Position Paper
OOPSLA'97 Workshop "Object-Oriented Design Quality"

Kai-Uwe Mätzel, Dirk Riehle

Ubilab, UBS, Bahnhofstr. 45, CH-8021 Zurich, Switzerland
e-mail: {maetzel | riehle}@acm.org

What is quality?

Quality is not an absolute intrinsic measure neither for tangible goods nor for intangible ones such as object-oriented design. Quality can only be measured relatively to a particular point of reference and depends on the assessor's view of the objects to measure as well as the reference point. The view defines the various aspects to be considered regarding measuring quality such as reliability, performance, or security.

These aspects span a multi-dimensional space in which no universal linear ordering relationship can be defined regarding quality. We refer to the view and its related multi-dimensional space as viewpoint. If we change the viewpoint then that what we consider good quality changes as well; if we change the point of reference then the degree of the quality changes.

In the realm of object-oriented design, quality is usually measured from several distinct viewpoints relative to a thought reference system. The viewpoints define domain-specific and/or domain-independent aspects like maintainability, evolvability, or reusability. More precisely, these are, for example, the average cost of maintenance per year, the required effort to evolve the system, and the probability of reuse of the system or parts of it - all of that along typical scenarios incorporating domain-specific as well as domain-independent factors.

In all cases, what we basically measure is the amount of redesign and implementation work necessary to make a system meet the scenario's requirements. This opens the way to "quality-generating" design properties. We can analyze which changes, adaptations, etc. are most frequent (either in respect to a particular or to the most common viewpoints) and look for design solutions that make these changes unnecessary or at least very easy to do. Hence, given a viewpoint, we can assess the quality of a design by investigating whether it exhibits the appropriate properties in the sound way or not.

There are already well established quality-generating properties in the form of design patterns [Gam95]. However, regarding quality according to a particular viewpoint, it is not sufficient to count the number of viewpoint related design patterns a design incorporates. More than that, it is necessary to investigate if the patterns are used at the right places and in the right combinations. For example, it is counterproductive to use the Visitor design pattern in a case where the algorithm to be performed over a heterogeneous data structure is never going to be changed. If the algorithm does not change, the high number of introduced indirections would only hamper the system's performance. Furthermore, the application of the Strategy pattern is practically useless if none of the creational patterns such as Abstract Factory or Prototype is involved because the decoupling could then only be incomplete.

Although with a different attitude, pattern languages [Cop97] and composite design patterns [Rie97] both focus on the effect of the combined application of design patterns and their interplay. Pattern languages emphasize the generative aspect by providing guidance in the application of patterns and demonstrating the need to use several patterns collectively. Composite design patterns describe recurring, complex organizational object structures that can be explained as a particular, constrained combination of a set of design patterns.

However, not all quality-generating properties can be found explicitly in the design. The quality of a design along the various quality dimensions of a given viewpoint also depends for example on the used implementation infrastructure. Implementing a design using a reflective object system usually leads to a higher degree of dynamic adaptability than building it using plain C++ even if the according issues are not explicitly addressed.

Quality regarding unanticipated software system evolution

Next, we will take the perspective of unanticipated software system evolution and discuss design quality according to this particular viewpoint. Regarding anticipated evolution, it is proven practice to make the potential subjects of change explicit parts of an object-oriented design. For example, the algorithm is the expected subject of change of the Strategy design pattern and, therefore, explicitly represented by a separate object. Thus, it can be easily exchanged.

In respect to unanticipated evolution, the subject of change is not known in advance. But what is known are the observable consequences of evolution. Intended collaboration between objects suffers from broken interfaces or changed semantics. Evolution causes collaboration mismatch. Thus, a design shows a high quality regarding unanticipated evolution if it allows to handle collaboration mismatch flexibly. One way to accomplish that is to make objects to a certain degree resistant to changes of their collaborators by using flexible object coupling mechanisms.

Elaborating flexible object coupling mechanisms makes them quality generating properties in the explained sense. Each flexible object coupling mechanism provides a certain degree of change resistance along certain dimensions of message-based object coupling. In [MB97], we elaborate the dimensions of object coupling and present in detail one particular set of flexible object coupling mechanisms. The table below provides an overview of this set of coupling mechanisms by showing the dimensions they are making more flexible.

	identity of partner	message format	dispatch strategy	lookup process	data representation	data model negotiation	event/request ratio	required/provided types	object modification	dyn. type modification	dyn. system structure mod.
self-describing method calls		**x**	**x**		**x**	**x**					
anonymous communication	**x**			**x**			**x**				
trading	**x**			**x**			**x**	**x**			
Dynamic Facade								**x**			**x**
roles								**x**	**x**	**x**	
Dynamic Objects									**x**	**x**	

Investigating these mechanisms and their relating dimensions as presented in [MB97], we discover that three different sets of mechanisms can be separated. All mechanisms of a set applied in combination provide essential flexibility regarding one particular problem field:

- Self-describing method calls, anonymous communication (e.g. publish/subscribe), and Dynamic Objects (objects whose structure and behavior can be dynamically changed) allow flexible bridging of collaboration mismatches regarding event-based collaboration.
- Self-describing method calls, object trading, and roles [RWL96] are well suited to prepare a system for unanticipated evolution in the case of request-based collaboration.
- Dynamic Facades are relevant for dynamic system structure modification.

Thus, we have design elements (the coupling mechanisms) and combination rules (the sets and the problem field they tackle) that provide for the flexible handling of collaboration mismatch and, therefore, if sensibly used in a system, “generate” a high quality regarding unanticipated evolution. This enables us to answer the question what a good object-oriented design, more precisely a good object-oriented design regarding unanticipated evolution, is. From the viewpoint of unanticipated evolution, we consider a design which makes use of the mentioned coupling mechanisms a good design if it uses the mechanisms according to the identified sets to make request- and event-based object collaboration as well as dynamic system structure collaboration flexible.

Since these coupling mechanisms are highly dynamic, they raise issues such as a higher probability of run-time failures, lower run-time efficiency, and stronger resource requirements. This reveals that quality-generating properties regarding one viewpoint can be quality-decreasing regarding another. Therefore, the application of those properties must always accommodate the requirements of various possible viewpoints of a design.

In the case of unanticipated evolution, faced with the mentioned problems, it becomes clear that the use of these coupling mechanisms has to be restricted to those places where they are really promising and necessary. Thus, a mechanism to determine these places is required which is given by Design for Slippage explained in [MB97] as well.

The authors

The authors works as researchers in the Distributed Object-Oriented Software Engineering Group at Ubilab, the UBS Information Technology Laboratory. Currently, they work on Geo, a distributed reflective object system. Kai focuses his research on software evolution. Previously, he worked on Beyond-Sniff, a distributed multi-user software development environment which was commercialized beginning of 1996. Dirk’y research focus is on metalevel architectures. He has a strong background in the Tools and Material Metaphor.

Workshop goals

Part of our work is to consult IT projects within UBS departments. So far, we have experience with only a very limited number of viewpoints, for example unanticipated evolution as presented above. We would like to share our experience and learn about other viewpoints. Furthermore, we would like to discuss whether quality-generating properties shift or change if the scale of a system is getting larger or not.

References

[Cop97] Coplien J. (1997). Pattern Languages. Journal of Object Oriented Programming, Vol. 9, No. 1, pp. 15-21.

[Gam95] Gamma E., Helm R., Johnson R. & Vlissides J. (1995). *Design Patterns: Elements of Reusable Design*. Addison-Wesley.

[MB97] Mätzel K.-U. & Bischofberger W. R. (1997). Designing Object Systems for Evolution. To appear. *TAPOS*, Vol.3, No.4.

[RWL96] Reenskaug T., Wold P. & Lehne O.A. (1996).*Working with Objects*. Manning.

[Rie97] Riehle D. (1997). Composite Design Pattern. To appear in *OOPSLA'97 Conference Proceedings*, ACM Press.