

Should Requirements Be Objects?

Martin Glinz
Department of Informatics
University of Zurich
Winterthurerstrasse 190
CH-8057 Zurich, Switzerland
<http://www.ifi.unizh.ch/~glinz>

Abstract. This position paper discusses arguments in favor and against regarding requirements as objects. As there are good arguments for both standpoints, a partial synthesis is developed. The synthesis is based on the idea of not treating requirements as objects, but using objects for organizing and understanding requirements.

Introduction: A naive approach to the problem

When we take a naive approach to the problem, we can easily demonstrate that every requirement is an object. Just consider Figure 1, which is a rather simplified model of a requirements specification. In this model, every requirement is an object. However, it is not this kind of objects that we are interested in. Moreover, if we examine the model in Figure 1 more closely, we find that requirements are *represented by objects* in that model, which does not necessarily mean that they *are* objects.

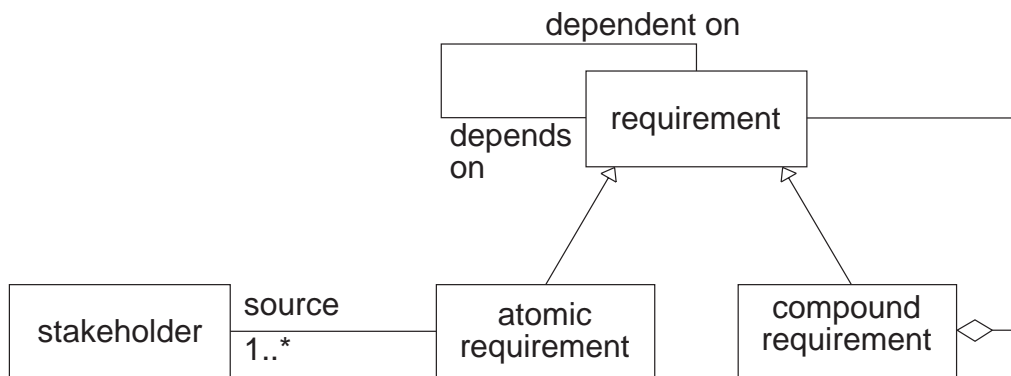


Figure 1. A simple conceptual model where every requirement is an object

So, what does it mean if we regard a requirement to be an object? Let's start from a definition: An *object* is an element in a problem or solution domain which has an identity and is clearly distinguishable from other objects. It typically encapsulates states and behavior.

As a requirement typically states properties or constraints that elements in a solution domain must have or satisfy in order to solve a problem from a problem domain, the notion of requirements being objects seems reasonable at a first glance.

In this position paper, I present arguments both in favor and against regarding requirements as objects and explore a partial synthesis of the two standpoints.

Point: Why requirements should be objects

Structuring a problem or a solution with objects allows encapsulation of states and behavior and, hence, abstraction, decomposition and information hiding. Abstraction, decomposition and information hiding in turn are the principal means for understanding complex systems of any kind.

When looking at traditional natural language-based requirements specifications, such specifications are frequently plagued by little structure, missing abstractions and no information hiding, which yields specifications that are hardly comprehensible. We could get rid of all these problems by treating requirements as objects and applying the abstraction, decomposition and information hiding capabilities that are inherent in object models.

Furthermore, if we employ state-of-the-art object-oriented design and implementation techniques, an object-oriented requirements specification would allow a seamless application of object-oriented software engineering methods through the complete development cycle, from inception to deployment. We would get a smooth transition from requirements into architecture and design and could apply round-trip engineering methods and tools. So why longer hesitate? Just let requirements become objects.

Counterpoint: Requirements as objects considered harmful

The promises of abstraction and comprehensibility sound good, but—treating requirements as objects is like making a problem fit a solution, instead of doing it vice-versa. What is a requirement? A requirement may be a goal, for example “The new CRM system shall reduce the number of customer complaints by at least 50%.” Is this an object? What does it encapsulate? Has it a state or behavior? Not really. So let’s try another kind of requirement. A requirement may be a function, for example “The system shall compute the maximum speed that the train can run with on the current track segment.” Is a function an object? Definitely not. So let’s again try another kind of requirement. A requirement can be a constraint, for example “In normal operating mode, the lift shall never move when the doors are not closed completely.” But again, a constraint is no object.

These are not just poorly picked examples. If you need more evidence that requirements are no objects, consider requirements in the form of use cases, qualities (such as usability), or properties (such as accuracy of computed values). Neither of these elements are objects. While there are some situations where requirements can be regarded as objects (for example, in the specification of an application domain), it should be clear now that the notion of requirements being objects is awfully wrong and, hence, has to be considered harmful.

Synthesis: Requirements are no objects, but they should be organized using objects

So what can we do if we follow the counterpoint argument that—at least in many situations—requirements are no objects, but also buy the argument that objects, abstraction and information hiding are the key to understanding complex systems?

In my opinion, a partial, highly beneficial synthesis is possible when we abandon the notion

of requirements to be objects and use objects for organizing requirements instead.

Let me explain what I mean by organizing requirements with objects. Typically, a given problem P can be decomposed into sub-problems so that every sub-problem encapsulates some part of the overall problem. This decomposition process can be applied recursively, yielding a problem decomposition hierarchy. Furthermore, when looking at a sub-problem as a whole, it typically has states and behavior. Hence, such a (sub-)problem hierarchy can be adequately represented by an object decomposition hierarchy, where the decomposition follows the principles of abstraction and information hiding. When we now look at the requirements for a system that solves problem P , we can determine for every requirement to which sub-problem(s) it pertains. Assume that a requirement r pertains to three sub-problems, say p_r , p_s , and p_t . Then we determine the lowest-level sub-problem p_k which contains the sub-problems p_r , p_s , and p_t and allocate r to the object representing p_k .

Thus we arrive at a hierarchically decomposed object structure of requirements which features abstraction and information hiding and, hence, exhibits the benefits of comprehensibility.

Object-like requirements become objects in this structure, others such as functions, use cases, properties, etc. that pertain to a range of sub-problems become a part of the lowest level object that encompasses this sub-problem range. Of course, if a requirement cross-cuts all sub-problems, it has to be allocated within the root object of the hierarchy. But—for understanding the problem this is the place where such a requirement belongs.

Please note that we are dealing with object decomposition hierarchies here, not with inheritance hierarchies. Not every object modeling language supports hierarchical object decomposition. For example, UML 2.0 [3] does, while UML 1.0 to 1.5 [2] don't. In my own research group, we have investigated the object decomposition problem for years and have developed a modeling language called ADORA [1] which is constructed upon this idea.

References

- [1] Glinz, M., S. Berner, S. Joos (2002). Object-oriented modeling with ADORA. *Information Systems* **27**, 6. 425-444.
- [2] *OMG Unified Modeling Language Specification*. <http://www.omg.org>
- [3] OMG (2003). *UML 2.0 Superstructure Specification*. OMG Final Adopted Specification, document ptc/03-08-02. <http://www.omg.org/cgi-bin/doc?ptc/2003-08-02>