

# A Rapid Development Process with UML

Giuliano Armano

DIEE, Dipartimento di Ingegneria Elettrica ed Elettronica,  
University of Cagliari  
Piazza d'Armi  
I-09123, Cagliari (Italy)  
Tel. +39-70-675.5878

armano@diee.unica.it

Michele Marchesi

DIEE, Dipartimento di Ingegneria Elettrica ed Elettronica,  
University of Cagliari  
Piazza d'Armi  
I-09123, Cagliari (Italy)  
Tel. +39-70-675.5899

michele@diee.unica.it

## ABSTRACT

Since "design-oriented" life-cycles came to their maturity, dramatic changes have been introduced as far as programming tools and computer hardware are concerned. Such changes made it possible to develop applications focusing on refactoring rather than on analysis and design. The underlying hypothesis is that by adopting suitable tools and target languages, refactoring would possibly cost less than the overhead introduced by modern A&D techniques. Recently, extreme programming has been proposed as an alternative to a "design-oriented" life-cycle.

In this paper we describe a software application developed using a software life-cycle that basically follows the guidelines suggested by extreme programming. Such an approach requires highly expressive programming languages and powerful CASE tools. UML has been selected as the underlying modeling language throughout the whole process, for it incorporates well-known diagrams for describing a software application from different perspectives. Smalltalk has been selected as target language, as it allows fast prototyping and early delivery. We claim that, for small and medium-sized projects, a life-cycle based on refactoring and supported by suitable languages and tools allows team productivity to be greatly enhanced.

## Keywords

Software Life-Cycle, Object-Oriented Analysis and Design, Minimal Methodologies, Extreme Programming, Refactoring.

## 1. INTRODUCTION

Object-oriented technology is extensively used in the field of software engineering. In particular, it appears to be the most suitable approach to perform conceptualization, analysis, design and implementation, all within a common framework. Up to 1997 more than 50 methods have been proposed and used to support object-oriented life-cycle software development (e.g., Booch [2] OOSE [7], OMT [11], Shlaer-Mellor [13], Wirfs-Brock [14], Coad-Yourdon [4]).

The standardization of the well known Unified Modeling Language ([3]) has dramatically changed the scope in object-oriented software development. In fact, the introduction of a standard language for describing

information, functional, and control models within the framework of object-oriented technology has made it possible to tackle software development issues from a common perspective. UML does not incorporate any particular process; however it encourages a process based on use cases [7], centered upon the architecture, iterative and, possibly, incremental.

Since UML has been standardized by OMG, their authors have been attempting to define a process general enough to be applied in most applications. This work leads to the definition of the so-called Unified Process <sup>1</sup> ([8], [9]). UP aims to apply software "best-practices" starting from the concept of "software generation".

According to UP, a software product is created in an initial development cycle, and it will evolve into its next generation by repeating a sequence of four phases: inception, elaboration, construction, and transition. According to the proposal of Boehm [1], each phase defines a point in time (a *milestone*) based on clear criteria. Thus, the above phases end up with the following milestones: life-cycle objective, life-cycle architecture, initial operational capability, and product release, respectively. Several iterations occur within a software generation, so that inception, construction, transition, and elaboration are different scenarios for iterations. It is worth pointing out that each iteration incorporates requirements elicitation, analysis, design, and implementation as core processes workflows, although the time spent on each "classical" workflow depends on the point in which the iteration occurs during the software generation. Roughly speaking, during inception preliminary iterations most of the time is spent on planning and requirements elicitation; during elaboration on requirements elicitation, analysis and design; during construction on implementation and testing; during transition on testing and deployment.

Several alternatives have been proposed that criticize the overhead introduced by a classical "design-oriented" approach, trying to simplify the software development cycle by concentrating on implementation, testing and refactoring (e.g., Minimal Methodologies [5], Scrum

---

<sup>1</sup> UP in the following.

Methodology [12], Extreme Programming<sup>2</sup> [6]). As this paper is mainly concerned with XP, let us now illustrate how it defines its own “best software practices”.

The rationale that supports the “extreme way” is that requirements will not be known at the beginning, as they will change along the way. Thus, instead of trying to capture and analyze requirements separately, it is better to define an alternative approach able to incorporate the design-for-change in a natural way. In other words, XP techniques stem from the consideration that life-cycle processes based on formal or semi-formal techniques introduce an overhead that might cost as much as (possibly more than) an approach based on refactoring, basically played at the implementation level. Of course, such a consideration could not be effective until languages and tools able to support Rapid Application Development [10] became common practice. It is worth pointing out that XP was conceived within Smalltalk environments. Due to its memory requirements, in the past Smalltalk was typically run on powerful (and expensive) computer systems, whereas nowadays it can be easily run on personal computers and used as a target language for iterative prototyping (which is the natural support for XP).

As a consequence of the underlying approach, the “extreme way” ends up with an iterative process strongly based on refactoring, testing, and a non-hierarchical team organization. The rule of thumb that highlights the process is “build for change instead of building for the future”. Thus, when a change is required, only “the simplest thing that could possibly work” is done, followed by a “merciless” refactoring.

The processes we have been outlining are very similar when considering their “natural” bias towards an

in the space of the architectural solutions that may be identified to solve a given problem. In particular, whereas the former basically follows a rather classical approach that uses A&D to avoid spending time in refactoring, the latter bases its process precisely on refactoring.

In this paper we describe a process that basically follows the guidelines suggested by XP, although here refactoring is incorporated within round-trip-engineering activities. Within such a process, UML and Smalltalk are used as modeling and target language, respectively. The former has been selected because it incorporates well-known diagrams for describing a software application from different perspectives (e.g., Use Cases, as well as Class, Interaction, State, and Deployment Diagrams), whereas the latter has been adopted as it allows fast prototyping and early delivery.

## 2. THE PROCESS

In the process we propose, used on a real project, we follow a spiral model where a complete round generates a software generation. For the sake of clarity, we preserved a process based on phases, as defined by UP, where iterations occur within different scenarios, depending on the phase currently being undertaken. Of course, the way core processing workflows (such as requirements elicitation, analysis, design, implementation, test, and deployment) are distributed along the life-cycle of a software generation has been customized following “XP-like” recommendations.

The main difference between XP and the life-cycle we adopted is on the scope of refactoring activities: whereas XP basically adopts refactoring at the implementation level, our approach extends it to design and analysis. From our point of view, this may be seen as a natural

	Inception	Elaboration	Construction	Transition
Business Modeling	high	decreasing	almost none	none
Requirements	low-increasing	high-medium	decreasing-low	low
Analysis & Design	almost none	high	decreasing-low	none
Implementation	none	increasing	high	almost none
Test	none	low	high	decreasing
Deployment	none	none	increasing	high
	~10%	~30%	~50%	~10%

Table 1. Life cycle phases together with core processing workflows in the Unified Process.

evolutionary life-cycle. In fact, iterative software development is widely accepted within the software engineers community as an essential characteristic of modern life-cycles. On the other hand, UP and XP are very different in the way they try to get a (local) minimum

evolution of the extreme way, as it continues to focus on refactoring, while extending it to modeling activities. A supporter of “pure” extreme programming would rather point out that such a choice leads back to a classical scheme, enhancing the overhead due to analysis and design versus implementation and testing. To extend refactoring without coming up against such a drawback, a tool able to perform round-trip engineering has to be

<sup>2</sup> XP in the following.

adopted, in order to facilitate moving from one level of abstraction to another.

For the sake of simplicity, while illustrating the process we adopted, we shall refer to the four main phases proposed by UP (i.e., inception, elaboration, construction, and transition). Before going into further details, let us illustrate a cross-reference table that basically recalls how and when core processing workflows occur within the phases defined in UP. As reported in Table 1, about 80%

strictly depends on the selected target language (Smalltalk), which strongly encourages and facilitates it.

To set up an environment able to support the proposed process, two different tools, i.e., Rational™ Rose and UMLTALK, have been used. The former is a well known commercial tool. The latter, developed at our department, is a tool able to update (or generate) an UML-compliant model starting from an application written in Smalltalk and vice-versa.<sup>3</sup> UMLTALK can also export its internal

	Inception	Elaboration	Construction	Transition
Business Modeling	high	decreasing	almost none	none
Requirements	low-increasing	high-medium	<i>medium</i>	low
Analysis & Design	almost none	high	<i>medium</i>	none
Implementation	none	<i>medium</i>	high	almost none
Test	none	<i>medium</i>	high	decreasing
Deployment	none	<i>medium</i>	<i>medium</i>	<i>medium</i>
	~10%	~10%	75%	~5%

Table 2. Life cycle phases together with core processing workflows in the proposed process.

of the time required to deliver a software release is spent on elaboration and construction, their ratio being about 60%.

The same cross-reference table is used to show the main characteristics of the process we adopted. As reported in Table 2 (italics have been used to stress where a change occurred), almost the same amount of time required to deliver a software release is spent on elaboration and construction, but their ratio changes (less than 15%). This basically means that elaboration has been “lightened” with respect to construction. In particular, let us point out that:

- requirements elicitation and A&D activities have been partially moved to the construction phase. This is basically due to the choice of lightening elaboration while increasing construction (by extending refactoring to analysis and design through a round-trip-engineering approach);
- implementation starts early (i.e., during the elaboration phase). In fact, coding is used at an early stage to verify critical, ambiguous, or incomplete requirements, as well as to anticipate the implementation of aspects deemed crucial for the system to be developed;
- testing activities are pervasive and basically follow the implementation’s workflow profile. In fact, according to XP, testing activities are very important and require a separate effort, usually aimed at implementing “test classes” (one for each class defined within the system to be developed);
- deployment occurs at each iteration; i.e., it is not delayed until the transition phase starts. Such a choice

representations into a “Petal” file format,<sup>4</sup> thus leaving the possibility of feeding Rose with a model created from a Smalltalk application. The reverse operation is also feasible, i.e., UMLTALK can update a Smalltalk application (or generate a Smalltalk skeleton) starting from a model imported from Rose. Such a capability has been used extensively within the application’s development, thus giving rise to a round-trip-engineering activity.

Let us now concentrate our attention on the process we propose by considering each single phase of it:

#### - Inception

According to the “extreme way”, user requirements are represented by means of use cases (called “user stories” in XP terminology), collected during brainstorming meetings held with domain experts. Use cases are basically aimed at eliciting domain classes from users. As use cases are an informal text-based description, we do not spend time on this issue, since the usual sensible recommendations apply to them (predefined structure, non ambiguous terminology, no redundant descriptions, etc.).

#### - Elaboration

Elaboration consists in performing analysis, defining the overall system architecture, and attaining a preliminary

<sup>3</sup> I.e., to update a Smalltalk application (or generate the skeleton of it) starting from a corresponding UML-compliant model.

<sup>4</sup> Any Rose model (by default) is stored using an internal file format called “Petal”.

design of the application. In the presented approach, these three activities are performed in the following steps.

1. Analysis is done using Class-Responsibility-Collaboration cards [14]. Here the focus is on characterizing classes related to the domain for which the application is intended to be run. This activity is usually started with brainstorming meetings, aimed at recording on actual CRC cards (with responsibilities and collaborations) the classes found examining and discussing use cases.

Then, these cards are transferred to UML class diagrams, drawn using only the subset of UML primitives that allow to implement CRC basic concepts. In particular:

- packages are used to partition the system into subsystems;
- classes (without attributes and operations) are used to represent CRC cards;
- class "documentation" slots are used to hold class descriptions and their responsibilities;
- dependency relationships are used to represent collaborations;
- inheritance relationships are used to represent the corresponding inheritance between classes annotated on the cards;
- notes are used to comment the diagrams.

In this way, the CRC analysis is documented with well defined diagrams, which can be incrementally modified. These diagrams are the starting point for the subsequent step.

2. System architecture definition is aimed at expanding the abstract view recorded with CRC cards and leads to further refine classes in terms of their attributes and operations. Now the focus is on adding structural information to domain classes. Furthermore, extra classes, strictly related to the application to be developed but still "visible" to the user,<sup>5</sup> can be added to the model.

The class responsibilities elicited in the previous step become attributes, associations, and operations. Collaborations are used to specify and refine operations, and to check the consistence of the model. Very often, collaborations links become associations and aggregations, since the fact that two classes collaborate reflects their structural dependencies.

At this level of detail, further classes and responsibilities that may arise are added to the model, and are also incorporated into the CRC analysis by means of an iterative process. Moreover, other classes belonging to the user interface or other subsystems are

defined and added to the resulting architecture. Packages are typically used to characterize high-level subsystems that exhibit a high degree of internal cohesion and external decoupling. Packages derived from the analysis can be further expanded, and other packages can be added, holding the added subsystems.

This activity is performed with extensive use of UML class diagrams, drawn using Rational Rose.

3. Design builds upon the overall architecture definition. In this step, further details are added to the model. For instance, let us consider an agency and its officers. During analysis it is specified that an agency has the responsibility to know its officers and to query their properties. This is reflected in agency responsibilities and in a collaboration between the two classes. During architecture definition these CRC concepts become an aggregation between an agency and its officers (an agency contains zero or more officers), and operations to manage such aggregation and to query the officers. During design it is specified that the aggregation is implemented using an ordered collection, and that officers are uniquely identified by an internal code.

Beside the augmentation of the model derived from analysis, while performing design all user interfaces are fully specified describing their widgets, events and call-back messages. Furthermore, the permanent storage of data is defined designing the database schema or the file formats, and the interfaces with external systems and devices are specified.

Eventually, the design model, written in UML, is automatically transformed into a set of Smalltalk classes, each with proper data structure, comments, (automatically generated) access methods, and the skeletons of other methods.

It is worth pointing out that state transition diagrams are routinely used to represent the dynamic behavior of a class. On the other hand, collaboration, sequence and activity diagrams of UML are used very seldom, if ever. The only motivation to use these diagrams is to document a complex scenario of interaction among objects, in order to make the model more understandable.

The whole elaboration phase has been kept as "lightweight" as possible, so as not to move too far from the "extreme way".

#### - Construction

Construction consists in implementing and testing the system to be developed. While implementing the system, several iterations may occur, basically centered upon refactoring. As already pointed out, refactoring does not usually occur just at the implementation level, i.e., it usually involves design activities, the overall architecture definition, and analysis (possibly together with further requirements elicitation). As a matter of fact, a round-trip-

---

<sup>5</sup> E.g., interface classes, protocols, controllers, etc.

engineering process has to be implemented during the construction phase. To put it into practice, UMLTALK is employed to link Smalltalk and Rose together. Starting from the initial architecture developed using Rose (during elaboration), UMLTALK can then be used (during construction) to import such an architectural description and to produce a Smalltalk skeleton of the program. At this point, Smalltalk coding can be performed until a change at the design or analysis level is required. In order to do this, the Smalltalk code is used as a source for updating the corresponding UMLTALK model. In particular, new methods may have to be incorporated into the model, and/or methods description may have to be updated according to the existing Smalltalk code.

Once performed a coding session, UMLTALK can be used to export the model in a "Petal" file format. In this way, when needed, Rose may be fed back and realigned with the updated model.<sup>6</sup> The problem of realigning model and code arises also when changes are performed within Rose. In such a case, they have to be transferred down to the Smalltalk code. Communication between a Rose model and the corresponding Smalltalk code is performed by using UMLTALK again, this time proceeding in the opposite direction. As a result, a round-trip-engineering process is implemented while performing construction. In practice, UMLTALK is basically used as a front-end to get Smalltalk applications being dealt with using a standard tool able to perform modeling according to a UML-compliant representation.

Let us note that the round-trip is performed starting from the architectural level, and does not extend up to the CRC analysis level. In this way, the "structured" class diagrams are kept updated as UML documentation of the system. On the other hand, the CRC diagrams reflect only the initial efforts in the development of the system.

As far as testing is concerned, it is worth noting that it is strictly coupled with implementation activities. In particular, for each class that belongs to the system being developed, a corresponding test-class must be defined and implemented. Thus, testing activities are uniformly distributed within an iteration instead of occurring mainly at the end of it.

During construction, at each iteration, one or more subsystems are partially refined and implemented. The outcome of each iteration is a prototype that incorporates part of the required functionalities. It is worth noting that the whole approach is iterative, and incremental. In fact, high-level subsystems may be developed separately according to a typical incremental approach, and implementation activities feed back design and/or analysis.

---

<sup>6</sup> The "Semantics" field of each operation has been selected and used to keep information about the Smalltalk code within Rose.

### *- Transition*

Transition consists in working on the application with the goal of delivering it to the end user. In UP, once the construction phase has been completed, usually several problems occur while attempting to adapt it to the working environment, trying to implement features that have been postponed, correcting some problems, etc. In particular, the act of adapting the application to the working environment involves deployment activities, which should typically occur at the transition phase. On the contrary, in the process we adopted, deployment is performed at the end of each iteration. In this way, the transition phase results in a very "light" activity. In fact, such an approach is strictly related to the target language (Smalltalk) which, from a conceptual point of view, does not distinguish between system and user-defined classes. Thus, in some sense, the system is always "ready-to-use" and would need a light deployment activity even if the application were developed using Distributed Smalltalk.

## **3. EXPERIMENTAL RESULTS**

We developed a system that falls within the class of business-oriented internet services. The system provides both an on-line and an off-line front-end. The former consists of a web service supplied to any potential business-man searching for a grant by the European Community, the state, and other national or regional bodies to set up a firm in Sardinia. The latter consists of a local service supporting domain experts in updating the information about grants.

After starting an internet connection by means of a standard web browser, the user is typically asked information about the business to be undertaken or improved. Depending on the given user profile, the system queries a data base containing information about all available grants, and selects the information that match the user profile. Results are automatically reported to the user by means of dynamic web pages. At this point, she/he can concentrate on a subset of the selected grants (if any) or begin a different query to the database. Of course, several queries can be repeatedly submitted by the same user and the application must be able to handle, at the same time, multiple queries submitted by multiple users.

To supply the required functionalities, the following subsystems have been provided:

- a database, containing laws and directives entailing financial support, as well as information about international, national, and regional bodies or authorities;
- a web interface, compatible with any web browser, able to create user profiles (one for each user connected to the web site), as well as to display, by automatically generating web pages, useful information resulting from queries performed on the database;

- an engine, able to perform suitable queries on the database, according to any given user profile;
- an interface for database maintenance.

A first attempt at capturing requirements made it clear that domain experts were having difficulties while trying to transfer their knowledge about the domain to be modeled. Starting from this lack of clarity and considering that only a few people were involved in the project, we decided not to let the usual roles of software architect, analyst and programmers be played within such a software project. Instead, we defined a more flexible team structure, composed of 4 people, adopting a non hierarchical team organization and founding our work on refactoring as the basic mechanism for process iterations.

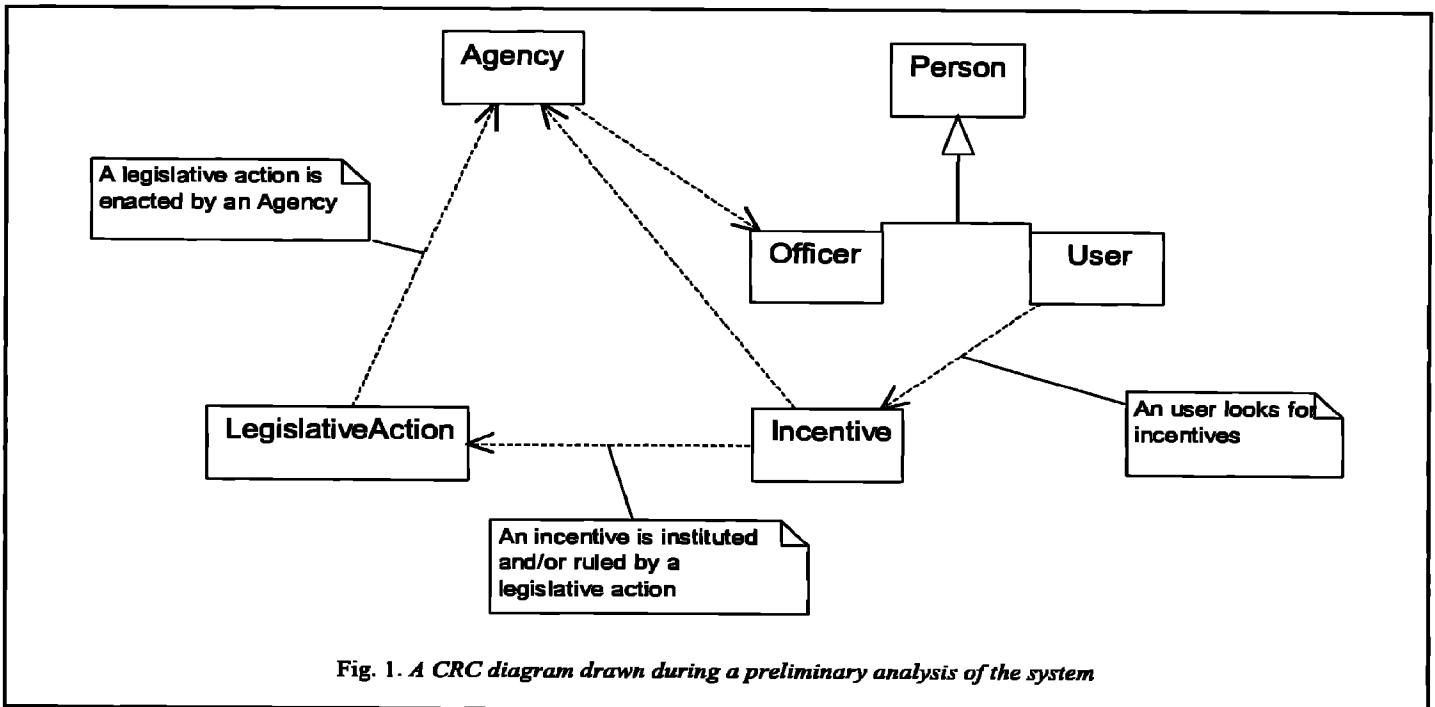
Requirements elicitation was done collecting from the users about 20 "stories" telling the forecasted use of the system. A "high-level" analysis was made with CRC cards, using real cards first, and then storing them as class diagrams in Rose. Fig. 1 shows one such diagram. Responsibilities are not shown here, although they have been incorporated within class documentation. During this activity 22 domain classes were found. Implementation of some classes has been done at this early stage, to verify the feasibility and consistency of critical aspects. Furthermore, subsystems (i.e. packages), as well as dependencies among classes, inheritance relationships, and responsibilities have been identified.

Then, domain classes have been refined, typically turning collaborations into associations and responsibilities into

attributes and operations, respectively. Fig. 2 illustrates a refined diagram (only few attributes are shown for the sake of simplicity). It is worth noting that collaborations have become associations and that 15 more classes (application-dependent classes) have been added to the former diagram. The whole system ended up with 65 classes. Every week an internal release was developed. One person was committed to develop test classes only. He did not work with the rest of the team and was accustomed to send his artifacts through Internet. This activity did not have any negative effect on the expected timing. After less than two months the part of the system consenting intensive data entry was released and immediately used by administrative personnel. The whole system was successfully deployed in time after five months from the beginning of the project. The estimated man power was about 10 man-months, since developers did not work fulltime on the project.

#### 4. CONCLUSIONS AND FUTURE WORK

In this paper we describe a software process strongly biased towards refactoring. It basically follows the guidelines suggested by XP; in fact, inception, elaboration and transition result in a very "lightweight" activity, whereas the main focus is on refactoring, implemented through round-trip-engineering techniques during construction. It is worth recalling that, within such a process, refactoring activities are not only used at the implementation level: they are extended to design and analysis, too. Of course, the developing team has to be



composed of few people, as communication between them is greater than that observed in classical teams. This happens for two different reasons: (i) the team is biased towards a “democratic” organization, and (ii) minimizing overall communication is no longer a process requirement.

UML has been selected as the underlying notation for representing the model, and Smalltalk has been selected as target language. To be able to perform round-trip-engineering, (in particular, to be able to move back and forth between analysis, design, and implementation levels), a suitable tool developed at our department (UMLTALK) has to be used throughout the whole process to keep the UML representation of the system synchronized with the corresponding Smalltalk code. In fact, UMLTALK can produce a UML-compliant model starting from a Smalltalk program, and vice-versa, and can export its internal representation into a “Petal” file format and vice-versa.

The proposed approach has demonstrated to be very effective in a small-sized project, enhancing productivity considerably, although its scalability up to middle-sized projects has still to be proven.

It is worth pointing out that the life-cycle we adopted takes into account the main criticisms moved by extreme programming to a classical “design-oriented” life-cycle.

We believe that, from a historical point of view, A&D has increased its importance within the software life-cycle depending on the assumption that it is costly to undo mistakes when playing mostly at the implementation level. Nevertheless, somehow surprisingly, a classical waterfall approach suffered from a similar drawback, as it pushes risks forward in time so that it is costly to undo mistakes from earlier phases. That is why, a modern “design-oriented” life-cycle has to be iterative and, possibly, incremental.

On the other hand, while “design-oriented” life-cycles came to their maturity, dramatic changes have been introduced as far as programming tools and computer hardware are concerned. Such changes made it possible to develop applications focusing on refactoring rather than on A&D. The underlying hypothesis is that by adopting suitable tools and target languages, refactoring would possibly cost as much as (or less than) the overhead introduced by modern A&D techniques. Moving in the same direction illustrated by extreme programming, we followed a process in which preliminary analysis and design are made “lighter” and most of the time is spent on refactoring. The main difference between our approach and the “extreme way” is that we performed refactoring within a round-trip-engineering cycle, so that the model

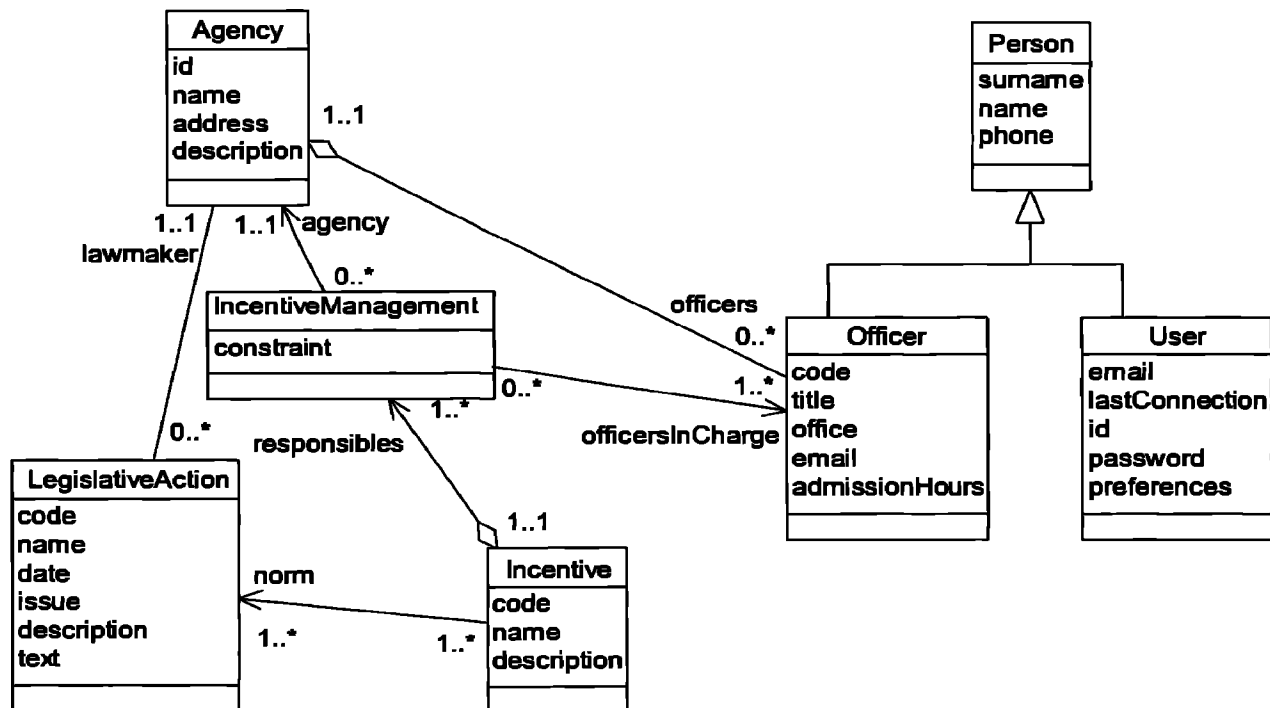


Fig. 2 A simplified “architectural” class diagram derived from the CRC diagram shown in Fig. 1.

and the corresponding Smalltalk implementation are continuously kept synchronized.

As far as future work is concerned, we are trying to give a suitable GUI to UMLTALK, in order to be able to directly perform round-trip-engineering, during construction, within a framework based on UML and Smalltalk as modeling and target language, respectively.

## 5. ACKNOWLEDGMENTS

Our thanks to all people involved in the project. A special thanks goes to Dr. A. Angius, President of "BIC Sardegna", the enterprise the software project has been developed for.

## 6. REFERENCES

- [1] B. W. Boehm, "Anchoring the Software Process," *IEEE Software*, pp. 73-82, July 1996.
- [2] G. Booch, "Object-Oriented Analysis and Design with Applications," Cummings, 1991.
- [3] G. Booch, J. Rumbaugh, and I. Jacobson, "The Unified Modeling Language User Guide," Addison-Wesley, 1998.
- [4] P. Coad and E. Yourdon, "Object-Oriented Analysis," Prentice-Hall, 1989.
- [5] A. Cockburn, "Surviving Object-Oriented Projects: A Manager's Guide," Addison-Wesley, 1997.
- [6] The best reference on Extreme Programming is in the Web site:  
<http://c2.com/cgi/wiki?ExtremeProgramming>.
- [7] I. Jacobson, M. Christerson, M. Jonsson P. van Overgaard, "OO Software Engineering, A Use Case Driven Approach," Addison-Wesley, 1992.
- [8] I. Jacobson, J. Rumbaugh, and G. Booch, "The Unified Software Development Process," Addison-Wesley, 1999.
- [9] P. Krutchen, "The Rational Unified process," Addison Wesley, 1998.
- [10] J. Martin, "Rapid Application Development," Macmillan, 1991.
- [11] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen, "Object-Oriented Modelling and Design," Prentice-Hall 1991.
- [12] K. Schwaber, "The Scrum Development Process," OOPSLA 95, Workshop on Business Object Design and Implementation, 1995.
- [13] S. Shlaer and S. Mellor, "Object-Oriented Systems Analysis: Modeling the World in Data," Prentice-Hall, 1988.
- [14] R. Wirfs-Brock, B. Wilkerson, and L. Wiener, "Designing Object-Oriented Software," Prentice-Hall, 1990.