Component-Oriented Programming

Clemens Szyperski[1], Jan Bosch[2], and Wolfgang Weck[3]

[1] Microsoft Research, Redmond, Washington
cszypers@microsoft.com - www.research.microsoft.com/users/cszypers/
[2] University of Karlskrona, Ronneby, Sweden
jan.bosch@ide.hk-r.se - www.ipd.hk-r.se/bosch/
[3] Oberon microsystems, Zurich, Switzerland
weck@oberon.ch - www.oberon.ch/

Abstract. This report summarizes the presentations, discussions, and thoughts expressed during the workshop sessions. *Full proceedings are available as a technical report* of the Department of Software Engineering and Computer Science at the University of Karlskrona/Ronneby, Sweden (http://www.ipd.hk-r.se/).

1. Introduction

WCOP'99, held together with ECOOP'99 in Lisboa, Portugal, was the fourth workshop in the successful series of workshops on component-oriented programming. The previous workshops were held in conjunction with the respective ECOOP conferences in Linz, Austria, Jyväskylä, Finland, and Brussels, Belgium. WCOP'96 had focussed on the principal idea of software components and worked towards definitions of terms. In particular, a high-level definition of what a software component is was formed. WCOP'97 concentrated on compositional aspects, architecture and gluing, substitutability, interface evolution, and non-functional requirements. WCOP'98 had a closer look at issues arising in industrial practice and developed a major focus on the issues of adaptation. WCOP'99 moved on to address issues of component frameworks, structured architecture, and some bigger systems built using components frameworks.

This report summarizes the presentations, discussions, and thoughts expressed during the workshop sessions. *Full proceedings are available as a technical report* of the Department of Software Engineering and Computer Science at the University of Karlskrona/Ronneby, Sweden (http://www.ipd.hk-r.se/).

WCOP'98 had been announced as follows:

> WCOP'99 is the fourth event in a series of highly successful workshops, which took place in conjunction with every ECOOP since 1996, focusing on the important field of component-oriented programming (COP).
>
> COP has been described as the natural extension of object-oriented programming to the realm of independently extensible systems. Several important approaches have emerged over the recent years, including

A. Moreira and S. Demeyer (Eds.): ECOOP'99 Workshops, LNCS 1743, pp. 184–192, 1999.

CORBA, COM (COM+, DCOM, ActiveX, DirectX, ...), JavaBeans. A component is not an object, but provides the resources to instantiate objects. Often, a single component will provide interfaces to several closely related classes. Hence, COP is about architecture and packaging, besides interoperation between objects.

After WCOP'96 focused on the fundamental terminology of COP, the subsequent workshops expanded into the many related facets of component software. WCOP'99 shall emphasis architectural design and construction of component-based systems beyond ad-hoc reuse of unrelated components. To enable lively and productive discussions, the workshop will be limited to 25 participants. Depending on the actually submitted positions papers, the workshop will be organized into three or four subsequent mini-sessions, each initiated by a presentation of two or three selected positions and followed by discussions. Instead of splitting the workshop into task forces, it is intended to provoke lively discussion by preparing lists of critical questions and some, perhaps provocative, statements (to be used on demand).

Position papers will be formally reviewed, each by at least two independent reviewers. As an incentive for submission of high quality statements, the best position statements will be combined with transcripts of workshop results and published.

Last year's WCOP'98 report observed: "The call for contributions in the area of systems rather than individual components and their pairwise coupling was addressed in only a minority of the submissions. It can be speculated that this is symptomatic for the relative youth of the component software discipline." Interestingly, this has changed and WCOP'99 did attract a large number of submissions in the area of component frameworks.

21 papers from nine countries were submitted to the workshop and formally reviewed. Eleven papers were accepted for presentation at the workshop and publication in the proceedings. About 40 participants from around the world participated in the workshop.

Based on the range of submissions and the relatively weak response from attendees when asked to submit ahead of time topics for discussion, the organizers decided to deviate from the announced workshop format. The workshop was organized into two morning sessions with presentations, one afternoon breakout session with four focus groups, and one final afternoon session gathering reports form the breakout session and discussing future direction.

2. Presentations

This section summarizes briefly the contributions of the eleven presenters.

Colin Atkinson (joint work with Thomas Kühne and Christian Bunse) pointed to what they call the "interface vicissitude problem." The observation is that the same logical operation requires different levels of detail at different architectural

levels, *all* of which are 'real' in that they are actually realized in an implementation. An example is a *write* operation that accepts some datum and writes it to a file. At a high level this might be architected as a method on a file object: `f.write(d)`. Within the file system implementation, the operation likely turns into `fm.write(f,d)`, i.e., a method on a file manager object that takes both the datum and the file as arguments. Finally, on the level of an ORB the operation might be coded: `orb.request("write",f,d)`. Obviously, none of these levels is the only "right" one—and, equally, none of the corresponding architectural views is "better" than any of the others. Instead, it is useful to view such architecture as *stratified*, where each strata corresponds to a particular refinement level. Colin and Thomas observed that reflective architectures are a special case of stratified architectures.

Günter Graw introduced an approach to the formalization of Catalysis, aiming at the specification of behaviour in component frameworks. His approach, called cTLA (compositional TLA) is based on Lamport's Temporal Logic of Actions and supports constraints on processes, processes interacting via joint actions, added flexibility over TLA, structured verification as is required to support components, and a basis to build theorems. Relating Günter's work to the idea of architectural strata above, it was observed that cTLA might be applied usefully to capture the mapping between strata as a set of local refinements with established properties.

Sotirios Terzis (in joint work with Paddy Nixon) observed that current concepts of trading in distributed systems are largely based on syntactic notions, which they perceive as insufficient for locating useful components. Therefore, they propose a Component Description Language (CDL) that, in conjunction with semantic trading, would allow to broaden the offering a trader comes up with when requested to locate a matching component. Attendants raised some concern whether such semantic trading was feasible.

Constantinos Constantinides (joint work with Atef Bader and Tzilla Elrad) presented an aspect-oriented design framework for concurrent systems. They view aspects and components as two orthogonal dimensions: aspects cut across a system and cannot be encapsulated. Examples include synchronization, persistence, error handling, and security. They propose a new kind of "module" that itself cuts across traditional modules (or components). Deviating from the standard "weaver" approach, which requires access to source code and which tends to explode in complexity with the introduction of new aspects, they propose to follow a moderator pattern. A moderator is an object that moderates between components and aspects by coordinating semantic interaction of components and aspects. Moderators aim to mingle binary code instead of source code–some kind of "just in time" weaving. To get moderators into the picture, all components have to be written to include fully abstract calls to moderators at critical points.

James Noble aimed to characterize the nature of component frameworks. Wondering about the difference between Brad Cox's "evolutionary programming" and component frameworks (with their provision for controlled evolu-

tion), James claims that the following three technical features are characteristic of component frameworks:

1. Component containment—the representation and implementation of a component is not exposed by its interfaces.
2. Multiple instances of interfaces—a single object (single identity) can implement multiple interfaces.
3. Interface dispatch—each interface potentially needs its own state and behaviour (the folding of interfaces with like-named methods, as in Java, is not generally acceptable).

During the following discussion it became clear that James refered to component models (such as COM) as component frameworks; his three characteristic properties therefore really characterize components, not component frameworks. Attendants wondered whether interface accessibility should be linked to levels of refinement (like architectural strata).

Alex Telea presented his Vission System, a component framework supporting the construction of rich and diverse visualization and simulation systems. As a mechanism, Alex enhanced C++ with a dataflow mechanism and provided his Vission environment as a frontend. The approach combines the advantages from object orientation (persistance, subtyping/polymorphism) with advantages from dataflow approaches (modularity, visual programming). His components use inheritance to incorporate aspects. Using meta-class wrapping, Vission can reuse the massive existing base of scientific and visualization code—his presentation included a number of impressive examples.

Ole Madsen reported that he had been intrigued by a suggestion by Clemens Szyperski at last year's WCOP, where Clemens suggested that first-class COM support in Beta would be a good way to help Beta off its "island". Ole had followed through since then and reported about the lessons learned from supporting COM in Beta. They used nested classes to implement multiple interfaces, which worked well and which naturally allows Beta to implement COM classes. While COM was found nice conceptually, there are a number of messy details such as the parameter space. The rather weak available "specification" of COM IDL was seen as a major obstacle.

Ulrik Schultz discussed his idea of "blackbox program specialization." Program specialization aims to configure a generic component to perform well in a set of specific situations. Specialization operations require access to the implementation, while the selection of a specialization operation depends on the intended usage. Thus, in a component world, it is the component implementer that understands how specialization can be applied and what benefits can be gained, but it is the client using a component that knows which particular specialization would be applied best. Ulrik therefore proposes that components should implement extra interfaces that provide access to specialization operations. These can then be called by the client in its particular context.

Henrik Nielsen (joint work with Rene Elmstrom) pointed out that there is a huge practical problem arising from the combination of blackbox abstraction and versioning. Fundamentally, with components one cannot trust what cannot

be *locally* verified. Therefore, detailed (formal) documents and guarding runtime checks (perhaps using wrappers) are all required. Further, since component management is quite cumbersome, he suggests that better tool support for dependency management among generations of groups of components is needed. Henrik pointed out that there is a Danish national initiative that addresses some of these issues: the Centre for Object Technology (`http://www.cit.dk/cot`).

Jing Dong (joint work with Paulo Alencar and Donald Cowan) focused on design components and the question on how to compose these properly. By "correct composition" he means that components don't gaine or lose properties under composition. He illustrated his point by showing how the iterator and the composite design pattern can be composed. To reason about composition of design patterns, patterns need to be enriched by theories. Composition then is theory composition and correctness of a composition can then be proved by showing that all involved component theories still hold. Jing claimed that this approach could be generalized to other (non-design) components. When asked, Jing explained that his specifications deal with aliasing explicitly and that state changes are restricted by specified constraints.

David Helton proposed the view that coarse-grained components actually work in practice and could be seen as an alternative to finer-grained ones embedded in component frameworks. He pointed to Gio Wiederholt's CHMIS project at Standord: a research system that uses coarse-grained components. David emphasized that such coarse-grained components are reality today, mentioning systems by SAP, Baan, and PeopleSoft as examples. Aiming for a synthesis, he suggests that several concepts from the small-grained component world should be questioned when considering coarse-grained ones instead. For example, are classes, polymorphism, and other OO concepts needed at this level? (He thought: probably not!) He proposed to eliminate at the highest level all of implementation and interface inheritance, polymorphism, and possibly even instantiation, and claimed that this was similar to what was done in the CHMIS project.

Jan Bosch closed the morning presentations with a brief statement of his own, entitled "Software Architecture / Product Line Architectures and the real world." His presentation started with the identification that components can be reused at three levels, i.e., across versions of a product—which is mostly understood, across product versions and a product family—the current state of practice is learning this, and, finally, across product versions and product families and organizational boundaries—except for narrow, well established domains, we are nowhere near learning this trick. His main advice to organizations interested in converting to component-based software engineering was to first establish a culture in which intra-organizational component reuse was appreciated, before proceeding with full-scale third party component reuse. The main argument for this was that, in his experience with a number or industrial software development organizations, establishing a software product-line, which is based on intra-organizational component reuse, already results in so many problems that adding the additional dimension of complexity caused by third party components may exceed the ability of the organization. In addition, the benefits of a

software product line over traditional product-oriented development are already quite substantial.

3. Breakout Session

The following five breakout groups were formed:

1. Architectural strata and refinements.
2. Adaptation, aspect moderation, and trading.
3. Component frameworks.
4. Implementation problems with "real COM".
5. Practice versus theory/academia.

Most of the groups had between five and twelve members (one had only three), all with a designated scribe and a designated moderator. The following subsections summarize each groups findings.

3.1 Architectural Strata and Refinements

This group did not agree on interaction refinement as *the* mechanism for architectural refinement, but did agree that interaction refinement is an important concept. The group observed that strata refinement is sometimes automatic (as is the case with remoting infrastructure) and that in such cases the stratification itself is abstracted into separate components. The group wasn't clear whether there was a difference between interaction refinement and normal decomposition (and if there was a difference, which these differences would be). They perceived a spectrum of solutions between "inlining" and abstraction into separate components.

3.2 Adaptation, Aspect Moderation, and Trading

This group came up with a taxonomy of adaptation levels:

- adaptation
  - system (adaptation from inside)
    - collection of components (replace component)
    - composition of components (rewire components) [1]
      - topology
      - semantics of connectors
  - component (adaptation from outside)
    - appearance (interfaces)

[1] Two remarks: (a) Colin Atkinson observed that this distinction might be equivalent to the system versus component distinction at the top-level, if properly applying interaction refinements; (b) Walt Hill pointed out that it would be useful to separate the specialization of adapters from the rewiring/reconfiguring step.

    * behavior
        · properties (data)
        · implementation/specialization (code)

They also noted further dimensions of adaptation:

- blackbox vs. whitebox
- functional vs. non-functional (perceived as a more difficult, less quantifiable, and therefore unclear front of attack)
- when to adapt? (with a full spectrum between compile-time and fully lazy - for safety and performace: do as early as possible)
- ORB vs. Trader vs. Specialization

3.3 Component Frameworks

This group tried to apply the notion of architecture classification (Atkinson et al.) to the Vission framework (Telea). They observed a tension for architectural classification: levels vs. layers. The traditional approach suggests the following layering picture:

<table>
<tr><td colspan="2">enduser</td></tr>
<tr><td>data flow manager</td><td>GUI manager</td></tr>
<tr><td colspan="2">meta classes</td></tr>
<tr><td colspan="2">C++ classes</td></tr>
</table>

Unfortunately, this layer structure is disturbed by cross-layer relations preventing any linear ordering of the bottom three layers. There are relations between layer 3 and 2, layer 2 and 1, and layer 3 and 1.

The same architecture is described very differently when using architectural strata. One stratum is assigned to each class of user of the system: one for endusers, one for application designers, and one for component designers.

<table>
<tr><th>stratum</th><th>structure</th></tr>
<tr><td>enduser</td><td>GUI Mgr dataflow components</td></tr>
<tr><td>application designer</td><td>GUI Mgr DF Mgr meta classes</td></tr>
<tr><td>component designer</td><td>GUI Mgr DF Mgr meta classes C++ classes</td></tr>
</table>

It is possible to understand strata top-down (similar to proper layers).

The group concluded that mapping user layers to architectural strata worked well in this example. Also, layer and strata views are complementary. An open research question is whether component relations can be constrained to reside within strata rather than crossing strata.

An interesting further observation is that, at least in this example, components are found in successive strata. It was speculated whether indeed those

things that—once revealed by a stratum—stay, can be captured by separate components. That is, strata-based diagrams seem to expose the notion of components (objects?) explicitly, as these are the entities that appear in successive strata.

A number of issues are considered open:

- Is traditional "strict" layering too rigid?
- Does it conflict with object segregation?
- Would the stratum model improve on this?
- What's the relation between strata and aspects? (Some relation seems plausible.)
- Is it a problem of layering that it tries to segregate interfaces *and* implementations?

3.4 Implementation Problems with "Real COM"

This group started discussing the Beta implementation of COM but then progressed to more general issues. A summary of the topics discussed let to the following laundry list:

- Components should have strong specifications (including pre/post- conditions and invariants).
- Making robust components on top of non-robust/not-well-defined platforms is *hard*.
- Technology-independent components are an important research goal; but this is *very* difficult. (Clarification: the independence looked for is independence from platform-specific services and implementations.)
- Component technology vendors need to provide examples and explanations which make it easier for implementers who are building components.

A question from the audience as to how to teach people to get all these things right was answered by one of the group members: "You don't. This will remain largely the expertise of a few." (A pesimistic view indeed—although backed by the industrial experience with other non-IT component technologies: after initial shakeouts only a few champions remain in the race, usually with a rich repertoir of proprietary expertise and technology.)

The suggestion that linguistic support for specification aspects was required (such as pre/post-conditions) was seen as a good idea.

3.5 Practice versus Theory/Academia

This group considered the success factors for using theory in practice:

- Mature domains: creation of stable domain models is possible.
- Degree of criticality/complexity: there is a compelling reason to reason.
- Organizational maturity level (in the CMM sense) required to get this right.

– Scoped usage of theory: don't try to throw the biggest hammer at everything. (This last point in particular requires great care. It is not obvious how to contain the "spread" of a theory.)

In the context of components the group observed:

– Theory/formal methods can be naturally confined to specific components, leading to a natural scoping approach.
– Components can bridge the gap between theory and practice: the results of applying a particular theory in to a specific problem can be reused in component form, helping to ammortize the initial investment.

There are two further issues:

– Politics/culture: component approach establishes new dependencies inside and between organizations.
– Cost/benefit: component approach requires an ammortization model.

4. Concluding Remarks

Concluding, we are able to state that, on the one hand, also the fourth workshop on component-oriented programming was a highly successful event that was appreciated by the participants, but, on the other hand, that the issues surrounding component-oriented programming have not been solved and that future events remain necessary to further the state of the art.

One interesting trend that we have identified over the years is a change in focus. Whereas the first workshops aimed at the individual components, especially during the last event there was a clear consensus that the focus should be on the component architecture, i.e., the cooperation between components. In particular, not the component-'wiring' standards, such as CORBA, COM and JavaBeans, but rather the system design level issues, such as software architectures and component frameworks.