Lightweight Object-Oriented Development: What's in it for Software Architecture?

Christian Heide Damm[1], Klaus Marius Hansen[1], Michael Thomsen[1], and Michael Tyrsted[2]

[1] Department of Computer Science, University of Aarhus, Åbogade 34, 8200 Aarhus N, Denmark
{damm,marius,miksen}@daimi.au.dk
[2] Ideogramic ApS, Mejlgade 45, 2., 8000 Aarhus C, Denmark
tyrsted@ideogramic.com

Abstract. Lightweight object-oriented development methods are gaining momentum in industry and academia. Crystal "Clear" and eXtreme Programming are prominent examples of this. By focussing on risk resolution through iteration and implementation on demand, such methods promise control over uncertainties in software development. However, evolution of a software is often only done through low-level, code-near refactorings. Through examples from industry and academia, we argue that an appropriate software architecture – an overall design – of a developed system nevertheless can be evolved through a continuous focus on system structure. In particular, the recurring decision on and assignment of development tasks should be combined with promotion of the software architecture to a first class citizen in planning, developing, and evaluating – also in light-weight software development.

1 Introduction

Light-weight development methods have recently emerged as an alternative to more "heavyweight" software development methods. Building on [14], light-weight development can be said to exhibit the following characteristics:

- *Adaption rather than prediction.* Light-weight development acknowledges that the context of a system is prone to change, and thus builds adaptability into the methods themselves: if requirements are able to change radically, so must the development methods.
- *People rather than process.* The techniques of a light-weight method are designed to support they way developers work most productively, and enjoyably, not to force developers to follow a rigid process.
- *Development rather than documentation.* The main goal of system development is a working software system. Light-weight methods emphasise this and try to reduce document production to a minimum while maximizing code-oriented activities.

Software development methods such as eXtreme Programming (XP) [2], and Crystal "Clear" [9] qualify as lightweight development methods. On the other hand an iterative experimental development process such as DSDM Rapid Application Development (RAD) [28] has a high focus on non-programming related artefacts and prescribes a relatively fixed process. Nevertheless the border between heavy- and lightweight development methods is not completely fixed: the huge Rational Unified Process framework [20] can, e.g., be instantiated to support eXtreme Programming [21].

There are two basic definitions of software architecture that also help guide how software architecture is handled during system development. The first focuses on structures of the software being developed:

> *The software architecture of a software system is the structure(s) of a software system in terms of components, their external properties, and their interrelationships [1, 16]*

The second focuses on software architecture as an an overarching understanding:

> *The software architecture of a software system is an overall understanding of the system that guides its construction and use [6, 17]*

Most lightweight development methods advocate the second definition: eXtreme Programming advocates the use of *metaphor* as guidance for the creation and development of software architecture in a project. Examples include "like a spreadsheet" and "contracts, customers, endorsements". The metaphor for a system is meant to be easy to understand so that it can travel between project participants and be easily re-constructed at all times during development.

The "Crystal" framework (or "family") of methods has a human-centric focus on productivity and habitability: The process should be effective while still being pleasant to work within. In this respect, the family opposes eXtreme Programming, because it claims that it is generally not possible to define a fixed set of practices that apply to a wide variety of development projects. So since people and projects differ, a whole family of methods is needed – with Crystal "Clear" being the lightest.

Crystal "Clear" is designed for small teams of up to six people. The only mandated techniques are milestone and risk management. How everything else is done is up to the development team according to the circumstances. Design – and software architecture – evolves through a cooperative "language game" [29] involving the team and the artefact being designed. Representation of the results relies on the highly developed cognitive and communicative abilities of people to allow for low volume and precision in deliverables.

In our discussions of software architecture, we will distinguish between 'refactoring' and 'architectural refactoring' [5].

Refactoring is considered with semantic-preserving transformations of classes and their relationships [24]. Architectural refactoring, however, is considered with functionality-preserving transformations of architectural structures. A given

system will thus present the same functionality to its users (e.g., end users or programmers) after an architectural refactoring with respect to some domain. Simplistically, the reasons for doing refactoring have to do with code-level problems whereas architectural refactorings are done based on technical as well as social considerations.

To discuss software architecture, and refactorings of software architectures, in light-weight software development, this paper presents two case studies of development projects: a contract development project running for almost two years, and a product development project running as a research project for more than a year with subsequent commercialization. Both used lightweight development methodologies and both were initially academic research projects.

Although the projects were research-oriented, they were also realistic in the sense that industrial partners were involved with quite realistic requirements and expectations. Even so, they represent technologically innovative development that has later made it into commercial products – the latter characteristic in particular has put the architectural practices discussed in this paper under pressure. In an industrial context, the results are relevant for projects of the same size as the projects described in this paper, because these are the kinds of projects that will typically benefit from light-weight development methods.

Based on these projects, we argue that an explicit focus on software architecture both as structure and metaphor could and should be used in light-weight software development in contrast to what is currently advocated in, e.g., XP. Furthermore, we discuss the concrete techniques used for architectural development.

The structure of the rest of the paper is as follows: In section 2, we discuss the Dragon Project's lightweight development and its use of software architecture. Section 3 discusses the Knight Project with a similar focus. Following this, section 4 sums up and discusses the previous sections, and section 5 concludes.

2 Lightweight Contract Development – The Dragon Project

The *Dragon Project* was initiated in March 1997 as a joint project between the University of Aarhus and a globally distributed shipping company in order to develop a series of prototypes of a Global Customer Service System (GCSS). The concept of GCSS had evolved within the business based on a global Business Process Improvement (BPI) process aiming at the coordination of globally distributed sites and supporting an effective and efficient customer service that, at the same time, supported local needs.

Three of the authors participated in the Dragon Project.

2.1 Development Process in the Dragon Project

The various business activities (white) and university research group activities (grey) are shown in Figure 1. Initially, the project group went through a four

month prototyping phase in which a number of important business areas were explored ("Prototyping"). Following the approval of the prototype by the company, the development continued in a number of iterations in which the prototype was evolved horizontally as well as vertically ("Experimental System Development").

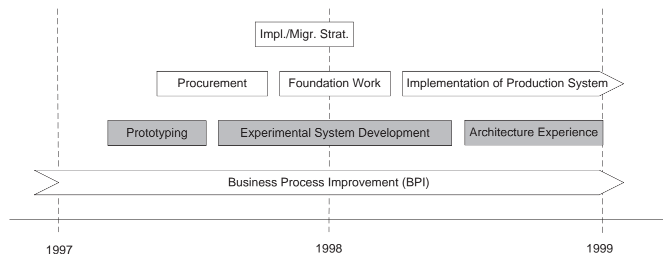

Fig. 1. Major activities concerning the Dragon project

The development process was inherently lightweight: through a multidisciplinary orientation to actual shipping practice, ethnographers, participatory designers, and object-oriented developers [4] collaborated closely to eliminate indirect communication and in a common orientation to an evolving prototype. The primary delivery of the project was the prototype and lightweight architectural descriptions of the system. [5] contains a discussion of the actual architecture of the system and the restructurings that shaped it.

2.2 Software Architecture in the Dragon Project

An architecture for the Dragon Project was drafted by two senior developers within the first weeks of the project. This was done since the prototype itself was envisioned as the primary way of conveying requirements to and from the business in contrast to traditional diagrams and documents. The architecture enabled extensive experiments with what was later termed "business functions", which are the essential work practices of customer service in shipping. One of the initial metaphors used were thus "flexibility": the architecture should be sufficiently flexible to allow for technical as well as functional experiments in the initial phases.

However, there was also a clear and concise structural understanding of the initial software architecture by the developers. This was expressed in initial pseudo code, in concrete code, in informal diagrams, and in verbal re-creations in discussions between developers.

Following the first experimental prototyping phase, the company decided to start the implementation of the final system based on the initial prototypes. Thus, the role of the Dragon Project changed: it should eventually provide a specification of the final system in terms of, e.g., support for important work processes and

client-side architecture. Nevertheless, the prototype should continue to be used as a vehicle for discussion with potential end users in presentations, workshops, and so on.

During this evolutionary development phase, a much clearer understanding of the necessary constituent parts of the system had been achieved, so that the architectural focus changed to structural *intradependencies* in contrast to the earlier focus on *interdependencies* between high-level components. Also metaphors changed character to, e.g., "Components for Work Processes" thanks to the initial technical and conceptual clarifications.

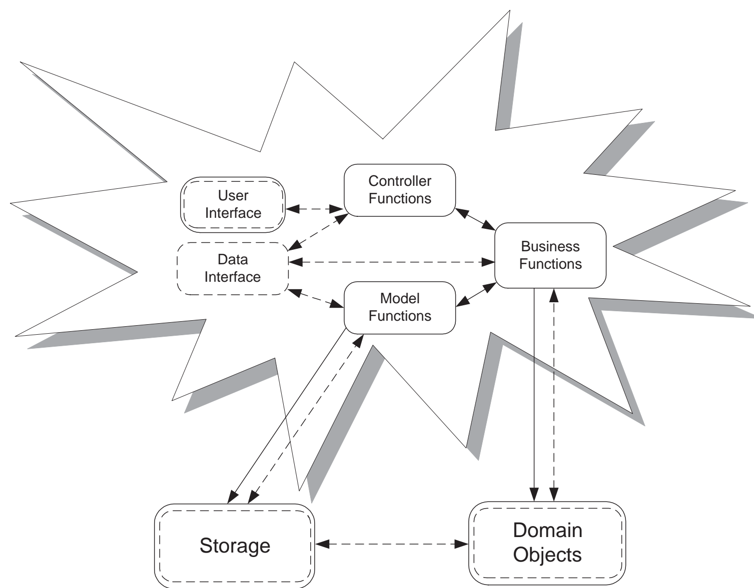

Fig. 2. Flexibility at the level of component boundaries

Figure 2 illustrates the architecture in the evolutionary development phase. The notation is taken from [5]. Simplistically, the rounded rectangles represent components, and the arrows represent flow. The dashed lines emphasise *data*, whereas the full lines emphasise *processing*. The star shape expresses *architectural uncertainty*. Figure 2 thus shows that the "business functions" were no longer the focus for experimentation. The new dominating metaphor, "Components for Work Processes", meant that the flexibility in the architecture at this point had to be mostly at the level of component boundaries as illustrated by the figure.

Informal architectural reviews had been a part of the iterative development cycles of the first phase. Because of the increased focus on technical issues, architectural reviews were conducted in conjunction to formal reviews with business.

Since software architecture was explicitly represented and developed, it became a natural candidate for focus during guided evolution, i.e., *refactoring*.

Refactoring on an architectural level is concerned with function-preserving transformations of architectural structures made in order to improve a software architecture. Architectural refactorings in the Dragon Project included among others redefining component boundaries, enabling flexibility in parts of the architecture in response to architectural uncertainties, and generalisation of, e.g., persistence mechanisms. [5] discusses this in more detail.

3 Lightweight Product Development – The Knight Project

The *Knight Project* was initiated in February 1999 to investigate how novel input devices can support object-oriented developers in making Unified Modeling Language (UML, [23]) diagrams. Concretely, the project used electronic whiteboards (essentially wall-sized touch-sensitive computer screens) in combination with gestural input (as known from handwriting recognition on Portable Digital Assistants (PDAs)) to create an application allowing physically co-located developers to collaboratively sketch diagrams in an intuitive and non-intrusive manner [10, 13, 12]. Figure 3 shows the tool in action on an electronic whiteboard.

All four authors participated in the Knight Project.

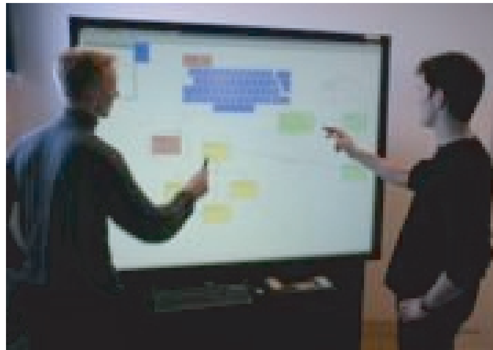

Fig. 3. Use of Knight/Ideogramic UML

3.1 Development Process in the Knight Project

The first part of the Knight Project was concerned research-oriented with a focus on supporting collaborative object-oriented modelling. In August 2000, the scope of the project changed, as the company Ideogramic [18] adopted the research prototype and made it into a commercial product, *Ideogramic UML*.

The project thus went through two main phases: an initial research phase and a following commercialization phase. These cannot be totally separated, though, as one of our goals is to enable continued research in the technologies that we build on. The phases are illustrated in figure 4.

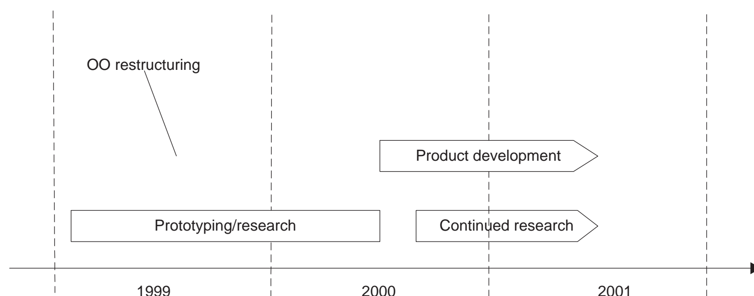

Fig. 4. Timeline of the Knight project

During the research part of the project, the development process was only loosely defined. There were four researchers working on the project, and the communication overhead was kept to a minimum. The development was divided into short development cycles – partly to structure the development, but also for practical reasons to meet deadlines for, e.g., presentations, papers, and evaluations.

Given that the work took place in a research context, there was no strict management in the project, and all decisions were taken at informal meetings between two or more of the researchers, and at semiformal meetings about once a week. At these weekly meetings, the researchers decided on a plan for the coming cycle, discussed status within the current cycle, and generally coordinated who should implement what.

The number of development artifacts was kept to a minimum as well. Beside the actual source code, the only other persistent, written artifacts were research papers, and occasionally written meeting minutes.

The development process has naturally become more well-defined in the commercialisation phase. From being virtually non-existing, the development plan now reaches several months out into the future. In this way, the project is different from eXtreme Programming, where the development "plan" is fixed only for the current interation. On the other hand, having long-range planning does not mean that the plan cannot be changed, and this happens frequently in the Knight Project, where feedback from users are taken very seriously. The nature of the Knight Project, i.e., developing a standard development tool, implies a weaker customer relationship than a contract development project, and having several customers instead of one implies more stable requirements.

The level of documentation has increased in the commercialisation phase compared to the research phase. All the major components are now described in special documents, but at a high level – further details can be found by looking

at the code, which also contains more comments than in the research phase. Although the level of documentation has increased, the development process is still completely based on the people and their knowledge. Pair programming [8] helps in distributing the knowledge.

3.2 Software Architecture in the Knight Project

The software architecture evolved quite a lot during the project. The different natures of the two major phases influenced how the software architecture evolved.

Research Phase Within the research phase, the product evolved radically. From a software architecture viewpoint the system evolved through three main variants:

- Initial horizontal prototype
- Object-oriented vertical prototype
- Support for tool integration

The initial horizontal feasibility prototype was built using the Tcl language [25]. There was no focus on creating a clean implementation, but on getting enough functionality as fast as possible to be able to try out various design and to facilitate early usability testing. This meant that the researchers, maybe even more often than not, resorted to reuse techniques such as cut-n-paste, and that there were very few discussions about the overall architecture.

As the researchers became confident that the research was fruitful, and as the time scope of the project thus extended, they became aware that the current architecture would become a liability. Further investigations showed that 1) the program could be implemented much cleaner using an object-oriented approach, and 2) re-implementing the prototype would be time-consuming, but not overwhelming. To be able to reuse parts of the considerable amount of code, the Itcl [22] object-oriented extension to Tcl was chosen. This would allow for a more gradual re-implementation, where the initial method bodies of main methods could be filled with the procedural statements of the prior main functions. This re-implementation can be seen at the first major architectural decision in the project.

Following the re-implementation, the researchers became more aware of the implementation, and given the large amount of time that had been spent on the re-implementation, they were no longer so inclined to “hack” the code. One example of this is the tool integration efforts that took place late in the phase: As part of the research, it was investigated how the tool could be integrated with other software development tools using both a standardised file format, and using runtime component connections [11]. Instead of “tweaking” the source code to allow for these experiments, the issue was discussed on several meetings, specifically how the code could be restructured for this purpose.

Commercialisation Phase The software architecture also evolved during the commercialization phase:

- Initial product version (support for multiple diagram types)
- Cleaned-up product version
- Support for non-UML diagrams
- The future: product line?

As the project changed from being a research project to a commercial project, things changed dramatically. As an example, during the research phase, there was much more focus on adding functionality than on fixing bugs. Also, the functionality requirements changed to a large extent: In the research project there was focus on novel aspects, whereas the product version was to include the functionality that was requested by the potential customers. One example of an architectural issue that arose from this is support for multiple diagram types. Following an "as simple as possible, but no simpler"-approach, such a requirement had not been implemented, and the abstractions for this were thus not present. The same kind of force was behind the third major architectural restructuring, and this time the code had to be restructured to support non-UML diagrams.

Following the overarching "just do it" metaphor in the research phase, the architectural refactorings in the commercialisation phase were governed by a number of different metaphors. One of the metaphors was that diagrams (i.e., the *UML metamodel* in the case of UML) should be an isolated part of the program in the sense that they should be a more or less passive part that could exist without the context of the application. This metaphor was used actively to guide all developers. The resulting architecture, in terms of the first definition in section 1, was a structuring of the code according to the blackboard architectural style [27], where other components can read or modify the diagrams (see figure 5) and register observers in order to be notified when parts of the diagrams are changed.

This focus on isolating the diagrams has in fact ensured a modular architecture, in which it is easy to add or remove components that read or modify the diagrams. For example, it is straightforward to create several "radars" (i.e., miniature displays) for the same diagram. Another benefit was that the "Tool Integrator" that implements the runtime tool integration described earlier, became an independent component, which was much easier to maintain than before the architectural refactoring.

A different type of architectural force arose about half a year later as the company hired two new developers. These were the first new developers that were going to work on the project since its beginning. To accommodate their introduction to the source code, specific code simplification and general clean-up tasks were incorporated into the development schedule several months before they were actually hired.

In the future, the software architecture needs to support several applications, i.e., it will evolve into a product line architecture [3]. A product line architecture is

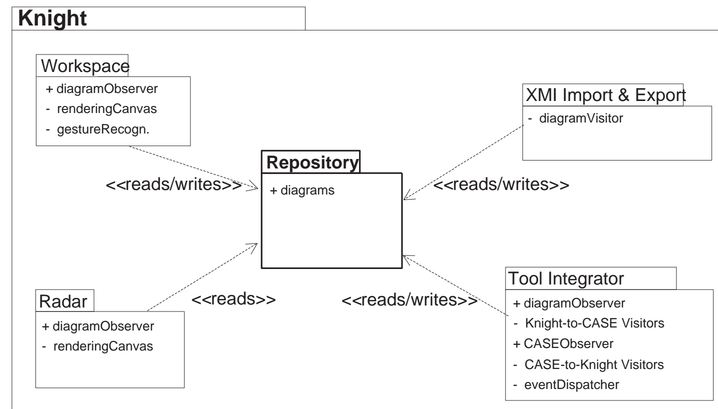

Fig. 5. Blackboard architecture

the structure and set of compontents that are shared between different products. As such, it includes the features that are shared between some or all of the products in the product line as well as the variability required to accommodate the differences between those products.

Although an effort has been done to identify and evolve reusable components whenever the same functionality was needed in several places, there still remains a lot of code that is specifically targeted towards the single product so far, namely Ideogramic UML. For example, the XML-based load/save component is currently targeted at UML, but the non-UML parts can to a large extent be reused in other XML-based load/save formats.

The transition to a product line was recently made mandatory since Ideogramic started developing a specialised diagramming tool to be used by business consultants.

However, restructurings like this invariably imply new bugs in the UML product, i.e., renewed debugging and testing resulting in a minor delay of schedule. A decision has therefore been made to extract only the most important parts of the UML product, and then let the other parts evolve gradually into shared components.

Architecture Reviews As explained in section 2, the Dragon project incorporated regular architecture reviews into the project plan, namely in connection with the business reviews. Compared to this rather systematic treatment of the architecture, the Knight Project employed a more pragmatic approach to evolving the software architecture: it was restructured whenever there was a need for it, as in the examples above with multiple diagram types and product line. This is true for both the research phase and the commercialisation phase.

4 Discussion

Section 1 characterized lightweight development and outlined potential conflicts development of software architectures. This section will try to relate the outlined properties to the use of software architecture and discuss this in relation to the presented case studies.

4.1 Adaption rather than Prediction

In the Dragon as well as the Knight Project it is interesting to note how architectural refactorings have guided the development of the architecture from specific to general. In the Dragon Project, the developed architecture supported a flexible addition of new business areas through business function components. In the Knight Project the architecture has gone from a very specific – and horizontal prototype – to one containing frameworks for the addition of new diagram types.

This generalisation is continuing in the Knight Project. As mentioned in section 3, we are currently setting up projects with external companies to extend the gesture-based modelling techniques to other problem domains than UML. It is our intention to gradually develop a product line architecture for this, *combined* with opportunities for continuing research in connection to the commercial products. It is our belief that the focus on architecture outlined in this paper in connection with our lightweight development process will enable this.

One of the main contributions of eXtreme Programming is its focus on *validation and verification* in contrast to *production* [26]. The whole apparatus of eXtreme Programming ensures that a direction set for development can and will be corrected if necessary through, e.g., continuous integration, unit testing, and refactoring. With respect to software architecture this view is very compatible with the metaphorical view of software architecture: The concrete structures and relationships between components are gradually evolved and validated using the metaphor.

Given this metaphor, it is one of the basic tenets of eXtreme Programming that evolution of a software system can be directed via a large number of small refactorings. The lack of focus on software architectural construction and evolution in Crystal “Clear” seems to imply the same sort of preconceptions. This is certainly necessary but, based on our experiences, we doubt that this is sufficient in general. In contrast to refactorings, some of the above-mentioned examples of architectural restructurings have been fundamental: from persistence via a persistent store to orthogonal, transparent persistence; from a single client to distributed clients; from procedural to object-oriented implementation.

Although our examples do not establish this in general, we believe that a focus on the larger scale structures – and metaphors – of software architecture is necessary in order to recognise and execute important refactorings of a software system.

4.2 People rather than Process

As for software development in general, it is naïve to assume that software architecture should be approached in one specific way. Every development project is unique: the actual constraints of the project and the developers assigned to it ensure this. A lightweight approach to software architecture will have to take this into account: As in the Crystal methods, the way software architecture is used will have to be crafted taking into account the unique characteristics of developers and development.

An explicit focus on software architecture holds great promises for accomodating different types of developers, though. One of the benefits we have experienced in the Knight commercialization phase, as discussed above, was that being able to describe the software architecture from a metaphorical level *as well as* being able to identify structures and relationships in the architecture helped to introduce new developers to the project.

Software architecture can in this perspective be said to be constraining: In the Dragon Project a new developer was introduced to the project at a late stage. Even though his job was to introduce new functionality quickly, the architecture constrained him to do it in a disciplined way. This points to that the constraints imposed by the software architecture – and implied process – should be evaluated and corrected as the project develops.

4.3 Development rather than Documentation

One of the future directions of our reflections on the Dragon, Knight, and other research projects is to categorize *types* of architectural refactorings. Theoretical work on this has already been undertaken by, e.g., [1] with the concept of *unit operations* and by [19] with the concept of *prototypical deltas*. What is really needed in order to support the development with refactorings on an architectural level are empirically based categorizations such as the ones that already exist for "small" refactorings [24, 15]. Combined with their context – technological and social – such refactorings could help provide a basis for development in which it is possible to a much larger extent to navigate the software architectural spaces of lightweight development projects. We believe that this will be very important in combining software architecture and lightweigth object-oriented development. And "develop rather than document".

Obvious other artefacts and technologies that are needed to pursue such a goal include tools and languages that embody software architectural entities as first-class citizens. Even though this will help close the gap between description and prescription in software architecture development, a challenge will still be to reconcile the metaphorical and structural views of software architecture.

4.4 Software Architecture in a Light-Weight Development Process

The Dragon and Knight Project case studies have argued that it is beneficial to complementarily focus on a metaphorical view of software architecture:

> *The software architecture of a software system is an overall understanding of the system that guides its construction and use*

and a structural view:

> *The software architecture of a software system is the structure(s) of a software system in terms of components, their external properties, and their interrelationships*

For each increment in functionality, the software architecture was evaluated and refactored if necessary. Change in metaphor was often connected to drastic changes in structure and often happened after a period of trying to "force-fit" a structural understanding of the system to a specific metaphorical understanding. This suggests that the intersection of changes in metaphor and structure are particularly interesting as they signal major changes.

A concrete problem is how to incorporate a focus on software architecture as advocated in this paper into existing light-weight methods such as XP and Crystal "Clear". In the case of XP, one could consider incorporating a new architectural "practice" as a replacement for the "Metaphor" practice. The Metaphor practice was alluded to in the introduction and is subsumed by the metaphorical view on software architecture.

In XP, the Metaphor is used as a "background reminder" of how the system should be constructed and thought of. It should be so concrete that the various stakeholders can use it as a common reference understanding of what the system is all about. We have argued that this should be complemented by a structural view on software architecture.

Both metaphor and structure should be used as guidance when making major changes to a software system – and both should be refactored (or just changed) as the system is evolved on an architectural level.

5 Conclusion

In this paper, we have argued that it is indeed possible to merge the benefits of explicit focus on software architecture in object-oriented development with lightweight object-oriented development methods. The resolution is based on a continuous, adaptive focus on software architecture on different levels. It thus emphasises proper working processes as an important factor behind obtaining a good software architecture. In particular, the recurring decision on and assignment of tasks should be combined with a continuous focus on software architecture.

6 Acknowledgements

The work reported in this paper has been supported in part by the Danish National Centre for IT Research (CIT, `http://www.cit.dk`), research grants COT 74.2 and 74.4.

References

1. Bass, L., Clements, P., & Kazman, R. (1998). *Software Architecture in Practice.* Addison Wesley Longman.
2. Beck, K. (1999). *EXtreme Programming EXplained - Embrace Change*, Longman Higher Education.
3. Bosch, J. (2000). *Design and Use of Software Architectures - Adopting and evolving a product-line approach*, Addison-Wesley.
4. Christensen M., Crabtree A., Damm C.H., Hansen K.M., Madsen O.L., Marqvardsen P., Mogensen P., Sandvad E., Sloth L., & Thomsen M. (1998). The M.A.D. Experience: Multiperspective Application Development in evolutionary prototyping. In Jul, E. (Ed.) *Proceedings of ECOOP'98*, Brussels, Belgium, July 1998, pp. 13-40.
5. Christensen, M., Damm, C.H., Hansen, K.M., Sandvad, E., & Thomsen, M. (2000). Design and Evolution of Software Architecture in Practice. In *Proceedings of TOOLS-Pacific 1999*, Melbourne, Australia, November.
6. Crispen, R.G. & Stuckey, L.D. (1994). Structural model: architecture for software designers. In *Proceedings of the 1994 conference on TRI-Ada*, pp. 272-281.
7. Cockburn, A. (1996). The Interaction of Social Issues and Software Architecture. In *Communications of the ACM* 39, 10.
8. Cockburn, A. & Williams, L. (2000). The Costs and Benefits of Pair Programming. In *Proceedings of eXtreme Programming and Flexible Processes in Software Engineering, XP2000.*
9. Cockburn, A. (1996). *Crystal "Clear": A Human-Powered Software Development Methodology for Small Teams.* Draft. Available at: `http://members.aol.com/humansandt/crystal/clear/`
10. Damm, C.H., Hansen, K.M., & Thomsen, M. (2000). Tool Support for Cooperative Design: Gesture Based Modeling on an Electronic Whiteboard. In *Proceedings of CHI'2000*, The Hague, The Netherlands, April 2000. New York: ACM Press.
11. Damm, C.H., Hansen, K.M., Thomsen, M., & Tyrsted, M. (2000). CASE Tool Integration: Experiences and Issues in Using XMI and Component Technology. In *Proceedings of TOOLS Europe 2000*, Mont St Michel & St Malo, France, June 5-8.
12. Damm, C.H., Hansen, K.M., Thomsen, M., & Tyrsted, M. (2000). Creative Object-Oriented Modelling: Support for Intuition, Flexibility, and Collaboration in CASE Tools. In *Proceedings of ECOOP2000*, Sophia Antipolis and Cannes, France, June.
13. Damm, C.H., Hansen, K.M., Thomsen, M., & Tyrsted, M. (2000). Supporting Several Levels of Restriction in the UML. In *Proceedings of UML'2000*, York, United Kingdom, October 2-6, 2000.
14. Fowler, M. (2000). Put Your Process on a Diet. In *Software Development.* December.
15. Fowler, M. (1999). *Refactoring. Improving the Design of Existing Code..* Addison-Wesley.
16. Garlan, D. & Shaw, M. (1993). An Introduction to Software Architecture. In Ambriola, V. & Tortora, G. (Ed.) *Advances in Software Engineering and Knowledge Engineering.* Series on Software Engineering and Knowledge Engineering, Vol 2, World Scientific Publishing Company, Singapore, pp. 1-39.
17. Hayes-Roth, B. (1995). An architecture for adaptive intelligent systems. In *Artificial Intelligence*, 72(12), pp. 329-365.
18. Ideogramic ApS. (2001). `http://www.ideogramic.com`.

19. Jacobsen, E.E., Kristensen, B.B., Nowack, P., & Worm, T. (1999). Software Evolution: Prototypical Deltas. In *Proceedings of TOOLS Asia'99*.
20. Kruchten, P. (2000). *The Rational Unified Process, An Introduction, 2ed.* Addison-Wesley.
21. Martin, R.C. (2000). *RUP vs. XP.* `http://www.objectmentor.com/publications/RUPvsXP.pdf`.
22. McLennan, M.J. (1993). [incr Tcl]: Object-Oriented Programming in Tcl/Tk. In *Proceedings of the Tcl/Tk Workshop*, University of California at Berkeley, June 10-11.
23. Object Managment Group (2000). *OMG Unified Modeling Language Specification.* OMG document `formal/00-03-01`, March.
24. Opdyke, W. (1992). *Refactoring Object-Oriented Frameworks.* Unpublished Ph.D. thesis, University of Illinois at Urbana-Campaign.
25. Ousterhout, J. (1990). Tcl: An Embeddable Command Language. In *Proceedings of the Winter 1990 USENIX Conference*, January 22–26, Washington, DC, USA.
26. Rawsthorne, D. (2001). Afterword. In Jeffries, R., Anderson, A., & Hendrickson, C. *Extreme Programming Installed.* Addison-Wesley.
27. Shaw, M. & Garlan, D. (1996). *Software Architecture – Perspectives on an Emerging Discipline.* Prentice-Hall, Inc.
28. Stapleton, J. (1995). *The Dynamic Systems Development Method Manual v2.0.* The DSDM Consortium. `http://www.dsdm.org`.
29. Wittgenstein, L. (1965). *Philosophical Investigations.* New York: The Macmillan Company.