Differentiating requirement types from instances: architectural thinking as a pre-design activity

R. Ramaswamy

Infosys Technologies Limited

3rd Cross, Electronics City, Bangalore 561 229, India
Ph: +91 80 852 0261 Fax: +91 80 852 0362
e-mail: ramkumarr@inf.com

Abstract

The success of an OO project hinges on the integrity of its OO architecture. How sure can one be that a proposed object model is sound and will not change *structurally* as requirements are 'fleshed in' during development? This question becomes particularly critical in a software reengineering/redesign scenario in which business rules may already be known and documented in some detail, so that the development team is expected to transition into design and implementation even before current business functionality is comprehensively understood. In such a scenario, thinking of software architecture as a design activity may not be entirely appropriate. In this paper we address the above issue by showing how the use of a simple but highly effective and shop-usable approach to understanding and abstracting a set of *requirements* leads to crucial early insights on OO architecture. We thus argue that architectural thinking can begin in a natural way during requirements specification. As a corollary, the approach also injects much-needed clarity into the definition of what must constitute an initial executable 'architectural slice'. While our approach is described in an OO context, the underlying ideas are easily seen to be applicable in more general situations.

Keywords

Object-oriented architecture, requirements specification, software reengineering

1 INTRODUCTION

Software architecture is viewed as one of the levels in software design (Shaw and Garlan, 1996). Specifically, it is viewed as the manifestation of the earliest design decisions about a system (Bass *et. al.,* 1998). In particular, an object-oriented architecture–consisting of a set of classes and a specification of their collaborations–embodies crucial structural decisions and has sweeping implications for detailed design, implementation and testing (Booch, 1996). The activity of object modeling of course, begins during requirements specification, but the objects identified at that stage are usually logical in nature, corresponding to *entity objects* (see (Jacobson *et. al.*, 1992) for a discussion of entity, interface and control objects). This yields a *logical object model* (LOM), which is structurally not very different from the traditional data model (see, e.g., (Reingruber and Gregory, 1994)). The LOM undergoes changes during architectural design–new objects may be added for control and interface, and existing ones may be split, combined or reorganized. The resulting object model is the *design* or *physical object model* (POM), and this is what drives detailed design and implementation. The POM represents key decisions on object structure and collaboration. For this reason, it is important to minimize structural changes to the POM once detailed design has begun.

It is particularly interesting to examine the forces that influence the requirements specification (RS) and design activities in a reengineering/redesign scenario. Such a scenario typically involves redesigning a chunk of an existing application (using, say, OO techniques) without significant change to existing functionality. Such an exercise is usually conducted to make the system more modular and flexible for maintenance and future enhancements. To the extent that business rules are documented in some detail, RS in such a context is reduced to a short systems study, constituting a relatively small proportion of total effort. Further, the development team is–perhaps unreasonably–expected to transition into design and implementation as soon as business functionality is 'broadly understood.' This forces the team to begin design without a comprehensive understanding of business rules, and consequently reduces the scope for evaluating and closing out on architectural options early during design. There is thus the apprehension that complex business rules lurking in some dark corners of the current system will, when excavated, upset the apple cart by requiring structural readjustment of the object model.

How must such a situation be handled? Specifically, we ask the following questions. What constitutes a 'broad understanding' of business requirements?

How early can ‘architectural thinking’ begin? How sure can one be that there will not be turbulent changes to the OO architecture once detailed design has commenced and requirements and business rules are excavated and ‘fleshed in’? We address these questions in the next section

2 REQUIREMENT TYPES AND INSTANCES

Consider the following passage which is an edited extract from an actual business rules document that we encountered, and appears to be written well enough to not require significant rewriting. (In the interest of brevity, we have provided a relatively simple though sufficiently illustrative example.)

AB0090032 Batch file description
This batch file will be received by the system once a day. Each batch file will contain about 300,000 records. Information in this file must be validated before the contents are stored in the database. Each record is composed of 30-43 fields containing detailed information about a credit card transaction. Some key descriptions are reproduced below:

1. The cardType field in each record must be Visa, MasterCard, Amex or Discover. This type must correspond with the txnType field in the same record; a '1' indicates Visa or MasterCard, and a '2' indicates otherwise.
2. The accountNumber for each transaction must have exactly 16 digits; in addition, the first 2 digits must be '55', ‘56', ‘65' or '66'.
3. The value of the NumRec field in the batch header record equals the number of records in the batch. Similarly, the TotVal field in the batch trailer record equals the sum of transaction values of all records in the batch. Sometimes there is a discrepancy, and if there is, an alarm must be raised.
4. If there is a transaction with IndType equal to 'UY', (indicating a hotel check-in), and another transaction with IndType equal to 'RU' in the same batch with the same account number, the txnQualifier in the two records must match.

We provided this passage as a sample extract of the documentation for a current system to a group of a dozen analysts, and asked them how they would proceed to assimilate its contents. Specifically, we invited opinions on whether they felt there was a need to edit, reorganize or summarize this extract in any way. One or two analysts felt that the document could be left as-is, while the rest felt they would like to add on various summary statements that typically read as follows:

There are 4 card types.

The system deals with only those account numbers that begin with certain prespecified values.

An alarm is raised if there is any discrepancy in the batch header or trailer

When interrogated, the analysts were unable to provide concrete justification for providing these summaries apart from ‘Summarization helps me get the big picture’. There was unanimity in the view that the summary itself would serve little

purpose once the requirements had been 'broadly understood' and the systems study phase was completed.

The notion of a *requirement type* lends sharper meaning to the phrases 'broad understanding' and 'big picture'. It focuses on identifying distinct *responsibilities* that need to be assigned to components in the OO architecture. (As we will see, this approach is analogous to data modeling, in which we focus on capturing entity types, rather than instances, during requirements specification and database design.) Consider the following summary of the above requirements extract.

There are three kinds of field validations:

Local validations, which verify that the given field belongs to an admissible domain

Intra-record validations, which verify that a given field's value is consistent with the value of other fields within the same record.

Inter-record validations, which verify that a given field's value is consistent with the value of fields in other records within the batch

This summary is qualitatively different from the earlier ones. It clearly identifies three *types* of validations (local, intra-record and inter-record), without spelling out any *instances* of these three types. For example, the rules about admissible values of card type as well as the rules about admissible values for account number may be viewed as *instances* of local validation. Similarly, the correspondence rules between card type and transaction type are instances of intra-record validation. How does this help? Different requirement types may need to be architected for differently (e.g., may require a different architectural policy or style), but multiple instances of a requirement type can be expected to be handled using the same policy (or very minor variations of the policy). This allows us to ignore multiple instances of each identified requirement type during RS; it is more important to focus on identifying as many requirement types as we can.

This approach achieves two objectives. One, it aids architectural decisions by bringing into sharp focus the spectrum of responsibilities embodied in the different requirement types. Two, it allows an analyst conducting a requirements study in a reengineering/redesign scenario to manage and time the transition into design, without being fully conversant with the body of current business rules.

To complete the illustration, let us return to the example passage above. Suppose we have classes Record and Batch (amongst others such as subclasses of Record) to hold, validate and store information in an incoming batch file. How would the three types of validations be handled by these classes? Typically, local validation of a field will be handled during object construction by a set() method on the appropriate class (in this case the appropriate subclass of Record) that contains the field as an attribute. Intra-record validation is also handled by the appropriate set() method during object construction: for example, setCardType() will check the value of the txnType attribute (assumed to have already been populated) before populating the value of cardType. Finally, inter-record validation can be handled

by the Batch class: each time an instance of Record is added to a Batch using the method Batch::addRecord(), its consistency with other objects already within Batch is checked within the method.

One of the important themes in software engineering is the separation of analysis and design concerns. For example, the notion of 'system essence' (McMenamin and Palmer, 1984) was mooted as a technique to ensure that a requirements specification does not get 'polluted' by decisions that reflect constraints imposed by implementation technology. The key to specifying requirements essence is to ask, 'Are there requirements that would cease to exist if I had perfect technology available to implement the system?' In a redesign/reengineering scenario, we would ask the same question with reference to the chunk of the system being reengineered (and which is within one's control). Clearly, the distinction between requirement types and requirement instances is independent of implementation technology. This means that the act of distinguishing between the two may legitimately belong to the RS phase, which implies that *the process of architectural thinking can begin naturally during requirements specification*.

A caveat, however is in order: our experience has shown that focusing on requirement types does lead to a temptation to cross the boundary and move deeper into 'nonessential' (in the sense defined above) architectural decision-making that should properly be done after RS. It is important to resist this temptation.

3 COROLLARIES

The articulation of requirement types throws up an unexpected benefit during incremental integration and testing. A crucial step in system development is the verification of an architecture by taking an *executable slice* through it. An executable slice 'carries out some or all of the behavior of a few interesting scenarios chosen from the analysis phase,' and 'should provide partial implementation for the entire domain model (Booch, 1996).' If the dominant risks to the project are technological, a vertical slice is recommended, while a horizontal slice is recommended if the dominant risks involve the logic of the system. In a complex business application such as one involving card transaction processing and settlement, a horizontal slice may be preferred. However, it is common to find developers arguing about what exactly should constitute this slice. Words such as 'interesting' and 'partial implementation' are usually too ambiguous and subjective to be shop-usable as-is. The notion of requirement types makes the answer relatively easy: take *one instance* of each requirement type and wire these together to create an executable slice. In the example passage above, we could implement clause #1 (one instance each of local and intra-record validation), along with the validation of the NumRec field in the batch trailer record (inter-record validation). This sets up a well-defined end-to-end processing sequence early in the game. Fleshing in the rest of the application then becomes a matter of adding in the remaining instances in an incremental manner.

Finally, the focus on capturing requirement types and excluding requirement instances is extremely useful in defining and monitoring the scope, milestones and deliverables of the RS phase in a reengineering/redesign scenario, in which there is often some pressure on the analyst to produce tangible output as evidence of sufficient understanding of the current system! This is small but important consideration from a practitioner's point of view.

4 SUMMARY AND CONCLUSION

In this paper, we have addressed the issue of architectural stability of the POM with special reference to a reengineering/OO redesign scenario. We have argued that thinking of software architecture as a design activity may not be entirely appropriate in such a scenario, due to the expectation that the team will begin design and implementation without a comprehensive understanding of business functionality. We have shown how focusing on requirement types helps the analyst drive key architectural decisions in a natural way during requirements specification without compromising on the principle of system essence. The focus on requirement types also helps the analyst manage and time the early transition into design despite not having a comprehensive understanding of business functionality. Finally, we have argued that the approach injects much-needed clarity into the definition of an executable 'architectural slice'. Our approach has evolved out of OO projects, but it is clear that the underlying ideas have broader applicability. Our experience has proven it a simple idea with a tremendous return on investment.

6 REFERENCES

Bass L., Clements P. and Kazman R. (1998) Software Architecture in Practice. Addison-Wesley Longman, MA.

Booch, G. (1996) Object Solutions. Managing the Object-Oriented Project. Addison-Wesley, MA.

Jacobson I., Christerson M., Jonsson P. and Overgaard G. (1992) Object-Oriented Software Engineering. Addison-Wesley, MA.

McMenamin S. J. and Palmer J. F. (1984) Essential Systems Analysis. Prentice-Hall, NJ.

Reingruber M. C. and Gregory W. W. (1994) The Data Modeling Handbook. A Best-Practice Approach to Building Quality Data Models. John Wiley, NY.

Shaw M. and Garlan D. (1996) Software Architecture. Perspectives on an Emerging Discipline. Prentice-Hall, NJ.

7 BIOGRAPHY

Ramkumar Ramaswamy received a PhD in Operations Research and Systems Analysis from the Indian Institute of Management, Calcutta, in 1994. He is currently an associate project manager at Infosys Technologies Limited. His interests include software architecture, methodologies for systems analysis and design and algorithms for combinatorial optimization.