more fun)
- Change interface (improve usability and children's comprehension of game)

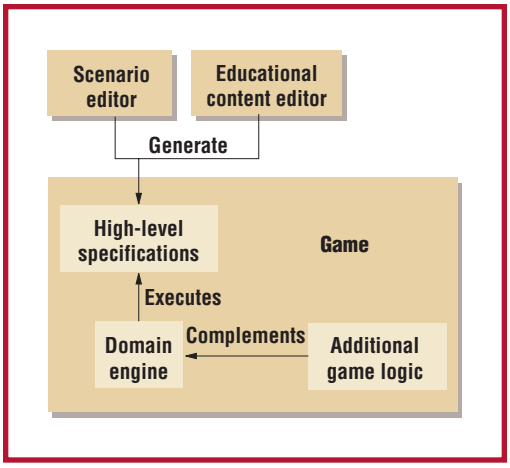


**Figure 2. The game architecture.**

- Modifications to each game's educational schema (change educational contents and adjust game's self-regulation mechanism, which reacts to user skill)

Because it was clearly inefficient to sit down and work out a stable design for each separate game, we decided to delimit the domain of the games by establishing a base of common features and properties. We could thus develop a specific set of tools for each domain as well as a game engine to support them. The purpose of these tools was to allow better maintenance by providing an automated and standardized specification modification process. We designed these tools to allow fast and efficient maintenance of software that wasn't completely designed yet and to let each specific game developer easily test his or her changes directly on the engine. By working with high-level specifications and covering a significant part of the final applications, these tools went one step beyond maintenance to support graphic design and implementation tasks.

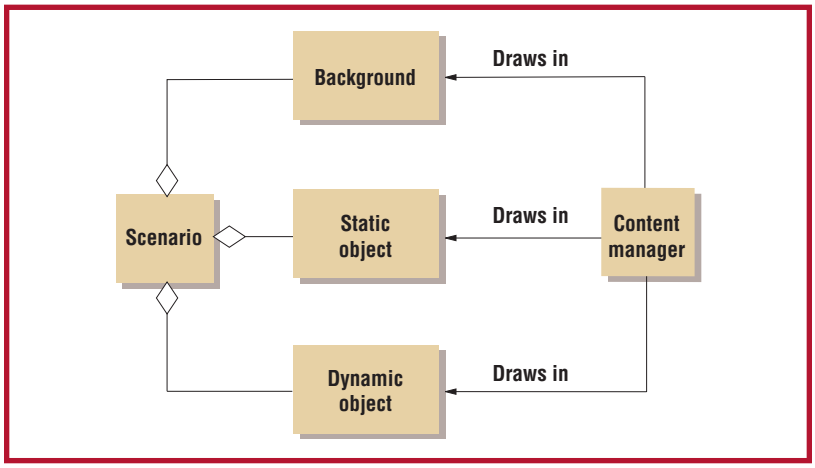


**Figure 3. The structure of the software domain engine.**

We decided to implement 2D, four-way scrolling games, populated with objects sharing a set of standard properties and common behaviors. We created another experimental case (a second version of Selva) that used our maintenance-oriented approach, which we respecified using the tools we already developed. Figure 2 shows the general game architecture that we defined.

## The tools

A game is made up of a sequence of scenarios, each one containing a decorative background, an infrastructure of static objects (such as floor, walls, and statues), and a set of dynamic objects (such as protagonists and enemies). A content manager decides when and what text and graphics to show using any of the scenario constituents. Under this scheme, a particular game adds its own specific logic, mainly by extending the funcionality of the objects shown. Figure 3 shows the structure of the software domain engine that executes the high-level specifications from the editors in runtime (explained further in the next section).

### Scenario editor and validator tool

The first three maintenance areas (game levels, game dynamics, and interface modifications) are related in that they deal with the specification and modifications that redefine different aspects of usability, playability, and entertainment. We considered these areas when we developed the scenario editor and validator tool under a standardized scheme all game designs could share. We developed the scenario editor and validator tool to generate more automated code and data than is generated in the traditional approach. This tool allows the specification of the standardized virtual game world (or scenario) in which the game takes place, including the background and the objects with which the protagonist can interact (such as the floor, boxes, and enemies).

Figure 4 shows how the designer specifies the properties that a given visual object has within the game. In Figure 4a, a palette lets us choose a graphic object (bitmap) from those that are available. These objects' attributes are defined with the properties editor (see Figure 4b)—for instance, the objects' position in the scenario, the way the

game's protagonist and antagonists will interact with the object (collision properties), and other scenario-specific attributes. When time- or event-driven dynamic objects exist, the designer uses the same properties editor to specify aspects such as movement trajectories, and simple behavior rules.

The scenario editor and validator tool also allows the automatic management and validation of system parameters such as video RAM usage (amount of tiles and sprites available) and ROM placement (automatic paging). For example, it can indicate the area with the greatest amount of tiles used. We can also modify the hardware-dependent system parameters from within the tool. This lets the tool produce game data for different hardware platforms in a highly automated fashion.

All these features help the tool generate scenario specifications that a common software engine can run compatibly with all the project's different game designs. This same tool, from a maintenance perspective, lets the designer adjust all kinds of game parameters such as artwork modification and object properties. This quality of the scenario editor—integrating implementation and maintenance in a common tool and process that provide deep automation—is the result of looking at maintenance from the get-go. The only drawback in this case is the loss of flexibility in software design, due to the generative approach taken.

### Educational content editor tool

We designed the educational content editor to manage the educational contents included in each game, which is the remaining maintenance area to be addressed. This tool and its scheme standardize and automate the educational aspects of all games, improving general maintenance and allowing for modifications of each game's specific educational schema. For example, the designer could specify general educational parameters such as how many different levels of educational content (levels of difficulty and different activities) to use and the exercises' maximal parameters. The exercise editor determines each exercise's actual contents; the editor allows for the definition of any particular characteristic pertaining to a certain exercise.

Additionally, the tool permits generation of reports for each game's contents, detailing all the design parameters. In essence, the tool allows for the specification of the design information that is relevant to each game's educational contents as well as each exercise's particulars.

This same tool, now considered from the maintenance perspective, lets us modify all the parameters and elements pertaining to the educational contents and exercises of each game. The tool focuses on the whole maintenance phase instead of just the implementation stage like a standard tool does, and it has the same functionality and report generation abilities. The standardized educational scheme allows an automated and more stable content generation process for all possible games within the domain.

We developed this tool figuring that its users would be mostly educators and psychologists with no particular experience in software development or this project's special nature. Experience shows that, for them, maintaining and developing was essentially the same—that is, they perceived the entire development process as iterative maintenance.

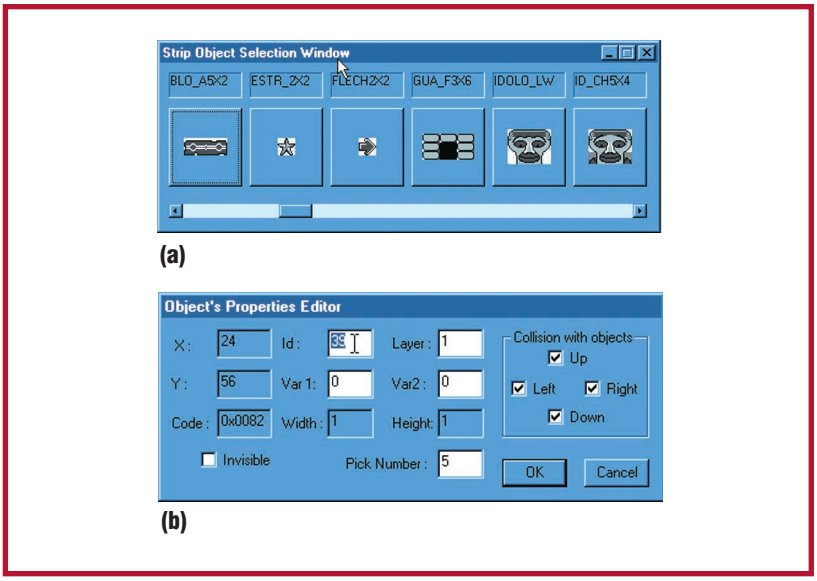


**Figure 4. (a) Object selection palette; (b) object properties editor.**

## Comparative analysis

In any video game process, there are two kinds of design. The first relates to the conceptual design, which produces a semiformal high-level specification of the game[13]—that is, a written or graphic script describing its characters, scenarios, and so on. This is the input for the software design itself, which consists of creating a software model for the target platform.

Tables 1 and 2 show the respective phases

**Table 1**

**Standard process tasks by project phase (traditional approach)**

| Process phase | Common tasks |
|---|---|
| Design | Conceptual design: story, characters, special objectives, interactions, educational content's presentation system<br>Scenario design specified in drawing tool<br>Design of custom educational contents |
| Implementation | Game dynamics; testing and tuning<br>Scenario data calculation and validation; testing<br>Educational content's presentation logic; testing<br>Educational content's data testing<br>Additional game logic; testing |
| Design maintenance | Conceptual design changes (major redesign decisions)<br>Scenario design modifications in drawing tool<br>Changes to educational content or addition of new set |
| Implementation maintenance | Game dynamics retuning<br>Modified scenario data calculation and validation; testing<br>Educational content's presentation logic modification; testing<br>Additional game logic modification; testing<br>Enhancement or refactoring of standard implementation tasks |

**Table 2**

**Maintenance-oriented process tasks by project phase**

| Process phase | Common tasks |
|---|---|
| Common base design (for all games) | Creation of scenario structure and game dynamics scheme<br>Creation of educational contents and configurable presentation scheme<br>Tool design |
| Specific base design | Conceptual design under common scheme restrictions: story, characters, special objects, objectives, interactions, configution of educational content's presentation system configuration<br>Design of overall scenario structure and game dynamics in Scenario Editor; automatic error-free data validation and generation<br>Educational content's presentation configuration in Contents Editor<br>Initial educational contents specified in Contents Editor |
| Design maintenance | Conceptual design changes, under common scheme<br>Editing of scenario's static structure and game dynamics edition in Scenario Editor; automatic error-free validation and generation of new data<br>Editing of educational content's presentation configuration in Contents Editor<br>Editing of educational contents in Contents Editor |
| Implementation maintenance | Tuning of game dynamics specification<br>Scenario data playability testing<br>Educational content's configuration tuning<br>Calculation of educational contents' data effectiveness<br>Additional game logic enhancement and changes; testing |

for both cases in our project, the control and the experiment. The tables also list the kinds of tasks that each phase includes.

The design and implementation tasks are related to the static structure of a scenario (walls, floors, and background), the game's dynamics (object movements, animations, and interactions), the educational content's presentation scheme configuration (the rules for choosing exercises from the content database), and the contents themselves (a pair of numbers to add and the alternatives to choose). They all must be repeatedly tuned and changed for experimental and debugging reasons. These tasks were easier to fulfill in the maintenance-oriented process, because the tool automated them. This created a safer maintenance job because the automatically generated code was stable. Only manually created data and logic were still susceptible to errors (after reaching tool and library stability), which, in this case, meant less code was susceptible to errors than in the control Selva (tools generated most of the code and data). Also, the existence of a common structure in the games made it easier to localize a game's changing points when fixing an error. Additionally, timelines became predictable, process control improved, and thus the project stayed on budget. However, the loss of flexibility imposed by the automation added restrictions to the games that made finding a successful final design difficult for some of them.

We used these new tool-based games for our experiment with the children. The experiment created the need for intensive testing, which produced special maintenance needs not present in the standard case—the maintenance approach's benefits became more important than ever. These extra maintenance tasks produced phases that consisted of

- comprehensive bug-finding sessions by game testers (before the children's experiment), reported to the development team for bug fixing and eventual redesign
- playability experimentation feedback, sent to the development team for maintenance updating of the software
- evaluation of educational content based on teachers' feedback and children's knowledge evaluations (to evolve contents into more effective exercises and configurations)

## Metrics comparison

Table 3 shows effort in person-hours for the different tasks involved in the life cycles of the standard Selva (control case) and the maintenance-oriented Selva (experimental case). First, we identify the costs of building a complete Selva scenario in each case. In the standard Selva, we used a commonly available drawing tool to build the static objects infrastructure and the dynamic objects of the scenario from scratch, with almost the same facilities provided by the specific editing tool used in the new Selva. This explains the similarity in costs. However, the hand-coded dynamics and the almost hand-made postprocessing of the scenario data in the standard Selva involved the most significant cost difference from the maintenance-oriented Selva, as Table 3 shows, because the scenario editor automated this task.

Analyzing the standard scenario modification, we see that the design change's cost is generally minimal, but the compiler-level data calculation, validation, and debugging make a difference again. Without the availability of the scenario editor, the team had to redo an average of half the postprocessing, given the fact that the scenario was extremely wide and short and that the mean change was located in the middle of it. Considering that each game had at least three scenarios and that implementation of the common scenario scheme design and the tool construction debugging consumed about 100 and 160 person-hours of work, the maintenance-oriented approach was more economically feasible even for the first releases of the games.

The educational dimension is another major area of maintenance. Table 4 shows the costs of educational-content-related tasks for each case. In the standard approach (single-game production), the educational content and its presentation were a single design task, specified in a text editor or spreadsheet and then included in the game along with the specific code needed. In the maintenance-oriented approach, the standard scheme separated the presentation configuration process from the content specification, the latter being formatted according to the configuration defined in the former. Although this diminished flexibility, the whole content design for a game took less time in this case due to the Content Editor's availability, and the automated implementation avoided the introduction of logic-related bugs in the process. Also, as we see in Table 4, the modification tasks are highly optimized in the maintenance-oriented case.

**Table 3**

**Scenario development costs for the control and experimental case**

| Task (person-hours) | Standard Selva (person-hours) | Maintenance-oriented Selva |
|---|---|---|
| Scenario specification from scratch | 10 | 8 |
| Complete scenario coding, calculation, validation, and debugging | 80 | 0 (automated) |
| Scenario design modification (tuning) | 1 | 1 |
| Calculation, validation, and debugging of modified scenario | 40 | 0 (automated) |

**Table 4**

**Contents development task costs for the control and experimental case**

| Task (person-hours) | Standard Selva (person-hours) | Maintenance-oriented Selva |
|---|---|---|
| Design of educational content and presentation | 10 | 8 |
| Implementation of content and presentation | 40 | 1 (automated) |
| Presentation or content modification | 6 | 1 (implementation automated) |

For a stable version, the cost of developing the scheme design and library implementation, along with the tool construction, meant a total of approximately 160 person-hours. Again, this investment was fully rewarded considering each game's extensibility and maintenance-driven life cycle and that there were at least four versions of each game with different contents and configurations.

The maintenance-oriented approach we adopted from the beginning of the process contributed to a definitive gain in lower and standardized maintenance costs. Process control and software stability were strengthened, because code was modified only in defined places and mainte-

**We believe this paradigm is still well suited to software development scenarios in which there is a large degree of instability in the initial specifications.**

nance did not introduce new code. Unfortunately, though, variability in game design decreased. For this approach to be successful, the software domain must be carefully circumscribed and defined so that you can use the same tools throughout the software cycle.

We believe this paradigm is still well suited to software development scenarios in which there is a large degree of instability in the initial specifications, as well as those in which the hardware platform is being modified or is unknown in the software development phase. It is also appropriate for those cases in which the software's operating conditions are uncertain and lots of changes and modifications will have to be made. It is particularly useful when these modifications are to be handled by people with little technical experience regarding the software in question.

Our approach might seem similar to the evolutionary model of software development (from the viewpoint of rapid prototyping).[5] However, we are less ambitious: we simply propose the development of maintainable software.


## Acknowledgments

**The Chilean National Science Foundation (FONDECYT) grant number 1000520 partially funded this work.**


## References


1. J.R. McKee, "Maintenance as a Function of Design," *Proc. Am. Federation of Information Processing Societies Nat'l Computer Conf.*, AFIPS, Reston, Va., 1984, pp. 187–193.
2. T. Guimaraes, "Managing Application Program Maintenance Expenditures," *Comm. ACM*, vol. 26, no. 10, Oct. 1983, pp. 739–746.
3. M. Hanna, "Maintenance Burden Begging for a Remedy," *Datamation*, Apr. 1993, pp. 53–63.
4. N. Gross et al., "Software Hell: Special Report," *Business Week*, 6 Dec. 1999, pp. 38–44.
5. I. Sommerville, *Software Engineering*, 5th ed., Addison-Wesley, Reading, Mass., 1996.
6. R.S. Pressman, *Software Engineering: A Practitioner's Approach*, 4th ed., McGraw-Hill, New York, 1997.
7. I.D. Baxter and C.W. Pidgeon, "Software Change through Design Maintenance," *Proc. Int'l Conf. Software Maintenance*, IEEE CS Press, Los Alamitos, Calif., 1997, pp. 250–259.
8. J. Han, "Designing for Increased Software Maintainability," *Proc. Int'l Conf. Software Maintenance*, IEEE CS Press, Los Alamitos, Calif., 1997, pp. 278–286.
9. M. Ramage and K. Bennett, "Maintaining Maintainability," *Proc. Int'l Conf. Software Maintenance*, IEEE CS Press, Los Alamitos, Calif., 1998.
10. T. Pigosky, *Practical Software Maintenance*, Wiley Computer Publishing, New York, 1997.
11. "Learning by Playing," Schools of Eng. and Psychology, Catholic Univ. Chile, 2000; www.ing.puc.cl/sugoi.
12. N.G. Schneidewind, "Characteristics of Software Maintenance," *Encyclopedia of Computer Science*, Macmillan Reference, London, 1999.
13. A. Rollings and D. Morris, *Game Architecture and Design*, Coriolis Technology Press, Phoenix, Ariz., 2000.



For more information on this or any other computing topic, please visit our Digital Library at http://computer.org/publications/dlib.


## About the Authors


**José Pablo Zagal** is director of online community development at Virtualia S.A. His research interests include human–computer interactions, computer game design and development, computer-assisted education, and online communities. He received a BS in engineering and an MSc in computer science from the Catholic University of Chile. Contact him at Kennedy 5757, Office 1502, Las Condes, Santiago, Chile; jp@virtualia.cl.



**Raúl Santelices Ahués** is a software engineer at Blue Planet Software. His research interests include virtual reality, computer game development, and real-time artificial intelligence. He received a BS in engineering and an MSc in computer science from the Catholic University of Chile. Contact him at Vasco de Gama 4702, Dept. 64, Las Condes, Santiago, Chile; rasantel@ieee.org.



**Miguel Nussbaum Voehl** is a full professor of computer science in the School of Engineering at the Catholic University of Chile. His research interests include knowledge engineering, artificial intelligence in human sciences, and technology applications in education. He received his EE from the Catholic University of Chile, an MSc in computer science from Georgia Institute of Technology in Atlanta, and a PhD in informatics from the ETHZ, Switzerland. Contact him at Pontificia Universidad Catolica de Chile, Escuela de Ingenieria, Departamento Ciencia de la Computacion, Casilla 306, Santiago 22, Chile; mn@ing.puc.cl.