Change-Oriented Round-Trip Engineering

First-class changes for Agile Software Development

Peter Ebraert* — **Ellen Van Paesschen**** — **Theo D'Hondt***

* *Programming Technology Laboratory*
Vrije Universiteit Brussel (VUB)
Pleinlaan 2, 1050 Brussel, Belgium
{pebraert, tjdhondt} @vub.ac.be

** *Laboratoire dÕInformatique Fondamentale de Lille*
University of Lille
1, 59655 Villeneuve dÕAscq, France
ellen.Vanpaesschen@lifl.fr

ABSTRACT. *In order to support runtime changes, automated testing and refactorings in Agile Software Development (AgSD), we propose to represent changes applied on a system under development, as first-class objects. We envision the integration of a Change Management System in an existing methodology for AgSD called* Advanced Round-Trip Engineering *(ARTE). We explain how* Change-Oriented ARTE *(COARTE) allows capturing, visualising, replaying and rewinding changes that have been applied on the modelling, implementation and runtime views of an ARTE environment, and automatically synchronizes them with the other views. In this setup tests and refactorings can be composed out of changes, while runtime change propagation is realized with different propagation strategies.*

RÉSUMÉ.

KEYWORDS: *Model-Driven Development (MDD), Round-Trip Engineering (RTE), Agile Software Development (AgSD), Runtime Software Evolution, Reflection.*

MOTS-CLÉS :

1re soumission à *Languages et Modèles à Objets (LMO 2007)*, le 29ième mars 2007.

1. Problem statement

Agile Software Development (AgSD) [COC 02] stresses a highly iterative and incremental development cycle in order to support *change* during the different phases of software engineering. This is achieved by prioritizing issues such as flexibility, end-user interaction, productivity, and individuality. There already are a number of frequently applied agile methods such as *Extreme Programming* (XP) [BEC 99], Test Driven Design (TDD) [BEC 03] and *Adaptive Software Development* [HIG 00].

There exist however a number of unaddressed challenges in the tool support for AgSD. In this paper, we focus on three related issues. First, because some software systems (such as critical systems) may not be stopped, changes sometimes need to be performed on *running systems* [ORE 98]. This is especially relevant in AgSD where systems are developed incrementally and are thus continuously changed. This issue will be further referred to as *runtime change propagation*. Reasons for not supporting *runtime change propagation* can be found in difficulties such as ensuring state transfer or managing system consistency – as we explain in [EBR 04b]. Second, *testing* plays a central role in TDD. *Unit tests* [BEC 97] are automated pieces of code that invoke different methods and then check assumptions on their behavior. It has to be possible to easily write the tests and quickly run them, repeatedly and automatically. Finally, automated testing gives rise to *refactoring* which is used for altering the structure of an existing implementation in order to improve its design quality while not changing any of its functional characteristics [OPD 92]. Agile methods such as XP continuously apply refactorings in order to facilitate adding new functionality or to improve the design quality after a change took place. Reasons for not supporting *testing* and *refactoring* can be found in difficulties such as replaying/undoing changes and keeping the different views synchronised when tests and refactorings are performed [EBR 04a].

2. First-class Changes in Agile Software Development

During AgSD, systems are created by iteratively and incrementally changing them. As such, many small changes are applied on the system under development. In [ROB 07], Robbes argues that considering changes as first-class[1] objects provides more accurate information about the evolution of a software system compared to traditional file-based techniques. *Change-oriented* software development centralizes these changes as the main entity of the development process. The entire set of changes represents the complete history of a software system, in the same incremental way it was created and manipulated. Holding all the information that specifies them, the changes can also be used for reproducing the software system and reasoning about it.

We believe using a change-oriented software development process for AgSD with first-class changes offers different advantages:

1. First-class change objects are objects that can be referenced, queried and passed along.

1) In AgSD, programmers create software systems by a trial-and-error approach. In this setting, changes are continuously *done and undone*. Traditional Interactive Development Environments (IDEs) include an undo mechanism, but do not support changes on the semantic level. We envision first-class change entities maintaining information for supporting the undoing and reapplication of changes.

2) Change objects can form an *extra level of abstraction* between the three different levels of software artefacts. This would bring along two benefits with respect to the propagation of changes. First, *different propagation strategies* could be adopted for synchronising the different software artefacts. Second, the changes could incorporate information for *incrementally adapting* the views on the different levels. If those views would have to be recomposed each time a change is made, we would have a lot of overhead, especially in AgSD.

3) Change objects can be used for capturing the programmer's intention for making certain changes. If a developer wants to rename a variable for example, he does not think about replacing all methods referencing it with new methods, even if that are the changes the IDE actually applies on the system. All these changes could be annotated with the programmer's intention. This would be particularly interesting in an AgSD context, where large amounts of small changes coexist. Grouping and documenting changes would *ease understanding* the system and *improve its maintainability*.

4) Change objects can include pre- and post-conditions, representing the invariants imposed by the system's meta-model. This information could be used for automatically detecting conflicts between the changes [HER 00]. This would improve both testing and debugging software systems and hence help *preserving system integrity* in the trial-and-error approach of AgSD.

3. Change-Oriented Advanced Round-Trip Engineering (COARTE)

In [Van 05] a new RTE practice called ARTE was proposed for supporting Agile Model-Driven Development which combines the highly iterative agile development cycle with the vision that software can be derived automatically from abstract models. In this approach, the system under development is represented by a high-level design view, a static implementation view and a runtime view that all consider one underlying model [Van 06]. We propose to extend the ARTE approach with an extra *change view* allowing the interaction with a management system for first-class changes. The envisioned approach (COARTE) provides *five interacting views* on the software system under development, which are synchronised by the *the COARTE Change Management System* as shown in figure 1. The following sections explain how we envision both.

3.1. *The Five COARTE Views*

The **Static Design and Implementation Views** respectively contain a UML class-diagram, and a view on the source code of the application. At this level, one can see

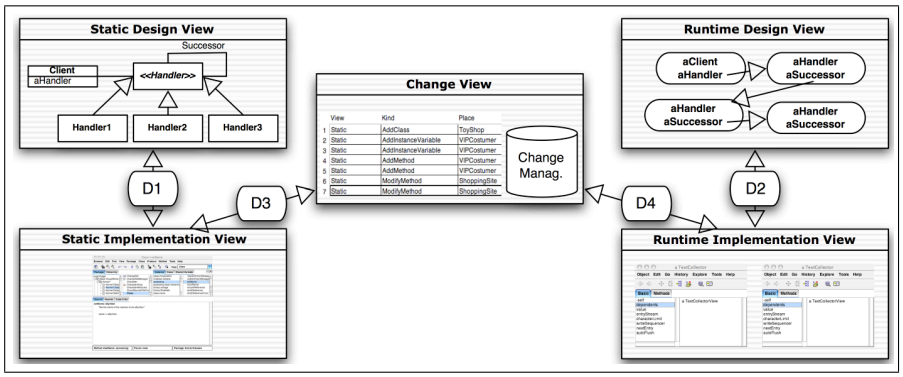

Figure 1. *Five interacting views in Change-Oriented Advanced Round-Trip Eng.*

how an application is designed and how it is implemented. These views are mostly consulted by the designers and the developers of the application. A typical change in the former view is adding a class to the design. An example of a change in the latter view is modifying the implementation of a method body. Most RTE tools include both a static design and implementation view.

The **Runtime Design and Implementation Views**, depict the objects that are at that moment *alive* in the system. Every object is represented by a rounded rectangle containing an ID and all the names of instance variables it contains. The values of those instance variables are other objects, that are again represented by rounded squares. Arrows from the instance variable names to other objects show that those objects are the actual values of those instance variables. They represent the references that are maintained within the former object and that allow the former object to send messages to its instance variables. The implementation view serves for inspecting the implementation of the living objects. An example of a change in this design view is changing a visual link between two objects and consequently change the value of a certain instance variable. A typical change in the implementation view, is the modification of the value of an instance variable of an object. Only few RTE tools such as SelfSync [Van 06] provide runtime views that are part of the RTE process.

The **Change View** is a view whose central part consists of a list of changes that have been or can be applied to the system. The changes in the list are actual references to first-class change objects, which can be inspected, altered and saved. Which changes are supported depends on the meta-model of the programming language of the system under development. It would, for instance, not make sense to support an `AddClass` change in a meta-model where there is no notion of classes. As we focus on object-oriented development, we chose to support the Famix meta-model [DEM 99b]. This model provides a language-independent representation of object-oriented source code and is used for exchanging information about object-oriented software systems [DEM 99a]. Examples of languages adhering to this model are Java or Smalltalk. We refer to [EBR 04a] for a detailed explanation of the changes.

3.2. *The COARTE Change Management System*

The envisioned COARTE change management system allows the *addition* and the *reuse* of changes. It should be possible to add new change objects in three different ways. First, the developer could create a new change object by selecting one of the provided changes of the meta-model which functions as a template, and instantiate it as desired. Second, each time a change is made to one of the other four COARTE views, a corresponding change object has to be created automatically. Finally changes have to be importable via files. Change objects can be reused, by grouping them into transactions, by undoing them and by reapplying them. The remainder of this section explains how the COARTE Change Management System would support (runtime) change propagation, testing and refactorings.

Change propagation could be supported in four directions (as denoted by $D1$ to $D4$ in figure 1). The change management has to propagate changes for ensuring the consistency in the different views on the system under development. This has to be done in an incremental way, so that the views do not have to be recomposed every time a change is made to one of them. The change management system can offer four different strategies for propagating changes to the runtime views and two different strategies for propagating changes to the static views. The first-class change objects can incorporate all the information the change management system needs for propagating changes to the different views.

Testing could be supported by letting a developer express a test as a number of changes. The first part of the test contains a boolean expression whose result determines whether the test succeed or fails. The second part of the test are changes that describe a number of compensating actions which are performed by the change management system in case the test fails. The compensating actions are executed for extending the design of a system in a test-driven way. Each test is stored and can be reused when needed.

Refactorings could be supported in two ways. On the one hand, a refactoring can be performed by the developer on the system under development while the change management system records the refactoring and instantiates change objects for it. On the other hand, the developer can select new or existing change objects and compose them into a new refactoring, which in its turn can be applied automatically as a transaction on the system under development. After its application, the refactoring is saved and can be reapplied or undone at all times.

4. Future Work and Conclusion

In this paper, we envision Change-Oriented Advanced Round-Trip Engineering (COARTE), as an extension of Advanced Round-Trip Engineering with first-class changes and with a change management system that aims at better supporting testing, refactoring and runtime change propagation in Agile Software Development. COARTE would include five views on a system under development: two statical views, two runtime views and a change view. We propose to represent changes as first-class objects based on the Famix meta-model for object-oriented systems. When

changes take place in the static or dynamic views, the change management system could record them in the change view and semi-automatically synchronise them to the other views in an incremental way. In this set-up, tests and refactorings can be composed out of changes, they can be recorded, re-applied, undone, reasoned upon and propagated where appropriate. We are working on the implementation of COARTE which is currently still immature: testing is not supported and only two propagation strategies have been implemented. The future work consists in adding these features.

5. References

[BEC 97] BECK K., *Kent Beck's Guide to Better Smalltalk*, Cambridge University Press, New York, USA, 1997.

[BEC 99] BECK K., *Extreme Programming Explained: Embrace Change*, Addison-Wesley Professional, October 1999.

[BEC 03] BECK K., *Test-Driven Development: By Example*, Addison-Wesley, Boston, 2003.

[COC 02] COCKBURN A., *Agile software development*, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[DEM 99a] DEMEYER S., DUCASSE S., TICHELAAR S., "Why Unified is not Universal?", *Proceedings of the Second International Conference on The Unified Modeling Language*, vol. 1723 of *LNCS*, Fort Collins, CO, USA, October 1999, Springer, p. 630–644.

[DEM 99b] DEMEYER S., TICHELAAR S., STEYAERT P., "FAMIX 2.0 - The FAMOOS Information Exchange Model", report , August 1999, University of Berne.

[EBR 04a] EBRAERT P., MENS T., D'HONDT T., "Enabling Dynamic Software Evolution through Automatic Refactorings", *Proceedings of the Workshop on Software Evolution Transformations (SET2004)*, Delft, Netherlands, november 2004.

[EBR 04b] EBRAERT P., VANDEWOUDE Y., D'HONDT T., BERBERS Y., "Pitfalls in unanticipated dynamic software evolution", CAZOLLA W., Ed., *In the proceedings of the Workshop on Reflection, AOP and Meta-Data for Software Evolution in conjunction with the 18th European Conference on Object-Oriented Programming*, 2004.

[HER 00] HERRANZ-NIEVA A., MORENO-NAVARRO J. J., "Generation of and Debugging with Logical Pre and Post-Conditions", *Automated and Algorithmic Debugging*, 2000.

[HIG 00] HIGHSMITH J. A., *Adaptive software development: a collaborative approach to managing complex systems*, Dorset House Publishing Co., Inc., 2000.

[OPD 92] OPDYKE W. F., "Refactoring Object-Oriented Frameworks", Phd Thesis, University of Illinois at Urbana Champaign, 1992.

[ORE 98] OREIZY P., MEDVIDOVIC N., TAYLOR R. N., "Architecture-based runtime software evolution", *Intl. Conf. on Software Engineering*, Kyoto, Japan, April 1998.

[ROB 07] ROBBES R., LANZA M., "A Change-based Approach to Software Evolution", *Electronic Notes in Theoretical Computer Science*, vol. 166, 2007, p. 93-109.

[Van 05] VAN PAESSCHEN E., DE MEUTER W., D'HONDT M., "SelfSync: a dynamic round-trip engineering environment", *Proceedings of the ACM/IEEE 8th International Conference on Model-Driven Engineering Languages and Systems*, Jamaica, October 2005.

[Van 06] VAN PAESSCHEN E., "Advanced Round-Trip Engineering (ARTE): An Agile Analysis-Driven Approach for Dynamic Languages", PhD thesis, Vrije Universiteit Brussel, 2006.