Evolution of a Model-driven Process Framework

Wilson Pádua
Federal University of Minas Gerais
Rua dos Oitis, 288 – Morro do Chapéu
34000-000 Nova Lima – MG – Brazil
E-mail: wppf@ieee.org

***Abstract*—We discuss the evolution of Praxis, a model-driven process framework, building on feedback from educational and professional applications, along the past fifteen years. We follow the evolution from Praxis first version to the current one, discussing what was introduced in each. For past and current versions, we classify model improvements, discussing their nature and rationale, derived from received feedback.**

Keywords—process; model-driven development; framework; reuse; model transformations; CRUD transactions; persistent data

I. INTRODUCTION

According to the CMMI [1], a defined software development process "*has a maintained process description, and contributes process related experiences to the organizational process assets*". By **process framework** we define a set of artifacts which includes process descriptions and other important kinds of assets, such as reusable libraries and guidance resources.

Such framework is defined as **model-driven** when models are its core artifacts, from which others are partially or completely derived. In this work, we describe how a model-driven framework evolved along fifteen years, through improvements suggested by feedback from both educational and professional applications.

The process framework whose evolution is discussed here is Praxis, whose primary purpose is to support software engineering course projects. As such, it has been used in the last fifteen years to support teaching in software engineering courses.

Moreover, Praxis has been systematically applied and evaluated by the author himself, in industry-oriented, graduate software courses. The results of this kind of application have been discussed elsewhere ([2], [3], [4], [5]). As shown there in more detail, the students in such courses were required to develop small applications using the complete process. Typical courses comprised four software engineering disciplines, with about 30 hours each, and typical student project had a size of about 100 to 150 function points.

Praxis-Synergia, a derived process tailored to real-life projects, has been applied in the development of applications in the range of hundreds to thousands of function points. This application is performed by Synergia, a university-based software engineering laboratory which develops real-life applications under contract, mostly for government organizations ([6], [7]).

The Praxis process has evolved along those years, mostly through feedback from both course and Synergia projects. This paper describes which changes were introduced during those years, as feedback from process use was collected and analyzed.

In section 2, we discuss the goals of process and modeling improvements, proposing a classification for them. In section 3, we present the evolution of a model-driven development process, oriented to support course projects, showing the improvements performed in each version, how such improvements were suggested by feedback from its application, and which kinds of change they caused. In section 4, we discuss current work. Conclusions are drawn in section 5.

II. PROCESS IMPROVEMENTS

A. Improvement goals

A major process improvement goal is to make it more **effective**, that is, help projects to accomplish their mission within specified constraints. For real-life projects, this usually means delivering a product with a satisfactory quality level, meeting the product requirements in a provable way. However, even a fully effective process will not allow competitive development if it is not also **efficient**, accomplishing such mission within its market budget and schedule constraints. In the evolution of processes, feedback from process application leads to actions to improve both the process effectiveness and efficiency.

For educational processes, effectiveness also means exercising the knowledge and skills that their application in course projects intends to impart. Efficiency also means keeping those projects within course budgets for time and effort.

B. Process artifacts

A software development process aims to produce **executable code**, such as application code and test scripts, together with their **environmental data**, such as database schemata, test data, configuration files, localized text, graphics and other resource files.

A number of other artifacts help delivering such code and data. Some may be generated by a tool, while others may require at least partially manual derivation. Such artifacts include:

- a set of **models**, which describe both the problem to be solved and the proposed solution, using often a graphic language such as UML;
- technical **documents** and **hyper-documents** for human consumption, such as requirements specifications, manual test scripts, visual prototypes, user manuals, on-line user help and hyper-document representations of the models themselves;
- managerial artifacts, such as **plans** and **reports**;
- **logs** where data are recorded, perhaps in a partially or totally automated way, such as work, appraisal and test logs.

C. Modeling improvements

Several kinds of process improvements actions call for changes in artifacts and practices. For the purposes of this discussion, we classify them in the following kinds: **artifacts reorganization**, **transformations streamlining**, **process simplification**, **reuse enhancements** and **guidance enhancements**.

Artifacts reorganization happens when there are structural changes in the set of process artifacts; in a model-driven framework, the most important are changes in its core models. The models in the set may change, or there may be changes in their internals. A case of special interest is **model layering**, where a model or model section is split in layers, such that a layer uses only elements defined in the same layer or in a layer below. Those layers may be mirrored in derived artifacts.

Transformations streamlining may be applied to the model transformations that generate derived artifacts, other models or other sections of the same models. This includes transformations automation, but also cases where transformations remain partially or totally manual. This may happen either because the derivation requires human design or choices, or because automation may not be cost-effective, at least in a given moment. In such case, streamlining helps to perform them in a more systematic and reliable way.

Process simplification means dropping artifacts or artifact sections that do not prove to be actually useful, thus reducing process overhead and making it more agile and efficient. However, the choice of artifacts to drop requires careful analysis, in order to avoid reducing quality assurance, which would cause an increase in rework, and this might erase the agility gains.

Reuse enhancements are actions that aim to promote reuse of both models and derived artifacts. Reuse is often the best way to improve process efficiency [8]. Compared to other ways, such as reduction in process overhead, reduction in project rework and automation through use of more powerful tools, it usually requires more long term investment, but offers larger returns.

Guidance enhancements apply to supporting process artifacts, in order to improve the way developers use the models. They include enhancements to the process descriptions, but also to reference materials, such as process use guides, process standards, artifact templates, application samples and teaching aids.

III. PROCESS EVOLUTION

A. Version 1.0

Praxis version 1.0 appeared in 2000, being fully described in a textbook published in 2001 (in Portuguese). It evolved from a previous process that had been developed in the preceding years, under contract from an industrial customer. From the beginning, it had as primary goal to support course projects, following the concept of Humphrey's processes ([10], [11], [12]). Moreover, it intended to exercise key concepts present in the UML and in the SW-CMM [13]. Its process artifacts were organized as specified by the IEEE software engineering standards, 1993 edition [14].

The process structure was loosely inspired on MBase [15]; it had similarities with other MBase descendants ([16], [17], [18]), but, overall, its lifecycle model was closer to cascade than to spiral. Several IEEE-style documents were its main artifacts; analysis and design UML models were used as means to organize information that should be present in the IEEE documents for requirements, design and test; several kinds of spreadsheets were used as source for management documents. A single analysis model, written using Rational Rose, was provided as an example.

B. Version 2.0

Version 2.0 appeared in 2003, together with the second edition of the textbook. This was the first truly model-driven version, where the analysis and design models had been improved and reorganized to hold all important technical information. The IEEE documents became model derivatives; experience had shown that they tended to be hard quite hard to use and update, for the kind of small applications developed in the course.

Table 1 summarizes the enhancements introduced in this version, classifying them according to the categories introduced in Section II.C, and stating their description and rationale.

TABLE 1. MODELING IMPROVEMENTS IN VERSION 2.0

Category	Description	Rationale
Guidance enhancement	Supply of sample application	Illustrate framework use
Artifact reorganization	Standardized use of Rational Rose views	Separate modeling concerns
Artifact reorganization	Model and code layering	Separate concerns, improve reuse

The sample application provided a full analysis model and its derived requirements specification, but the design model, its derived documents and the application code were only partially implemented, since course projects specified a whole application, but did not have time to implement more than one or two functions. Support was provided for development of Java stand-alone applications with Swing user interfaces; this remained the standard environment for the following versions.

The models used Rational Rose standard views: a use case view specified functional requirements, in the analysis model, and user interface design, in the design model; a logical view modeled problem concepts and structural requirements as conceptual classes, in the analysis model, and internal design classes, in the design model. The modeling tool allowed forward and reverse engineering, between design model and application code.

A layered architecture was adopted for the logical view in the design model and the application code; it used the boundary, control, and entity layers proposed by Jacobson et al. [16], plus a persistence layer which translated persistent data between object-oriented and relational representations, and a system layer, encapsulating environment services. The process lifecycle model became closer to spiral, although the analysis model and its derived specification were still expected to be complete at the end of the second project phase.

C. *Version 2.1*

Intermediate minor versions of the process framework have the purpose of testing enhancements which, if successful, are definitively adopted in the following major version. Such intermediate versions are not fully documented in the textbook; supplementary material is supplied when they are tested. Version 2.1 introduced important changes, which were tested, evaluated and later retained in Version 3.0. Part of the course projects whose results were analyzed in the published papers ([4], [5], [6], [7]) used Version 2.1. Table 2 summarizes its enhancements.

TABLE 2. MODELING IMPROVEMENTS IN VERSION 2.1

Category	Description	Rationale
Reuse enhancement	Reuse framework for application code and design model	Ease development of common CRUD functions
Reuse enhancement	Application-independent persistence layer	Allow applications focus on problem-oriented code
Process simplification	Migration of data from documents to model attachments	Avoid duplication between model and documents
Process simplification	Migration of data from documents to spreadsheets	Collect data to provide quantitative feedback
Artifact reorganization	Requirements and design prototypes	Supplement models with visual aids
Reuse enhancement	Reuse framework test code and model	Ease test-driven development
Guidance enhancement	Guidance through a process model	Provide structured supplementary information
Process simplification	Chain of management artifacts	Streamline project management

A major difference from Version 2.0 was the introduction of a reuse framework, for artifacts related to implementation: application code, design model, and code for unit tests. This framework provided a persistence layer that used Java reflection to become completely independent from applications, following the design proposed by Ambler [19]. The reuse framework, also called Praxis, provided (mostly abstract) base classes that supported simple CRUD functions (managing persistent objects which contained primitive fields only) and CRUD with a single strong detail (managing persistent objects whose fields might contain collections of other objects with independent lifetime).This reuse framework was influenced by the experience with a similar framework in one of the first industrial applications projects.

One of the published papers ([7]) discusses in detail the benefits brought by the reuse framework. Thanks to it, it became possible to fully implement applications with at least a hundred function points, corresponding to about five CRUD functions, enough to exercise most of the techniques taught in the supported courses.

Non-UML requirements, design and test information, formerly present in the IEEE documents only, were reshaped as attachments to the analysis and design models; the corresponding documents became mere formatted reports, which might be mechanically extracted from the models. Thereafter, they were gradually dropped from the course projects. The IEEE documents remained in the sample application, however, to illustrate how they might look; the professional variant, Praxis-Synergia, automated the extraction of the IEEE requirements specification, since most clients required it as a contractual reference.

All the management artifacts became spreadsheets; IEEE documents were replaced by spreadsheets with the same content. Their focus shifted from mere fulfillment of IEEE standards, to become means for collection of useful size, work and quality data, providing quantitative feedback to process evolution.

The framework included support for the creation of low-fidelity requirements prototypes and high-fidelity design prototypes. The sample application used a spreadsheet for the requirements prototype, and a technical drawing tool for the design prototype, both generating HTML. Prototypes did not add information to the models, but provided visual feedback, especially to end users.

Test-driven development began to be used in this version, using JUnit [20] scripts to drive and test each application layer. True system tests, acting on actual user interfaces, were not used, because the then available tool used a non-standard script language. However, they were simulated, in a somewhat contrived way, by JUnit tests that exercised fields and commands in the boundary layer. The reuse framework included test script base classes containing most test procedures logic, allowing their specializations to focus on providing application-specific test data and comparing actual against expected application results.

Since the course textbook did not change for this process release, supplementary process information was supplied to the pilot classes, as an UML **process model**. This represented the process itself as use cases and classes, using Rational stereotypes for business process modeling. As with the other models, required non-UML information was supplied by model attachments.

Management artifacts were organized in a chain of derivation that started in a requirements database maintained using the Rational RequisitePro tool. It kept trace relationships from the primary requirements, expressed by analysis model

use cases and persistent classes, to derived items in both models, using the integration with Rational Rose provided by that tool. For the primary requirements, this database held also function point counts and their rationale. Extracted functional size reports fed the estimates performed by project planning spreadsheets. The planning artifacts fed project control reports, which compared expected and actual project performance.

D. Version 3.0

The third and current edition of the textbook reflected new or upgraded relevant software standards, such as UML 2.0 [21], CMMI [1], the 2003 collection of IEEE software standards [22], PMBoK [23] and SPEM 2.0 [24]; UML 2.0 and SPEM deeply affected modeling. Most of the practices of the Extreme Programming [25] agile methodology were adopted; however, models remained in the framework core, unlike XP and more like Agile Modeling [26].

The adoption of such standards in the process aimed to give Praxis users the opportunity to use in practice some of the most important standards then available. In fact, all of those standards have had only minor revisions and improvements, to this date. Table 3 summarizes its enhancements.

TABLE 3. MODELING IMPROVEMENTS IN VERSION 3.0

Category	Description	Rationale
Artifact reorganization	Models migration to Eclipse	Support UML 2.0; sunset of former tool
Guidance enhancement	Process models migration to EPF	Support SPEM; richer on-line documentation
Transformation streamlining	Rich stereotype profile	Embed more data in the models
Transformation streamlining	XML attachments	Ease transformations and visualization
Artifact reorganization	Using activities to model scenarios	Improved use case modeling
Artifact reorganization	Partitioning models into views	Matching models to process steps
Reuse enhancement	Partitioning into framework and product levels	Organize reusable elements in a framework
Artifact reorganization	Internal view layering	Separate architecture, structure and behavior
Artifact reorganization	Layering test view and code	Ease use through separation of concerns
Reuse enhancement	Migration of persistence layer to Hibernate	More powerful persistence. using free components
Process simplification	Improve chain of management artifacts	Artifact streamlining; better data quality assurance

To adopt UML 2.0, and because sunset of Rational Rose was expected, models migrated to IBM Rational Architect, embedded in Eclipse. The vendor-provided conversion tool offered limited help, since UML 2.0 brought useful new modeling facilities to, such as richer sequence diagrams to model interactions. Constructs such as selections, iterations and use references could be formally documented, and more formal definition of specialization helped to reuse collaborations and use cases.

SPEM allowed much better documentation of the process itself. Most of the process model migrated to EPF Composer [27], providing better on-line process reference. However, a smaller UML business model was kept, to document structural relationships among process concepts, not well supported by EPF, which focuses on representation of process dynamic.

UML 2.0 and the Rational Architect provided rich support for stereotype profiles. In the previous version, stereotypes had the limited purpose of providing visual representations, to enhance diagrams clarity. Now the Praxis profile was developed, allowing models to use UML tagged values (called stereotype properties, in the modeling tool) to hold much of the requirements and design information, formerly kept in model attachments.

Those properties provided readier access to both human users and tool extensions, especially when a set of scalar data was associated to a single UML element. For instance, details of I/O requirements were kept in stereotypes for boundary classes; business rules, use case preconditions and post conditions, non-functional requirements, design rules and design decisions were kept in stereotyped UML constraints; persistence requirements and design data went to stereotyped persistent classes.

On the other hand, collections of data associated to sets of UML elements were kept as attachments, since in this case the bare modeling tool was not easy to use. The tool provided an API for Java plug-ins, more convenient than the Microsoft OLE model used in the previous generation. This was used by the professional Praxis-Synergia variant, to provide visual support to more complex data extraction facilities, such as function-point counting ([9], [28]), and generation of requirements specifications and requirements and design prototypes [9].

The development of plug-ins was deemed too expensive for the educational version. Instead, much of attachments migrated to XML, using XSL style sheets to provide visual representations. Spreadsheets remained in use for management artifacts were calculations were required. In the standard version, the prototypes in the sample application were created directly in HTML, using an HTML visual editor. In other cases, such as test data and user messages, a simple conversion transformed XML attachments into Java property files, queried by the application at run-time.

Richer support and formalization of use cases allowed the replacement of the text attachments that described use cases scenarios by activity diagrams. Better formalization of use case specialization and the use of UML 2.0 elements improved use case modeling.

Praxis retained a sharp distinction between modeling **what to do** (problem specification, corresponding to the Requirements and Analysis disciplines) and **how to do it** (solution design, corresponding to Design, Test and Implementation). Such distinction is not in other model-driven, transformation-based proposals, such as AndroMDA [29], Jarzabek and Trung [30], and Mashkoor and Fernandes [31]. Indeed, Praxis models names changed to **Problem model** and **Solution model**, to emphasize this distinction. The Problem model should be technology-independent and sole source for problem complexity measures, such as function point counts.

To a given Problem model might correspond several Solution models, if several solution are developed, using different architectures and technologies.

Instead of the fixed major divisions imposed by Rational Rose, the new tool allowed partitioning the models in **views**, sections corresponding to the steps followed in the development of each function. In the Problem model, the **Requirements** view models requirements at a user-oriented, higher level, using use cases for procedural descriptions of the required functions, and constraints for business rules and non-functional requirements. The **Analysis** view uses conceptual-level classes to hold more detailed requirements, such as required I/O fields and commands, and persistence requirements; collaborations of those classes must realize the use cases in a convincing way. In the professional practice, the requirements view is built during JAD-style workshops (as described by McConnell [32]), reconciling perhaps conflicting requirements of different users, while the analysis view reflects detailed interviews conducted with individual users.

The Solution model has a **Use** view, to model the external product design, that is, its user interfaces and interactions of those with the user and among themselves (such as navigation and changes in appearance). This view expresses design decisions which must match the problem requirements, but include consideration of usability, architecture and implementation issues, for a given technology. Few other methodologies provide that kind of view, and still fewer use UML models, such as RUP-UX [33], but the use of the professional variant has proved it to be one of the most useful applications of models. In that variant, a plug-in allows automated generation of visual design prototypes from the use view, allowing prototype and model to keep synchronized.

Other Solution model views are the **Test** view, which derives from the use view a set of test model elements, from which manual and automated test scripts and data may be generated; and a **Logical** view, which represents internal application design. Forward and reverse engineering facilities provided by the modeling tool are used to keep the last two views synchronized, respectively, with test and application code.

Layering was now performed at three levels. In a first level, reuse was supported by dividing the framework into a **framework level**, containing reusable models, mostly composed by abstract classes, use cases and collaborations; reusable code libraries, matched to the test and logical views of the framework-level Solution model; and reusable artifacts such as XSD schemata, XSL style sheets and spreadsheet templates.

In a second level, each view had an **architecture** section, where requirements and design rules and decision were expressed as stereotyped constraints; a **structure** section, where those constraints were attached to classes, attributes, operations and relationships; and a **behavior** section, where those classes participated in **collaborations**, containing interactions derived from use case scenarios, in a continuous chain.

The third level was used to partition some views in layers that matched code layers. The Analysis view has boundary, control and entity class layers; the Logical view contains additional persistence and system layers.

Fully automated functional tests were introduced, using IBM Rational Functional Tester, which provides Java test scripts, tightly integrated with Eclipse. To ease the use of its somewhat complex API, and provide some degree of technology independence, the test model view and code were also layered and employed reuse. A **common** layer, shared by all kinds of tests, holds test data, represented by classes whose instances contains **test case** data, with similar test cases sharing **test entities**.

Above the common layer, test view and code split in a **black-box** layer, containing system tests, and a **gray-box** layer, containing unit tests for the application entity and control layers. Boundary layer unit tests were no longer used, since, by process guideline, this layer must handle presentation only, delegating functionalities such as field validation to the layers below. To further tame complexity and provide separation of concerns and technology independence, the black-box layer uses three layers of classes: **test inspectors** move data in and out of the user interfaces; **test checkers** compare expected and actual results; and **test procedures** implement the interactions in test collaborations.

Persistence handling underwent a significant change. It was decided to adopt Hibernate [34], instead of improving the previous specific object-to-relational translating mechanism, which had very limited capabilities. This decision was based both on the good results of Hibernate adoption in the professional version, and the acceptance of that tool by the market, especially with the JPA API, much easier and more convenient that then previous Java EJB persistence mechanism. However, to allow upper layers to retain a simple view of persistency, based on specialization of a `PersistentObject` class, JPA calls were encapsulated in in a thin façade layer that offered the same API as the layer in the previous version.

Very few changes were needed in the entity and control layers, mostly to allow for a few collateral effects of the way Hibernate handles its persistence cache. If the relational table names adhere to Hibernate defaults, a minimum of JPA persistent annotations has to be present in the application code, just to mark persistent classes and their methods which return persistent collections.

In this version, the chain of management artifacts continues to start in the requirements database, since this might also be integrated with Architect. Most changes aimed to streamline them further, while a keeping some redundancy among artifacts, to provide consistency checks for the collected data. In several cases, this redundancy allowed detection of incorrect and even faked data, which incurred heavy penalties during course projects grading.

A significant improvement was the adoption of COCOMO [35] for the estimation and planning of project work. Although somewhat old and hard to integrate with the remaining process tools, COCOMO has proved itself very useful to this date.

E. Version 3.5

TABLE 4. MODELING IMPROVEMENTS IN VERSION 3.5

Category	Description	Rationale
Transformation streamlining	Richer stereotype profile	Ease of data extraction via reports
Transformation streamlining	Match XML files to stereotypes	Systematic extraction of complex data
Reuse enhancement	Framework-level Problem model	Promote fitting requirements to reuse
Reuse enhancement	Richer CRUD patterns	Support richer variation in reuse

Version 3.5 was developed as a stepping stone to Version 4.0. This version was the first international edition of the framework: all artifacts were translated to English, as well as the user interfaces of the sample application, using the Eclipse support of Java externalized strings. Table 4 summarizes its enhancements.

The framework stereotype profile was enhanced to ease the extraction of derived artifacts. These are useful in the professional environment, easing models use by large teams of developers, many of which not highly UML-proficient, as our experience with industrial projects has shown. For the educational version, simpler data sets are extracted using BIRT, a report generator provided by Eclipse.

Generation of problem-level prototypes required more complex data extraction; for this, stereotyped properties structure match an external XML representation, for which XSL style sheets provide just-in-time HTML generation. This allowed a still manual, but very systematic matching between prototype and model.

In the previous versions, there was no framework-level Problem model, since this was much smaller and simpler that the Solution model. Crude reuse might be performed with copy-and-paste from the sample application. However, a framework-level Problem model was introduced in this version, building on the experience from the professional version. Real-life projects have suffered from low reuse that causes loss of productivity. Reuse of Problem model elements might help the requirements engineers to try and fit user-required functions into standardized framework-supported patterns, allowing the provider to charge less for those functions.

In the educational version, it is expected that this would guide the students to fit the requirements proposed for their projects into reuse patterns. In past projects, sometimes students found out that their originally proposed functions were too hard to implement, using the existing patterns. In such cases, they were allowed to change the requirements, but some time and work had already been wasted before they realized this.

In most cases, implementation difficulties were found with functions that did not fit the patterns present in the Solution model and code; that is, CRUD with a single strong detail. For other kinds of detail, the sample application provided some example functions, but adapting them was much more difficult than reusing the framework. The current version provides support for multiple detail collections, including weak ones (data whose lifetime is bound by the master instance lifetime). Handling such collections builds on UML parameterized classes, in the Solution model, and Java generics, in the code.

IV. CURRENT STATUS

A. Version 4.0

Currently, Version 4.0 is under development. This version is not very different from Version 3.5, perhaps reflecting stability of the framework. Most improvements were minor; Table 5 shows the few major ones.

TABLE 5. MODELING IMPROVEMENTS IN VERSION 4.0

Category	Description	Rationale
Reuse enhancement	Simplified boundary layer as prototype	Design prototype becomes way to boundary development
Artifacts reorganization	Support for Selenium tests	System test more similar to unit test, in open-source environment
Guidance enhancements	Support for Vaadin boundary layer	Many web-oriented applications

For solution-level prototypes, it is possible to use HTML prototypes, as done for problem-level. However, it was found in former versions that a significant amount of Javascript code is needed to have a prototype with significant behavior. Reuse of prototype common elements reduces the amount of Javascript needed for each new prototype, but a different solution was tried and accepted for this version.

Several difficulties found with the IBM Rational Functional Tester tool prompted us to go back to Junit-based system tests. By now, system-level testing in a JUnit environment had become much more powerful with the appearance of tools such as Selenium [36], supplemented by JUnitParams [37].

With our new approach, a preliminary version of the boundary layer is used as the solution-level prototype. Only minor modifications are then needed to change that into the definitive code. Some of the differences between the two versions correspond to differences between actual and simulated control and entity layers; some stem from validity checks that are too heavy for prototypes; some correspond to features that only the actual version allows being thoroughly tested; and some are code optimizations that should be done in the final code only. Differences usually amount to less than 5% of the code.

Also, an additional version of the boundary layer infrastructure was provided to support web-based applications that use Vaadin [38] components. This adds to the existing support for local, Swing-based interface components.

B. Some examples

In a Problem model, Fig. 1 shows the scenarios for a complex use in an application. However, no specific event flow steps are necessary for any flow. These are all described in the CRUD abstract use case, which the Program Management concrete use case specializes, as shown in Fig. 2. Only stereotyped properties need be instanced; an example for a

scenario is shown in Fig. 3. In this case, the scenario instantiates the event flow shown in Fig. 4.

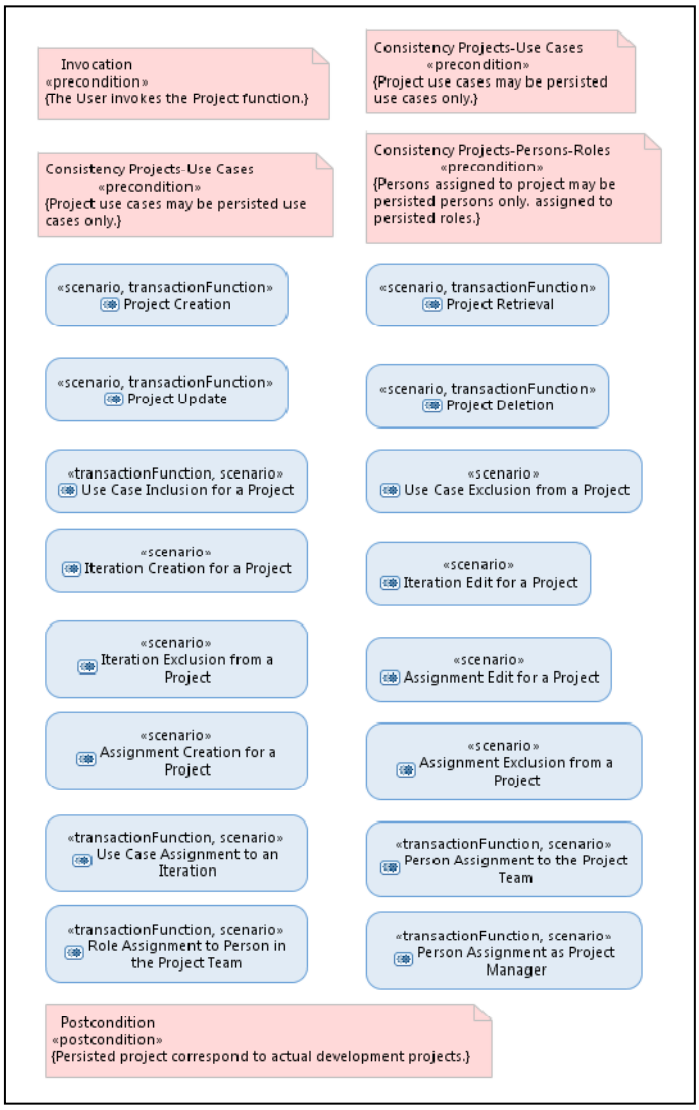

Fig. 1 Requirements for a use case

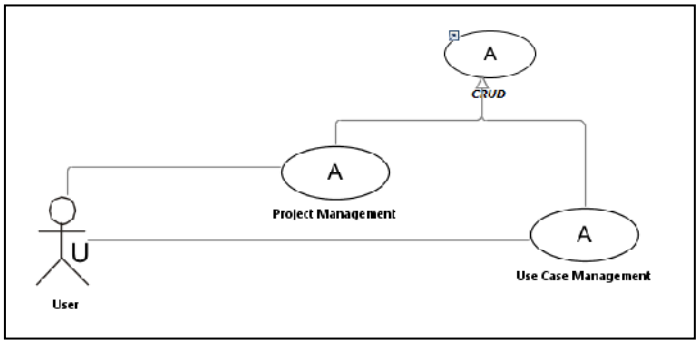

Fig. 2 Sample use cases

Property	Value
scenario	
changeDescription	Revision of tool purposes within intended toot suite.
changeStatus	1 - Added
developmentStatus	7 - Validated
scenarioKind	2 - Subflow
stability	0 - Low
transactionFunction	
DET	Person name and e-mail.
FTR	Project, Person
functionKind	2 - EQ
NDET	2
NFTR	2

Fig. 3 Stereotype properties for a scenario

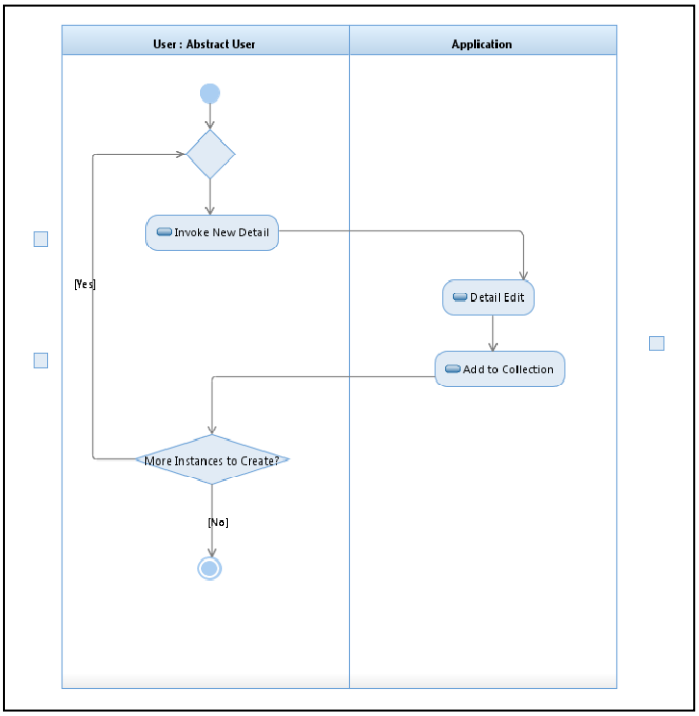

Fig. 4 A reusable scenario

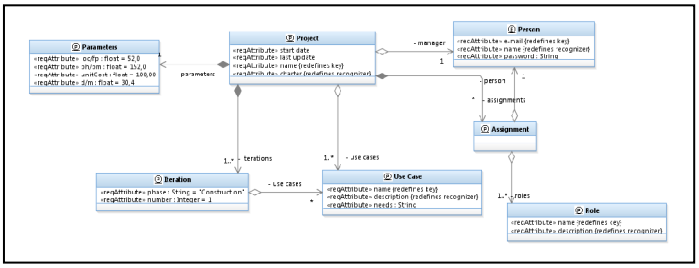

Fig. 5 Problem model entity layer

The Problem model is completed by an Analysis view; samples for the entity and boundary layer are shown in Fig.5 and Fig. 6, respectively. For the entity layer, all shown classes inherit from a `PersistentEntity` framework class, which means they represent information persisted in the database.

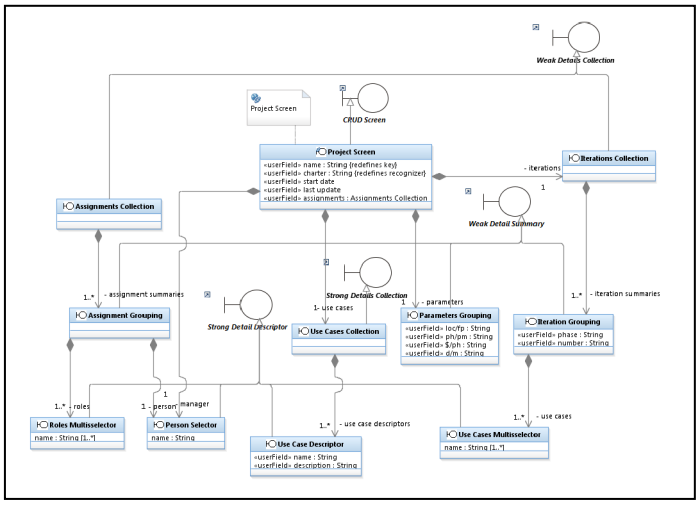

Fig. 6 Problem model boundary layer

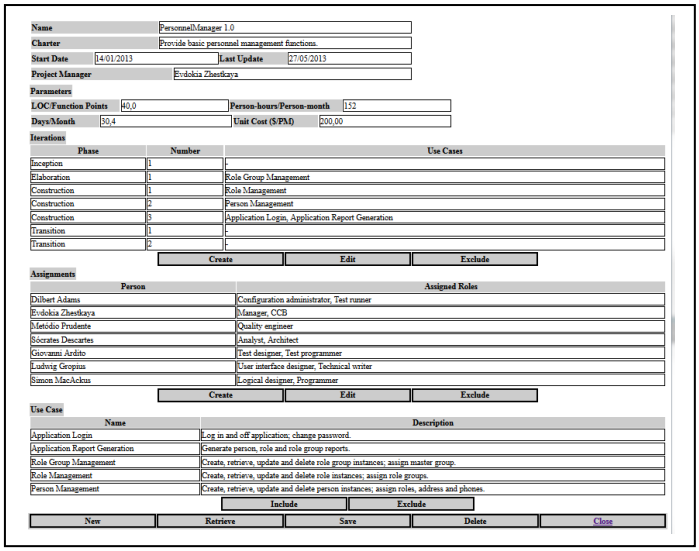

Fig. 7 Application interface prototype

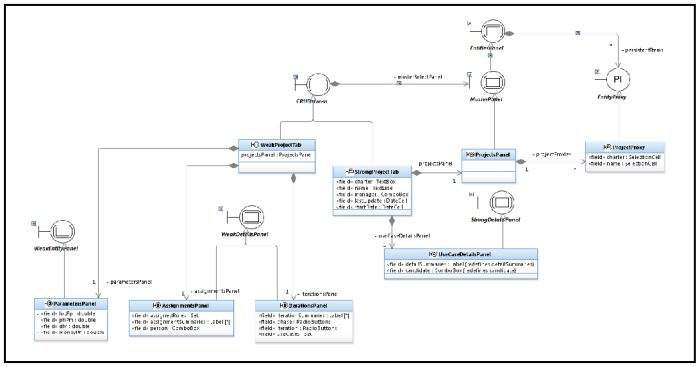

Fig. 8 The Use view of the Solution model

Fig. 7 shows the prototype that corresponds to the model in Fig. 6. In the Solution model, the Use view, shown in Fig. 8. This presents a view of the user interface that is closer to reality than the Problem model view shown in Fig. 6.

Design rules

Group	Name	Body	Level	Rationale	Remarks
User errors handling					
	Undo and redo	Undo and redo facilities will not be provided.	Application	Architectural simplification.	Non-commited transactions may aborted via the Cancel comman Commited transactions should b undone and redone manually, or, in worst cases, the database shou be restored.
User help handling					
	On-line user help	On-line help documentation shall be provided as hypertext.	Application	Easy to build and to use.	Standard presentation is a single-page HTML.
	User interface customization	User interface customization facilities will not be provided.	Application	Architectural simplification.	Browser customizations availabl as usual.
	User skill levels	User interface will not be differentiated according to user skill levels.	Application	Architectural simplification.	Knowledge of Praxis project artifact conventions should be necessary and sufficient.

Design decisions

Group	Name	Body	Level	Rationale	Tactics	Remarks
Decisions						
	Use Case Management use collaboration	The Use Case Management use collaboration shall specialize the CRUD use collaboration.	Application	Reuse of external design.	Modifiability: maintaining semantic coherence; information hiding; mediation; configuration files.	No interactions shall be redefined or added.
	Project Management use collaboration	The Project Management use collaboration shall specialize the CRUD use collaboration.	Application	Reuse of external design.	Modifiability: maintaining semantic coherence; information hiding; mediation; configuration files.	No interactions shall be redefined or added.

Fig. 9 Design rules and decisions

Fig. 9 shows an example of the documented design rules and decisions that are attached to the Use view.

Number	Identification	Description	Kind	Test data	Expected data	Required data	Forbidden data	Expected message	Key expected	System test
1	INPrR	Invalid Name Project Retrieval	%none%	INPr	-	-	-	PROJECTS - INVALID NAME	false	false
2	NEPrR	Non-existing Project Retrieval	%none%	NPPr	-	-	-	PROJECTS - NON-EXISTING PROJECT	false	false
3	PPrR	Persistent Project Retrieval	%none%	PPr	PPr	-	-	%none%	false	false
4	PrRL	Projects Retrieval and Listing	C-PERL	PPr	-	PPr	-	%none%	true	true
5	PrCIN	Project Creation with Invalid Name	C-IEC	INPr	-	-	-	PROJECTS - INVALID NAME	false	true
6	PrCEN	Project Creation with Existing Name	C-EEC	ENPr	-	PPr	-	PROJECTS - EXISTING NAME	false	true
7	PrCIC	Project Creation with Invalid Charter	C-IEC	ICPr	-	-	-	PROJECTS - INVALID CHARTER	false	true
8	PrCILU	Project Creation with Invalid Last Update	C-IEC	ILUPr	-	-	-	PROJECTS - INVALID LAST UPDATE	false	true
9	PrC0PM	Project Creation without a Project Manager	C-IEC	Pr0PM	-	-	-	PROJECTS - NO PROJECT MANAGER	false	true
10	PrC0UC	Project Creation without Use Cases	C-IEC	Pr0UC	-	-	-	PROJECTS - NO USE CASES	false	true
11	PrC0It	Project Creation without Iterations	C-IEC	Pr0It	-	-	-	PROJECTS - NO ITERATIONS	false	true
12	PrC0As	Project Creation without Assignments	C-IEC	Pr0As	-	-	-	PROJECTS - NO ASSIGNMENTS	false	true
13	PrCILF	Project Creation with Invalid LOC/FP	C-IEC	ILFPr	-	-	-	PROJECTS - INVALID PARAMETER LOC/FP	false	true
14	PrCIPP	Project Creation with Invalid PF/PM	C-IEC	IPPPr	-	-	-	PROJECTS - INVALID PARAMETER PF/PM	false	true
15	PrCIDM	Project Creation with Invalid D/M	C-IEC	IDMPr	-	-	-	PROJECTS - INVALID PARAMETER D/M	false	true
16	PrCISP	Project Creation with Invalid $/PH	C-IEC	PPr	-	-	-	PROJECTS - INVALID PARAMETER $/PH	false	true
17	VNPrC	Valid Non-existing Project Creation	C-VNEC	NPPr	NPPr	-	NPPr	%none%	true	true
18	PrUIN	Project Update with Invalid Name	C-IEU	INPr	-	NPPr	-	PROJECTS - INVALID NAME	false	true
19	PrUEN	Project Update with Existing Name	C-IEU	ENPr	-	PPr	-	PROJECTS - EXISTING NAME	true	true
20	PrUID	Project Update with Invalid Charter	C-IEU	ICPr	-	NPPr	-	PROJECTS - INVALID CHARTER	false	true
21	PrUILU	Project Update with Invalid Last Update	C-IEC	ILUPr	-	NPPr	-	PROJECTS - INVALID LAST UPDATE	false	false
22	PrU0PM	Project Update without a Project Manager	C-IEC	Pr0PM	-	NPPr	-	PROJECTS - NO PROJECT MANAGER	false	false
23	PrU0UC	Project Update without Use Cases	C-IEC	Pr0UC	-	NPPr	-	PROJECTS - NO USE CASES	false	true
24	PrU0It	Project Update without Iterations	C-IEC	Pr0It	-	NPPr	-	PROJECTS - NO ITERATIONS	false	true
25	PrU0As	Project Update without Assignments	C-IEC	Pr0As	-	NPPr	-	PROJECTS - NO ASSIGNMENTS	false	true
26	PrU	Project Update	C-VEEU	UPPr	UPPr	NPPr	UPPr	%none%	false	true
27	PrD	Project Deletion	C-PED	UPPr	-	UPUC	-	%none%	false	true
28	NEPrD	Non-existing Project Deletion	%none%	NPPr	-	-	NPPr	PROJECTS - NON-EXISTING PROJECT	false	false

Fig. 10 Test cases

Number	Identification	Description	Scalar fields	Weak entities	Strong entities	Weak collections	Strong collections
1	PPr	Persistent project.	PersonnelManager 1.0 Provide basic personnel management functions. 14/01/2013 27/05/2013	VPa	PP4	VI,VI1,VI2,VI3,VI4,VI5,VI6 VA,VA1,VA2,VA4,VA5,VA6	PUC,PUC1,PUC2,PUC3,PUC4
2	PPr1	Persisted project 1.	Translator 1.0 Translate a text file, looking up strings in a dictionary. 09/07/2013 06/10/2013	VPa1	PP3	VI1,VI12,VI13,VI14,VI15 VA7,VA8,VA9,VA10,VA11,VA12,VA13,VA14	PUC5,PUC6
3	NPPr	Non-persisted project.	ProjectManager 1.0 Provide support for managing basic project data. 11/03/2014 04/07/2014	VPa2	PP3	VI21,VI22,VI23,VI24,VI26,VI27 VA7,VA15,VA16,VA17	PUC7,PUC8,PUC1,PUC2
4	INPr	Project with invalid name.	;#$=-§ Provide support for managing basic project data. 11/03/2014 04/07/2014	VPa2	PP3	VI21,VI22,VI23,VI24,VI26,VI27 VA7,VA15,VA16,VA17	PUC7,PUC8,PUC1,PUC2

Fig. 11 Test entities

Fig. 10 shows an attachment to the Test view that displays the test cases that a system test must perform for validation. The data used by the tests are referred in tables, a small part of

which is shown in Fig. 11. The data shown here describe sample projects, whose details are defined by other described in the same way, which are part of each project collections. Strong collections are those that survive the project existence and are therefore managed in other pages, while weak collections do not and are wholly managed within the project.

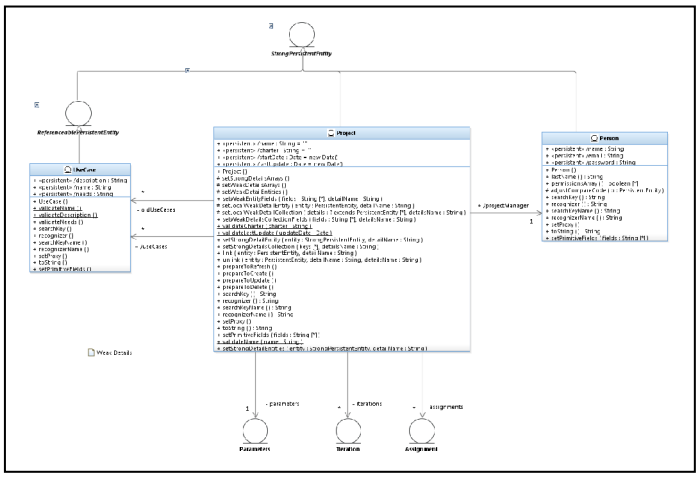

Fig. 12 Logical model – strong entities

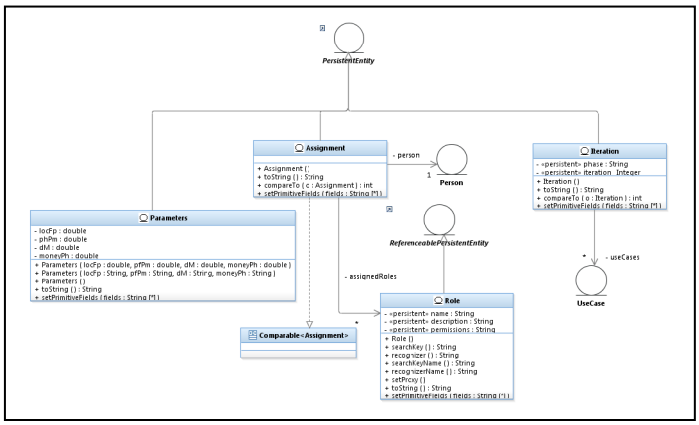

Fig. 13 Logical model – weak entities

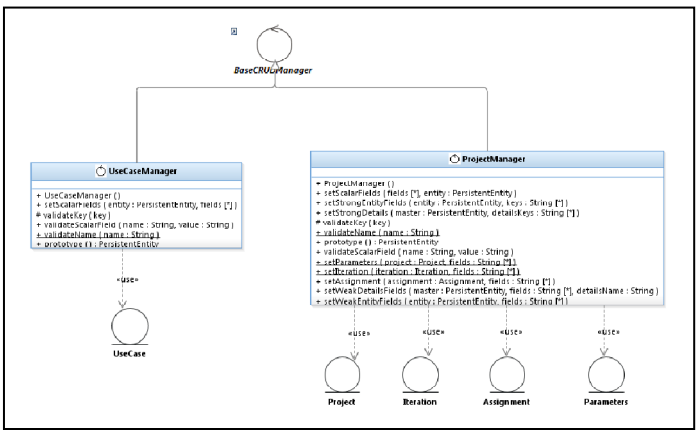

Fig. 14 Logical model – control

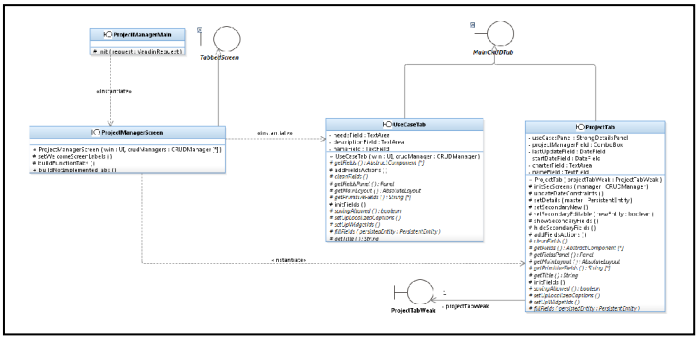

Fig. 15 Logical model – boundary (screens)

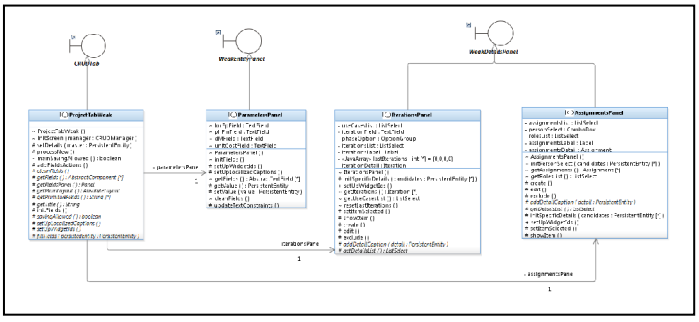

Fig. 16 Logical model – boundary (fields)

Fig. 12 to Fig. 16 illustrate the resulting application logical model, which is organized into entity, control and boundary layers. Fig. 17 and Fig. 18 show views of one selected project information. Weak details are shown in a separate page, because their information is very extensive.

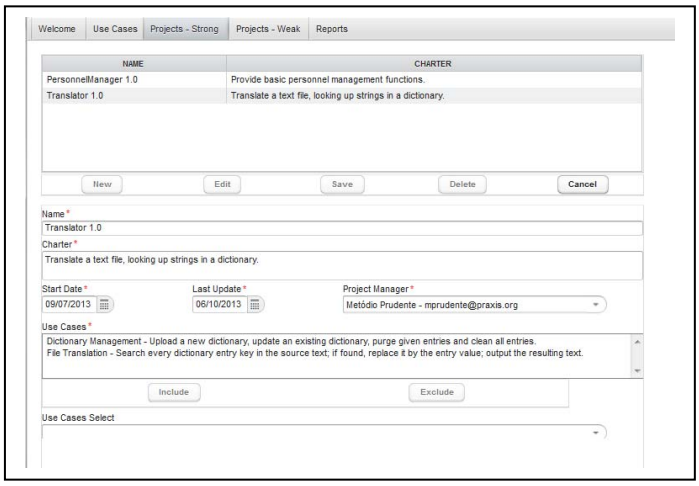

Fig. 17 Executed program – main projects page

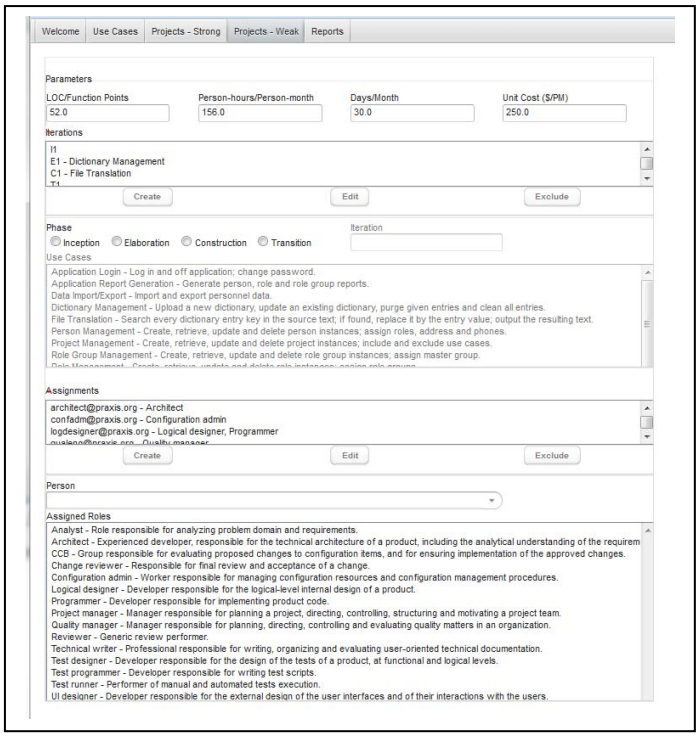

Fig. 18 Executed program – weak details page

V. *FUTURE WORK AND CONCLUSIONS*

During the years where this author lectured practice-oriented courses using Praxis as the process for writing course applications, it was possible to experiment with its use as a process to develop course applications. Currently, as the author has ceased to teach regular courses, a fourth edition of the Brazilian textbook is being written, consolidating in Praxis 4.0 what was learned in its fifteen years of development.

In the last ten years, agile methods became increasingly used, and our professional environment was no exception. UML proficiency has remained a rare asset in the professional market, and Synergia has had to face such reality.

Currently, Synergia demand has switched to a number of smaller projects, together with maintenance of the old projects; the oldest Synergia project is still maintained and updated, after fifteen years of use. Therefore, Synergia has switched to a current process mostly based on the Scrum agile practices [39], together with using Kanban [40] for maintenance projects.

It is intended to extend the Praxis family with an agile variation, which should profit from such experience. In such a version, information contained in the stereotyped properties should be held in spreadsheets, equivalent to those currently extracted from the models by BIRT.

The evolution of the Praxis process and framework was mostly driven by feedback from both course projects and real-life systems. This aligns to a major goal of Synergia: promotion of synergy between academic research, practice-oriented education, and real-life software development, reflected in its very name.

ACKNOWLEDGMENTS

We thank IBM Rational for supporting this work, within the IBM Academic Initiative.

REFERENCES

[1] CMMI Product Team, *CMMI for Development, Version 1.3*, CMU/SEI-2010-TR-033, Software Engineering Institute, Nov. 2010, available at www.sei.cmu.edu/library/abstracts/reports/10tr033.cfm.

[2] W. Pádua, "A Software Process for Time-constrained Course Projects", Proc. 28th International Conference on Software Engineering (ICSE '06), IEEE Press, May 2006, pp. 707-710, Shanghai – China, doi.acm.org/10.1145/1134285.1134397.

[3] W. Pádua, "Using Model-Driven Development in Time-Constrained Course Projects", Proc. 20th Conference on Software Engineering Education and Training (CSEET '07), IEEE Press, pp. 133-140, Dublin, Ireland, Jul. 2007, doi.acm.org/10.1109/CSEET.2007.55.

[4] W. Pádua, "Using Quality Audits to Assess Software Course Projects", 22th Conference on Software Engineering Education and Training, pp. 162-165, Hyderabad, India, Feb. 2009, doi.acm.org/10.1109/CSEET.2009.12.

[5] W. Pádua, "Measuring complexity, effectiveness and efficiency in software course projects", Proc. 28th International Conference on Software Engineering (ICSE '10), IEEE Press, May 2010, vol.1, pp. 545-554, doi.acm.org/10.1145/1806799.1806878 .

[6] B. Pimentel, W. Pádua, C. Pádua, and F. T. Machado, "Synergia: a software engineering laboratory to bridge the gap between university and industry", Proceedings of the 2006 international workshop on Summit on software engineering education (SSEE '06), ACM, New York, NY, USA, pp. 21-24, doi.acm.org/10.1145/1137842.1137850.

[7] V. A. Batista, D. C. C. Peixoto, W. Pádua and C. I. P. S. Pádua, "Using UML Stereotypes to Support the Requirement Engineering: a Case Study", Proceedings of the 2012 International Conference on Computational Science and Its Applications – ICCSA 2012, Salvador, Brazil, 2012.

[8] M. L. Griss, "Software reuse architecture, process, and organization for business success", Proceedings of the Eighth Israeli Conference on Computer Systems and Software Engineering, IEEE Computer Society, Jun 1997, pp. 86 – 89, doi.acm.org/10.1109/ICCSSE.1997.599869.

[9] C. Jones, Assessment and Control of Software Risks, Yourdon Press – Prentice-Hall, 1994.

[10] Watts S. Humphrey, A Discipline for Software Engineering, Addison-Wesley, 1995.

[11] Watts S. Humphrey, Introduction to the Personal Software Process, Addison-Wesley, 1997

[12] Watts S. Humphrey, Introduction to the Team Software Process, Addison-Wesley, 1999.

[13] M. C. Paulk, B. Curtis, M. B.h Chrissis and C. V. Weber, Capability Maturity Model for Software, Version 1.1, CMU/SEI-93-TR-24, Software Engineering Institute, Feb. 1993.

[14] IEEE, IEEE Standards Collection – Software Engineering, IEEE, New York – NY, 1994.

[15] B. Boehm, "Anchoring the Software Process", IEEE Software 13(4), Jul. 1996.

[16] I. Jacobson, J. Rumbaugh and G. Booch, The Unified Software Development Process, Addison-Wesley, 1999.

[17] S. W. Ambler, J. Nalbone and M. J. Vizdos, The Enterprise Unified Process: Extending the Rational Unified Process, Prentice Hall, 2005.

[18] Ph. Kruchten, The Rational Unified Process - An Introduction, 2nd Edition, Addison-Wesley, 2003

[19] S. W. Ambler, The Design of a Robust Persistence Layer for Relational Databases, An AmbySoft White Paper, available at www.ambysoft.com/downloads/persistenceLayer.pdf, Jun. 2005.

[20] P. Tahchiev, F. Leme, V. Massol, and G. Gregory, JUnit in Action, Second Edition, Manning Publications, Jul. 2010.

[21] OMG, Unified Modeling Language: Superstructure, version 2.1.2, Formal/2007-11-02, Nov. 2007, available at www.omg.org/spec/UML/2.1.2/Superstructure/PDF.

[22] IEEE, IEEE Software Engineering Collection on CD-ROM, IEEE, New York – NY, 2003.

[23] Project Management Institute, A Guide to the Project Management Body of Knowledge (PMBOK® Guide), Third Edition, 2004.

[24] OMG, Software & Systems Process Engineering Meta-Model Specification, v2.0, Formal/2008-04-01, Abr. 2008, available at www.omg.org/spec/SPEM/2.0/PDF.

[25] K.Beck, Extreme Programming Explained: Embrace Change, 2nd Edition, Addison-Wesley, 2004.

[26] S. W. Ambler, "Agile Model Driven Development Is Good Enough", IEEE Software 20(5), 70 – 73, Sep. 2003.

[27] P. Haumer, "Increasing Development Knowledge with EPF Composer", Eclipse Review 1(2), Spring 2006, available at www.haumer.net/paper/EPFC-eclipsereview.pdf.

[28] D. Garmus and D. Herron, Function Point Analysis: Measurement Practices for Successful Software Projects, Addison-Wesley, 2000.

[29] AndroMDA, Generate components quickly with AndroMDA, available at www.andromda.org.

[30] S. Jarzabek and H. D. Trung, "Flexible generators for software reuse and evolution", Proceeding of the 33rd International Conference on Software Engineering (ICSE '11), ACM, New York, NY, USA, 920-923, doi.acm.org/10.1145/1985793.1985946.

[31] A. Mashkoor and J. M. Fernandes, "Deriving Software Architectures for CRUD Applications", The FPL Tower Interface Case Study, Proceedings of the International Conference on Software Engineering Advances (ICSEA '07), IEEE Computer Society, Washington, DC, USA, doi.acm.org/10.1109/ICSEA.2007.25.

[32] S. McConnell, Rapid Development: Taming Wild Software Schedules, Microsoft Press, 1996.

[33] J. Heumann, "User experience storyboards: Building better UIs with RUP, UML, and use cases", The Rational Edge, Nov. 2003.

[34] Ch. Bauer, G. King, Java Persistence with Hibernate, Manning Publications, Dec. 2006.

[35] B. Boehm, C. Abts, A. W. Brown and S. Chulani, Software Cost Estimation with Cocomo II, Addison-Wesley, 2000.

[36] Selenium. "Selenium HQ Browser Automation", available at www.seleniumhq.org.

[37] GitHub. "Pragmatists/JUnitParams", available at github.com/Pragmatists/junitparams.

[38] Vaadin, "Thinking of U and I", available at vaadin.com

[39] K. Schwaber and J.Sutherland. "The Scrum Guide", available at hwww.scrumguides.org/scrum-guide.html .

[40] J. Hurtado. "Open Kanban - An Open Source, Ultra Light, Agile & Lean Method", available at agilelion.com/agile-kanban-cafe/open-kanban