
Software Modeling in Essence

Master's Thesis submitted to the
Faculty of Informatics of the *Università della Svizzera Italiana*
in partial fulfillment of the requirements for the degree of
Master of Science in Informatics
Major in Software Design

presented by
Remo Lemma

under the supervision of
Prof. Dr. Michele Lanza
co-supervised by
Fernando Olivero

June 2012

I certify that except where due acknowledgement has been given, the work presented in this thesis is that of the author alone; the work has not been submitted previously, in whole or in part, to qualify for any other academic award; and the content of the thesis is the result of work which has been carried out since the official commencement date of the approved research program.

Remo Lemma
Lugano, 22 June 2012

To Valentina

Everything you can imagine is real.

Pablo Picasso (1881-1973)

Abstract

The design of any object-oriented system starts with (software) modeling, a process during which one identifies the system's core concepts and how they relate to each other. Among the various ways of performing this activity, there are lightweight means, such as pen & paper/whiteboard or CRC cards, and on the other end of the spectrum we have complex full-fledged UML editors. The advantages of the first ones are their immediacy, speed and playful nature, i.e., their informal essence supports the creative modeling process, but their output is difficult to store, process, and maintain. The latter ones, while making amend for these problems, are tedious and often not trivial to use, thus hindering both creativity and productivity. We believe that there is a possible middle ground which maximizes the good of both worlds, while keeping the bad at bay. This middle ground is best treaded using the emerging technology of touch-based tablet computers.

In this thesis we present a novel modeling methodology based on a minimalistic set of elements: the essence of object-oriented software modeling. We also describe a simple, yet powerful, visual metaphor based on a matrix, which has been designed to explicitly leverage the modeling activity. Finally, we illustrate `CEL`, an iPad application which implements our modeling methodology. This tool has been designed to support the process of rapidly and interactively creating, manipulating and storing language independent software models. These models can then be used to generate the corresponding skeleton code in any language of choice.

Acknowledgements

The first person I would like to thank is my advisor, Prof. Dr. Michele Lanza. I have to thank you for your advices and your brilliant ideas. Since I walked in for the first time into the Programming Fundamentals I class (five years ago), your door has always been open to discuss about any project or idea. This has been of immeasurable help for me throughout my whole studies.

I have also to thank Fernando Olivero for co-advising this work: Your pure object-oriented thinking and your revolutionary ideas have been a great inspiration for me.

I also want to thank Alberto Bacchelli for all the interesting discussions we had, for discovering an interesting tool every day, and for reviewing the drafts of this thesis. Thanks also to Dr. Marco D'Ambros for his constructive ideas and the feedback given during the meetings.

I also want to express my gratitude to the REVEAL group as a whole: You are an amazing team of excellent scientists and working with you has been an honor and a great opportunity.

I want to thank all my friends, who supported me inside and outside of the university. Friendship is one of the most treasured gifts and I hope to be important to you as you are to me. I will not mention you one by one because I am afraid of forgetting someone. Nevertheless, I owe a special mention to Patrick Zulian, Teseo Schneider and Luca Ponzanelli. You guys are extremely talented computer scientists and working with you made me grow as a person and as a professional. I know you have a glorious future in front of you and I also truly believe that in a couple of years we will consider the projects we are developing together right now as the start of a common success. I hope you are learning from me as much as I am doing from you.

Before going into more personal acknowledgments I want to thank all the people who participated in the evaluation of this work.

I have to thank my parents for supporting me throughout my entire life: I would not have completed my studies without knowing that I can always rely on you. I also have to say thank you to my sister for being there in the important moments of my life and to my little nephew Daniel: You are able to transform my gray days into shining moments.

I also would like to acknowledge the entire Marafioti family: You welcomed me as one of the family from the very first moment and always tried to support me whenever I had a difficult time.

Last, but surely not least, I want to thank with all my heart my girlfriend Valentina. You always believed in me and supported all my choices, even the more questionable ones. I honestly do not know how you can bear with me all the time, but thank you for not giving up on me. I love you.

Contents

Contents	xi
List of Figures	xiii
List of Tables	xv
1 Introduction	1
1.1 Contributions	3
1.2 Structure of the Document	3
2 Related Work	5
2.1 Modeling Techniques	5
2.1.1 Lightweight Modeling: Pen & Paper / Whiteboard / CRC Cards	5
2.1.2 Heavyweight Modeling: UML Editors	7
2.2 Context-related Approaches	9
2.2.1 IDEs Enhancements & New Metaphors	9
2.2.2 Avant-garde Programming	10
2.3 Summary	11
3 Philosophy	13
3.1 Modeling Philosophy	13
3.1.1 Entities	13
3.1.2 Relationships	14
3.1.3 Reflections	15
3.2 The Matrix Approach	16
3.3 Final Considerations	17
4 CEL	19
4.1 CEL in a Nutshell	19
4.2 Modeling Framework	20
4.2.1 Design Concepts	21
4.2.2 Action Observation	21
4.2.3 Extensibility	21
4.3 Model Visualization	22
4.3.1 Entities	22
4.3.2 Relationships	23

4.4	The User Interface	26
4.4.1	Enter the Matrix	26
4.4.2	Semantic Zoom	27
4.4.3	Third Zoom Degree: Parameters	32
4.4.4	The Projects Index	33
4.5	Interaction	34
4.5.1	Design Guidelines	34
4.5.2	Gesture-based Interaction	35
4.5.3	Keyboard-based Interaction	41
4.5.4	Menu-based Interaction	42
4.5.5	Toolbar-based Interaction	43
4.6	Summary	43
5	Evaluation	45
5.1	Experimental Design	45
5.1.1	Research Questions	45
5.1.2	Data Collection	46
5.1.3	Object Systems	47
5.1.4	Tasks & Treatments	48
5.2	Experiment Operation	49
5.3	Discussion & Results	50
5.3.1	Participants Characterization	50
5.3.2	Completion Time Analysis	50
5.3.3	Research Questions Debriefing	51
5.3.4	User Feedback	53
5.4	Threats to Validity	54
5.4.1	Internal Validity	54
5.4.2	External Validity	55
5.5	Reflections	56
6	Conclusions	57
6.1	Retrospective	57
6.2	Future Work	58
6.3	Closing the Circle	60
A	Handout	61
B	Experimental Data	67
C	Practical C_{EL}	69
C.1	Project Import	69
C.2	Model Export	69
C.2.1	Project Export	70
C.2.2	Image Export	70
C.2.3	Code Export	71
C.3	Reflections	74
D	Language Templates	75

Figures

2.1	Enhanced lightweight modeling techniques	6
2.2	Traditional UML editors	7
2.3	Touch-based UML editors	8
2.4	Mainstream IDEs, based on the WIMP metaphor	9
2.5	IDEs based on alternative metaphors	9
2.6	IDEs based on emerging technologies	10
4.1	CEL modeled in CEL itself on the iPad	20
4.2	Entities in CEL	22
4.3	Selected entities in CEL	22
4.4	CEL depicting relationships (the default visualization)	23
4.5	CEL depicting relationships for a selected entity	24
4.6	Channel size depends on the number of passing relationships	25
4.7	The graph constructed on top of the grid	25
4.8	The different matrix visualizations	26
4.9	Birds-Eye semantic level	27
4.10	Overview semantic level	28
4.11	Edit semantic level	29
4.12	Details semantic level	30
4.13	Internals semantic level	31
4.14	The internals of a method: parameters' visualization	32
4.15	The projects index view	33
4.16	Move of a selection of entities	37
4.17	Line gesture uses	38
4.18	Shape gesture usage	40
4.19	Keyboard-based interaction view	41
4.20	Menus of CEL	42
4.21	Toolbar of the matrix view	43
C.1	Model Export UI	70
C.2	Language templates import	72

Tables

4.1	Summary of the gestures used in CEL	35
5.1	The list of questions that guided the semi-structured interview	47
5.2	Treatments of the evaluation	49
5.3	Completion time for each task for each participant	50
B.1	Part I of the answers to the screening questionnaire: general information	67
B.2	Part II of the answers to the screening questionnaire: experience levels	67
B.3	Part I of the answers to the debriefing questionnaire: experience review (1 to 3)	68
B.4	Part II of the answers to the debriefing questionnaire: experience review (4 to 6)	68
B.5	Part III of the answers to the debriefing questionnaire: approach feedback	68
B.6	Part IV of the answers to the debriefing questionnaire: task difficulty	68

Chapter 1

Introduction

Any system is the concrete representation of an idea: a complex, abstract, concept. While creating such a concrete artifact, trying to manage the entire complexity of the process immediately at the beginning leads to failure. To tackle this issue, and correctly address the creation of any system, one should employ modeling techniques.

A (software) model is a simplified representation of the reality, which only includes information strictly serving the purpose at hand [Fav04]. (Software) Modeling is the activity during which such artifacts are produced. This process is mostly performed in the early stages of (software) system design. While modeling, one tries to identify the core concepts of the system to be built, sketching also their relations and reasoning about their behavior. For instance, in Object-Oriented Programming (OOP) these core concepts are mapped to classes, which in turn, through methods and variables (*i.e.*, fields), describe how objects will behave at runtime.

Among the different methodologies that can be used to model, we can identify two classes of approaches, the informal and the formal one, each one with advantages, limitations and drawbacks.

Informal modeling techniques include pen & paper, whiteboard and specific approaches such as CRC Cards [Wil95, BS97]. These methodologies do not set any limit around the creative process that modeling really is. However, the artifacts produced with such techniques are often volatile (*i.e.*, not persistent) and also difficult to share, process and maintain. The produced models have often a reduced communication power and induce a problem of knowledge sharing: It is difficult for someone not present during the modeling stage to understand the model and to comprehend the design decisions that were taken. Furthermore, the maintenance problem is a crucial drawback of these techniques, since design drift is often unavoidable in real-world systems [MNS01] and it is a root cause of software aging [Par94].

Formal modeling, instead, is expressed using diagrammatic visual notations. The main representer of this category is UML, the unified modeling language [FS00], which can be considered a *de facto* industry standard. UML is so widely used and taught, that sometimes it is even used as a synonym of modeling. This type of modeling is mostly performed through the aid of computer applications, such as UML editors (*e.g.*, ArgoUML and UML Lab¹). The formalisms introduced by these visual languages and the uniformity imposed by the use of digital means (*i.e.*, the look and feel of the models is always very similar and the notation used is consistent and commonly accepted) make the produced models easy to process. Moreover, the applications used to create such formal diagrams support knowledge sharing and reduce maintenance problems.

¹<http://argouml.tigris.org/> and <http://www.uml-lab.com/>

However, we claim that these applications, and particularly UML editors, inhibit the creative nature of modeling: One spends more energy in reproducing the right steps for creating correct, understandable and visually clean diagrams rather than in exploring possible design alternatives.

Furthermore, the tight integration of modeling applications (*e.g.*, UML Lab) with integrated development environments (IDEs), can lead to further problems during the modeling phase. We state that these two phases should be clearly separated, because their tight coupling can lead to models which are too much polluted by implementation-specific elements (*e.g.*, programming language specific features, *etc.*). Moreover, this tight integration can cause, especially in semi-professional or academic environments (*i.e.*, the environments where there is no strictly regulated development and design process), the tendency to progressively abandon the modeling activity, pouring all the energy and the effort directly into the programming phase. Thus, newcomers would not be able to take advantage of the abstraction layer provided by software models to become familiar with the system.

We believe a more flexible, tool-based, solution is needed, one that empowers developers to rapidly craft persistent, easily maintainable and processable software models. This solution should take advantage of the middle ground which exists between the informal and the formal approach, maximizing their benefits, limiting the possible drawbacks and taking a step back from the bad practice of coupling the modeling and the implementation phases.

The emergence of touch-based tablet computers (such as the iPad²) creates a perfect opportunity to fully harness this middle ground. Touch-based devices stimulate a playful freeform environment similar to pen & paper/whiteboard, giving the possibility to fully express creativity and stimulating the rapid exploration of design alternatives. The portability of such devices enables modeling in any setting and environment and their computational power supports processing, storage, and maintenance of models.

However, we do not advocate using tablet computers as mobile whiteboards, or porting UML editors to tablet PCs. These solutions would suffer again from limitations and/or drawbacks very similar to the ones we discussed above, and would not really place themselves in the middle ground that we are targeting.

In this thesis, we present CEL, an iPad application, which incorporates what we believe being the essence of object-oriented software modeling. Our approach is based on the minimal number of elements which we claim are necessary to model a software system. CEL has been constructed keeping simplicity and intuitiveness as general design guidelines, yet providing all the means to create and manipulate complete and useful object-oriented software models. CEL has been developed on top of a minimal matrix-based visual language and uses the available touch and gesture-based interaction methodologies to design software models, which can be stored, exported, and used to generate skeleton source code. We know from literature that software maintenance absorbs up to 90% of total software costs [ZSG79, Erl00], of which 60% is used to understand the system at hand [Cor89]. The lack of user understanding is in fact a key problem [LS81]. We help in mitigating this issue by furnishing easily maintainable models, which can be analyzed and understood by exploiting the visual and highly interactive user interface (UI) of CEL.

²<http://www.apple.com/ipad/>

1.1 Contributions

The contributions of this thesis can be summarized as follows:

- An innovative modeling approach based on the essence of the object-oriented paradigm, which allows the creation of language agnostic models.
- A new visual metaphor designed to directly leverage the modeling activity.
- A standalone application framework that can be employed to create modeling applications based on our philosophy on any platform of choice.
- An interactive modeling application, which exploits touch-based tablet computers and that is explicitly designed to support model comprehension.

1.2 Structure of the Document

The rest of the document is structured in different chapters, described in the following.

- **Chapter 2** presents the work related to our research. We focus particularly on the differences between these approaches and our research. Moreover, the chapter presents also the work which influenced the design and the development of CEL.
- **Chapter 3** illustrates the fundamentals of our research. It illustrates the modeling philosophy we have created, our vision about software. Is also presents the user interface metaphor we devised to fully support our modeling technique.
- **Chapter 4** describes CEL, a highly interactive iPad application, which allows users to create, manipulate, store and export object-oriented software models. We present its user interface, the matrix metaphor and the interaction methodologies, justifying our choices and explaining how they can be exploited.
- **Chapter 5** illustrates a qualitative evaluation of CEL, describing the methodology we used to gather information, describing the possible threats to validity of the experiment and discussing the results and the feedback we obtained.
- **Chapter 6** reviews and concludes this thesis. We take a step back discussing what we have done so far, present our ideas in the context of the future development of CEL , and describe the future research directions that we are willing to explore.
- **Appendix A** illustrates the handout used during the evaluation.
- **Appendix B** describes all the relevant data gathered during the evaluation.
- **Appendix C** presents a practical view of CEL, illustrating features which leverage the created models. We focus on the possibility of exporting skeleton code by exploiting language templates. These templates can be used to describe any programming language of choice.
- **Appendix D** presents the details of the language templates, the approach we use to describe, in any programming language of choice, how models should be exported.

Chapter 2

Related Work

Modeling, in an abstract fashion, the software system currently being designed, is a fundamental activity of the whole development process. The research on software modeling, and methodologies to support this activity, has been active since many years, and we believe will continue in the next decades. In this chapter we present the work related to our research. We first describe the different methodologies which are adopted nowadays to perform software modeling and, afterwards, we present the context-related work which influenced our research.

2.1 Modeling Techniques

The different modeling techniques in use today range from informal, usually lightweight, means, to heavyweight, formal, methodologies. In the subsequent sections we present each modeling methodology separately, emphasizing its advantages and limitations, highlighting also the differences with our approach.

2.1.1 Lightweight Modeling: Pen & Paper / Whiteboard / CRC Cards

When confronted with design problems it is common practice for developers to sketch potential solutions using lightweight, informal, means such as pen & paper / whiteboard [CVDK07]. CRC cards, in the same way, keep the complexity of a design at minimum, using small cards on which a modeler writes the essential information about an object-oriented class. These techniques fully support the highly creative endeavor required in software modeling [Pet09] and favor the exploration of new design solutions. The possibility to freely sketch parts of a design, easily changing and redrawing parts, without having to follow strict rules or standards is a key advantages of these approaches. Sketching plays an important role in the design process, allowing to move from an abstract to a more concrete view of a system [Goe91]. Sketches are a natural extension of design thinking, being flexible and adapting well to the changes of focus of the users [Pet09]. These lightweight modeling means guarantee also the possibility to decide how many details should be specified for each part of the model. They also allow to choose if, and in which measure, to take advantage of other approaches, such as formal notations, favoring on-the-fly modifications and creation of new notations [DH07]. The low technical skills and external support required by these practices favor an immediate usage of the methodology and do not involve a steep learning curve.

Unfortunately all these advantages are counteracted by drawbacks burdening the artifacts (*e.g.*, sketches or CRC cards) obtained using these techniques. The static nature of this output, combined with the impossibility of manipulating the design, without erasing parts or drawing/writing over them is a major problem. In fact, software designers often have the necessity and the desire to manipulate the design at hand in a more complex fashion [DH07]. These artifacts are not digitalized, which makes them easy to lose and almost impossible to store. A direct consequence of this problem is the difficulty to share these outputs. Moreover, the possibility to comprehend the diagrams or the CRC cards is highly dependent on the drawing/writing skills of the user. The absence of a formal, consistent, visual aspects is a further obstacle, which also inhibits the communication power of such models. The difficult maintainability of these models and the impossibility for a person, absent during the modeling stage, to reconstruct the design decisions, and the subsequent difficulty in understanding the final model, are other fundamental drawbacks. In fact, these lightweight techniques are valuable not particularly for the produced artifacts, but more for the insights gained during their usage [SSR03].

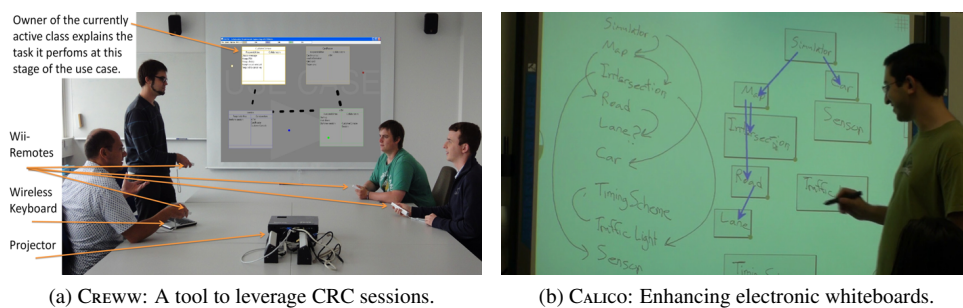


Figure 2.1. Enhanced lightweight modeling techniques

There are different tools that have been built with the objective of limiting these drawbacks. CREWW [BDL11], depicted in Figure 2.1a, alleviates the weaknesses of the low-fi CRC sessions by aiding the process with computer tools, a projector and using Wii-Remotes¹ as input devices. CREWW records the development sessions and helps with the storage and the organization of the generated CRC cards in a digitalized form. The sessions can be replayed to better understand the evolution of the model, and review/revise the decisions that have been taken. This tool tackles the problem of CRC cards digitalization, yet the authors advocate the usage of UML, to formalize the model in the last step of CRC sessions. With our approach we want to propose a new solution using an innovative, simpler, visual language not based on UML.

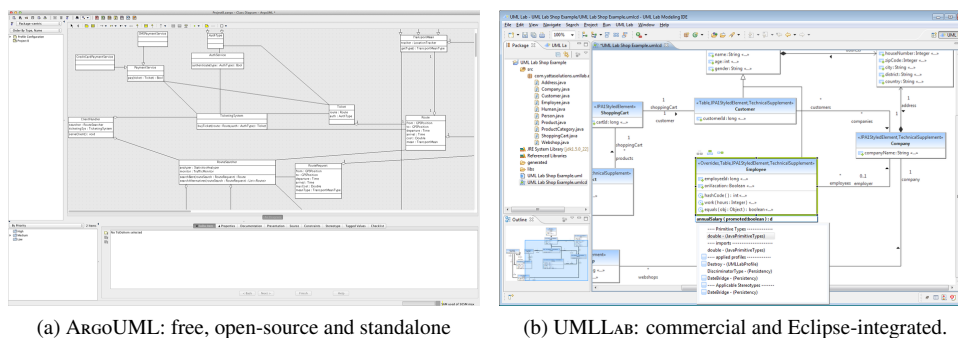
CALICO [MBD⁺10], instead, has been constructed, as shown in Figure 2.1b, to leverage electronic whiteboards. In particular, this application has been designed to preserve all the positive aspects guaranteed by the whiteboard modeling technique, adding the chance to easily switch between multiple sketches, organizing and favoring the reuse of user-defined notations and structures. In our research we do not support the concept of electronic whiteboards, because many of the problems described above yet remain, for example the dependence from drawing/writing skills and the limited communication power of such models. With our work we aim to solve most of these issues and we believe that another approach has to be pursued.

¹<http://www.nintendo.com/wii/what-is-wii/#/controls>

2.1.2 Heavyweight Modeling: UML Editors

UML editors are the computer applications which are used to exploit heavyweight, modeling techniques, mainly represented by formal visual languages. UML, the unified modeling language, is one of these visual notations. It is heavily employed in the industry and is nowadays commonly accepted as a standard in the context of software modeling. UML and UML editors solve many of the drawbacks afflicting lightweight modeling means which we described in Section 2.1.1. First of all, the applications takes care automatically of the visual consistency (*i.e.*, no artistic talent is required to the modeler) of the model. Combining the uniform look & feel of the diagrams with the strict, commonly accepted, rules imposed by the visual language, the understandability of the models is greatly increased. The communication power of these models is remarkable and it is one of the most important reasons which have supported the rise of UML. The diagrams are also easy to store (*i.e.*, they are stored as files on a hard-disk for example) and share, using common technologies such as e-mails, Skype² or any other application which supports file sharing. Many UML editors also support the concept of knowledge sharing and they considerably reduce the maintenance problems, mainly because of the possibility of editing models in non-trivial ways (*i.e.*, more operations than adding and deleting are supported).

However, as described for lightweight modeling methodologies, also the heavyweight ones, including UML editors based modeling, suffer from drawbacks, which again hinder the advantages we presented above. In particular, the strict rules and formalisms imposed by formal visual languages hamper considerably the creativity of modelers. They are more focused on creating correct, clean and commonly understandable diagrams, rather than trying out new solutions and evaluating different alternatives. Although these digitalized models are easy to share, very often the available formats (*i.e.*, not image-based such as screenshots, but textual representations of the model such as XML files) are not compatible across different applications. Moreover, the considerable amount of knowledge which has to be mastered to become proficient with UML editors leads to a prohibitive learning curve and makes them tedious to use.



(a) ARGOUML: free, open-source and standalone

(b) UMLLAB: commercial and Eclipse-integrated.

Figure 2.2. Traditional UML editors

Because of the undisputed domination of UML in the industry, these drawbacks have not stopped the spread neither of UML itself nor of UML editors. The number of open-source/free and commercial versions of such products is nowadays considerable. These applications have evolved over time and two main school of thoughts can be identified: standalone tools (*e.g.*, ARGOUML illustrated in Figure 2.2a) and IDE-integrated editors (*e.g.*, UML LAB depicted in Figure 2.2b).

²<http://www.skype.com>

From this point of view, our beliefs follow the same philosophy as the first ones, solely focusing on the modeling process and separating it completely from the programming activity. The latter ones, instead, realize the idea of integrating within IDEs all the tools helpful for software development. Among the different types of tools there are also modeling applications. As we already discussed, this approach might cause the progressive abandonment of the modeling activity and/or the construction of models which are too much influenced by language-specific features.

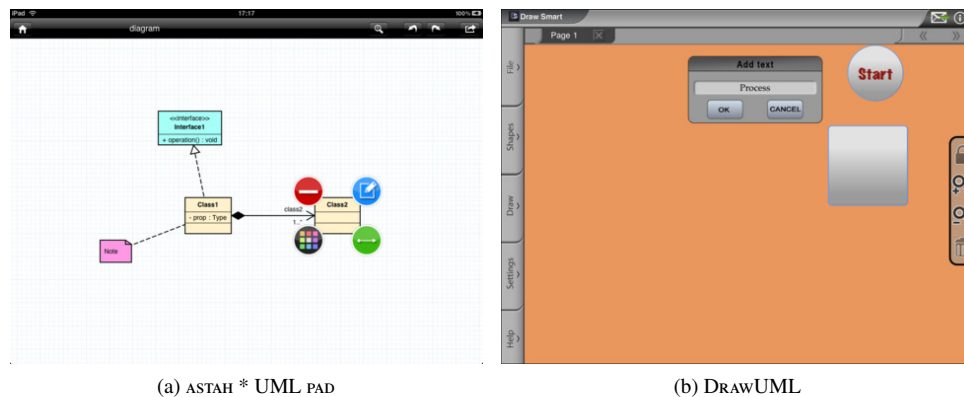


Figure 2.3. Touch-based UML editors

New technologies and devices have been explored also in the context of supporting UML-based modeling. Two different iPad applications, *ASTAH* UML PAD*³, illustrated in Figure 2.3a, and *DRAW UML*⁴, depicted in Figure 2.3b are examples of such interest. The first one provides the user with all the basic elements to sketch a UML class-diagram and the possibility to export it via e-mail as an image or in an XML format readable by the desktop application which is produced by the same company⁵. The latter one, instead, allows to create different types of diagrams, gives the chance to work on different models concurrently and supports the sending of these artifacts via e-mail. Nevertheless, these applications seem not able to compete with full-fledged desktop UML editors and are more a complementary tool to rapidly sketch part of a design rather than standalone applications. They lack features which are considered fundamental (*e.g.*, code export). With CEL we aim to create a totally independent, competitive and standalone product.

Overall, the biggest difference between our ideas and UML editors is the decision of taking a step back from current modeling practices and propose a new alternative to UML-based modeling. We believe that the drawbacks afflicting these approaches can be eliminated by using less rigid, yet formal visual languages, which should prefer creativity and design exploration instead of detailedness.

³<http://astah.net/editions/pad>

⁴<http://itunes.apple.com/us/app/draw-uml-for-ipad/id428468147>

⁵<http://astah.net/editions/uml>

2.2 Context-related Approaches

We present in this section the bleeding edge of research in the context of programming methodologies, which has influenced the design of our user interface metaphor and is related to the development of CEL. We separate work on IDEs and new metaphors from revolutionary approaches applied on touch-based devices.

2.2.1 IDEs Enhancements & New Metaphors

Nowadays, programmers write software in IDEs, which include many tools that enable all the programming-related activities to take place within the same environment.

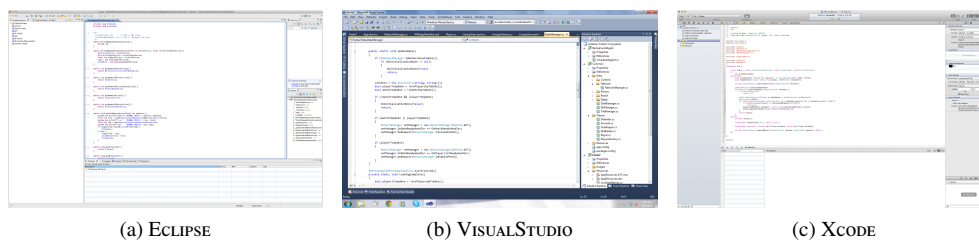


Figure 2.4. Mainstream IDEs, based on the WIMP metaphor

Mainstream IDEs (*e.g.*, ECLIPSE⁶, VISUALSTUDIO⁷, XCODE⁸, all shown in Figure 2.4) are built around the WIMP metaphor, making use of Windows, Icons, Menus, and a Pointing device to visualize and manipulate a textual representation of the program, *i.e.*, the source code. Lately, researchers proposed environments built around alternative metaphors, bringing objects closer to programmers, in order to provide a higher level of abstraction in the user interface.

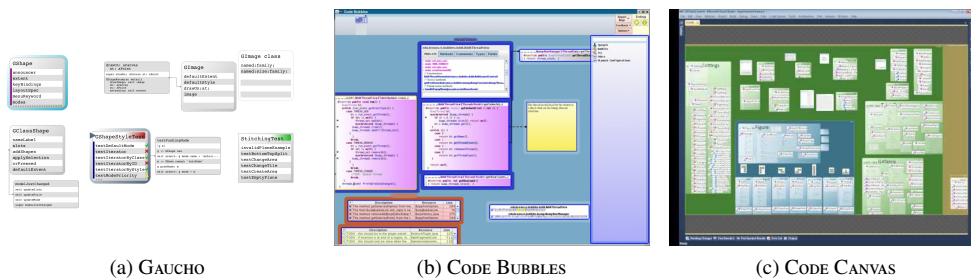


Figure 2.5. IDEs based on alternative metaphors

With GAUCHO [OLL10], depicted in Figure 2.5a, Olivero *et al.* have created a pioneering development environment based on the concept of direct manipulation: The user writes programs by directly manipulating graphical shapes representing objects. The user interface is zoomable and the overall IDE has been kept lightweight and simple by limiting the presence of external tools.

⁶<http://www.eclipse.org>

⁷<http://www.microsoft.com/visualstudio/> from Microsoft: <http://www.microsoft.com>

⁸<https://developer.apple.com/xcode/> from Apple: <http://www.apple.com>

GAUCHO has been proven helpful in enhancing and facilitating program understanding, signaling the possibility that alternative, visual, IDEs may be superior to the current mainstream IDEs as program comprehension aids [OLDR11]. In designing CEL, we followed the same minimalistic style present in GAUCHO and adopted direct manipulation as the main interaction paradigm.

The same results, in terms of program comprehension facilitation, have been achieved by CODE BUBBLES [BZR⁺10], illustrated in Figure 2.5b. This IDE depicts code fragments as lightweight, fully editable, bubbles, that can be grouped together, thus creating concurrently visible working sets. Furthermore, this tool significantly reduced navigation interactions compared to traditional IDEs, revealing also possible cognitive benefits of the bubble-metaphor approach. This metaphor has been created to support scalability and the possibility to concurrently work with large numbers of fragments. The IDE provides a 2-D continuous pannable horizontal virtual workspace, in which the user is freely able to place and organize the (groups of) bubbles. In CEL we went further, providing an infinite 2-D space. CODE BUBBLES adopts a free-layout, implementing sophisticated methodologies to automatically (re-)arrange bubbles in case of overlaps or other layout problems. We developed, instead, a matrix-based metaphor, which constrains the user in placing elements inside a grid, but automatically avoids the layout problems intrinsic to free layouts (*e.g.*, overlaps). We believe that the multiple working sets which can be created with CODE BUBBLES can be easily replicated in our matrix-based approach by exploiting the position given to the elements inside the grid.

Generally speaking, we use abstraction in CEL to present a concise view of the entities of the model under construction: A simple named rectangle following a color scheme. This is comparable to how the mentioned tools abstract away from treating source code as mere text.

The infinite 2-D surface we provide in CEL, is based on the canvas metaphor, which has been harnessed also in CODE CANVAS [DR10], as shown in Figure 2.5c. This tool proposes another innovative approach, in which the user is provided with an infinite zoomable interface where editable content and project-related information coexist, allowing information needs to be met through visualizations. This design aims to leverage spatial memory, reduce disorientation and support numerous analyses, by making it easy to synthesize information.

2.2.2 Avant-garde Programming

Standard PCs have been the main digital platform on which the design and the development of software products have been executed since the birth of software engineering. Research on using other platforms, such as tablet computers, in software development is going on since several years [EB04]. The improvement of new technologies (*e.g.*, touch-based input, innovative game controllers, *etc.*) and their increasing importance, have renewed the interest of researching these alternative means to support the software development process.

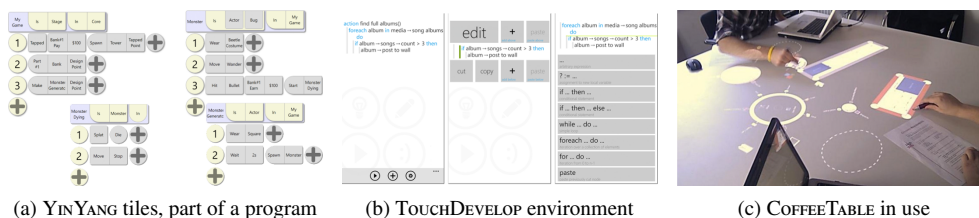


Figure 2.6. IDEs based on emerging technologies

New possibilities to program everywhere derive from the proliferation of small/medium-sized touch-based devices (*e.g.*, smartphones and tablets). In fact, many recent research projects targeted the means (*e.g.*, environments, languages, *etc.*) that are needed to program, and which new platforms currently lack. For example, YIN YANG [MCD11] is a visual programming language explicitly designed to allow game software development on tablet computers. As shown in Figure 2.6a, the programming model of this (visual) language is based on tiles and behavior constructs, simplifying the structure of a program, to best fit on the targeted devices. To be competitive with capable programming languages, it supports the definition and reuse of new abstractions. The main aim of the authors was to create an environment and a language to overcome the challenges and the problems which may hinder the programming experience on touch-based tablet computers.

Supporting programming on touch-based devices, especially smartphones, has been investigated also by Tillmann *et al.* with TOUCHDEVELOP [TMdHF11]. This application, designed for Windows Phone, currently targets students and hobbyists more than professional developers, favoring the design of applications for personalizing the phone. TOUCHDEVELOP consists of a development environment, illustrated in part in Figure 2.6b, based on a simplified, scripting language, which has built-in primitives to access sensors and, will allow, in future versions, thanks to abstraction to share data between clients and the cloud hiding the tedious details.

Different technologies have been explored by Hardy *et al.* with COFFEE TABLE, an IDE built around a shared interactive desk, as depicted in Figure 2.6c. This environment supports the traditional development methodologies whilst facilitating the creation of a shared vision of the project, a key aspect of successful design. COFFEE TABLE involves the combination of different elements such as physical hardware (a table, a short throw projector, infrared pens and Wii-Remotes), client-side software that developers use to write code, and server-side software which manages the different clients rendering also the interactive visualization.

All these projects have largely adopted emerging technologies, experimenting solutions which may create a solid ground for further research, and realizing tools which support and may leverage the programming activity, and software development in general. In our research we aim to do the same, yet focusing exclusively on the modeling phase.

2.3 Summary

In the previous sections we have described the work directly relevant for our research and other projects which influence, at least in part, our choices. We analyzed the state of the art in the context of software modeling: A multitude of approaches that can be classified in a spectrum which ranges from lightweight, informal, methodologies to heavyweight, formal, means. The middle ground between these two approaches is currently not exploited. With our research we aim to combine the advantages of both worlds and mitigate their drawbacks. The possibility of harnessing new technologies and devices such as touch-based tablet computers is valuable, because they naturally mimic the creative environment offered by lightweight modeling techniques. These new digital technologies offer also computing power and storage to create, process, share and maintain the artifacts, some of the key advantages proposed by heavyweight modeling methodologies. The community encourages the creation of applications meant to support the software development process on these devices [Che11]. Although different researches have been done in this context, there is plenty of work to be done and solutions to be explored.

In Chapter 3 we present our approach, which consists of a novel modeling technique and of a user interface metaphor which has been explicitly designed to support the modeling activity.

Chapter 3

Philosophy

In our research we have addressed the design of a new modeling technique and the planning of an innovative visual metaphor to be used to best support the software modeling activity. In this chapter we present the solutions we have created.

3.1 Modeling Philosophy

Our modeling philosophy aims to fully support the creative production of processable, maintainable, sharable and persistent software models. The minimalistic set of elements provided to the user is composed by entities to represent concepts and relationships to link them.

3.1.1 Entities

To create the set of elements which the user exploits to model software systems we started by distilling from the mainstream paradigm which we support (*i.e.*, the object-oriented programming (OOP) paradigm), the core constructs that we believe are necessary to build a system. Classes are the building blocks of the object-oriented paradigm [Rie96], and they describe typologies of objects present in the real-world system. Classes and objects (instantiations of classes), contain state (*i.e.*, fields), which describe the encapsulated data, and behavior (*i.e.*, methods), which outline the actions executed at runtime. Methods also contain parameters, yet these are used only to refine the structure of the method entity (*i.e.*, parameters are not essential constructs). In some programming languages, (*e.g.*, Self [SMU95]), the methods and fields entities are fused together into one single element (*i.e.*, slots). We decided to keep these two pieces of information separate, because especially at the modeling level, the possibility of having an easy way to understand the dynamics involving a class, and being able to rapidly identify the typologies of data owned, is of valuable importance. Furthermore, this distinction can help to identify, already in the early stages of the design, potential and critical flaws (*e.g.*, futile hierarchy, data classes [LMD10]).

In modern object-oriented languages this minimalistic definition of OOP has been extended with the introduction of other elements. For example, packages/namespaces elements contain classes which are tightly related. We claim that these elements are artificial constructs that aim at structuring the code and are not essential to the object-oriented paradigm. Thus, we have not included packages/namespaces among the elements we provide in our approach.

We also decided to not discriminate among different typologies of classes (*e.g.*, abstract classes, interfaces or pure virtual classes, *etc.*). Although this distinction is useful in the implementation phase, we do not deem it relevant for the modeling stage, where the designer wants to deal only with few basic concepts.

We support typing information in an optional fashion. Types are not essential for the pure modeling activity, as in most cases the meaningful denomination of an entity is enough. Nevertheless, in particular situations (*e.g.*, to declare special data structures like maps or lists) they can become valuable. Thus, the necessity of specifying types should be left up to the user. We do not force the specification of types, but we guarantee the possibility of enriching with typing information the entities which need more refinement. Types are available for fields, parameters and methods (*i.e.*, the return type).

We currently do not support types in our modeling approach. They are useful in particular situations (*e.g.*, to declare special data structures like maps, lists, *etc.*) to avoid the pollution of the name, but in the normal case the information provided with a meaningful denomination is enough.

Because we want to produce language-agnostic models, the absence of unnecessary entities is the wisest solution. These elements, and many others (*e.g.*, visibility modifiers, *etc.*) create unwarranted problems to the users (*e.g.*, which kind of class should be used), and are often directly related to specific programming languages, which would pollute our approach. Overall, we believe that combining these three simple concepts (*i.e.*, classes, methods and fields), a user can model any, trivial or complex, software system of choice. Having one single element for each kind of entity (*i.e.*, objects definition, behavior and state) allows users to focus on the true essence of designing a software system without having to specify tedious details.

3.1.2 Relationships

When modeling a software system, one needs to sketch collaboration among entities. In other modeling languages (*e.g.*, UML), relationships are particularly tedious to specify, because there are numerous typologies to be remembered and used in the correct way. Moreover, a lot of additional details (*e.g.*, cardinality, labels, *etc.*), are supported and should be specified. We have reduced all this complexity to two simple relationship types: inheritance and generic.

Inheritance Relationship

In object-oriented programming, inheritance is a technique which favors code reuse by making derived classes inherit the state and behavior of the parent(s). Inheritance is also tightly related to subtyping: The "is-a" relationship with a base class or more (*e.g.*, a Dog is an Animal), which usually involves inheriting the parent's state and behavior. Some programming languages allow to distinguish between a code-reuse oriented inheritance (*e.g.*, private inheritance in C++) and subtyping. Inheritance, especially when used in conjunction with subtyping, enables the creation of a hierarchy which can give a better understanding of the software system and its structural organization. In our approach we use inheritance relationship as a pure subtyping mechanism. We deem code reuse not to be a priority during the modeling phase, whereas defining an hierarchy of the classes to be valuable to refine system entities. As we claim that one single type of class is sufficient, there is also no need to further differentiate among various kinds of inheritance relationship as done, for example, in UML (*i.e.*, realization and generalization). We also fully support multiple inheritance, a technique which is often not available in modern object-oriented programming languages, mainly due to the implementation difficulties and drawbacks it introduces [Sin95].

Nevertheless, multiple inheritance is more expressive than simple single inheritance, allowing the user to show some of the system dynamics in a more comprehensible and detailed way (*e.g.*, a PhD student is a university employee and a student at the same time). While modeling, there are no implementation or efficiency drawbacks, thus adopting multiple inheritance, at least in early design phases, might improve the expressiveness of the created artifact.

Generic Relationship

In UML different types of relationships (*i.e.*, association, aggregation and composition) are used to denote coupling between elements. Even though the differences among the various connections might indicate how the software system has to be implemented (and how strong the relationships are), we argue that this distinction might unnecessarily complicate the model in the early stages of the design phase. In our research we decided to represent all types of connections, except inheritance, by means of a generic relationship. Constructing models using only one single generic type of relationship, instead of many specific ones, leaves more freedom in the implementation phase. This reduces the chances of having code early drifting apart from the model because, for example, of language-specific features which preclude the exact replication of some very detailed design decisions. The generic relationship should be used to not only indicate a strong link between entities (*e.g.*, a class invoking a method of another class), but also to signal a conceptual bound (*e.g.*, a class is related to another one, but the designer has not yet decided how).

3.1.3 Reflections

Our approach supports only few basic concepts, which we believe represent the true essence of object-oriented modeling and are at foundations of OOP. As we underlined in Section 2.1, all the existing modeling techniques have numerous advantages but also relevant drawbacks. We do not advocate using any of these methodologies, trying to solve their problems. First of all such a solution is hardly applicable, because many drawbacks are tightly related to the technique itself. Secondly, many of these modeling means are commonly accepted and changing them would lead to problems which may be even bigger than the ones created by a novel approach. For these reasons we have decided to devise a complete new modeling philosophy.

Our research tackles the problem from a novel perspective. We do not want to force users to create complete and fully detailed models, but we target the initial stages of modeling, where the focus is more on discovering and describing the essence of a system. We believe that the few elements we provide are sufficient to create accurate, meaningful models, yet keeping them lightweight and easy to understand. Because users do not have to delve into time-consuming details, they have the possibility to explore more alternatives, favoring creativity over complexity and detailedness.

From a technical point of view, the aforementioned methodology has been designed without relying on specific technologies. Although we advocate using touch-based tablet computers could improve the modeling process, our philosophy may be applied on different platforms without any risk or problem.

From a pragmatic point of view, our approach is minimalistic and may be criticized for tackling the problems of mainstream modeling techniques in an extreme manner. We know that our methodology is extremely simple. But that is exactly what we were aiming for, simplicity is not a synonym for functional poverty. Our modeling technique requires very little additional knowledge, yet with the few key concepts they are provided with, users can achieve non-trivial results. Citing White, "Design is not the abundance of simplicity, it is the absence of complexity" [Whi02].

3.2 The Matrix Approach

Software modeling is a highly creative process. Such processes require experimentation, the exploration of various alternatives and continuous feedback regarding the progresses made. User interfaces should support the non-linear and iterative nature of creative processes [TM02]. We believe that an ad-hoc user interface metaphor best fits these requirements. Moreover, in the context of software modeling, highly usable, custom, user interfaces can leverage model comprehension and enhance productivity.

The layout of entities is a fundamental element of any user interface. It is of major importance especially in computer applications which use particular visual languages to depict their content. These software systems are usually based on the concept of free layout. Among these applications there are also UML editors and modern, innovative, IDEs such as the ones presented in Section 2.2.1. Free layouts are powerful: They allow users to have full control over the user interface and place elements wherever they like. This characteristic maximizes flexibility and favors the creation of custom visualizations, which can be adapted in real-time to best-fit the purpose at hand. However, free layouts do not come without a price and their flexibility can also be considered a tremendous weakness. Without positioning and size constraints, the user has to personally care about the layout, or the application has to integrate a layout engine. Manual (re)organization is tedious and frustrating, resulting also in loss of time, which cannot be used to actually design the system. On the other hand, the process of automatically organizing the content of a user interface is a non-trivial operation. This process usually involves the usage of advanced algorithms such as corner stitching [Ous84] or other ad-hoc heuristics [BZR⁺10]. These methodologies output astonishing results, avoiding overlaps and cleverly calculating the new positions. Nevertheless, their results are greedily based on algorithmic goals, such as space or displacement minimization. These metrics, although providing optimal solutions, do not take any human factor nor the true meaning represented by the elements into consideration. Thus, there is the chance of having an optimal layout from an algorithmic point of view, which is suboptimal for the designer. In fact, the layouts obtained adopting automatic elements reorganization can be less than satisfactory, limiting comprehension [SM09].

Overall, our modeling approach aims to favor exploratory design with the final goal of rapidly producing easily comprehensible artifacts. We believe that these goals would be too penalized by the adoption of a free layout methodology. Thus, we have crafted an innovative user interface metaphor based on a matrix. The matrix subdivides a theoretically infinite 2-D space into cells, separated by a grid. This approach limits the user's freedom, because entities can be placed only inside cells. Despite that, there is still the possibility to autonomously decide in which part of the matrix to place the entities, which is, in our vision, a sufficient degree of freedom. The intrinsic order and the uniform visual appearance of models depicted using a matrix-based user interface can be important factors to enhance understandability and the overall communication power of these artifacts. Moreover, in a matrix, the addition of new elements do not directly influence the entities which have been already placed, avoiding annoying and costly repositioning. The extreme simplicity and the automatic organization provided by this matrix metaphor are properties which spare to the user a lot of time which can be employed to pursue different design alternatives. In a matrix-based approach, proximity is automatically exploited, giving the chance to group (conceptually) related elements together by simply placing them in adjacent cells. The same design principle can be used, manually, also in free layouts. In both methodologies, recognizable clusters of elements can be created also by exploiting different principles, such as (color) similarity [LHB03].

The adoption of a matrix-based approach has also a considerable advantage when dealing with relationships. Especially in software modeling, being able to clearly identify how the elements are correlated is of remarkable importance. In free layout based applications, because the placement of entities is done relative to the users's desires, depicting connections in a non-disturbing, easily understandable way is a difficult task. A possible alternative would be to lay out elements based on their relationships, grouping connected elements together. Unfortunately, this solution is even more difficult to realize and does not guarantee satisfactory results. Generally speaking it is hard to find an appropriate representation which does not overcomplicate the entire visualization and does not pollute the entities portrayal (*e.g.*, when the visualization of relationships overlaps with the depiction of some elements). On the contrary, using our matrix-based approach we are able to exploit the grid of the matrix to picture the relationships. The grid is regular and does not overlap with the visualization of the elements, thus we are able to depict relationships in a clean and unobtrusive way.

3.3 Final Considerations

Our matrix metaphor is suited to support few entities and, as we have demonstrated, it is also suited to depict and handle relationships. We do not claim that the matrix metaphor is the only one suitable for our modeling methodology. But, compared to other modeling methodologies, having designed the technique and the user interface metaphor in parallel is a considerable advantage. Such an approach granted us the chance of handling and solving early different relevant problems (*e.g.*, relationships depiction, entity positioning, *etc.*).

We created, following our modeling philosophy, an application framework. On top of this framework we have put our overall approach into practice, by designing an iPad application named CEL. The main view of the tool, the one used to model software systems, is based on the above mentioned matrix metaphor. In Chapter 4 we describe CEL and analyze the framework. We also present in details the user interface of the iPad application and the available interaction methodologies, mostly based on touch-based input.

Chapter 4

CEL

In this chapter we present CEL, the iPad application which integrates the modeling idea we exposed in Chapter 3. In the subsequent sections, we first rapidly present a general overview of our product. Afterwards, we analyze and discuss different aspects of CEL, such as its user interface and the supported interaction methodologies.

4.1 CEL in a Nutshell

CEL, depicted in Figure 4.1, is an iPad application that provides the essential means to model object-oriented software systems, integrating our vision about software modeling.

CEL tackles the modeling process from a different perspective, being innovative in many aspects, sitting at the intersection among IDEs, conventional modeling tools, and tablet devices. Because CEL wants to isolate as much as possible the modeling activity, it intentionally lacks support for common tasks such as coding, testing, running, and debugging programs.

Our modeling approach has been created on top of a visual language which users can adopt to create and manipulate models composed by interconnected named entities, which in turn can include other entities. The produced models are (programming) language-agnostic and are based on a minimal set of constructs, which we believe are sufficient to model any kind of software system. This approach aims to favor exploratory modeling practices.

The interconnected entities are placed into matrix cells, being our main user interface constructed on a matrix metaphor. The user can act on the matrix, instantiating new entities or relationships, and manipulating the existing ones. CEL allows also the creation of a selection to focus particularly on a group of elements and act on it. To mitigate the problem caused by the small screen size of the iPad, we have implemented the concept of semantic zoom, which allows users to switch rapidly from a global view to a more detailed one.

To exploit completely the touch-based device that we have targeted with our product, almost all interactions are performed using gestures, keeping the keyboard/button-based interactions to the bare minimum (*e.g.*, to name an entity, to activate undo & redo functionalities, *etc.*).

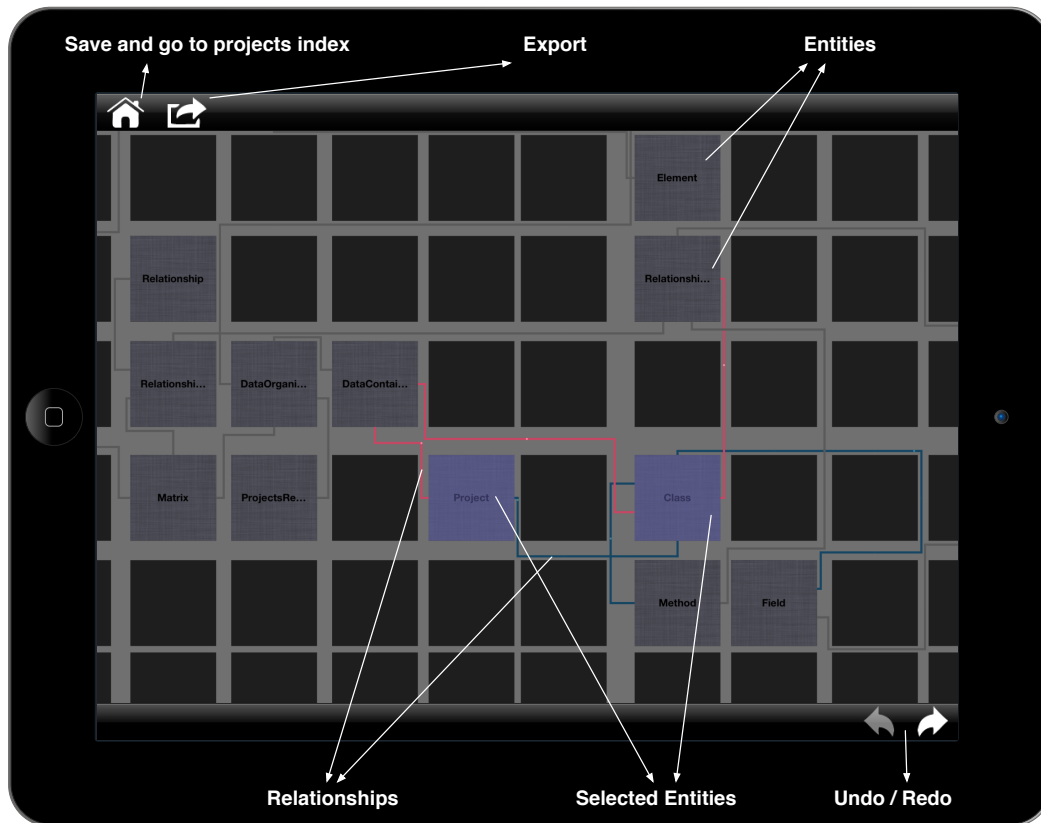


Figure 4.1. CEL modeled in CEL itself on the iPad

4.2 Modeling Framework

Since the early stages of our research, we decided to build CEL as a touch-based tablet application. The iPad is only one of these many devices, which are becoming more and more popular¹. Moreover, as already mentioned in Section 3.1.3, our modeling idea does not rely on any particular technological support. Thus, we have planned to implement our first prototype already keeping in mind the possibility of porting such a product on other devices and/or platforms. We decided to implement the modeling idea itself and the features tightly bound to it (*e.g.*, the export engine, mainly used to produce skeleton code, as explained in Appendix C.2.3), completely separated from any platform/device/view-dependent element.

We produced a standalone, extensible, C++ application framework. We have opted for C++, because it is a programming language which is widely supported on all modern operating systems. Also the most modern and popular Java-based platforms support C++ code, through the use of native SDKs². The possibility of directly reusing the framework allows the employment of our modeling philosophy on other platforms, without having to rewrite source code, thus saving a considerable amount of development time (and costs).

¹According to a research carried out by Google, in 2011 11% of US citizens owned a tablet http://services.google.com/fh/files/blogs/Google_Ipsos_Mobile_Internet_Smartphone_Adoption_Insights_2011.pdf

²<http://developer.android.com/sdk/ndk/> and <http://developer.blackberry.com/native/>

4.2.1 Design Concepts

Besides the entities needed to represent the elements described in Section 3.1.1 and Section 3.1.2, the application framework is built on two main concepts of *Data Organizer* and *Data Container*.

Data organizers are objects that know how to arrange, using specific data structures (*e.g.*, matrices, lists, *etc.*), child entities. The main data organizer that has been currently implemented, and which is at the base of CEL, is the matrix, as required by the user interface metaphor described in Section 3.2.

Data containers are entities that embed other elements. Therefore, a class is a data container as well as the project entity. Each instance of such containers can be assigned any kind of data organizer; the programmer chooses one that fits well for the purpose of the data container. The matrix is not necessarily the best choice for every data representation. For example, we have designed a list-based data organizer which best fits the containment of parameters inside methods. Data organizers have to be chosen carefully analyzing the data contained in a container and, in case of need, a new organizer should be created.

4.2.2 Action Observation

In building our framework we have made extensive usage of the command pattern [GHJV99], which states that an element, a command, encapsulate all the necessary information to perform some action at a later time. In our case the commands (actions) contain the necessary information to change the state of an entity. Each action is composed of a target element, a sender, and a transition of state (from a start value to an end value) of a certain property. These objects can be applied and unapplied, giving the chance to easily switch between different states of an object.

The actions are tightly coupled with our observer system (based on the observer pattern [GHJV99]). This component implements the basic functionalities to query entities and register as observers, in order to receive notifications about certain types of events (represented by actions).

Even though this approach induces some overhead in terms of living objects (as a new action has to be instantiated every time the state of an entity is changed), it provides numerous advantages in terms of control and implementation. For example we built the entire undo/redo system based on an component which registers the history of changes applied to the entire project: The actions are preserved and, thanks to the apply/unapply design of these objects, it is easy to go back and forth in the history. The change data can be collected also for other purposes, such as evolution analysis. Software evolution [Leh80] related research is mainly based on source code. Applying such studies also to software models can help in early detecting design problems. Late design flaws detection is extremely problematic, mainly because of the exponential growth of fixing costs [Boe81]. The analysis of the actions performed on a system can be also valuable to reconstruct the behavior of the user and discover possible interaction problems (*e.g.*, repeated deletion of entities immediately followed by the undo operation may imply a problem afflicting the deletion gesture).

4.2.3 Extensibility

The application framework is extensible: New data organizers, entities (*e.g.*, new class types), and relationships can be added by simply subclassing the appropriate base class(es). The framework offers also the possibility to add new actions, thus allowing the collection of new data for different types of analysis. This extensibility can be used for experimenting and testing new modeling ideas, as well as to implement and use standard components that are adopted in a custom design process (*e.g.*, standards of a company).

4.3 Model Visualization

In `CEL` we use the entities present in the application framework described in Section 4.2, that is in turn constructed on top of the few elements provided with our minimalistic modeling philosophy (see Section 3.1.2 and Section 3.1.1). In this section we present the different visual representation of such entities, explaining also the properties of such representations. We realized very simple visualizations, adding visual effects only when needed to keep visual noise at the bare minimum.

4.3.1 Entities

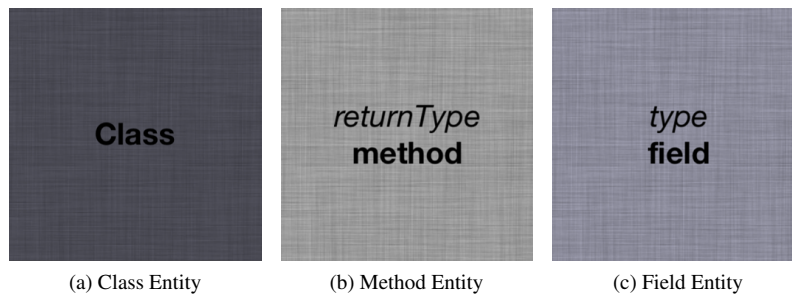


Figure 4.2. Entities in `CEL`.

Since `CEL`, and the philosophy behind it, supports object-oriented software design, `CEL`'s main elements are the building blocks of any object-oriented system: classes. Each cell of the main matrix (*i.e.*, the first matrix which is accessible when opening a project) can be filled with class entities. Methods and fields are contained inside those entities. Parameters are contained inside methods, but they are not organized using a matrix-based approach (consult Section 4.4.3 for the details).

Figure 4.2 depicts how `CEL` visually differentiates the various entities types by assigning each of them a distinct color. Exploiting this visual difference it is possible to quickly recognize the kind of entity being manipulated. From a visual point of view, we decided to keep the representation as simple as possible: A square containing a label (*i.e.*, a name or a signature) and eventually the typing information. We use the italicized version of the font to ease the distinction between the name and the type. We only added a mild pattern background to avoid the visualization from being too flat. Elegant solution can be achieved with an absolute minimum of components [MS94].

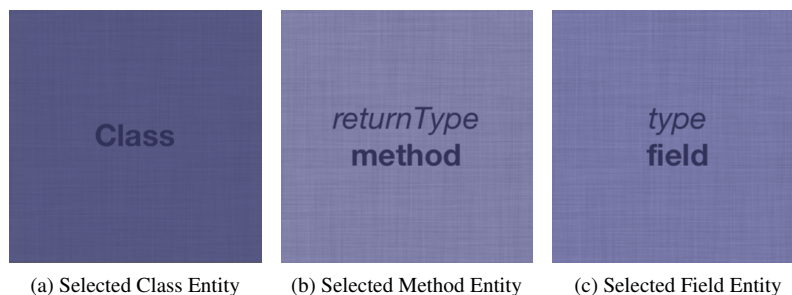


Figure 4.3. Selected entities in `CEL`.

When creating a child element of a class, a field or a method is instantiated depending on the specified signature. We have preferred a Java-like notation over the message passing notation (*i.e.*, like Smalltalk messages). Thus, when adding an entity to a class, CEL checks whether it has parameters or `()` at the end of its signature and, in case, it creates a method, otherwise a field.

The UI of CEL allows the user to select multiple elements. When models grow in size, being able to easily focus only on a portion of the entire model becomes a valuable feature. Figure 4.3 illustrates how the different entities look like when they are selected. The blue overlay mitigates the visual difference among the entities, but allows to exploit color similarity. This principle overcomes proximity from a cognitive point of view [Rus37], and makes the selection appear as a single element.

4.3.2 Relationships

As we already presented in Section 3.2, our matrix-based approach is ideal to contain and represent connections among matrix cells (*i.e.*, entities). The possibility to exploit the grid of the matrix to lay out and display relationships, without polluting the visualization with edges crossing entities is essential to keep the overall user interface ordered and clean. Figure 4.4 shows the default visualization of relationships.

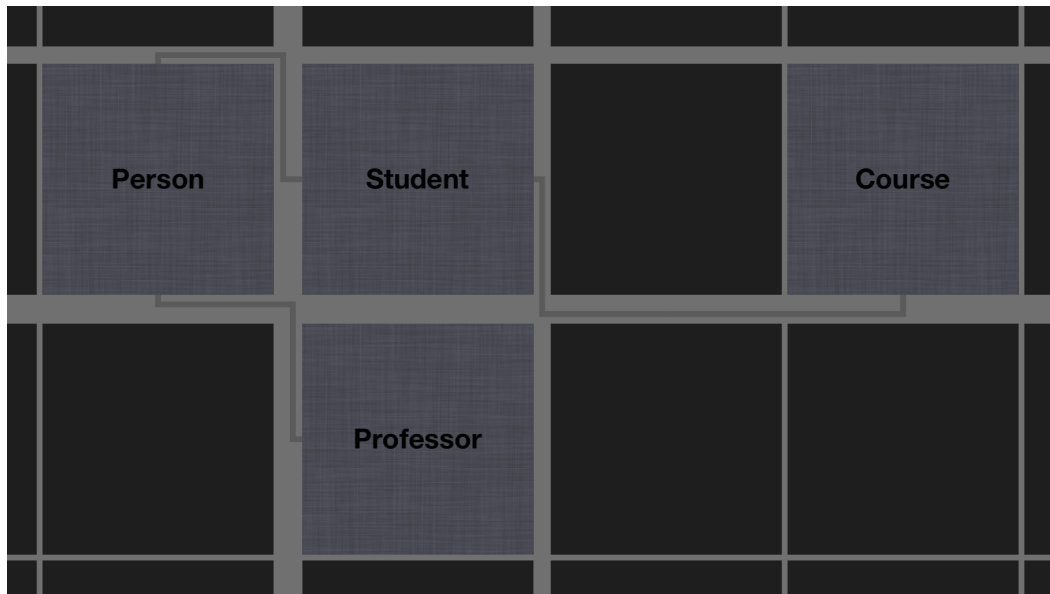


Figure 4.4. CEL depicting relationships (the default visualization)

We opted for keeping relationships always visible, but by default all the types of connections share a common visual aspect (*i.e.*, a polygonal chain colored with a tint similar to the one used for the grid) and it is impossible to distinguish them. Such a uniform visualization is essential for preserving a comprehensible interface. In fact, because we keep relationships always visible, the probability of having numerous connections depicted at the same time is considerably high. Utilizing a common visual aspect is a good trade off between low visual noise and the amount of information a user can extract from such a representation (*i.e.*, it is still possible to see, qualitatively, how many connections there are and which entities are the most referenced ones).

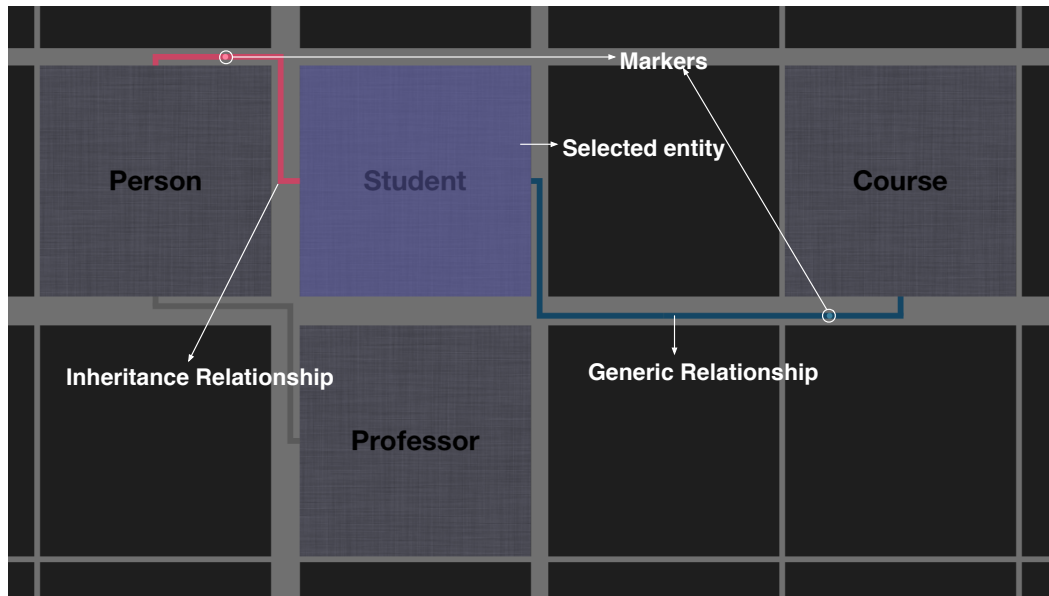


Figure 4.5. CEL depicting relationships for a selected entity

Figure 4.5 depicts how relationships are visualized when they are related to a selected entity. The colors used for the two types of connections are red for inheritance and blue for generic relationships. This solution allows users to progressively discover information depending on the entities in focus.

Similarly, we argue that the direction of a relationship is not fundamental, thus it should become visible only for those relationships belonging to the set of elements currently in focus. To visualize the direction of a connection, we use a little marker, as shown again in Figure 4.5, which moves repeatedly along the polygonal chain from the source entity to the target element.

Initially, the grid delimiting the matrix cells, is of limited size, not able to contain concurrently many, clearly distinguishable, relationships. The grid channels grow according to how many relationships pass through them. For example, the left channel of the entity depicted in Figure 4.6a contains 2 relationships and grows to contain three connections, as depicted in Figure 4.6b. Enlarging the grid channels has two main benefits:

1. The paths of all relationships are always intelligible.
2. Users can benefit from the grid to spot the entities involved in many relationships, as they are surrounded by larger channels.

Entities involved in many relationships can be usually considered important elements of the software system being modeled. Thus, this approach facilitates considerably model comprehension.

However, enlarging channels may also introduce visual clutter, disrupting also the uniform visual appearance of the matrix and of the entities. Moreover, users are usually interested only on the relationships belonging to the entities currently visualized. To solve these issues, CEL adapts the channel width basing the calculations only on the connections that are currently being displayed. This methodology costs some interface resizing, because, potentially, the grid channels have to be resized whenever the size or the position of the matrix are changed. Nevertheless, this strategy avoids disturbing situations where large channels are rendered, but no relationship is actually displayed.

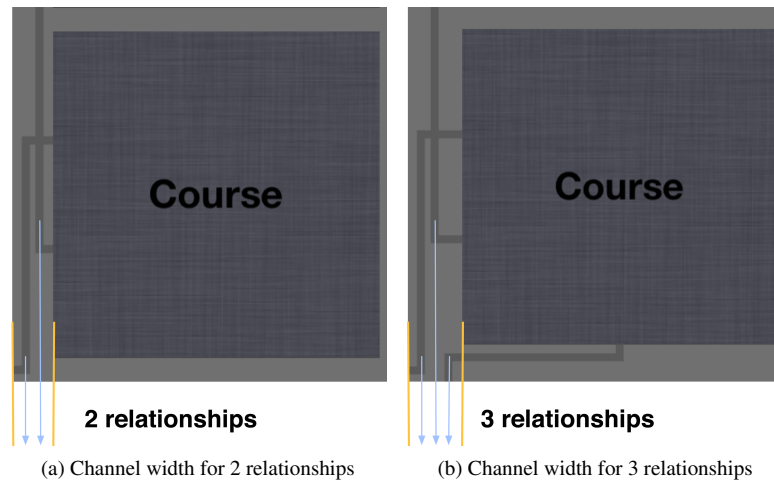


Figure 4.6. Channel size depends on the number of passing relationships

Path Calculation

The path (*i.e.*, the polygonal chain) of each relationship is calculated using the Dijkstra algorithm [CLRS09], which uses a greedy strategy to solve the single-source shortest-paths problem on a weighted (non-negatively), directed, graph.

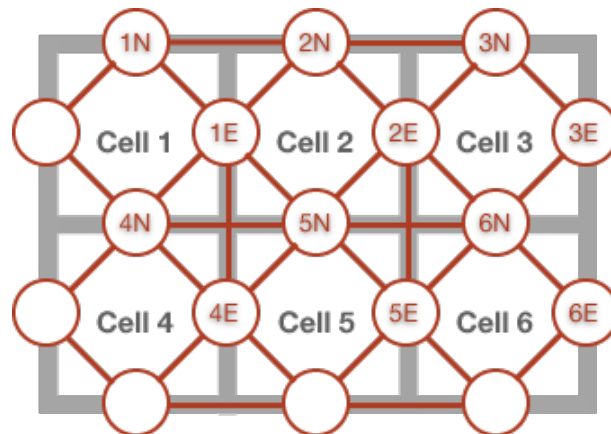


Figure 4.7. The graph constructed on top of the grid

When calculating the paths, we transform the matrix in an undirected graph and, for efficiency reasons, each cell references two new nodes of the graph (*i.e.*, the north and east). A schematic representation of our approach is shown in Figure 4.7. Each edge connecting two nodes has a weight, which is increased whenever the edge is used by a path of a relationship. To avoid cycles in a path, while exploring the graph, each edge can be traversed only in one direction.

We tuned the strategy employed to calculate the weights of the Dijkstra algorithm to limit the growth of the channels: The trade-off is between channel width (*i.e.*, we do not want very large channels) and intuitiveness (*i.e.*, the path should be as short as possible).

4.4 The User Interface

The user interface of CEL is based on our matrix metaphor. CEL works in all the orientations supported by the iPad, and the UI adapts automatically itself to the new orientation when the device is rotated. The overall UI has been adapted to best fit the modeling activity on a touch-based tablet computer, taking into consideration also the conventions and standard imposed by iOS³, the operating system which runs on the iPad. In this section we present the details about the matrix visualization implemented in CEL. We describe the methodology we used to solve the problem of having a small screen size: the semantic zoom. Finally we also describe the projects index, the first view which is encountered when the application loads.

4.4.1 Enter the Matrix

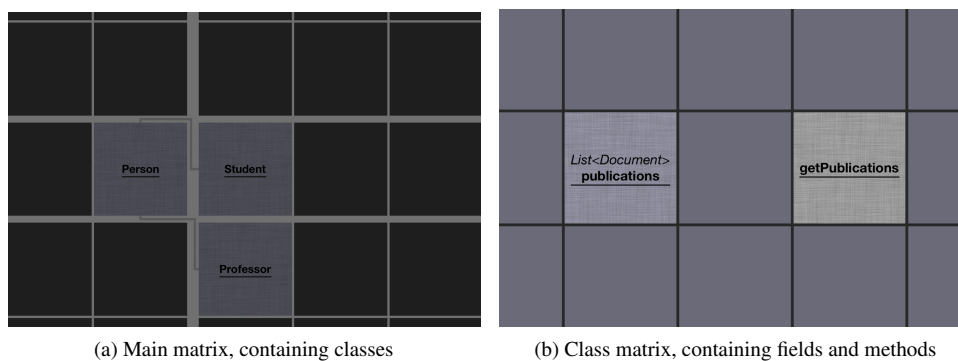


Figure 4.8. The different matrix visualizations

The main view of CEL relies on the matrix metaphor, described in Section 3.2, to subdivide the theoretically infinite 2-D space available to the user. This kind of view has been used to both visualize the classes of a model (*i.e.*, a project, see Figure 4.8a) and to depict the entities contained in these classes (see Figure 4.8b). We decided to employ the same metaphor for both visualizations to have a consistent view, and lower the number of notions that the user has to master in order to use CEL. Moreover, as in the case of class entities, it is valuable to have the possibility of grouping related methods and fields near each other (*e.g.*, methods which the user knows that will use a field).

Although from a conceptual and an interactive point of view the two matrices are exactly the same, we have decided to use a different color scheme for each of them. Each entity in CEL is bound to a personal color scheme, which describes how the entity itself should be colored (consult Section 4.3.1 for details about entities visualization) and how the related grid should be styled. The default color scheme we have implemented has been studied to be non disturbing and allow the rapid distinction of all the different elements. The color of the two matrices (*i.e.*, the background of the cells) have been chosen to be radically different (*i.e.*, almost black for the main matrix and light gray for the class matrix). Thus, the user can identify on-the-fly which matrix is currently being displayed. Other visual cues, discussed in Section 4.4.2, have been used to preserve the outer context also when the class matrix is shown and the internals of a class is being modeled. All the interaction methodologies available for the matrix-based views are presented in Section 4.5.

³<http://www.apple.com/ios/>

4.4.2 Semantic Zoom

Applications for tablet computers must deal with the available screen size, that is considerably smaller with respect to modern computer displays. CEL overcomes this problem by transforming the simple matrix (*i.e.*, the infinite 2-D space) into a semantic zoomable interface. When users zoom in, they progressively unveil details about the entities, as well as new operations and actions.

We opted for a semantic zoom because we argue that when users zoom out considerably, they are interested in obtaining the "big picture" of the system and they do not need detailed information. Furthermore, by allowing the user to perform complex operations on small entities (due to the low zoom level), we would induce errors, because their visual representation would be difficult to be distinguish, select and manipulate. Semantic zoom is also optimal for high zoom levels, because in this case the visual depiction of entities becomes big. Adding more fine-grained details is a good way to exploit the available space. Overall, the quantity of information displayed has to be proportional to the available space, avoiding the introduction of visual clutter, one of the most common defects affecting modern user interfaces [MS94].

CEL supports five types of semantic levels and below we briefly present them one-by-one, explaining the rationale behind their realization and their characteristics.

Birds-Eye

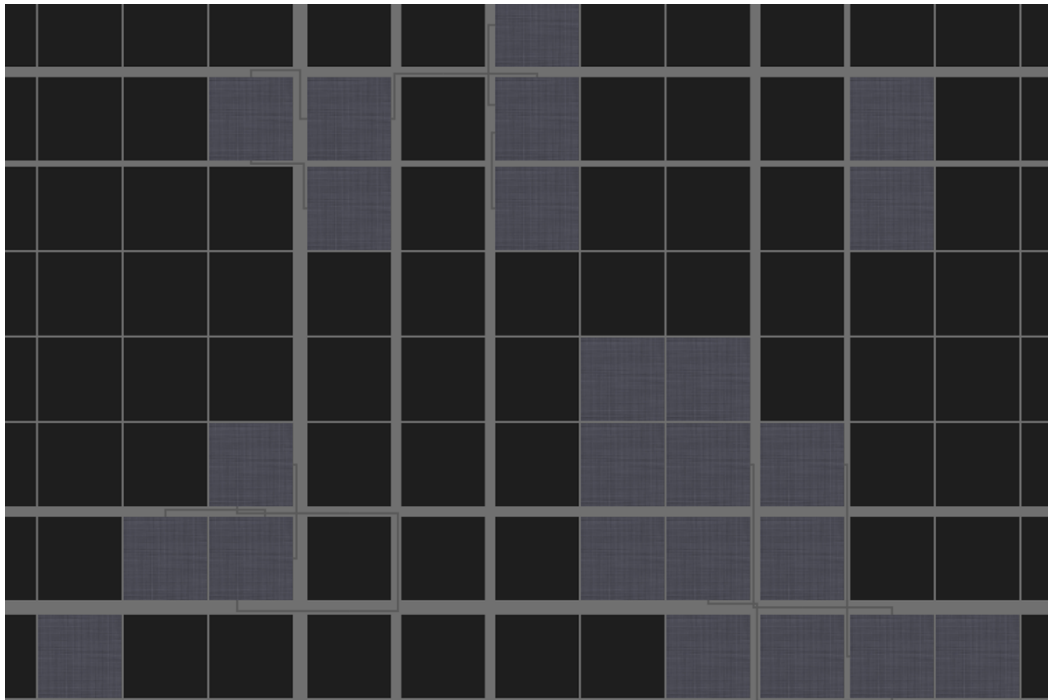


Figure 4.9. Birds-Eye semantic level

This is the lowest semantic level and at this zoom level entities appear as shown in Figure 4.9. Names are not displayed, because the aim of the birds-eye semantic level is to provide the user with the "big picture" of the model and, thus, the details are not relevant.

This visualization is ideal for moving from one portion of the matrix to another one. Having a global overview of the model is also precious to identify groups of related entities. Finding such clusters gives the chance of rapidly discovering important parts of a model. The number of elements displayed concurrently is considerable and the channels of the grid will be enlarged according to all the relationships involving such entities. Consequently, there is a significant probability of spotting core entities by simply analyzing the entities placed in the neighborhood of the crossing point of large channels.

Unfortunately, this low zoom level is detrimental for almost all operations which work on single entities. Thus, at this level, users can only select and move entities, and both operations work best when applied on a group of elements of considerable size.

Overview

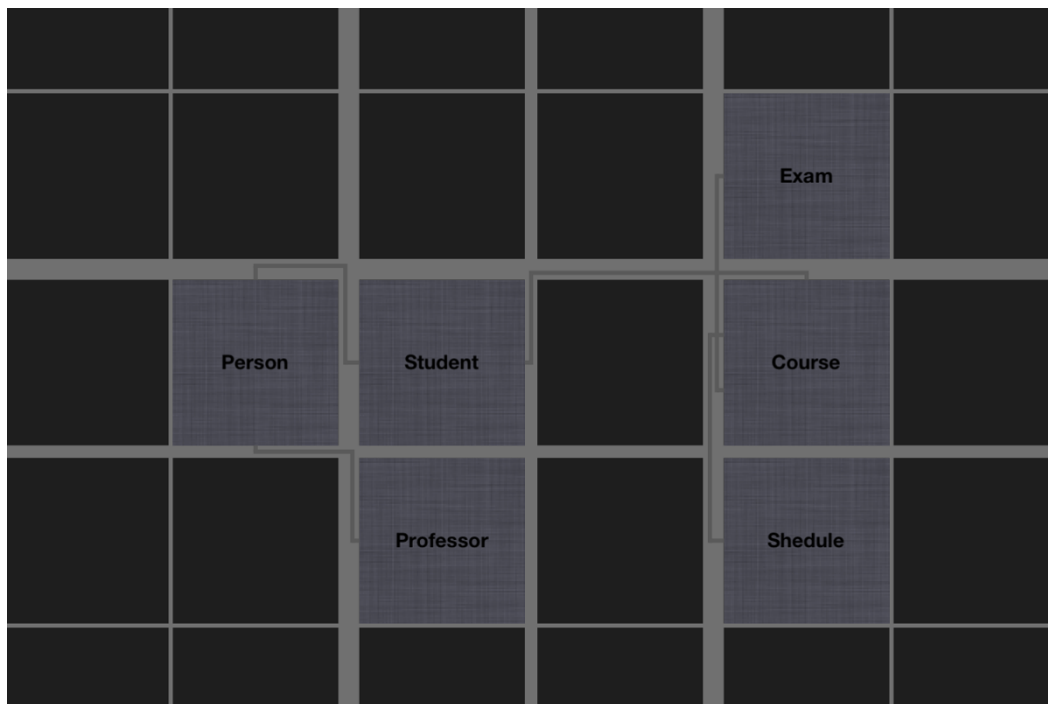


Figure 4.10. Overview semantic level

As shown in Figure 4.10 at this level CEL displays the names of the entities. Labels are now readable and entities are big enough to be handled also singularly. In fact this semantic level is fundamental: It provides all the necessary operations to design a concrete software model. The size of entities makes them easily manipulable, yet it is still possible to navigate smoothly through the matrix in order to construct new parts of the system in a different portion of the 2-D space.

Several new operations are provided: interaction using context menus, creation and deletion of relationships and entities, and the possibility of directly jumping into the internal semantic level (*i.e.*, zooming into an entity). All the necessary details about how users have to interact to issue these commands are provided in Section 4.5.

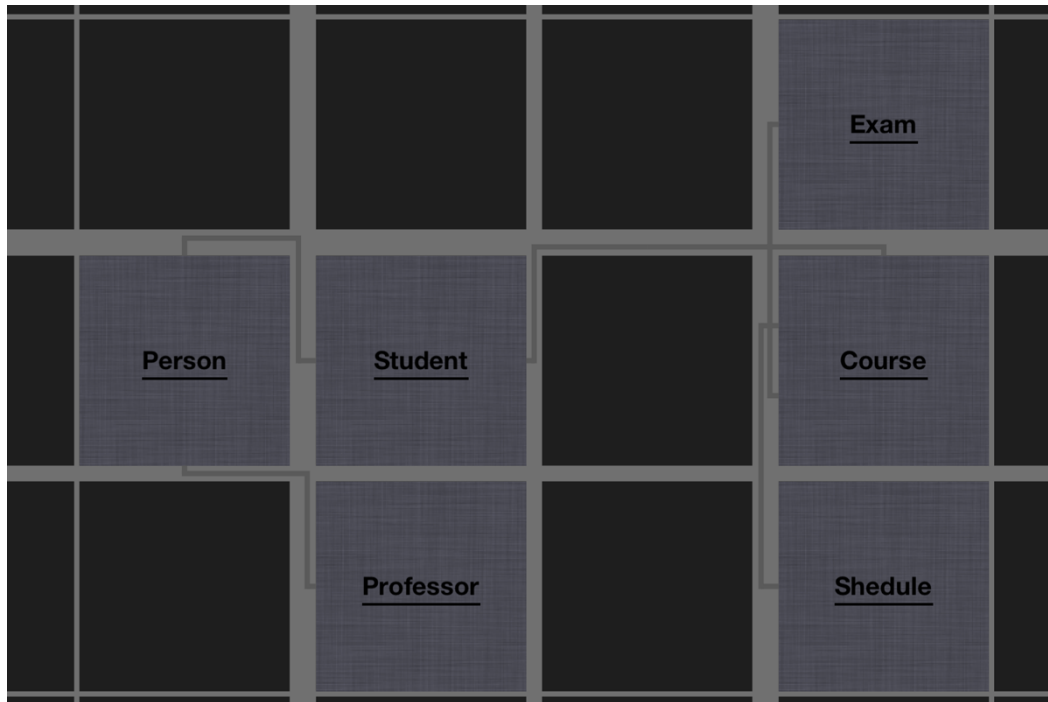
Edit

Figure 4.11. Edit semantic level

Currently, this level enables only one new operation: editing the name of entities via gestures. The new semantic level is denoted by a visual cue of underling the name, as shown by the entities illustrated in Figure 4.11. The edit level has been introduced, with the aim of enabling complex (optional) gesture-based interactions that need a considerable precision.

This semantic level is the level encountered when a new project is created and the matrix is shown. We have decided to propose this as the default semantic level for three reasons:

1. This zoom level includes entities' sizes which favor their manipulation.
2. At the beginning of a new project it is common practice to start by creating a couple of key entities. Thus, there is no need to work at a low zoom level, hindering gesture-based interaction, which becomes definitely harder the more the entities become smaller.
3. At this semantic level all the operations on entities (not on their internals) become available.

In contrast to the overview semantics, this level is more focused on favoring the manipulation of entities which are close to each other. Navigation is still smooth, but, due to the increased zoom factor, the number of interactions (*i.e.*, pans) needed to move far away is increased.

Details

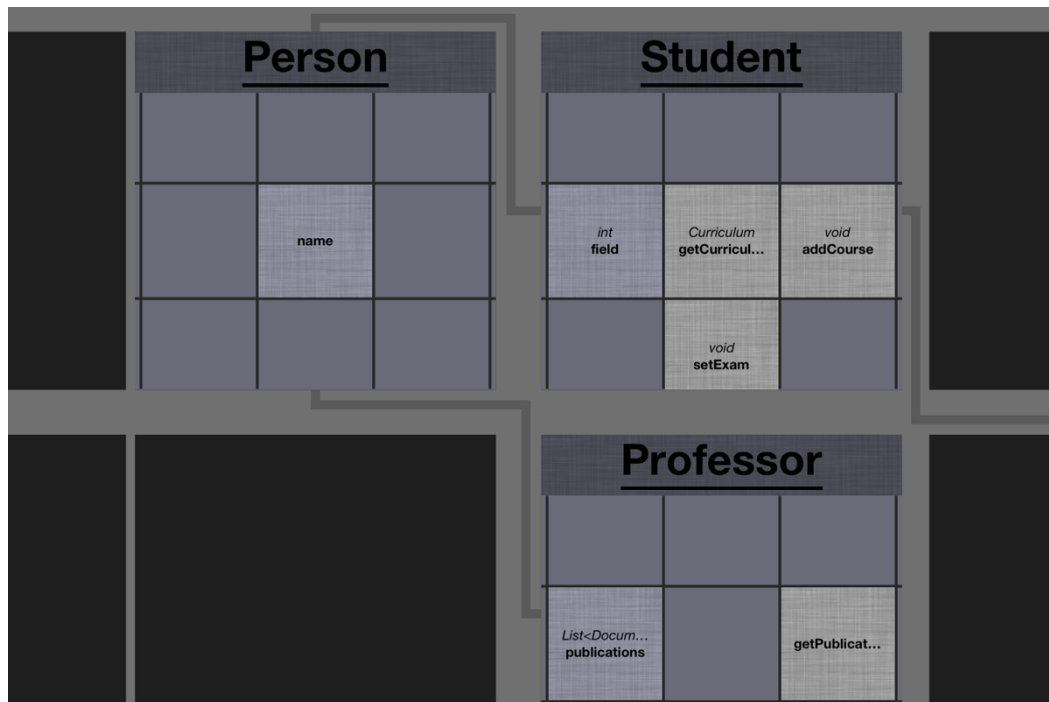


Figure 4.12. Details semantic level

As shown in Figure 4.12, when entering this semantic level, all the visualized entities reveal their composition: The embedded content (*i.e.*, in case of class entities, the content is composed by methods and fields) becomes visible.

This zoom factor induces three main problems:

1. The effort required by movement actions is considerable, making navigation is suboptimal.
2. Due to their considerable size, entities are difficult to manipulate.
3. The number of entities which fit into the screen is very limited.

These problems cannot be easily solved, and we do not claim our approach being the solution to any of them. We believe that, once arrived at entities which are that big, the user is not interested anymore in navigating the entire model or in modifying entities. We claim that it makes more sense to allow users to rapidly discover details about the visualized entities, which can be easily depicted thanks to the considerable amount of space available for each of the elements. The displayed content is valuable for model comprehension, because it allows users to instantly gain an insight about the internal structure of all the visualized entities.

The internal matrix is read-only and should only be used to acquire information without the need of using the internals semantic level. The visualization of the content can be navigated, but child elements cannot be manipulated.

Because parameters are depicted using another metaphor, the view which depicts the details of a method is discussed separately in Section 4.4.3.

Internals

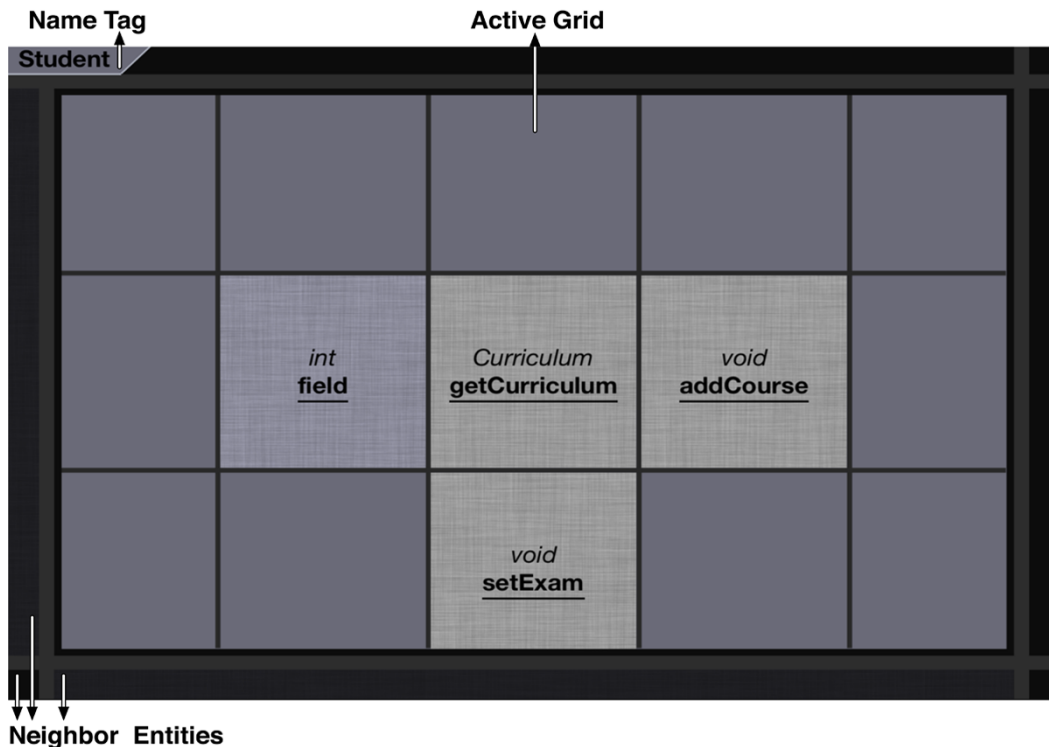


Figure 4.13. Internals semantic level

Figure 4.13 depicts the highest semantic level, which can be accessed only by explicitly zooming into an entity. Such interaction is explained in Section 4.5. The active matrix supports the same interaction methodologies as the parent one. Users do not need to learn new notions and can reuse the knowledge acquired to interact with the main matrix to create, delete and manipulate child elements.

Zooming completely into an entity gives the possibility to fully concentrate on modding this particular element, with all the visualization supporting this particular task. Nevertheless, preserving the outer context is fundamental for being able to return to the global overview later, in case of slips in the interaction (*i.e.*, the user did not want to zoom in). Moreover, in case of prolonged absence from the modeling activity (*i.e.*, the user has to go away for some time), the user has to be able to recover the context. For supporting context preservation the visualization of the active matrix has been scaled, in order to allow the insertion of different visual cues:

1. The top left corner features a menu tag, displaying the name of the zoomed entity.
2. A simplified version of the parent matrix, with no relationships, is still visible around the active matrix. We consider neighbor elements fundamental to preserve the outer context.

Because users can model internal content only by zooming into an entity, this approach is optimal for creating models in which users first lay out entities and then progressively analyze each of them and refine the initial concept. In fact, this approach is convenient to concentrate on a single element, but does not allow on-the-fly modifications of the content.

Again, the internals of a method (*i.e.*, parameters) are discussed separately in Section 4.4.3.

4.4.3 Third Zoom Degree: Parameters



Figure 4.14. The internals of a method: parameters' visualization

We believe that our matrix metaphor is not suited for depicting parameters inside methods. In fact in the case of these entities, the order is extremely important, and a list representation is more reasonable. Because iOS users are used to standard list widgets, we decided to reuse them, customizing their aspect to recall the overall visual aspect of CEL.

As depicted in Figure 4.14a, the details semantic level of a method already shows the list which can be scrolled, but not modified. Lists are not zoomable, but they adapt to the current layout, in order to show more text and entries if there is enough available space.

Figure 4.14b shows the internal semantic level of a method. The layout is identical to the internals of a class. All the information previously available to reconstruct the outermost context (*i.e.*, the class name and the neighbors of the class) are preserved and other elements (*i.e.*, a tag with the name of the method being manipulated and the neighbors of the method) are visualized to again preserve the intermediate context (*i.e.*, the one of the class matrix).

The list widgets support all the actions executable on parameters. The plus button on the top right of the list can be used to add a new parameter using a keyboard-based view (consult Section 4.5.3). Each entry contains a control on the right which can be employed to move the item. Swiping on an entry, this is substituted with a menu. This situation is depicted in Figure 4.14c. The menu contains the entries to delete the parameter or to edit its signature (*i.e.*, name and type). These interaction typologies are not as fast as the ones we present for the matrix-based views in Section 4.5, yet they are often used for list widgets. To exit the view the three-finger pinch gesture has to be used.

4.4.4 The Projects Index

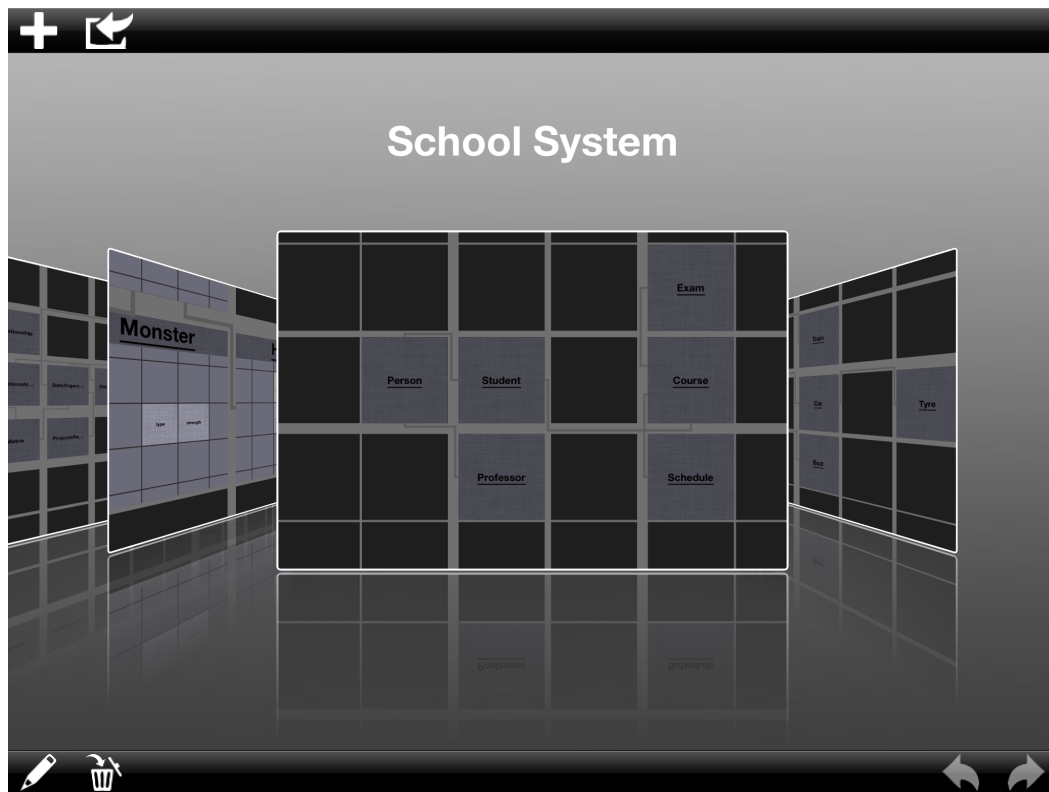


Figure 4.15. The projects index view

Although the matrix-based views are the main components of the user interface of CEL, the first screen, shown at the start of the application, is different and is depicted in Figure 4.15. This view enables users to create, delete, modify and access software models. The existing projects are alphabetically ordered and visualized using a cover-flow effect. We show the name of the project on top and a preview of the matrix underneath. This image is produced when the project is saved and thus, users have a concrete help in remembering what they were working on in this project.

Projects Index Interaction

The interactions possible on this view are very limited. The user can pan with one finger the cover flow to change the current project and tap the preview image to open (*i.e.*, zoom into) it.

We use toolbars for the rest of the operations. On the top toolbar there are two buttons. The first is used to create a new project, which is automatically opened once created. The second can be used to import a project into CEL (see Appendix C.1 for the details). Operations to act on the project in focus are placed in the bottom toolbar on the left: edit the name of the project and delete it. Finally, on the bottom right, we have placed the undo/redo functionalities. Custom gesture-based interaction is not applicable in this particular view, because it would conflict with the habit that has been created by the extensive usage of cover-flow in many different applications. The matrix-based views of CEL, use more complex interaction methodologies, which are presented and discussed in Section 4.5.

4.5 Interaction

Interaction is, after the screen size, the major issue that software designers have to face when creating an application for modern touch-based tablet computers. The different methodologies available (*i.e.*, gestures, keyboard-based interaction, standard widgets, menus and toolbars) have to be carefully investigated, finding the solution(s) which best fit to the purpose at hand. In this section we first present the general guidelines we followed to support interaction in CEL. Afterwards, we discuss each typology of interaction, explaining how it is used and why.

4.5.1 Design Guidelines

When performing any kind of operation, a user should be able to focus on the current task without any major interruption or disturbance [Ras00]. Since we want CEL to follow this guideline, we imposed ourselves two main design rules:

1. No confirmation message dialogue should be used. These messages repeatedly plague users with questions and block their flow of work. Because errors are rare, the answer given to these tedious questions is often the same. Thus, users start to automatically and unconsciously tap/click to enter the habitual answer (*e.g.*, yes, delete the file). Such behavior disrupts the fundamental utility of confirmation messages: The notification that an irreversible operation is going to be performed (*e.g.*, emptying the trash). Citing Raskin, "Any confirmation step that elicits a fixed response soon becomes useless" [Ras00].

To create a confirmation-free UI, we introduced an undo/redo system that captures any relevant event (*i.e.*, events that modify the model) and gives the possibility to undo it. Each project keeps its own history of events, which is deleted only once the project is closed. As we presented in Section 4.4.4 there is also a history of commands for the projects index. This history is erased only once the application is shutdown. The possibility of undoing operations such as projects' deletion is valuable, and is actually very rare in modern computer applications.

2. Waiting times must be reduced as much as possible. Waiting time is wasted time, which introduces frustration and seriously damages users' attention. Such *dead periods* are related to costly operations (*i.e.*, actions which require a significant amount of time and/or resources to be completed) or to resources locking (*i.e.*, some resources cannot be accessed by others, before the current operation is finished).

We designed CEL to be able to lock as few resources as possible. The crucial operation we had to optimize was the save of projects: Users can continue interacting with the application and with any project, except the ones being saved. We also designed the tool to perform all costly operations, which do not directly involve the current visualization (*e.g.*, the production of the preview image of a saved project), on a separate thread in order to cut waiting periods.

With these two rules settled, we chose the best interaction methodology for CEL. Touch-based tablet computers do not come with a pointing device such as the mouse (or the trackpad) in standard computers (unless one attaches an external support device). Moreover, the available digital keyboard needs training to be used proficiently. Widget-based UIs (*i.e.*, using toolbars, menus and buttons) have also significant drawbacks: toolbars often have small icons, thus creating usability issues and it is hard to create fully-interactive applications using only such means. For these reasons, we have decided to base the interaction offered by CEL on the innovative interaction methodology provided by touch-based devices: the gestures.

4.5.2 Gesture-based Interaction

Gestures are movements/actions captured on a (multi-)touch sensing surface. One can create complex gestures, or rely on the simple, standard, ones provided by most operating systems running on touch-based tablet computers (*e.g.*, tap, double-tap, pinch, *etc.*). Gestures are powerful means to create interactive applications, but the more gestures are employed, the more the usability of an application depends on them: On their easiness in usage and on the ability of users in utilizing such an innovative interaction methodology. Even simple gestures can become problematic for people not used to touch-based interfaces. This interaction technique introduces a learning phase which is necessary to become familiar with touch-based devices, to get used to the sensitivity of gestures, and to train and acquire a sufficient level of experience.










Gesture		Action(s)
Single Tap		<ul style="list-style-type: none"> • Create a new entity. • Open a context-menu for an entity.
Double Tap		<ul style="list-style-type: none"> • Zoom into an entity.
Pinch		<ul style="list-style-type: none"> • Zoom in and out in the matrix view. • Zoom out from an inner matrix.
Three-finger Pinch		<ul style="list-style-type: none"> • Zoom out from an inner matrix.
Two-finger Pan		<ul style="list-style-type: none"> • Pan the matrix view. • Move a selection of entities.
Line		<ul style="list-style-type: none"> • Edit the name of an entity. • Create and delete relationships.
Cross		<ul style="list-style-type: none"> • Delete entities.
Three-Finger Down Swipe		<ul style="list-style-type: none"> • Clear the selection.
Shape		<ul style="list-style-type: none"> • Create and manipulate a selection.

Table 4.1. Summary of the gestures used in CEL

The gestures used in CEL and their mapped action(s) are summarized in Table 4.1. We believe that gestures involving more than 2 fingers have to be used only for rapid and simple movements. Also, moving three or more fingers on the touch-based display occupies a significant amount of physical space, which makes it difficult to perceive what is happening on the screen during the gesture. These gestures should be mapped to non-essential operations, or shortcut commands created for proficient users.

In mapping gestures and operations, we decided to minimize the use of modes [Ras00]. Modes arise from the use of the same gesture for different purposes, depending on the state of the system. Although we agree that modes are bad, the interaction capabilities offered by touch-based devices are more limited than the ones offered by standard computers and, thus, the reuse of gestures is often unavoidable. Nevertheless, we agree with Norman that if modes have to be used, we can limit errors by providing clearly visible feedback on the state of the system [Nor83].

All the gestures employed in CEL are discussed in the following.

Single Tap

The use of this gesture indicates, generally speaking, that the user wants to *do something*. Usually, some of the most fundamental actions are assigned to this gestures, because it is simple to execute and it is the interaction which resembles at most the mouse click (*i.e.*, one of the fundamental interaction methodologies in standard computer applications).

We have assigned this gesture the fundamental role of instantiating the creation of a new entity when an empty cell in the matrix is tapped. If, instead, users tap on an existing entity, its context menu (discussed in Section 4.5.4) opens up.

Double Tap

We keep this gesture consistent with its behavior in iOS: bound to the concept of zoom-in and zoom-out. Double tapping an entity allows users to zoom into the selected element, showing the *internals* semantic level. Because there are three different zoom levels (*i.e.*, classes, methods and fields, and parameters) we have not employed this gesture to allows users to zoom out from the internal view of an element.

Pinch

The pinch gesture is well-known to most touch-based device users. It is used to perform zoom operations and we have reused the same concept also in CEL. Additionally, we have added a secondary feature to this gesture, allowing users to zoom out from the internals of an entity when the minimum zoom level is surpassed. When acting on the initial matrix, the minimum zoom level cannot be surpassed. The same applies for the maximum zoom factor, which has been introduced to avoid situations in which the user is confronted with one single entity occupying the whole screen.

Three-finger Pinch

This custom gesture allows to rapidly zoom out from entities without the need of zooming out long enough to surpass the minimum zoom factor. With a single, rapid, movement the internals view is dismissed. The similarity of this gesture with the standard pinch one (*i.e.*, only another finger has to be placed on the touchscreen) favors its learning.

Two-finger Pan

Panning is an action associated with the concept of movement. We decided to use a two-finger gesture to move our view. We discarded the usual, and simpler, one-finger pan, because of the conflicts with other gestures (*i.e.*, shape, cross and line gestures).

If the user starts a two-finger pan gesture outside of a selection of entities the entire matrix is moved. In the opposite case, the user modifies the position of the selected elements. Although using this gesture for two different purposes can be considered a mode, the entities selected are clearly highlighted (*i.e.*, we provide visual feedback) and the overall action associated to the gesture, *i.e.*, moving, remains consistent.

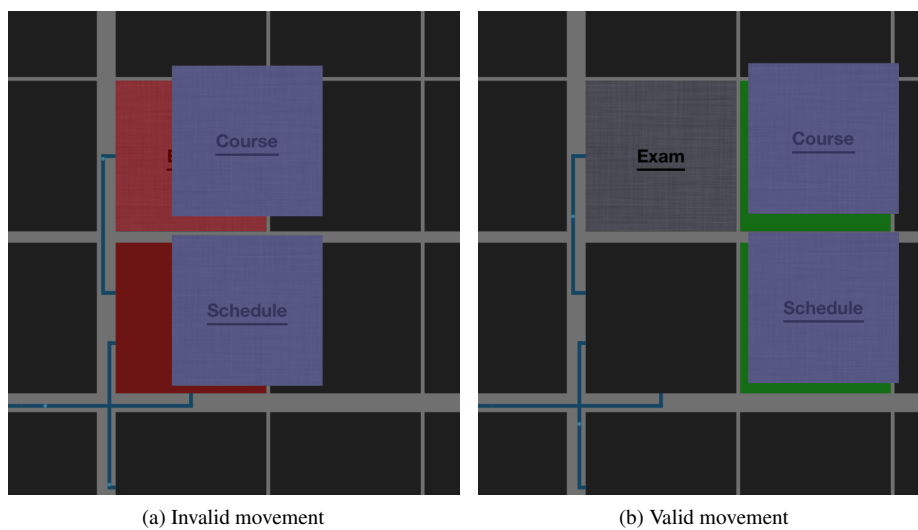


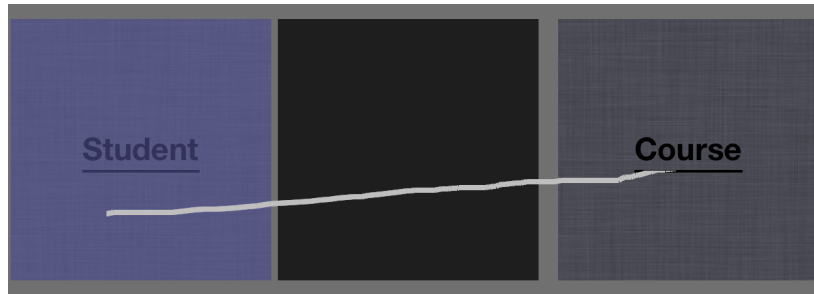
Figure 4.16. Move of a selection of entities

When a selection is moved, as showed in Figure 4.16, an overlay is added, as a visual cue to signal whether the movement is legal (green) or illegal (red). When a legal movement is ended (*i.e.*, the user removes the fingers from the touch-screen), entities are placed at the new position and all the paths for the relationships of the selected entities are recalculated. The recalculation is done at the end, because on-the-fly computation would be too costly. If the movement was illegal, entities return to their original site.

The matrix is also equipped with an auto-pan feature that can be exploited in both cases, while panning the matrix and while repositioning a selection of elements. To activate the auto-pan functionality, it is sufficient to put the fingers at the borders of the matrix, and the application will automatically start to move the matrix in the desired direction. Multiple directions can be combined at once (*e.g.*, up and left, down and right, *etc.*). When moving a selection of entities, the auto-pan shifts the matrix in the opposite direction with respect to the one suggested by the movement: This allows to accommodate the real intention of the user (*i.e.*, if the user wants to move the selection right, the matrix has to pan left and show the content on the right).

Line

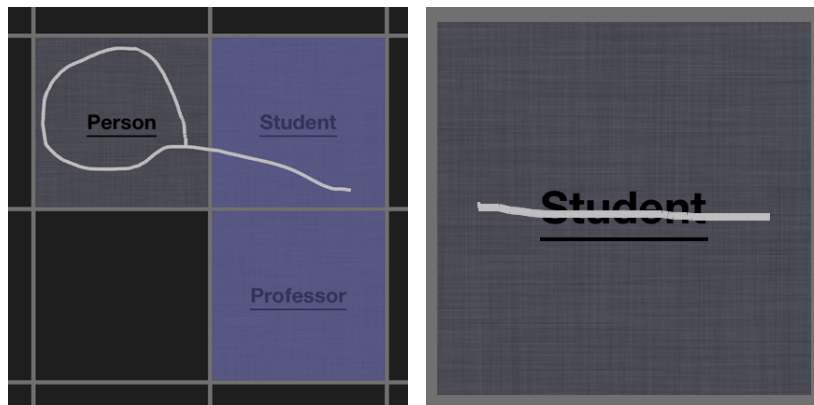
The custom line gesture accomplishes different tasks: handle relationships (*i.e.*, creation and deletion) and allow the renaming of entities.



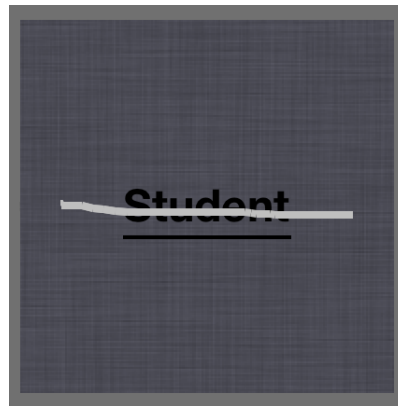
(a) Create generic relationship



(b) Delete relationship



(c) Create inheritance relationship



(d) Edit entity name

Figure 4.17. Line gesture uses

All the different actions assigned to the line gesture (or to its derivatives) are summarized in Figure 4.17. In the following we explain in the details how they differ and why they do not conflict.

To create a relationship the user has to draw a line from a selection of elements (*i.e.*, the sources of the connection) to another entity (*i.e.*, the target of the connection). To distinguish between which of the two different typologies of relationships supported in CEL (*i.e.*, inheritance and generic) is going to be created, we have designed two ad-hoc line-based gestures, shown in Figure 4.17a and Figure 4.17c. The inheritance creation gesture is a line with a closed shape at the end (*i.e.*, like a lasso) which encapsulates the center of the targeted entity. The generic relationships creation gesture is a simple line. They are similar enough to recall the same concept (*i.e.*, relationship instantiation), yet they are still easily distinguishable. We opted for assigning the simplest gesture to the generic relationship because it encapsulates many different meanings (*e.g.*, use, containment, *etc.*), and, thus, it will be probably used more frequently.

Currently, relationships can be instantiated only at the same abstraction level (*i.e.*, classes with classes, methods with fields and methods, and fields with methods and fields). By design choice, the inheritance relationship has to be instantiated from the subclass to the superclass: The direction indicates that the source of the relationship is a subclass of the target. Moreover, for inheritance relationships, the application checks whether the opposite connection has been established. In fact, inheritance relationship can be instantiated only in one direction (*i.e.*, if A inherits from B, B cannot inherit from A). On the contrary, generic relationship can be instantiated in both directions without any limitation. Nevertheless, in both cases, CEL checks, before actually instantiating a relationship, whether this connection does not already exist, and, in case, it does nothing.

To delete relationships, we decided to create a gesture similar, but not equal, to the one employed to delete entities (*i.e.*, the cross). The selected approach is illustrated in Figure 4.17b. Since relationships are drawn as polygonal chains of straight lines, the line drawn with the gesture intersect the visualization forming a cross, thus recalling the same gesture used for entities. As we discussed in Section 4.3.2, details about relationships are available only for those which involve a selected entity. Thus, to minimize errors, to delete a relationship one of the involved entities has to be selected. Deletion may conflict with the creation of generic relationship. However, deletion can be achieved by drawing very short lines which, very often, would have no reason to cross multiple entities. Thus, we have decided to give priority to entities creation, and if the gesture matches, the deletion is aborted.

The last interaction type that involves the line gesture is the renaming action. As depicted in Figure 4.17d, by striking the label of an entity, one can edit its name. This interaction is limited to the rectangle representing an entity, and does not conflict with the other uses of the line gesture.

Cross

This custom gesture has been created to delete entities. A cross drawn over multiple entities deletes them all. Thus, this gesture allows the rapid erasion of entire parts of the matrix at once.

In iOS, deletion is usually accomplished with a (slow) combination of gestures and buttons. We claim that a cross gesture can be naturally mapped to the concept of erasing content. The gesture recognition system has been tuned to not require the drawing to be extremely precise.

Three-finger Down Swipe

This gesture clears the entire selection (*i.e.*, deselects all entities). Metaphorically this interaction replicates a *sweep* and its main purpose is to help users to rapidly deselect every entity, in order to create a new selection and be sure to not have undesired elements in it.

Shape

The shape gesture is probably the most powerful custom gesture we have designed. The possibility to draw any kind of closed shape to activate this gesture mitigates the problems related to proficiency and ability in drawing: No extreme precision is needed. If no other gesture is compatible with the performed movement, the shape drawn is automatically closed, making this gesture even faster to perform (*i.e.*, users do not have to manually close the shape).

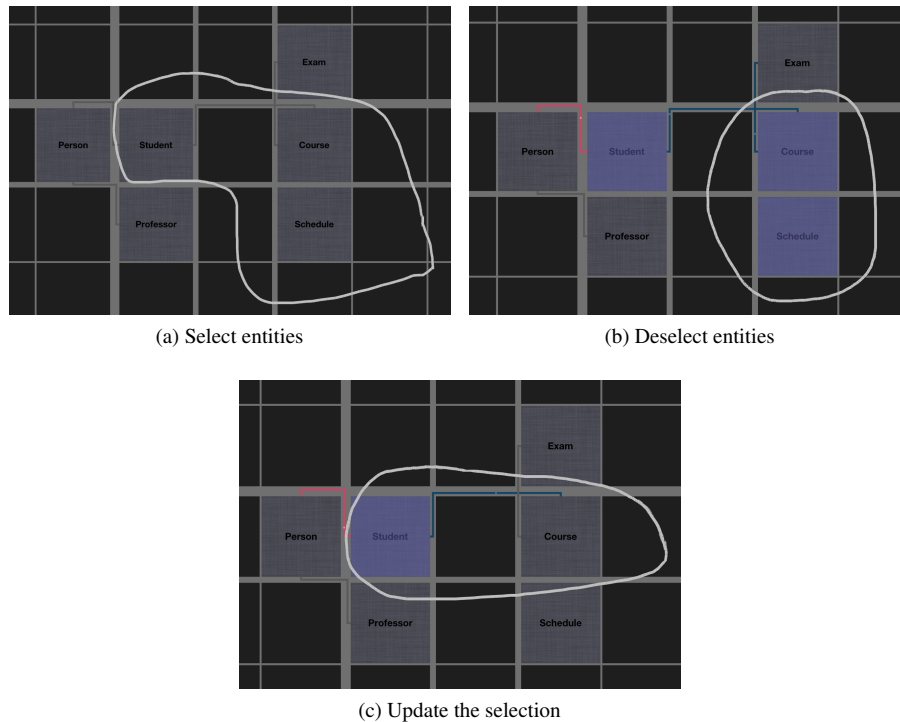


Figure 4.18. Shape gesture usage

Each shape gesture acts on the global selection that is assigned to each matrix view. The two trivial cases are shown in Figure 4.18a and Figure 4.18b. In the former, the user can add unselected entities to the selection. In the latter, the user deselects the entities which are contained in the shape.

Figure 4.18c shows a more complicated and controversial case, in which the shape drawn surrounds both selected and unselected entities. We had two options to handle this case: Implement the shape as a toggle (*i.e.*, invert the selection) or treat the entities contained in the shape as a single element. We opted for the second strategy. Therefore, if all the surrounded entities are selected, the group gets deselected, otherwise, all the entities are selected. In the example, the *Course* entity is added to the selection, which already includes the *Student* element.

A completely different approach would have been to treat the cases of Figure 4.18a and Figure 4.18c as the creation of new selections, avoiding the possibility to modify a selection once it has been created (*i.e.*, selections become immutable). Such a solution is simpler to understand (*i.e.*, you do not forget to have entities selected in some place of the matrix), but it requires to create selections, even complex ones, with one single shape gesture.

4.5.3 Keyboard-based Interaction

Physical keyboards have been used for decades and the proficiency that users have achieved is considerable. With their support, keyboard-based interaction is one of the most rapid ways to interact with a digital device. This does not hold on touch-based tablet computers, because the digitalized representation of the real device presents numerous usability issues [McD11]. Moreover, recent studies have been demonstrated how users are significantly more efficient with physical keyboards [CNP⁺ 10]. Such results are a clear signal that currently physical keyboards are definitely superior to their digitalized versions, also due to the habituation that users have developed towards these devices. It is hard to state if one day these results will be reversed.

When using keyboard-interaction on touch-based devices, one should consider also the training needed to become barely capable of using the digitalized version of the real device. In fact, the differences in terms of *sensations* are considerable (*i.e.*, the physical key is consistent, the user can feel it, a sensation which is not possible to have with digitalized keyboards) and they can severely affect the performance (*e.g.*, words typed per minute, errors, *etc.*) of users. We decided to limit keyboard-based interaction as much as possible, employing it only for (re)naming entities.

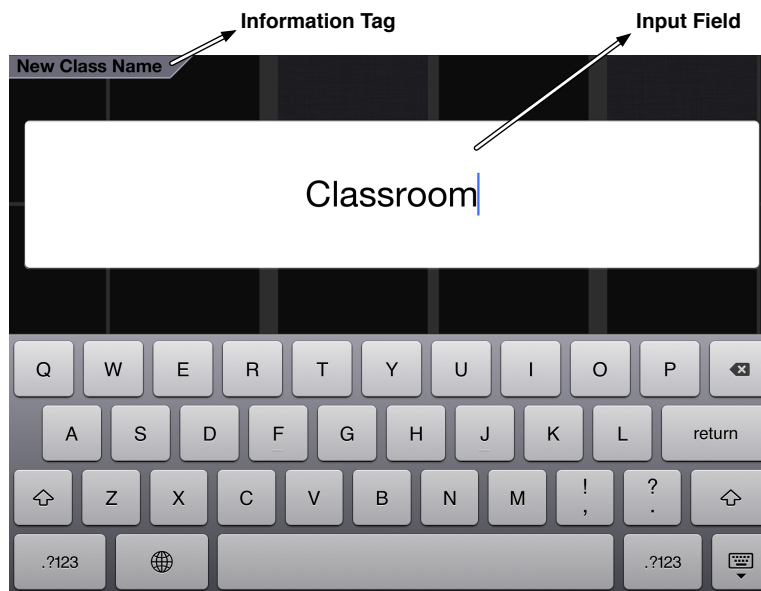


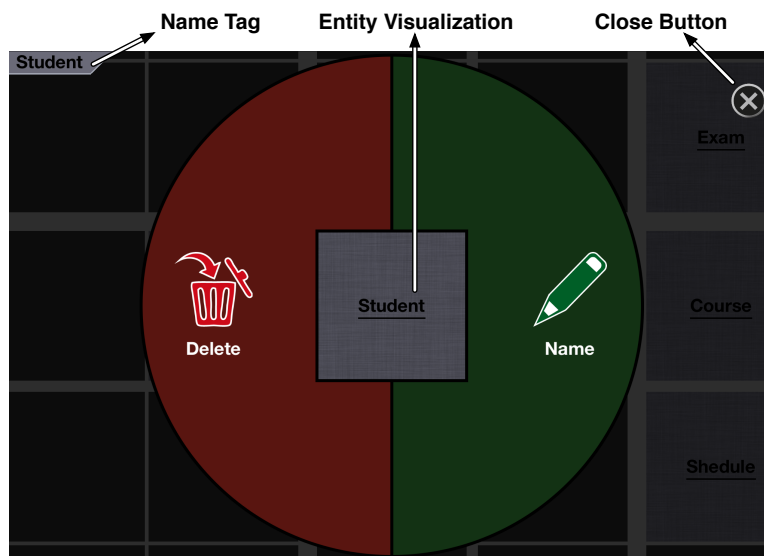
Figure 4.19. Keyboard-based interaction view

When keyboard-based interaction is necessary, the view depicted in Figure 4.19 is shown. The visualization is dominated by the presence of the keyboard and of one single input field, as we want the user to focus on the operation at hand: the manipulation of the name of an entity. We use a small information tag in the upper left part of the screen to show the action which is currently being performed (*e.g.*, name a new class, edit the name of a field, *etc.*).

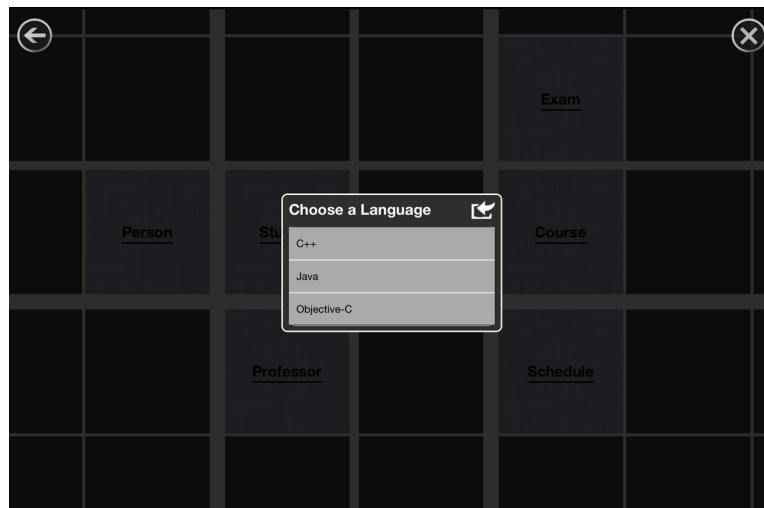
To quit the keyboard-based view, users have simply to hit the return key. If the entered name is invalid, CEL signals the error by highlighting in red the text and do not close the view. When assigning the name to a newly created entity, CEL handles the input of an empty name by not instantiating the new element. When editing an existing entity's name, instead, the input of an empty string leads to the preservation of the old name.

4.5.4 Menu-based Interaction

Although in *CEL* gesture-based interaction plays a prominent role, menus accommodate less experienced users. For menus with few elements, *CEL* supports pie menus, as opposed to traditional list-based menus, because they provide lower error rates and minimize the target seek time [CHWS88]. Pie menus also provide increased target size and reduced distance from any element to any other entry (*i.e.*, the elements are all at the same distance from the center, favoring a rapid movement to the correct target). When menus are of considerable length, or the number of elements is not fixed, list-based menus are used.



(a) Pie-menu like context menu



(b) List menu

Figure 4.20. Menus of *CEL*

In Figure 4.20a we illustrate a usage of a pie-menu in CEL: A context menu is opened when an entity in the matrix is tapped. When a menu, as in the example, references an entity, the visualization of the element remains visible in the middle of the screen, surrounded by a full-screen pie menu. Other elements of the matrix are shown with low opacity, to preserve the outer context. A name tag is showed in the upper left part of the screen, to show the complete name of the entity, which may not fit in its visualization. A generic pie menu (*i.e.*, with no name tag and no entity visualization) is used to export the model, a functionality discussed in detail in Appendix C.2.

Figure 4.20b shows an example in which a list-based menu is used: The selection of a language in which skeleton code should be exported (see Appendix C.2.3 for the details). Because the number of elements which populate the menu is variable, we adopt a list-based menu. An alternative could have been to employ pie-menus until a certain number of entries is reached. However, this solution would harm the habits of users by suddenly modifying the visual aspect of a component, which may have been constant for a significant period of time.

If multiple menus have been visited, a back button, which can be used to navigate to the ancestors, is shown in the top left part of the screen. To close any menu, users can use the close button in the top right portion of the screen.

4.5.5 Toolbar-based Interaction

In CEL we use toolbar to provide quick access to features that may be invoked at any moment, regardless of the context. In each toolbar we represent actions by means of icons (*i.e.*, iconic representation [LHB03]). For space problems we have not introduced labels. We partially make amend for this problem by choosing only commonly-accepted, not arbitrary, icons. To be in line with the rest of the UI, also the icons are very minimalistic and simple.

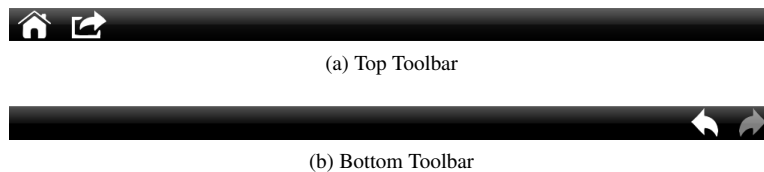


Figure 4.21. Toolbar of the matrix view

The matrix includes a top toolbar, shown in Figure 4.21a, which contains a home icon to save the model and return to the projects index and a second icon used to export the model (see Appendix C.2). The bottom toolbar, depicted in Figure 4.21b, includes icons to use the undo/redo system.

Toolbar-based interaction is possible only when both, keyboard-based and menu-based interaction, are not active. Otherwise toolbars are disable or not even present (*i.e.*, like the bottom toolbar in keyboard-based interaction).

4.6 Summary

In this section we have introduced our modeling framework and CEL, our highly interactive iPad modeling application. We have shown its user interface and analyzed the available interaction methodologies. In Chapter 5 we present an initial qualitative evaluation we have conducted to verify our choices and gather feedback.

Chapter 5

Evaluation

In Chapter 4 we presented CEL, an iPad application which integrates the modeling methodology we have devised (see Section 3.1). We believe that our approach is beneficial for the software modeling process. By conducting evaluation experiments we are able to gather data, feedback and suggestions which we can use to draw conclusions about our work and to further improve it.

As a first evaluation we decided to not go for a qualitative experiment such as [WLR11], but to organize a qualitative evaluation to assess the goodness of our choices. In the following sections we introduce our experiment, we explain how we executed it, we discuss the possible threats to validity, and, finally, we present the results, the feedback and the impression we gathered from the evaluation.

5.1 Experimental Design

The experiment has been designed to produce a first qualitative evaluation of our choices which we have integrated in CEL. In the experiment each participant is also confronted with ARGOUML, a well-known UML editor. We do not aim to prove that CEL is better (or worse) than ARGOUML, but we decided to show to users the two very different modeling approaches. A comparison with lightweight modeling techniques (*e.g.*, whiteboard), would also have been interesting, but for this experiment we decided to limit the scope and compare our work with a digital, heavyweight, methodology.

5.1.1 Research Questions

Although this qualitative user study has been mainly designed to gather feedback on our modeling technique and on different aspects (*i.e.*, user interface and interaction methodologies) of CEL, we also devised two research questions:

RQ1 How do participants adapt to different modeling techniques which provide distinct sets of elements to create a software model?

RQ2 Which strategy is commonly employed by users to identify core concepts of an unknown software model and how do they adapt to different visual cues?

RQ1 is mainly related to the differences between modeling approaches. RQ2, instead, is related to model comprehension and investigates how and if users adapt their strategy to understand an unknown model to the application in use (*i.e.*, exploiting the available tools and/or visual cues).

5.1.2 Data Collection

As suggested by literature [Cre06, Bar08] we have employed a mixture of methods of data collection to have various artifacts to analyze and be able to better interpret the results. The following data sources have been collected:

- **Questionnaires** regarding the participants' experience level and skills, and regarding the experiences in the experiment.
- **Interviews** to gather participants' feedback on their usage of CEL, and to have a double check on the answers given in the debriefing questionnaire.
- **Artifacts** produced during the experiment (*i.e.*, software models), in order to analyze how users modeled the given systems using the two different tools.

Below we briefly explain how each data source was collected and how the resulting data were used during the analysis phase.

Questionnaires

Participants were asked to answer two questionnaires (see the initial and final part of the handout in Appendix A) for different purposes:

1. **Screening questionnaire.** Before executing the actual experiment, participants were asked to provide preliminary information. They were mainly asked to provide personal data and technical experience information. The personal information have been used mainly for contact purposes and for creating statistics about the overall group of participants. The technical experience information, instead, have been employed during the data analysis phase.
2. **Debriefing questionnaire.** After finishing the experiment, participants were kindly asked to answer another questionnaire to give immediate feedback about the different aspects investigated in the evaluation. They were first asked to rate CEL and ARGoUML on different criteria. Afterwards, participants were asked to give their opinion on different fundamental aspects of CEL. In the third part of the questionnaire they had to evaluate the difficulty of the tasks they executed. In the last part of the questionnaire, participants had the chance to write their opinion on both tools and, in the end, give feedback on the experiment.

Interviews

At the end of each experiment, we conducted a semi-structured interview with the participant. The interview was brief (10-15 minutes) and was done 5 minutes right after the experiment. In this period of time participants had the chance to mentally review their experience.

The main purpose of the interview was to collect further feedback, but we also wanted to double check the answers given during the debriefing questionnaire. The questions that guided the interview can be found in Table 5.1. The set of questions was enriched or slightly changed depending on the answers given by the participant during the debriefing questionnaire.

The relevant answers (*i.e.*, the ones which were different from the questionnaire) and interesting statements have been transcribed. We used these data to illustrate the opinion of users and support the finding arising from the overall experiment.

1	How was the overall experience with CEL? Was it difficult to use?
2	Were the gestures difficult to perform? What would you change?
3	Did you try to reorganize the model? Was it difficult?
4	Which strategy have you employed to find the core concepts in tasks B1 and B2?
5	Did you miss the different kinds of relationships? And the different typologies of entities?
6	Was the zoomable interface intuitive?
7	How do you rate the choice of using the zoom-in to model the details of an entity?
8	How do you rate the approach of enlarging the channels? And of showing a marker to indicate the direction of relationships?
9	Which were the main problems you encountered in operating with CEL?
10	What should we improve in CEL? Do you have any other suggestion or idea?

Table 5.1. The list of questions that guided the semi-structured interview

Artifacts

The models produced with both CEL and ARGOUML have been preserved in order to analyze how the participants modeled the software systems and which are the differences that can be spotted. These hypothesis have been confronted with the answers given during the interview and in the debriefing questionnaire. The conclusions arisen from the analysis of the artifacts have been used as support for the overall conclusions which can be drawn from the evaluation.

5.1.3 Object Systems

As we discuss in Section 5.1.4, in our experiment participants are asked to model two software systems (one with CEL and the other with ARGOUML) and to analyze two already created software models (again one with CEL and the other with ARGOUML). Thus, we had to find four different object systems. Reusing the same system for different tasks would make no sense.

The systems needed for such tasks should be non-trivial to be challenging for advanced users, and be representative of a real-world use case. Nevertheless, the systems should not be too complicated, both to accommodate the time constraints which have been imposed during the experiment, and to be understandable also for beginner users.

We have discarded the idea of producing the object systems ourselves: We wanted to be transparent and not risk to produce systems which would favor, in some way, the modeling approach of CEL over the one of ARGOUML. We have chosen four design exercises created by Prof. Cesare Pautasso¹ *et al.* in the context of Software Architecture, a Master course taught at Faculty of Informatics of the the Università della Svizzera Italiana². Two simple systems have been employed for the modeling tasks and two more complex ones have been used for the comprehension tasks. These latter tasks needed already created software models: We did not create the models ourselves, but produced them based on solutions which were proposed during the course. The models have been designed in both CEL and ARGOUML and are identical in terms of entities, positions (when possible), and relationships.

¹<http://www.pautasso.info/>

²<http://www.inf.usi.ch>

The four systems can be briefly described as follows:

- S1 Star Bux DJ.** This system models a radio service for a coffee chain (*i.e.*, Star Bux). The DJ can upload new songs in the system and create a playlist for each day. Each store retrieves the playlist, downloads the songs and plays them. Moreover, each store has a Music Box which can be used to create, burn and buy a CD.
- S2 ATM System.** The produced model should mimic a distributed Automated Teller Machine (ATM) system. Customers can withdraw from any bank which uses this system transparently (*i.e.*, the user should not see a difference if she is withdrawing money from her bank or from another one). Each bank has its own account system that has to be reused and at the end of the day the ATM sends reports to any bank involved in the transactions which have been logged.
- S3 Book A Trip.** The system is concerned with the realization of a novel interface for a legacy system used to book flights. Clients should be able to electronically reserve a flight, print tickets and cancel reservations. The system should also provide a new security layer and record all the history of trips for further data analysis.
- S4 Public Transport.** This system models an integrated service that supports people while using the public transport of a city. The system exploits GPS positioning of all transport means to suggest routed (and means) to the users. The system seamlessly integrates the schedules of different transport means. Moreover, the service takes traffic into account and users themselves can signal problems (*e.g.*, accidents). Finally, users can also directly pay tickets by SMS or using an app on their smartphone.

5.1.4 Tasks & Treatments

The assignment was composed of four tasks, time-limited up to 20 minutes (see Appendix A for the complete handout). However, we have not been not strict on timing (*i.e.*, each subject was given couple of minutes more to finish her work if the task was almost completed). The tasks were divided into two groups:

- **Modeling Tasks: A1 + A2**

These tasks provided participants with a textual description of systems S1 and S2. The goal of the tasks was to model the software system in a sufficiently detailed manner, including, when needed details (*i.e.*, fields and methods) and relationships. The participants were free to stop before the expiry of the 20 minutes, if they were convinced that the model they produced was sufficient to represent the given description.

- **Comprehension Tasks: B1 + B2**

These tasks required to the participants to analyze the software models of the systems S3 and S4. The participants were asked to give a brief description of the overall purpose of the modeled system, to identify the key entities (describing them briefly) and report the strategy they employed to find such information. The subjects could stop whenever they finished answering, also if the limit of 20 minutes was not reached.

Treatment	Tool A1	Tool A2	Tool B1	Tool B2
T1	CEL	ARGOUML	CEL	ARGOUML
T2	ARGOUML	CEL	ARGOUML	CEL

Table 5.2. Treatments of the evaluation

For each set of tasks one had to be executed using CEL and the other employing ARGOUML. Because it is hard to assess that the two tasks in each group have the same difficulty, we decided to create two different treatments, which are summarized in Table 5.2. Each treatment was exactly identical except for the application used in each task.

5.2 Experiment Operation

Instead of including warm-up tasks in the experiment, each of the evaluations was preceded by a brief training session, in which the user was introduced to both CEL and ARGOUML. During this preparation, which lasted maximum 10 minutes for each tool, we presented the main aspects and concepts required to perform the experiment. We briefly describe the training session for each tool in the following:

- CEL
The user has been introduced to the matrix-metaphor and to the different interaction methodologies used in CEL. The methodologies to create, manipulate and delete entities have been presented, together with the semantic zoomable interface and all its different zoom levels.
- ARGOUML
The basic elements of UML were explained, showing how to create, manipulate and delete classes and relationships. Also the user interface of the tool was presented. We focused also on some elements which are not present in CEL, as for example different kinds of relationships (association, aggregation, composition).

Each session involved only one participant at a time and in each of the different experiments the tools were furnished always following these guidelines:

- CEL was installed on a version of *The new iPad* (*i.e.*, iPad of 3rd generation) running iOS 5.1 and with no other application neither started nor in background. The two models for tasks B1 and B2 were already loaded (although only one had to be used) and the application was presented with the projects index view. The version of CEL evaluated in this experiment is the one described in this document except for the typing system which was not present. Thus, the views were slightly, but not significantly, different.
- ARGOUML v.0.34 was installed on MacBook Pro running Mac OS X Lion (*i.e.*, 10.7.3). A fresh installation was provided at every participant and the correct .zargo file needed for task B1 or B2 was also provided.

We kept the time using an alarm clock and annotated the timing information on the handout. Such data was then used in the data analysis as an initial and informal feedback on the efficiency (*i.e.*, completion time) reached by the participants with both tools.

Overall, we were able to perform six different experiments (*i.e.*, three for treatment T1 and three for treatment T2) which took place at the Università della Svizzera Italiana in Lugano.

5.3 Discussion & Results

In the following we discuss the results and the feedback gathered during the different runs of the experiment. First we characterize the participants, mainly using the data collected in the screening questionnaire. Subsequently, we discuss the completion time results and answer the research questions. Finally, we also present the feedback we gathered during the experiment regarding different aspects of our research. All the data can be consulted in Appendix B.

5.3.1 Participants Characterization

The six subjects of our experiments were three Bachelor students in computer science (P1, P2 and P3), and three Master students, again in computer science (P4, P5 and P6).

As expected, the bachelor students had little experience in modeling software systems, as opposed to master students, which rated themselves advanced or experts. However, all the participants had at least basic knowledge about UML. Independently from the experience in modeling, all the participants indicated as their favorite modeling mean the whiteboard (or paper sketches). This can be intended as a further indication that current digital modeling techniques are not sufficiently good to support early design phases.

Except for P1 and P6 nobody had previous experience in using ARGOUML, which was beneficial for the experiment because they obviously had no experience also with CEL.

Three subjects (P1, P2 and P6) possess(ed) an iPad or a similar device. Nevertheless, also from data gathered during the interview, all the subjects stated that evaluating their experience with the iPad (or a similar device) was difficult, especially because most applications running today on these devices are very similar to the ones found on standard computers.

5.3.2 Completion Time Analysis

Participant	CEL usage time (mins)				ARGOUML usage time (mins)			
	A1	A2	B1	B2	A1	A2	B1	B2
P1	20	-	7	-	-	14	-	8
P2	-	14	-	14	20	-	13	-
P3	22	-	16	-	-	23	-	18
P4	-	9	-	5	10	-	7	-
P5	20	-	13	-	-	21	-	8
P6	-	24	-	7	21	-	7	-
Time total	62	47	36	26	51	58	27	34
Tasks total		109		62		109		61

Table 5.3. Completion time for each task for each participant

The number of participants is not sufficient to draw statistically relevant conclusions. However, the time measured during the experiment for each single task (consult Table 5.3) suggests that CEL is not inferior neither for modeling creation nor for comprehension tasks. Four participants (P2, P3, P4 and P5) were actually faster in building the model with CEL than with ARGOUML. Three subjects (P1, P3 and P5) were faster in comprehending the models. Two subjects (P3 and P5) were faster in all the tasks using CEL, the opposite has not happened (*i.e.*, nobody was always faster using ARGOUML).

The tasks themselves seem to not have influenced the results, because the timing information are opposite. This may mean that two tasks were favoring CEL over ARGOUML and the other two had the opposite effect.

Overall, these results are positive, exceeding our initial expectations. The previous experiences with UML modeling, the novel approach proposed by CEL, and the familiarity with standard computer interaction methodologies (and metaphors), are factors which heavily penalize the time efficiency that a new user can reach using CEL. These results point out the concrete possibility that CEL might leverage, after an initial learning phase, the modeling activity and surpass UML-based editors in terms of time efficiency.

5.3.3 Research Questions Debriefing

In this section we discuss and answer each research question individually. We support the results with evidence arisen from the interview and answers given in the debriefing questionnaire. The findings cannot be seen as statistically relevant, due to the small number of observations performed. Nevertheless, they give an initial answer to the research questions, giving us the possibility to adjust our research based on these initial findings.

RQ1: How do participants adapt to different modeling techniques which provide distinct sets of elements to create a software model?

From tasks A1 and A2 we have observed how users differently adopted the techniques proposed by CEL and by ARGOUML.

While using ARGOUML only three people sporadically employed different types of relationships, yet all the participants mainly used associations. Nobody used abstract classes nor interfaces, but this behavior could also be a direct consequence of how the systems were described (*i.e.*, very concretely). This methodology of employing UML was shared by both advanced modelers and beginners.

While modeling with CEL the people used all the elements we provided with our methodology, especially generic relationships were exploited to provide information about how entities collaborate.

All six participants were satisfied of the minimalistic approach proposed by CEL. They said that the limited number of entities and relationships were enough, exactly what they needed. Three of the subjects asked for a typing system.

Generally speaking, we noticed that people do adapt to different techniques and seem quite flexible. They are able to rapidly switch context among different modeling methodologies, without having major issues in moving from a touch-based solution to a desktop-based environment. However, techniques such as UML offer many different elements to detail a software model, but, if they are not obliged to adopt them, the users seem to employ only few of these options. Such behavior and the desire for a simple modeling technique have been confirmed also by our participants:

"In UML I spend a lot of time dealing with different relationships, but I do not need them." (P4)

"Few relationships and entities are sufficient and help me in focusing only on modeling." (P5)

From this first qualitative evaluation it seems that users are able to adapt to different modeling methodologies, but they seek simpler solutions with respect to mainstream, digital, modeling methodologies. Although ARGOUML provides many more elements than CEL, users perceived both approaches as functionally equivalent, yet they preferred our simpler methodology (as suggested by Ockham's Razor design principle [LHB03]).

RQ2: Which strategy is commonly employed by users to identify core concepts of an unknown software model and how to they adapt to different visual cues?

The strategies employed by the subjects to identify the core concepts of the systems provided in tasks B1 and B2 can be summarized as follows:

- P1 mainly focused on the internals of an entity (*i.e.*, fields and methods), while P6 mainly targeted entities referencing many relationships. Nevertheless, they both used the information gathered while looking at the entities to learn about the overall system and use this information to change the search criteria. P6 further refined this strategy by also looking at entities' names. This approach was more problematic to adopt in CEL because names are not always visible. P1 exploited the zoomable interface of CEL to rapidly switch from a global view of the system to a more local one and finally to a visualization of the internals of a particular entity.
- P2, P4 and P5 identified core components by exclusively considering the number of relationships. While using CEL P2 the subject used highlighting to focus on some of the entities which was candidates to become key elements and observed the direction of the connections following the marker. Also P5 refined the comprehension methodology while using CEL: The participant first looked at the width of the grid channels, then highlighted the entities in the neighborhood of crossing points to observe the number of connections.
- P3 used a different technique for each tool. While using CEL the subject focused on entities with numerous relationships, exploiting also the direction information (*i.e.*, the marker which moves inside the connection). When using ARGOUML, instead, the participant looked at the fields and the methods information. The subject stated that the change of methodology was due to the impossibility of rapidly understanding the direction of relationships.

In object-oriented systems core concepts are implemented in key entities, which in turn are represented by (key) classes. Tahvildari *et al.* define key classes as follows [TK04]:

"Usually, these most important concepts of a system are implemented by very few key classes, which can be characterized by a number of properties. These classes which we called key classes manage a large amount of other classes or use them in order to implement their functionality. The key classes are tightly coupled with other parts of the system."

The participants identified key entities using relationship-based or internals-based (*i.e.*, analyzing methods and fields) search techniques. As suggested by the definition, both information should be used. Nevertheless, we can state that the simplistic methodologies used by the subjects are reasonable (yet not perfect). Relying on the name of entities as done by P6, instead, can be misleading, especially in models which are not well designed. Understanding a software system is a complex operation and design a perfect methodology to find the key entities is a difficult task. What we observed in our experiment is that users prefer to use a rapid technique, being aware that it is not perfect. They prefer to learn about the system while analyzing some entities and, eventually, refine the search.

From this first evaluation we can learn that users employ simple, rapid, reasonable, yet not perfect techniques to find key entities of an unknown software model. They adapt without major issues to the user interface they are confronted with and exploit the available visual cues. In fact, except for P5, all other participants used some peculiarities of the user interface (*e.g.*, the zoomable interface of CEL or the easy access to fields and methods in ARGOUML) or visual cues (*e.g.*, the marker for indicating the direction of relationships) to enhance their basic analysis methodology. Thus, a well-designed user interface can enrich model comprehension.

5.3.4 User Feedback

The information and data we gathered to answer the research questions were valuable, yet we also collected feedback on CEL itself, trying to have feedback on our choices. In the following we discuss different important aspects of our application, reporting positive and negative points highlighted by the test subjects.

Modeling Methodology

As already discussed in Section 5.3.3, in particular answering to RQ1, the participants were particularly satisfied of our modeling approach. They claimed that it provided all the necessary means for modeling a software system, except (for three people) a typing system, which could help in refining some entities. The minimalism of the methodology was also seen as potentially beneficial in terms of comprehension speedup:

"CEL provides minimal information, which is potentially good for fast understanding." (P6)

The possibility to create relationships among methods and fields was also appreciated:

"The possibility to create relationships among methods allows to model the dynamics and the dependencies between them." (P5)

Matrix Metaphor & Zoomable User Interface

Except for P5, the matrix metaphor was said to give enough freedom while modeling. All participants, except P3, agreed that the structure provided by the matrix-based interface was beneficial. P3 desires a further way to structure the model by explicitly clustering entities together. An idea we are already developing, a tagging system, has been accepted as a valid solution. P1, P2 and P5 stated that the impossibility of directly viewing and interacting with methods and fields might slow down the modeling process: A problem that we have surely to further investigate.

The zoomable interface was a great success. All the participants claimed that it is useful to view the system as a whole, but also to concentrate, when needed, on single entities:

"I could easily have an overview of the entire system." (P1, P2)

"Intuitive zoomable interface" (P3)

P5 and P6 reported that names which are too long cannot be read directly in the matrix because they are abbreviated with dots (*i.e.*, with "... " at the end of the name). This approach forces users to zoom into the entity to read the entire name and adds tediousness to the modeling experience. We are aware of this problem and are working on a zoom-lens system which should allow users to enlarge parts of the matrix and solve the issue.

The relationships depiction and organization were judged positively:

"Great visualization of relations." (P2, P3)

"I do not have to loose time reorganizing edges." (P6)

P2 gave negative feedback on the representation of the direction of long relationships. The subject claims that the marker visualization is a good choice, yet more than one should be employed.

Interaction

Gestures were heavily experimented by all subjects. P3, P4 and P6 found them not all intuitive, yet easy to learn. P1, P2 and P5 found them intuitive.

The interactions which have been mostly appreciated were the gestures used to create entities and relationships and the methodology employed to distinguish fields and methods:

"Good creation gestures." (P4)

"Creating methods just by adding () is cool." (P6)

The navigation was also appreciated. The interaction methodologies provided by a touch-based tablet computer were effective and helped in gaining flexibility:

"Great navigation." (P2)

"Touch-based user interface with good navigation." (P4)

Some gestures were difficult to use for some subject. The double tap was often exchanged by mistake with the single tap gesture (P1, P4 and P5). P4 proposed to add a zoom-in entry in the context menu opened with single-tap, a solution that we are willing to explore. The scattered selection mechanism annoyed P4 and P6: They believe a simpler solution (*i.e.*, select a group can be done only with a single gesture) would be better. Moreover, P2 asked for a new gesture to directly center the matrix around a group entities.

P5 and P6 also reported problems while using the digitalized keyboard. They claim this issue considerably hinders typing performance. We agree with the two subjects, but we cannot directly solve this issue. We have already limited the keyboard-based interaction to the bare minimum and we are actually studying an alternative solution which involves handwriting recognition. Of course also the use of an external, physical keyboard would solve this issue (as suggested also by P6).

Unfortunately, menu-based and toolbar-based interaction have been rarely used and, thus, we do not have any concrete feedback on them yet.

5.4 Threats to Validity

In this section we present the possible threats to internal and external validity which may have affected our experiments and how we addressed them.

5.4.1 Internal Validity

The internal validity refers to the uncontrolled factors that may influence the effect of the treatments. In the following we discuss different of these factors.

Subjects

To reduce the threat of having subjects with no competence at all in the field of software modeling, we first briefly interviewed the subjects, ensuring they knew the basics of at least one modeling technique. Although 2 of them rated themselves completely new to modeling, they all knew the fundamentals of UML.

Tasks

The tasks may have been biased to the advantage of CEL. Creating a software model is the main purpose of UML editors, thus the first two tasks are, in our opinion, safe. The problem was addressed, instead, in the model comprehension part. The models presented to the users have been first designed with ARGOUML and optimized for its user interface. Afterwards we have reused the exact same disposition on CEL. Our tool was clearly penalized by this decision, because the way elements are laid out in ARGOUML is also related to the overall structure of the model (*i.e.*, to avoid overlaps and ugly layout). This is not the case in CEL, where each entity can be placed near related elements, without having to care about unrelated entities and relationships.

Baseline

The choice of a competitor may have, favored CEL: Subjects may dislike the chosen contender. Thus, participants may have been unconsciously willing to emphasize every positive aspect of CEL, or ignore defects, due to their revulsion against the competitor. To mitigate this threat, we have chosen as a contender ARGOUML, a well-known UML editors. Moreover, we kindly asked people to give feedback on each tool exclusively, without directly comparing them.

Session differences

There were six runs of the experiment and the differences among them may have influenced the results. To mitigate this threat, we annotated an initial setup and used the same configuration for each run. Before each experiment we recharged the iPad, re-uploaded a fresh installation of CEL with the needed models, reinstalled ARGOUML and rebooted the computer. We also tried to find a quiet place to run the experiment to minimize external distractions and noise.

Training

We have done only a brief introduction to both tools and this may have influenced the results. Although many subjects did not have prior experience with ARGOUML they all knew the basics of UML, which was already an advantage compared to CEL. Moreover, the novelty of our approach combined with the new device we targeted (*i.e.*, a touch-based tablet computer like the iPad) were clear disadvantages for CEL. All six subjects agreed that it was feasible to learn the basics of ARGOUML in few minutes, because it resembles many other popular desktop applications. CEL instead, has another kind of learning curve, mainly due to the particular interaction methodologies. A level of proficiency with the iPad can be reached only after different, intense, hours of use.

5.4.2 External Validity

External validity refers to the generalizability of the experiment's results. In the following we discuss different factors which are related to this subject.

Subjects

Three participants rated themselves as modeling newcomers and three as advanced or experts. We targeted this kind of balanced separation to mitigate representativeness issues. This may not have been the best way to select a representative set of subjects, yet estimating other relevant factors (*e.g.*, iPad proficiency) is hard even for the subjects themselves.

Tasks

Our choice of tasks may not reflect real world situations. A1 and A2 (*i.e.*, modeling creation tasks) are, in our opinion, well-designed, because the initial design of a software system is a procedure that is usually performed in any software development lifecycle. We added to tasks B1 and B2 (*i.e.*, model comprehension tasks) one question asking which was the overall purpose of the system, exactly to mitigate this issue and mimic a real-world situation in which a developer is confronted with an unknown model. In fact, it is more probable that the developer will be asked to work on the system (and thus she has to understand it all), instead of simply having to find some key entities.

Object systems

The representativeness of our object systems is another threat. The systems were borrowed from a university course. Although they are not trivial, they do not resemble real world systems and we are aware of that. Nevertheless, designing or understanding a real world system would have required too much time. We believe that, for an initial qualitative evaluation, the chosen systems are complex enough, especially because also beginners were involved.

Experimenter effect

One of the experimenters is also one of the authors of the approach and the developer of the tool. To avoid the threat of being unfair during the judgment of the results, we used the solutions provided during the course from which we borrowed the object systems. These solutions were detailed and furnished all the information we needed to judge the answers. We did not evaluate the created models, we only judged tasks B1 and B2, assessing whether the subject really understood or not.

5.5 Reflections

We know that this evaluation is not statistically significant and we are aware of the fact that we will need (many) other experiments to assess the quality of our approach. Nevertheless, we believe that this experiment is an excellent starting point. The subjects were enthusiastic about CEL and they claimed that our modeling technique may be superior to current mainstream modeling methodologies. The experiment was also designed to be as fair as possible, yet the fundamental differences between an iPad and a well known desktop computer may have even disadvantaged CEL. From the evaluation we were also able to gather ideas and suggestions which we plan to take into consideration in the future steps of our research. As an example, the typing system has already been implemented.

The evaluation presented in this chapter concludes the thesis. In Chapter 6 we summarize the work done so far and explain the possible research directions we are willing to explore.

Chapter 6

Conclusions

This chapter closes this document by taking a step back from what we have described so far. We first review all of our work. Subsequently, we illustrate the future work we are planning for CEL and the research directions we are willing to explore in this research.

6.1 Retrospective

Modeling is a fundamental phase of any development process. The aim of our research is to create a new approach towards software modeling, taking a step back of current, mainstream, methodologies. We aim to position ourselves in the middle-ground between lightweight, informal, and playful, modeling methodologies and heavyweight, formal, and tedious digital modeling techniques. Both worlds have advantages, which we target to provide, and different limitations, which we intend keep at bay. We review and summarize our contributions as follows:

- **An innovative modeling approach based on the essence of the object-oriented paradigm, which allows the creation of language agnostic models.**

In Section 3.1 we presented an innovative, minimalistic modeling philosophy, which employs only the few necessary elements at the base of the object-oriented paradigm. Our approach requires the knowledge of very few concepts and does not require the specification of various, unnecessary, and tedious details. We believe that the time employed in overcomplicating the software model should be used to explore different design alternatives. However, the few elements we provide are sufficient to create fully usable models.

- **A new visual metaphor designed to directly leverage the modeling activity.**

The matrix visual metaphor we presented in Section 3.2 has been designed to fully support the modeling activity, ensuring that all the layout-related problems, the organization of relationships and the repositioning issues are automatically managed by the interface itself. Moreover, such a visual representation gives a uniform view to all the models and provides an intrinsic structure to the model. The matrix can be also exploited to easily group correlated entities together, by placing them near each other.

- **A standalone application framework that can be employed to create modeling applications based on our philosophy on any platform of choice.**

We encapsulated our modeling philosophy into a standalone C++ application framework, which we presented in Section 4.2. This framework offers all the means to implement modeling applications based on our own approach. The framework is flexible and easily extendable. It also offers the possibility of collecting all the data on a model, being able to study the evolution of such artifacts.

- **An interactive modeling application, which exploits touch-based tablet computers and that is explicitly designed to support model comprehension.**

In Chapter 4 we described *CEL*, our highly interactive iPad modeling application. The tool has been constructed on top of our application framework and its user interface is based on our matrix metaphor. The application fully supports the creative nature of the modeling process, by allowing the modeling activity to take place in any setting or environment. Moreover it produced easily processable, maintainable and sharable software models. We use a semantic zoomable interface to overcome the problem related to the small screen size. This kind of interface also allows to easily move from a global overview of the model to a detailed representation of a single entity. Overall, the interface and the interaction methodologies, mainly based on gestures, have been designed to furnish an enjoyable user experience. Through different means (*e.g.*, highlighting, enlarged channels, *etc.*) our application is also aiming at supporting the model comprehension task.

As presented in Chapter 5, we also ran a first qualitative evaluation, which was precious for gathering feedback about our research. The results have been remarkable and the subjects were impressed by our approach, signaling the possibility that *CEL* may be superior to mainstream digital modeling methodologies. Moreover, in Appendix C we illustrate in detail other features of *CEL*, focusing particularly on the ones which allow users to exploit the models created with our application (*e.g.*, skeleton code export).

6.2 Future Work

We are currently finalizing the procedure to submit a first version of *CEL* to the AppStore¹, the official platform from which iPad applications can be purchased and downloaded. We will publish the tool free of charge. Establishing a user base can provide us feedback complementary to the one gathered with formal evaluations. Once *CEL* will be published we will also offer the possibility of downloading the application framework.

However, there are plenty of possibilities to further expand our research and improve our work. From a first evaluation our approach seems to be a considerable contribution for the software modeling process. We plan to run other quantitative and qualitative evaluations to gather additional feedback and statistically relevant results. In the following we describe some other research directions and further improvements we are willing to tackle in the near future.

¹<http://www.apple.com/itunes/>

Model Extensions

As our philosophy states, only few essential entities are necessary to model a software system. Nevertheless, the possibility to add complementary, visual, elements could be a valuable extension.

We would like to evaluate the use of a tagging system which allows to annotate entire groups of entities and create proper working sets within regions of the infinite 2D space. Such system may be also employed to mimic the structure given by packages/namespaces, without polluting our modeling approach.

We plan to exploit the typing system by automatically instantiating generic relationships among reference entities (*e.g.*, the creation of a field of type B in class A will lead to the automatic creation of a relationships going from A to B).

Interaction Enhancements

The user interaction methodologies are a fundamental component of CEL. To enhance the interactivity of our application, we are continuously evaluating different alternatives, such as simpler custom gestures and pen-based interaction. Handwriting recognition (mainly using a pen) is becoming more and more popular on the iPad and is a valid alternative to keyboard-based interaction. We are planning to give to users the possibility of employing both methodologies.

Another enhancement which we are currently evaluating is the possibility to allow users to directly transfer entities from one data container to another one, in order to facilitate model refactoring. Ease the refactoring of models, especially complex ones, can be extremely beneficial.

We are also currently implementing an entity search system. This is a fundamental enhancement, because developers navigating through an infinite 2D space viewed as a matrix may lose orientation within the overall landscape. Although this issue can be partially mitigated by zooming and panning the matrix, the possibility of searching an entity and instantly jumping to its position is more effective.

We also want to improve scalability by implementing *portal cells*, which allow to reference and directly visualize a related project (*i.e.*, one could separate the model and the view in two projects).

User Interface Enhancements

The user interface is a fundamental part of our application and we are evaluating different ideas to make it cleaner and simpler. We are particularly focusing our attention on the representation of parameters inside methods.

To be more compliant to iPad (and general iOS) standards, we also intend to introduce a configuration panel where the style of our tool and the different options can be managed. In this panel we will also introduce an item to directly provide feedback.

Collaborative CEL

Modeling (software) systems is a creative process, yet we do not believe that it has to be a single person activity. We want to introduce in CEL the possibility of creating projects in which multiple modelers are involved. All the actions which modify the model are signaled in real-time to all developers which are currently active. Otherwise the changes are notified once the modeler opens CEL. This solution avoids the dependency from an SCM system. The modeling framework has already been built to support collaborative projects. However, the infrastructure to be used and the changes to be applied to the user interface need further investigation.

Model Data Analysis

In Section 4.2 we presented the modeling framework, illustrating the possibility of collecting data about the modeling activity and on the evolution of the produced artifact. These data may be used as complementary information to spot potential problems in CEL. We plan to integrate an internal, automatic, data collection system. The gathered data can be adopted for personal use and can be also anonymously shared with us. We are planning to use these data to study how CEL is employed to create software models. Moreover, if we are able to collect the same information for other modeling tools we could analyze the differences to detect what is the true impact of our approach and quantify how beneficial (or not) it is for software modeling

CEL Port

Touch-based tablet computers can considerably support software modeling. They are small, can be carried anywhere and used everywhere. We are also willing to investigate, in a future step of our research, to investigate the implications, especially regarding the interaction methodologies, of porting CEL to a platform using a large-sized touch sensing surface (*e.g.*, Microsoft Surface²).

6.3 Closing the Circle

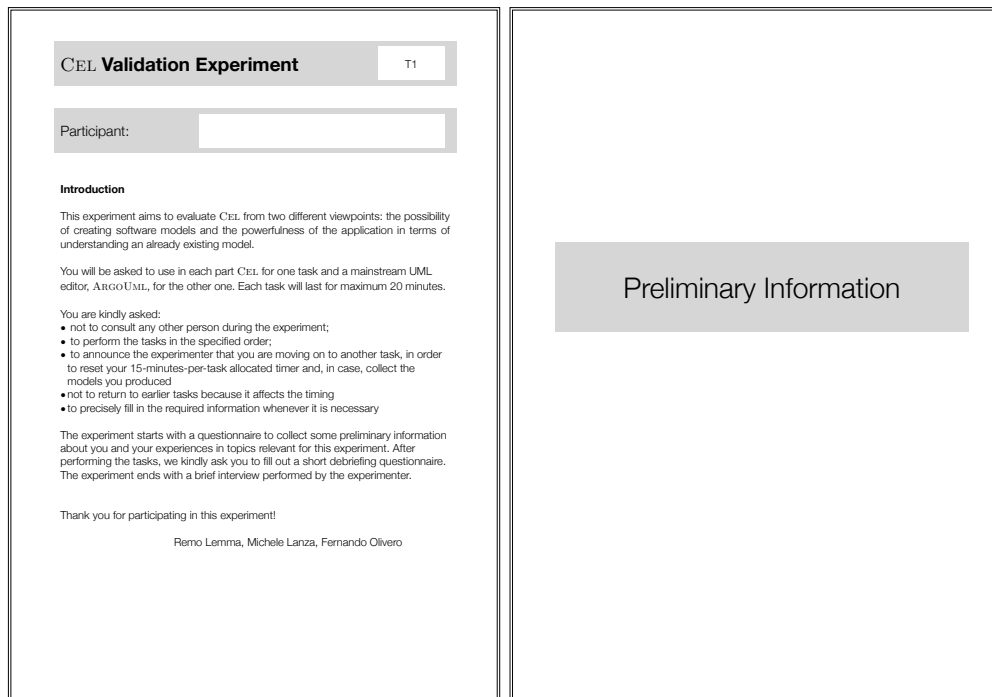
Software modeling is a fundamental activity. The evolution of this process is currently threaten by the scarce research interest in finding new alternatives to mainstream modeling methodologies, namely UML. This research has proposed a valid alternative closer to the creative essence of software modeling. We believe that our work has shown that there are plenty of possibilities to augment current modeling tools. The research on revolutionizing programming methodologies is vivid and we truly believe that the same phenomena can be applied to software modeling.

²<http://www.microsoft.com/surface>

Appendix A

Handout

The participants to our qualitative experiment were given a handout with instructions about the assignment and the tasks. In the following, an illustration of the handout is shown.



<p>Please fill out these information about you and your experience. The data will remain confidential and will be anonymized when published.</p> <p>E-mail Address:</p> <p>-----</p> <p>Gender:</p> <p><input type="checkbox"/> Male</p> <p><input type="checkbox"/> Female</p> <p>Age: (for statistical purposes only)</p> <p>----- years.</p> <p>Nationality: (for statistical purposes only)</p> <p>-----</p> <p>Affiliation: (i.e., University, Company, etc.)</p> <p>-----</p> <p>Job Position: (i.e., Developer, Project Manager, MSc Student, etc.)</p> <p>-----</p> <p>Do/Did you possess an iPad or a similar device?</p> <p><input type="checkbox"/> Yes</p> <p><input type="checkbox"/> No</p>	<p>What is your favorite mean to model software systems?</p> <p><input type="checkbox"/> Whiteboard / Paper Sketches</p> <p><input type="checkbox"/> UML Editors</p> <p><input type="checkbox"/> No modeling, I directly start programming in the IDE</p> <p><input type="checkbox"/> None of the above: ..-----</p> <p>Experience level in: (A subjective assessment of your skills)</p> <table border="1"> <thead> <tr> <th></th> <th>None</th> <th>Beginner</th> <th>Intermediate</th> <th>Advanced</th> <th>Expert</th> </tr> </thead> <tbody> <tr> <td>OOP</td> <td><input type="checkbox"/></td> <td><input type="checkbox"/></td> <td><input type="checkbox"/></td> <td><input type="checkbox"/></td> <td><input type="checkbox"/></td> </tr> <tr> <td>Software Modeling</td> <td><input type="checkbox"/></td> <td><input type="checkbox"/></td> <td><input type="checkbox"/></td> <td><input type="checkbox"/></td> <td><input type="checkbox"/></td> </tr> <tr> <td>Using ARGOUML</td> <td><input type="checkbox"/></td> <td><input type="checkbox"/></td> <td><input type="checkbox"/></td> <td><input type="checkbox"/></td> <td><input type="checkbox"/></td> </tr> <tr> <td>Using the iPad</td> <td><input type="checkbox"/></td> <td><input type="checkbox"/></td> <td><input type="checkbox"/></td> <td><input type="checkbox"/></td> <td><input type="checkbox"/></td> </tr> </tbody> </table> <p>Number of years of: (the years you spent in acquiring your experience)</p> <table border="1"> <thead> <tr> <th></th> <th>less than 1</th> <th>1 to 2</th> <th>3 to 5</th> <th>6 to 9</th> <th>10 and more</th> </tr> </thead> <tbody> <tr> <td>OOP</td> <td><input type="checkbox"/></td> <td><input type="checkbox"/></td> <td><input type="checkbox"/></td> <td><input type="checkbox"/></td> <td><input type="checkbox"/></td> </tr> <tr> <td>Software Modeling</td> <td><input type="checkbox"/></td> <td><input type="checkbox"/></td> <td><input type="checkbox"/></td> <td><input type="checkbox"/></td> <td><input type="checkbox"/></td> </tr> <tr> <td>Using ARGOUML</td> <td><input type="checkbox"/></td> <td><input type="checkbox"/></td> <td><input type="checkbox"/></td> <td><input type="checkbox"/></td> <td><input type="checkbox"/></td> </tr> <tr> <td>Using the iPad</td> <td><input type="checkbox"/></td> <td><input type="checkbox"/></td> <td><input type="checkbox"/></td> <td><input type="checkbox"/></td> <td><input type="checkbox"/></td> </tr> </tbody> </table>		None	Beginner	Intermediate	Advanced	Expert	OOP	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	Software Modeling	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	Using ARGOUML	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	Using the iPad	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>		less than 1	1 to 2	3 to 5	6 to 9	10 and more	OOP	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	Software Modeling	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	Using ARGOUML	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	Using the iPad	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
	None	Beginner	Intermediate	Advanced	Expert																																																								
OOP	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>																																																								
Software Modeling	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>																																																								
Using ARGOUML	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>																																																								
Using the iPad	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>																																																								
	less than 1	1 to 2	3 to 5	6 to 9	10 and more																																																								
OOP	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>																																																								
Software Modeling	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>																																																								
Using ARGOUML	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>																																																								
Using the iPad	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>																																																								

<p style="text-align: center;">Current Time - Notify the experimenter</p> <div style="border: 1px solid black; border-radius: 10px; width: 100px; margin: 20px auto; padding: 10px;"> <div style="display: flex; justify-content: space-around; align-items: center;"> <div style="text-align: center;"> <p>—</p> <p>hours</p> </div> <div style="text-align: center;"> <p>·</p> <p>minutes</p> </div> <div style="text-align: center;"> <p>·</p> <p>seconds</p> </div> </div> </div>	<p style="text-align: center;">Model Creation Task A1</p> <p>Given the following informal requirements and the usage scenarios, please create the software model that you think is the most appropriate. Please do not only model the class entities, but for each entity add the details (i.e., methods, fields and relationships) which you find important.</p> <p>To model this software system you are required to use CEL.</p> <p>Requirements</p> <p>Design a radio service for the coffee chain Start Bux (SB), almost the most frequented coffee chain in the country. (They have several shops around the main cities). The informal requirements given to you are the following:</p> <ul style="list-style-type: none"> - In the central office, the Start Bux DJ composes a playlist each day for the next day. Sometimes the DJ may need to add new songs to the system. - Each store daily receives the playlist from the DJ and the songs are played locally by the shop (no streaming!). Missing songs are downloaded from the system (each store maintains a cache of the most recent 500 songs played). - In each store, there is a Music Box, i.e., a smart station to sell music CD-roms. User chooses the songs and the system checks their size in minutes (does they fit with the CD size?). The user pays in cash directly at the Music Box, and the system burns the CD-rom. - The system allows one to consult the list of the last 10 songs played in the shop. <p>Scenarios</p> <p>The system must realize the following two main scenarios:</p> <p>Let's play some music The DJ uploads some new songs in the system and then creates the playlist for the next day. The day after, before the opening time, the store retrieves the playlist and updates its song cache. At the opening time the store starts playing the songs.</p> <p>Make a gift to your partner The customer enters the SB store, consults the list of the recently played songs and then composes a CD-rom. The payment process works fine and the songs are written to the CD-rom.</p>
---	---

Current Time - Notify the experimenter

hours
:
minutes
:
seconds

Task A2

Model Creation

Given the following informal requirements and the usage scenarios, please create the software model that you think is the most appropriate. Please do not only model the class entities, but for each entity add the details (i.e., methods, fields and relationships) which you find important.

To model this software system you are required to use ANTOUML.

Requirements

Design a distributed ATM (Automated Teller Machine) system.

- Customers can use the ATM from any bank to withdraw cash from their bank account.
- Each bank has its own system to deal with accounts (checking access rights, balance, etc...), that must be reused.
- Each ATM keeps a list of the transactions performed, so that banks can keep track of the amount of money they owe each other
- At the end of each day, each ATM sends a report to the banks involved in each transaction. This information is accessible from stations installed in the banks themselves by bank clerks.

Scenarios

The system must realize the following scenarios:

My bank A customer goes to an ATM of his/her bank to withdraw cash. The ATM machine itself (locally) verifies the correspondence between customer's card and PIN. The customer asks for cash, the ATM connect the bank system, check the availability on customer's account, log the operation and give cash.

Another Bank A customer goes to an ATM of a bank different from his/her own bank to withdraw cash. The ATM machine itself (locally) verifies the correspondence between customer's card and PIN. The customer asks for cash, the ATM connect the bank system, check the availability on customer's account, log the operation and give cash.

End of day reporting and back-end processing At the end of the day each ATM produces and sends reports to the other banks, one report for each bank involved in any transaction of the day. At the beginning of the day after, bank clerks consult the transaction logs of own bank.

Current Time - Notify the experimenter

hours
:
minutes
:
seconds

Task B1

Model Comprehension

Using C++L, open the *Project A* project, which includes a partial model of a software system. Please carefully answer the following questions.

Which are, in your opinion, the three key entities of the model and what do they represent?

1.
2.
3.

Which strategy did you use to identify the key entities?

.....

.....

.....

What are the main functionalities of the software system represented by this model?

.....

.....

.....

Current Time - Notify the experimenter

____ : ____ : ____
hours minutes seconds

Model Comprehension Task B2

Using ArgoUML, open the Project B project, which includes a partial model of a software system. Please carefully answer the following questions.

Which are, in your opinion, the three key entities of the model and what do they represent?

1. _____

2. _____

3. _____

Which strategy did you use to identify the key entities?

What are the main functionalities of the software system represented by this model?

Current Time - Notify the experimenter

____ : ____ : ____
hours minutes seconds

Debriefing

Please revisit your experience with CEL and ARGOUML, and share with us your thoughts by completing this table.

	Very Difficult	Difficult	Intermediate	Easy	Very Easy
Overall Usability					
CEL	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
ARGOUML	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
Model Navigation					
CEL	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
ARGOUML	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
Entities Manipulation					
CEL	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
ARGOUML	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
Modeling Entities Details					
CEL	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
ARGOUML	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
Identify Core Concepts					
CEL	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
ARGOUML	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
Model Reorganization					
CEL	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
ARGOUML	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>

Please revisit your experience with CEL and select the answer which you find the most appropriate.

The few entities types present in CEL were...

What I needed, nothing less, nothing more

Too limiting

Too many

Other:

The few relationship types present in CEL were...

What I needed, nothing less, nothing more

Too limiting

Too many

Other:

About the matrix...

It constrained too much the positioning, and gave no structure

It gave me enough freedom, but no structure

It constrained too much the positioning, yet giving structure

It gave me enough freedom and structure

Other:

The zoomable interface was...

Too confusing

Useful to concentrate on a single entity

Other:

The gesture were...

Not intuitive and difficult to learn

Not intuitive but easy to learn

Intuitive

Other:

Please rate, regardless of time pressure, how difficult it was to comprehend the specification and model the software systems presented in Tasks A1 and A2.

	Very Difficult	Difficult	Intermediate	Easy	Very Easy
Task A1: Star Bux DJ	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
Task A2: ATM System	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>

Please rate, in a general fashion, how difficult did you find the models presented in Tasks B1 and B2.

	Very Difficult	Difficult	Intermediate	Easy	Very Easy
Task B1	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
Task B2	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>

Which were the positive aspects of CEL?

Which were the negative aspects of CEL, and/or what are the improvements or new features that we should provide?

<p data-bbox="384 333 799 400">Which were the positive aspects of AncoUML?</p> <div data-bbox="384 412 799 613"></div> <p data-bbox="384 636 799 703">Which were the negative aspects of AncoUML?</p> <div data-bbox="384 714 799 916"></div>	<p data-bbox="887 333 1302 400">Enter comments and/or suggestions you may have about the experiment, which could help us improve it.</p> <div data-bbox="887 412 1302 613"></div> <p data-bbox="887 636 1302 703">Please insert any other comment/suggestion/finding that you want to share with us. Ideas, comments even critiques are always appreciated. (optional)</p> <div data-bbox="887 714 1302 916"></div>
---	---

Appendix B

Experimental Data

For the sake of transparency and repeatability, we make available the participants' answers to the two questionnaires. For space constraints we abbreviated the data, which can be anyway easily interpreted with the support of the handout presented in Appendix A.

Table B.1 and Table B.2 contain the answers to the screening questionnaire, whereas Table B.3, Table B.4, Table B.5, and Table B.6 contain the answers to the structured part of the debriefing questionnaire. We do not report the answers to the open-ended questions (*e.g.*, positive aspects of CEL, negative aspects of CEL, *etc.*), because it would be difficult to report them in a summarized form. Moreover we actually used this feedback (and cited many of the answers) in Chapter 5.

Table 16 contains the answers to the debriefing questionnaire.

Id	Treatment	Gender	Age	Country	Affiliation	Position	iPad	Favorite Mean
P1	T1	M	21	IT	USI	BSc Stud.	Yes	Whiteboard
P2	T2	M	24	CH	USI	BSc Stud.	Yes	Whiteboard
P3	T1	M	22	IT	USI	BSc Stud.	No	Whiteboard
P4	T2	M	26	CH	USI	MSc Stud.	No	Whiteboard
P5	T1	M	25	IT	USI	MSc Stud.	No	Whiteboard
P6	T2	M	30	CH	USI	MSc Stud.	Yes	Whiteboard

Table B.1. Part I of the answers to the screening questionnaire: general information

Id	OOP		Modeling		ARGO UML		iPad	
	Level	Years	Level	Years	Level	Years	Level	Years
P1	Interm.	1-2	Interm.	<1	Beginner	<1	Interm.	<1
P2	Beginner	<1	Beginner	<1	None	<1	Adv.	1-2
P3	Adv.	1-2	None	<1	None	<1	None	<1
P4	Expert	10+	Adv.	10+	None	<1	Interm.	1-2
P5	Expert	3-5	Expert	3-5	None	<1	Expert	<1
P6	Expert	3-5	Expert	3-5	Beginner	<1	Interm.	<1

Table B.2. Part II of the answers to the screening questionnaire: experience levels

Id	Overall Usability		Model Navigation		Entities Manipulation	
	CEL	ARGOUML	CEL	ARGOUML	CEL	ARGOUML
P1	Easy	Interm.	Interm.	Interm.	Easy	Easy
P2	Diff.	Very Diff.	Interm.	Easy	Easy	Very Diff.
P3	Interm.	Diff.	Easy	Diff.	Interm.	Diff.
P4	Easy	Easy	Very Easy	Very Easy	Interm.	Interm.
P5	Very Easy	Interm.	Very Easy	Diff.	Easy	Interm.
P6	Easy	Easy	Interm.	Easy	Interm.	Diff.

Table B.3. Part I of the answers to the debriefing questionnaire: experience review (1 to 3)

Id	Entities Details		Identify Concepts		Reorganization	
	CEL	ARGOUML	CEL	ARGOUML	CEL	ARGOUML
P1	Easy	Easy	Interm.	Easy	Interm.	Interm.
P2	Diff.	Very Diff.	Interm.	Diff.	Easy	Diff.
P3	Easy	Diff.	Very Easy	Very Diff.	Interm.	Easy
P4	Interm.	Very Easy	Easy	Easy	Interm.	Very Easy
P5	Interm.	Easy	Easy	Interm.	Easy	Very Diff.
P6	Diff.	Easy	Very Easy	Easy	Diff.	Diff.

Table B.4. Part II of the answers to the debriefing questionnaire: experience review (4 to 6)

Id	Entities	Relationships	Matrix	Zoomable	Gestures
				Interface	
P1	Good, but types needed	What needed	Good, except for internals	Useful	Intuitive except double-tap
P2	What needed	What needed	Freedom but no structure	Useful	Intuitive
P3	What needed	What needed	Freedom and structure	Useful	Easy to learn but not intuitive
P4	What needed	What needed	Constraints but structure	Useful	Easy to learn but not intuitive
P5	What needed	What needed	Freedom and structure	Useful	Intuitive
P6	What needed	What needed	Freedom and structure	Ok but limiting for names	Easy to learn but not intuitive

Table B.5. Part III of the answers to the debriefing questionnaire: approach feedback

Id	Task A1	Task A2	Task B1	Task B2
	P1	Interm.	Interm.	Interm.
P2	Diff.	Diff.	Diff.	Very Diff.
P3	Easy	Interm.	Easy	Diff.
P4	Easy	Easy	Easy	Easy
P5	Very Easy	Very Easy	Easy	Easy
P6	Very Easy	Interm.	Very Easy	Interm.

Table B.6. Part IV of the answers to the debriefing questionnaire: task difficulty

Appendix C

Practical C_{EL}

C_{EL} has been mainly designed to provide an interactive and simple environment to model software systems. However, while implementing the tool, we augmented its functionalities by introducing features that can be exploited with the created models. In this section we present them: the modeling import and export features.

C.1 Project Import

To import a project into C_{EL} the import button on the top toolbar of the projects index has to be used (see Section 4.4.4 for the user interface description). Once tapped, the button opens a keyboard-based view (consult Section 4.5.3 for the details on keyboard-based interactions and on the view) in which the user can input a URL which points to a file containing the description of a software model designed with C_{EL}. The file is checked for being a valid C_{EL} project. If it's the case, the project is saved on the local disk and is opened. The projects can be exported directly from C_{EL} itself, as explained in Appendix C.2.1.

Currently, the only way to import model is through the use of HTTP. Description of models attached to e-mails cannot be currently opened with C_{EL}. We plan to cover this lack shortly. However thanks to file hosting services, this problem is at least mitigated. A user can place the file in the cloud directly from the iPad (*e.g.*, DropboxApp for the iPad) and then access the public URL using C_{EL}.

C.2 Model Export

Once a software model has been created, the possibility of sharing the artifact with other people is fundamental. Especially in the case of touch-based tablet computers, which are not the primary platform employed by developers, such feature is of extreme importance. In C_{EL}, to start the export procedure, the user has simply to open a project and then tap on the export button placed on the top toolbar (see Section 4.5 for the details about these interactions).

As illustrated in Figure C.1a, we provide three different formats in which the model can be exported. Each of them will be presented in one of the subsequent sections. Once the user has chosen the appropriate format and, in case, configured the necessary options, the produced artifacts can be shared using the means shown in Figure C.1b.

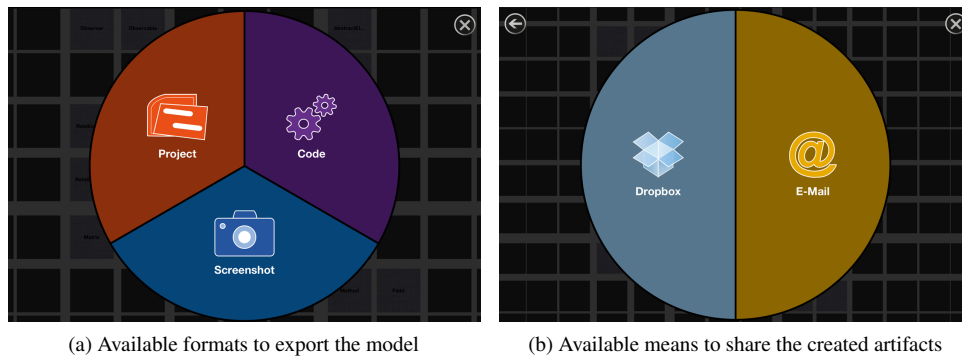


Figure C.1. Model Export UI

We support the upload of artifacts on Dropbox¹, one of the most famous and used file hosting services. The possibility of easily sharing files with different devices and users by simply uploading them in a shared folder is a simple and powerful methodology which is gaining striking popularity. We also support the possibility of attaching the exported artifacts to emails, as they are still one of the most used communication mean. However, sharing via email will work only with the Mail app² correctly configured (*i.e.*, a working email account has to be setup).

In the following sections we explain the details about the different export formats.

C.2.1 Project Export

This format consists in a file containing a JSON³-based representation of the model at hand. This file can be read by CEL itself and, in fact, the main use of this export format is to allow the import of projects into CEL (see Appendix C.1).

C.2.2 Image Export

This format captures the current state in which the project is visualized and saves it in a PNG file. The possibility of sharing (parts of) the model with its native visual representation is valuable for different reasons:

- Show an overview of the model.
- Show details about a particular part of the system while the system is still being modeled.
- Explain the model visually, by means of one or more screenshots if the iPad is not available.

The possibility to easily highlight parts of the system, showing in details the relationships, is a precious way to enrich the screenshots. Of course, being this a static representation, the dynamic behavior present in CEL (*i.e.*, the direction of the relationships) is lost.

¹<http://www.dropbox.com/>

²<http://www.apple.com/iphone/built-in-apps/mail.html>

³<http://www.json.org/>

C.2.3 Code Export

Mainstream digital modeling applications (*i.e.*, UML editors) furnish a skeleton code export feature by implementing a code generation engine. The possibility of automatically generating source code from software models is valuable. This skeleton code can be employed as a first guideline to implement the system. Moreover, using code directly generated from the model ensures, at least in the initial phases, a perfect synchrony between the model and the implementation.

Modern UML editors also implement a reverse engineering mechanism, which can help to keep code and model in sync, or to understand a system, by generating its model from the code. With our research, we target the early stages of software modeling, thus such a feature is not a priority for us. Nevertheless, its implementation may be valuable to enhance the comprehension of a system.

What we believe being crucial, to exploit the models created with CEL in real world scenarios, is a flexible and easy way to produce skeleton source code. The main issue when dealing with code generation is the possibility of supporting a multitude of programming languages. In fact, the number of available programming languages is impressive. Moreover, languages are continuously adopted and abandoned. Usually, modeling tools have a fixed set of supported languages which can be used to generate skeleton source code. However, create an ad-hoc solution for each programming language is unfeasible and supporting only few of them may limit many users. Furthermore, the creation of new languages also encourages the use of a more generic and flexible approach.

In CEL we designed a code generation engine which natively supports the addition of other languages. Programming languages are formal and are defined by an unambiguous grammar. However, to export the skeleton code based on the created software model, such grammars are not necessary: A simple definition of how the entities have to be exported is sufficient. Moreover, CEL does not target only highly experienced developers, and the creation and/or modification of formal grammars is a difficult task. Thus, we designed a language template based approach, in which programming languages can be described by simply specifying how the key entities of CEL should be exported. This methodology requires very little knowledge and, therefore, also beginners can create new language templates to satisfy their needs.

The programming languages natively supported by Cel are C++, Java and Objective-C. In the following, we explain how custom language templates can be imported into CEL, and, afterwards, we explain our approach in detail, emphasizing its flexibility.

Language Templates Import

As we already stated, CEL natively supports only few languages, but new templates can be easily imported through the use of HTTP URLs. When source code is chosen as the export format, the view depicted in Figure C.2a is shown. Tapping on one of the entries in the list will tell CEL to use the selected language to produce the skeleton source code. The import icon on the top right of the list can be used to open a keyboard-based view (see Section 4.5.3 for the details) to enter an HTTP URL. The URL content is analyzed to check whether it represents a valid language template. In the affirmative case, the template is imported and the visualization is updated to contain the newly available language. As depicted in Figure C.2b, new entries are added to the list with a different color, to signal that they are not natively supported languages. The imported languages are saved on the iPad and can be reused for every other project, unless they are deleted. To delete an imported language, the user needs to swipe on the list entry and tap the delete icon which appears when the swipe gesture is performed (see Figure C.2c).

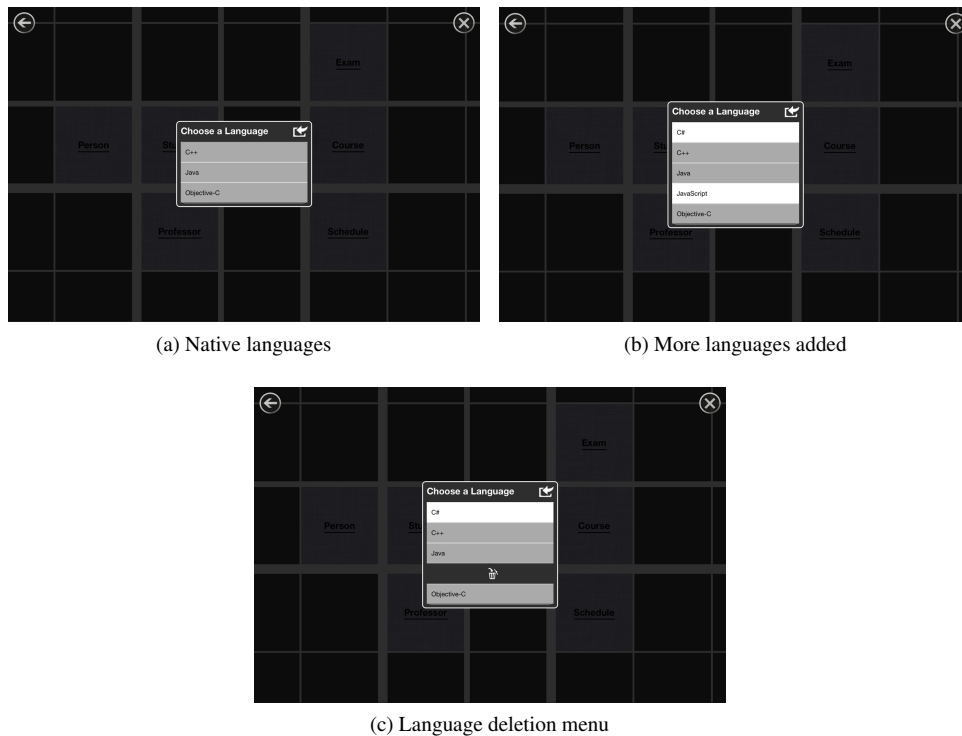


Figure C.2. Language templates import

Language Templates Analysis

In designing language templates, we aimed at equipping users with a simple and flexible format to describe any programming language.

The templates have to be expressed using the JSON format: A lightweight, text-based and open-source standard designed for human readable data interchange. We claim that such format is simple to understand and utilize. It also gives a simple structure to the data, which helps readers to comprehend better the contained information.

The templates are a mix of very few statements of source code, structural elements imposed by our design and the tags which will be substituted by CEL with the concrete values. An example of a tag is the name of a class, specified as $\{NAME\}$, in language templates. The tags should be used to define which parts of an entity should be exported and how. All the exportable entities, their eventual sub-entities (*i.e.*, entities which are specified to characterize the parent element) and all the available tags are presented in Appendix D.

To clarify all the concepts related to our methodology we show in the following a portion of two language templates: one for Java (*i.e.*, method call based syntax) and one for Objective-C (*i.e.*, message based syntax). We analyze two fragments of the templates to give more detailed information on how language templates should be built. Moreover, this comparison serves to show the flexibility and generality of our approach. A complete version of the two language templates, which should be understandable after the explanations provided in this section, can be found in Appendix D.

Listing C.1. Java method export definition

```
"method": {
  "export": "\n\t{RETURN_TYPE} {NAME} ({PARAMETER_LIST}) { }",
  "defaultReturnType": "void",
  "parameterList": {
    "*": "{PARAMETER_TYPE} {PARAMETER_NAME}",
    "separator": ", ",
    "defaultType": "Object"
  }
},
```

Listing C.2. Objective-C method export definition

```
"method": {
  "export": "\n({RETURN_TYPE}) {NAME}{PARAMETER_LIST};",
  "defaultReturnType": "void",
  "parameterList": {
    "0": "({PARAMETER_TYPE}) {PARAMETER_NAME}",
    "*": "{PARAMETER_NAME}: ({PARAMETER_TYPE}) {PARAMETER_NAME}",
    "separator": " ",
    "defaultType": "id"
  }
},
```

In Listing C.1 the Java definition for exporting a method entity is presented, whereas in Listing C.2 the definition has been adapted to support the syntax of the Objective-C programming language. We have chosen methods as an example because these elements are syntactically very different in method call based languages with respect to message based languages.

To ease the understanding of the language templates we analyze each field separately and compare, when necessary, the Java template with the Objective-C one:

- **export:** This field defines how the entity (in this case a method) has to be exported. It should usually contain the tag of the name and the parameter list. The return type should be obviously specified only if needed. The export field of Objective-C is slightly more complex because the return type has to be encapsulated in curly brackets. The export field can also contain stylistic elements, such as newlines (*i.e.*, \n), in order to create a skeleton code which is easily readable.
- **defaultReturnType:** If the return type is not present in the method entity, the value provided in this field is used. A user can specify an empty value if this field is not required.
- **parameterList:** This field defines how each parameter of a method should be exported. It is actually represented by another JSON object which contains the following fields:
 - **defaultType:** The default type which should be used for a parameter, if the entity has no type specified.
 - **separator:** This field indicated how the different parameters should be separated. In Java the parameters are separated by a comma (*i.e.*, ,). In Objective-C the parameters are actually placed inside the name, yet they should be preceded by a space. Thus we adopted the space character as a separator.

- *****, **0**, **1**, *etc.* Because parameters are represented by a list, we allowed users to fully control each element of the list and specify a different export behavior for each of them. This can be achieved by inserting a field named with the index of the element which should have a different behavior (*i.e.*, *0* or *1*). Of course these fields are used only if the specified parameter is present. The ***** field is used to encapsulate the default behavior for all the list elements which have no ad-hoc field defined. In Java each parameter is simply represented by the parameter type and the name. In Objective-C each parameter defines also a part of the name of the method. Thus we used the name of the parameter both as part of the name and as parameter. However, this behavior should affect all the parameters except the first one, for which the name part has not to be specified. Therefore, we specified a field *0* which acts differently.

While developing this approach we had to face the tradeoff between simplicity and functional richness. We chose to provide as many features as possible, at the cost of losing simplicity. We partially make amend for this problem by using the JSON format, which is well-known, structured, and human readable.

The example we presented and the subsequent explanations show the flexibility and powerfulness of our approach. Language templates are an effective mean to allow the generation of skeleton code in any language of choice. They can be easily defined without having to deal with formal grammars. Also the style of the produced skeleton code can be controlled and adapted to specific needs or tastes.

C.3 Reflections

The work we presented in this section has still to be improved and some features have to be implemented in order to allow users to exploit the full potential (*e.g.*, language template import) of our approach. However, the possibility of exporting and importing models from and into CEL are fundamental features which make the models designed with our tool usable in real world situations. Moreover, modeling is not necessarily an individual activity. The possibility of sharing the created artifacts is fundamental to favor, yet indirectly, collaborative modeling.

The chance of sharing screenshots of the model is also valuable: It allows to share information with people which may not have an iPad. This gives us the possibility to have more people adopting our visual metaphor and getting to know our modeling philosophy. Also the skeleton source code generation is a precious feature, especially because it creates the necessary point of contact between the modeling phase and the programming activity. Although these activities should be performed separately, at some time the system being designed has to be implemented.

Appendix D

Language Templates

In this appendix we describe language template in more details. First we present the different entities which are present in the templates and the tags defined by each of them. Afterwards we show two templates (one for Java and one for Objective-C) which illustrate how this approach can be flexible and might be used to export models in any programming language.

Entities & Tags

In the following, we describe all the key entities modeled in the language templates and for each of them we list the available sub-entities, the tags and their meaning.

Class Entity

This entity represents the class model entity. It contains the inheritance sub-entity which defines how the information about inheritance should be exported. The available tags, which are substituted by CEL with their actual value, are:

- **NAME:** This field represents the name of the class.
- **INHERITANCE:** This inheritance information, which are exported according to the inheritance sub-entity.
- **FIELDS:** The fields of the class, which are exported according to the field entity of the language template.
- **METHODS:** The methods of the class, which are exported according to the method entity of the language template.
- **PARENT_NAME:** The name of the parent entity, which for classes is by default not present.

Inheritance Entity

This entity is a child of the class entity and defines how the information about inheritance should be handled. The inheritance entity contains a *default* field to regulate how the default case (*i.e.*, there is at least one superclass) should be handled. Moreover, it has also to contain a *null* field, which regulates how to handle the case in which the class being exported has no superclass defined. The available tags are:

- **SUPERCLASS_LIST:** This field represents the list of potential superclasses. This field is generated according to the *superclassList* field.
- **SUPERCLASS_NAME:** This field stands for the actual name of the reference superclass.

Method Entity

The method entity symbolizes the method model entity. It contains the parameter sub-entity which defines how the parameters of the method should be treated and exported. The tags which will be searched and substituted are:

- **NAME:** This field represents the actual name of the method, with no parameters.
- **PARAMTER_LIST:** This field stands for the concatenation of all the export strings generated using the parameter sub-entity (*i.e.*, the concatenation of all parameters).
- **RETURN_TYPE:** This field denotes the return type of the method.
- **PARENT_NAME:** The name of the parent entity, which is usually a class.

Parameter Entity

This entity is responsible for exporting each parameter of a method and concatenate them into a list. The available tags are:

- **PARAMETER_NAME:** This field symbolized the name of the parameter.
- **PARAMETER_TYPE:** This field is the representation in the language template of the type of the parameter.

Field Entity

The field entity represents the field model entity. The tags which can be used to customize how fields are exported are the following:

- **NAME:** This field represents the name of the field.
- **FIELD_TYPE:** This field stands for the type of the field.
- **PARENT_NAME:** The name of the parent entity, which is usually a class.

Examples

In Listing D.1 we show the language template for Java (a method call based language), whereas in Listing D.2 we illustrate the entire language template for Objective-C (a message based language).

Listing D.1. Java Language Template

```
{
  "lang": "Java",
  "extension": ".java",
  "open_tag": "{",
  "close_tag": "}",

  "class": {
    "export":
      "class {NAME} {INHERITANCE} { {FIELDS} \n {METHODS} \n}",
    "inheritance": {
      "default": {
        "export": "extends {SUPERCLASS_LIST}",
        "superclassList": {
          "*": "{SUPERCLASS_NAME}",
          "separator": ", "
        }
      },
      "null": {
        "export": ""
      }
    }
  },

  "method": {
    "export":
      "\n\t{RETURN_TYPE} {NAME} ({PARAMETER_LIST}) { }",
    "defaultReturnType": "void",
    "parameterList": {
      "*": "{PARAMETER_TYPE} {PARAMETER_NAME}",
      "separator": ", ",
      "defaultType": "Object"
    }
  },

  "field": {
    "export": "\n\t{FIELD_TYPE} {NAME};",
    "defaultType": "Object"
  }
}
```

Listing D.2. Objective-C Language Template

```
{
  "lang": "Objective-C",
  "extension": ".h",
  "open_tag": "{",
  "close_tag": "}",

  "class": {
    "export":
      "@interface {NAME} {INHERITANCE} {\n
        @private {FIELDS} \n} {METHODS} \n@end",
    "inheritance": {
      "default": {
        "export": ": {SUPERCLASS_LIST}",
        "superclassList": {
          "*": "{SUPERCLASS_NAME}",
          "separator": ", "
        }
      },
      "null": {
        "export": ""
      }
    }
  },

  "method": {
    "export":
      "\n({RETURN_TYPE}) {NAME}{PARAMETER_LIST};",
    "defaultReturnType": "void",
    "parameterList": {
      "0": ": ({PARAMETER_TYPE}) {PARAMETER_NAME}",
      "*":
        "{PARAMETER_NAME}: ({PARAMETER_TYPE}) {PARAMETER_NAME}",
      "separator": " ",
      "defaultType": "id"
    }
  },

  "field": {
    "export": "\n\t{FIELD_TYPE} {NAME};",
    "defaultType": "id"
  }
}
```

Bibliography

- [Bar08] R.S. Barbour. *Introducing Qualitative Research: A Student's Guide to the Craft of Doing Qualitative Research*. Sage Publications, 2008.
- [BDL11] Felix Bott, Stephan Diehl, and Rainer Lutz. CREWW: collaborative requirements engineering with wii-remotes (NIER track). In *ICSE '11: Proceeding of the 33rd International Conference on Software Engineering*. ACM Request Permissions, May 2011.
- [Boe81] Barry W Boehm. *Software Engineering Economics*. Prentice Hall, 1 edition, November 1981.
- [BS97] David Bellin and Susan Suchman Simone. *The CRC Card Book*. The Addison-Wesley series in object-oriented software engineering. Addison Wesley, 1997.
- [BZR⁺10] Andrew Bragdon, Robert Zeleznik, Steven P Reiss, Suman Karumuri, William Cheung, Joshua Kaplan, Christopher Coleman, Ferdi Adeputra, LaViola, Joseph J, and LaViola, Joseph J. Code bubbles: a working set-based interface for code understanding and maintenance. In *CHI '10: Proceedings of the 28th international conference on Human factors in computing systems*. ACM Request Permissions, April 2010.
- [Che11] B X Chen. The iPad falls short as a creation tool without coding apps. *Wired Magazine*, 2011.
- [CHWS88] J. Callahan, D. Hopkins, M. Weiser, and B. Shneiderman. An empirical comparison of pie vs. linear menus. In *Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 95–100. ACM, 1988.
- [CLRS09] Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, and Clifford Stein. *Introduction to Algorithms*. The MIT Press, third edition edition, July 2009.
- [CNP⁺10] B. Chaparro, B. Nguyen, M. Phan, A. Smith, and J. Teves. Keyboard Performance: iPad versus Netbook. *Usability News*, 12(2), 2010.
- [Cor89] Thomas A Corbi. Program Understanding: Challenge for the 1990s. *IBM Systems Journal* (), 28(2):294–306, 1989.
- [Cre06] John W Creswell. *Qualitative Inquiry and Research Design: Choosing among Five Approaches*. Sage Publications, Inc, 2nd edition, December 2006.

- [CVDK07] Mauro Cherubini, Gina Venolia, Rob DeLine, and Andrew J Ko. Let's go to the whiteboard: how and why software developers use drawings. In *CHI '07: Proceedings of the SIGCHI conference on Human factors in computing systems*. ACM Request Permissions, April 2007.
- [DH07] Uri Dekel and James D Herbsleb. Notation and representation in collaborative object-oriented design: an observational study. In *OOPSLA '07: Proceedings of the 22nd annual ACM SIGPLAN conference on Object-oriented programming systems and applications*. ACM Request Permissions, October 2007.
- [DR10] Robert DeLine and Kael Rowan. Code canvas: zooming towards better development environments. In *ICSE '10: Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering*. ACM Request Permissions, May 2010.
- [EB04] Stephen H Edwards and N Dwight Barnette. Experiences using tablet PCs in a programming laboratory. In *CITC5 '04: Proceedings of the 5th conference on Information technology education*. ACM Request Permissions, October 2004.
- [Erl00] L. Erlikh. Leveraging legacy system dollars for e-business. *IT Professional*, 2(3):17–23, May 2000.
- [Fav04] Jean-Marie Favre. Foundations of Model (Driven) (Reverse) Engineering : Models - Episode I: Stories of The Fidus Papyrus and of The Solarus. *Language Engineering for Model-Driven Software Development 2004*, 2004.
- [FS00] Martin Fowler and Kendall Scott. *UML distilled - a brief guide to the Standard Object Modeling Language (2. ed.)*. Addison-Wesley-Longman, 2000.
- [GHJV99] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns. Elements of Reusable Object-oriented Software*. Addison Wesley Longman, printed in India by Eastern Press, second ise reprint edition, 1999.
- [Goe91] Vinod Goel. *Sketches of thought: a study of the role of sketching in design problem-solving and its implications for the computational theory of the mind*. PhD thesis, University of California at Berkeley, Berkeley, CA, USA, 1991.
- [Leh80] M. M. Lehman. Programs, life cycles, and laws of software evolution. *Proceedings of the IEEE*, 68(9):1060–1076, 1980.
- [LHB03] William Lidwell, Kritina Holden, and Jill Butler. *Universal Principles of Design*. Rockport Publishers, October 2003.
- [LMD10] Michele Lanza, Radu Marinescu, and S Ducasse. *Object-Oriented Metrics in Practice: Using Software Metrics to Characterize, Evaluate, and Improve the Design of Object-Oriented Systems*. Springer, softcover reprint of hardcover 1st ed. 2006 edition, December 2010.
- [LS81] Bennet P Lientz and E Burton Swanson. Problems in Application Software Maintenance. *Commun. ACM* (), 24(11):763–769, 1981.

- [MBD⁺10] Nicolas Mangano, Alex Baker, Mitch Dempsey, Emily Navarro, and André van der Hoek. Software design sketching with calico. In *ASE '10: Proceedings of the IEEE/ACM international conference on Automated software engineering*. ACM Request Permissions, September 2010.
- [McD11] Sean McDirmid. Coding at the speed of touch. In *ONWARD '11: Proceedings of the 10th SIGPLAN symposium on New ideas, new paradigms, and reflections on programming and software*. ACM Request Permissions, October 2011.
- [MNS01] Gail C Murphy, D Notkin, and K J Sullivan. Software reflexion models: bridging the gap between design and implementation. *IEEE Transactions on Software Engineering*, 27(4):364–380, April 2001.
- [MS94] Kevin Mullet and Darrell Sano. *Designing Visual Interfaces: Communication Oriented Techniques*. Prentice Hall, 1 edition, December 1994.
- [Nor83] Donald A. Norman. Design rules based on analyses of human error. *Commun. ACM* (), 26(4):254–258, 1983.
- [OLDR11] Fernando Olivero, Michele Lanza, Marco D'Ambros, and Romain Robbes. Enabling program comprehension through a visual object-focused development environment. *VL/HCC*, pages 127–134, 2011.
- [OLL10] Fernando Olivero, Michele Lanza, and Mircea Lungu. Gaucho: From Integrated Development Environments to Direct Manipulation Environments. In *Proceedings of FlexiTools 2010 (1st International Workshop on Flexible Modeling Tools)*, 2010.
- [Ous84] John K Ousterhout. Corner Stitching: A Data-Structuring Technique for VLSI Layout Tools. *IEEE Trans. on CAD of Integrated Circuits and Systems* (), 3(1):87–100, 1984.
- [Par94] David Lorge Parnas. Software Aging. In *Proceedings 16th International Conference on Software Engineering (ICSE 1994)*, pages 279–287. IEEE Computer Society, 1994.
- [Pet09] Marian Petre. Insights from expert software design practice. In *ESEC/FSE '09: Proceedings of the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering on European software engineering conference and foundations of software engineering symposium*. ACM Request Permissions, August 2009.
- [Ras00] Jef Raskin. *The Humane Interface: New Directions for Designing Interactive Systems*. Addison-Wesley Professional, April 2000.
- [Rie96] Arthur J Riel. *Object-Oriented Design Heuristics (paperback)*. Addison-Wesley Professional, 1 edition, May 1996.
- [Rus37] G.P. Rush. *Visual grouping in relation to age*. Archives of psychology. Columbia university, 1937.
- [Sin95] Ghan Bir Singh. Single versus multiple inheritance in object oriented programming. *OOPS Messenger* (), 6(1):30–39, 1995.

- [SM09] B. Sharif and J.I. Maletic. The effect of layout on the comprehension of UML class diagrams: A controlled experiment. In *Visualizing Software for Understanding and Analysis, 2009. VISSOFT 2009. 5th IEEE International Workshop on*, pages 11–18, September 2009.
- [SMU95] Randall B Smith, John Maloney, and David Ungar. The Self-4.0 user interface: manifesting a system-wide vision of concreteness, uniformity, and flexibility. In *OOPSLA '95: Proceedings of the tenth annual conference on Object-oriented programming systems, languages, and applications*. ACM Request Permissions, October 1995.
- [SSR03] Martina Schütze, Pierre Sachse, and Anne Römer. Support value of sketching in the design process. *Research in Engineering Design*, 14(2):89–97, 2003.
- [TK04] Ladan Tahvildar and Kostas Kontogiannis. Improving design quality using meta-pattern transformations: a metric-based approach: Research Articles. *Journal of Software Maintenance and Evolution: Research and Practice*, 16(4-5), July 2004.
- [TM02] Michael Terry and Elizabeth D Mynatt. Recognizing creative needs in user interface design. In *C&C '02: Proceedings of the 4th conference on Creativity & cognition*. ACM Request Permissions, October 2002.
- [TMdHF11] Nikolai Tillmann, Michal Moskal, Jonathan de Halleux, and Manuel Fahndrich. TouchDevelop: programming cloud-connected mobile devices via touchscreen. In *ONWARD '11: Proceedings of the 10th SIGPLAN symposium on New ideas, new paradigms, and reflections on programming and software*. ACM Request Permissions, October 2011.
- [Whi02] A.W. White. *The Elements of Graphic Design: Space, Unity, Page Architecture, and Type*. Allworth Press, 2002.
- [Wil95] Nancy M Wilkinson. *Using CRC Cards — An Informal Approach to Object-Oriented Development*. SIGS Publications, Inc., 1995.
- [WLR11] Richard Wettel, Michele Lanza, and Romain Robbes. Software systems as cities: a controlled experiment. In *ICSE '11: Proceeding of the 33rd International Conference on Software Engineering*. ACM Request Permissions, May 2011.
- [ZSG79] M.V. Zelkowitz, A.C. Shaw, and J.D. Gannon. *Principles of software engineering and design*. Prentice-Hall software series. Prentice-Hall, 1979.