On the Definition of Service Granularity and Its Architectural Impact

Raf Haesen[1,2], Monique Snoeck[1], Wilfried Lemahieu[1], and Stephan Poelmans[2]

[1] Department of Decision Sciences & Information Management,
Katholieke Universiteit Leuven, Belgium
firstName.lastName@econ.kuleuven.be
[2] Hogeschool-Universiteit Brussel, Belgium
firstName.lastName@hubrussel.be

Abstract. Service granularity generally refers to the size of a service. The fact that services should be large-sized or coarse-grained is often postulated as a fundamental design principle of service oriented architecture (SOA). However, multiple meanings are put on the term granularity and the impact of granularity on architectural qualities is not always clear. In order to structure the discussion, we propose a classification of service granularity types that reflects three different interpretations. Firstly, *functionality granularity* refers to how much functionality is offered by a service. Secondly, *data granularity* reflects the amount of data that is exchanged with a service. Finally, the *business value granularity* of a service indicates to which extent the service provides added business value. For each of these types, we discuss the impact of granularity on a set of architectural concerns, such as performance, reusability and flexibility. We illustrate each granularity type with small examples and we present some preliminary ideas of how controlling granularity may assist in alleviating some architectural issues as we encounter them in a large-sized bank-insurance company that is currently migrating to SOA.

Keywords: granularity, service oriented architecture, component based development, architectural qualities, impact analysis.

1 Introduction

Service granularity generally refers to the size of a service. The fact that services should be large-sized or coarse-grained is often postulated as a fundamental design principle of service oriented architecture (SOA). This advice is a rather obvious consequence of the quest for design artefacts that are defined at a high level of abstraction. Indeed, business people are generally not interested in fine-grained, implementation-level concepts for the construction of automated support for their work. Instead, they prefer to use and reuse automated chunks of functionality (or services) that correspond to units of work as they are used to handle them. These units are typically broader in scope than units that are processed in a software program. For example, services that provide support

Z. Bellahsène and M. Léonard (Eds.): CAiSE 2008, LNCS 5074, pp. 375–389, 2008.

for (parts of) business processes offer a high amount of functionality and are therefore labelled as coarse-grained.

It is interesting to compare services to other units of software construction that were proposed earlier, such as objects and components. The transition from objects to components and then to services is generally associated with an increase in granularity, i.e. from fine-grained objects, to coarser-grained components and even more coarse-grained services [1,2]. In what follows we briefly elaborate on these transitions.

The object oriented paradigm introduced, among others, the idea to create units of abstraction that are close to real-world concepts. However, the resulting objects turned out to be too fine-grained and biased towards implementation to be useful for the development of business applications. These issues were partly solved with the introduction of component based development (CBD), which promotes the creation of coarser-grained components. To further stress the importance of making abstractions that are recognisable for the business, the difference between generic software components and business components was made. A *business component* is generally defined as a software component that implements functionality from a particular business domain [3,4]. In general, a business component encapsulates a business-level entity or process. Therefore business components tend to be defined at higher (and hence improved) levels of abstraction.

The step towards service oriented computing (SOC) caused a further increase in granularity. While components are building blocks for applications, services are access points to an implementation that potentially covers multiple applications. As already stated, these services encapsulate business-level functionality that may even cover (parts of) enterprise-wide processes. As a consequence, their granularity is coarser than that of objects and components.

Instead of merely advocating for coarse-grained services, it is more appropriate to firstly acknowledge that *the spectrum of possible service granularity levels has become wider*. Indeed, we will show that both coarse-grained and fine-grained services can have positive impact on the architecture. As a consequence more refined judgments to control granularity are required. A few unanswered questions concerning service granularity are:

- What is the impact of service granularity on architectural qualities, such as performance, reusability and flexibility?
- How can service granularity be measured?
- Is there an upper limit for service granularity? In other words, are there any criteria that rather favour finer-grained services?

Defining granularity is quite complex since it cannot draw on theoretical groundings. Indeed, granularity can hardly be measured in terms of absolute numbers, because of the subjectivity of the related concepts that may determine the granularity in question. For example, a service may be defined in terms of an activity that is executed by that service. However, the concept 'activity' itself has a vague, hierarchical nature: it can represent a simple state change, the

work performed by one actor in one unit of time, or even a complete business process (see e.g. [5]). This makes it far from straightforward to define granularity in terms of executed activities.

In what follows we attempt to provide initial answers to the above questions about service granularity. The paper is organised as follows. Section 2 gives an overview of related work. Section 3 classifies multiple interpretations of service granularity from an interface point of view. For each granularity type, we present some small examples and we discuss the impact of granularity on architectural qualities. In Section 4, we discuss the difference between the interface and realisation viewpoint on granularity. Section 5 briefly discusses some evaluation tracks and outlines areas of future research. Finally Section 6 concludes the paper.

2 Related Work

A multitude of scientific papers, industrial papers and web entries touch upon the topic of service and component granularity. Until now, most attention was paid to measuring and assessing the impact of granularity of *components*. As argued by Herzum and Sims [4, pg. 38], component granularity is defined recursively, since a component can be defined as the composition of finer-grained components. This recursion can be discrete or continuous, respectively depending on whether the granularity levels are predefined or not. Herzum and Sims prefer the discrete form since it caters for reduced levels of design complexity. They distinguish between system-level components, business components and distributed components in descending order of granularity. System-level components are composed of business components, while a business component is composed of distributed components.

Since component based development mainly focuses on reuse, the relationship between granularity and reusability is widely discussed. Despite the general tendency towards design artefacts of increasing granularity levels, some refined observations were made [6,7]. Firstly, coarse-grained components have high reuse efficiency (because of a high contribution to the system) but low reusability (because of highly specific problem solving capabilities). Furthermore, the coarser the granularity is, the lower the composition cost is because of the fewer number of components and interactions that are required. Finally, Wang et al. [8] argue that, if a component cannot absorb requirement changes through configuration (e.g. business rules, parameterisation, etc.), then its granularity should be decreased. Besides the impact on reusability, Vitharana et al. [9] concluded a negative correlation between granularity and other managerial goals such as cost effectiveness, customization and maintainability. On the other hand, increasing levels of granularity tend to ease component assembly.

Sims [10] gives some clues of how service granularity may be measured, i.e. (1) by counting the number of components invoked through an operation on a service interface, (2) by counting the number of function points for a component, or (3) by counting the number of database tables updated. As an alternative to the

latter, the number of update operations invoked on a component can be counted or the number of types in the information model if both read and update access are relevant.

Besides these quantitative results, many authors provide an overview of general design principles to optimize service and component granularity. In what follows we give an overview of some of these principles:

- The 'right' granularity of a service or component generally varies over time [4]. A service or component that seems appropriate nowadays was maybe unsuited a few years ago because both markets and technology constantly evolve. For example, since SOA enables searching for services at runtime (e.g. facilitated by the UDDI standard), registry management and brokering are typical services that were less important before the introduction of SOA. Moreover, when particular vertical service or component standards mature, the corresponding industries can be relieved from searching for appropriate granularity levels.
- Good candidates for business components or services represent real and independent concepts to business domain people [4,11,12]. In other words, they should not be based on implementation concepts and the scope should be understandable without further context information.
- Herzum and Sims [4] give additional heuristics to identify right-sized business components: they should be easily marketable, highly usable and reusable; they should support autonomous development and should correspond to units of stability. Furthermore they should adhere to several cohesion principles, i.e. temporal (provide for development and evolution stability), functional (combine logically related functions), run-time (run e.g. computing-intensive tasks in the same address space) and actor (users of a given component should be similar) cohesion.
- If service granularity is defined in terms of the number of operations delivered [12], a service should not be too coarse as it will increase the number of consumers. Hence, a possible service change may impact many consumers. Furthermore, a huge list of operations does not provide a clear overview of which functionality is offered.
- A service should contain support for transaction integrity and compensation [13,14]. Put otherwise, all activities executed by the service should be in the scope of one transaction. If a service fails during a transaction, it should provide a compensation mechanism to undo possible changes.
- Finding the right granularity is a matter of balancing between multiple criteria [15]. For example, coarse-grained services require less network roundtrips as the execution state is contained in the message. On the other hand, small services generally require uncomplicated input data and are more easily composed.

This literature overview shows that the existing knowledge about service granularity is quite fragmented: each author takes a particular view on the subject to devise criteria for granularity optimisation, without making the considered

context explicit. In the following section we attempt to consolidate and extend the insights on service granularity.

3 Service Granularity Types

In order to structure the discussion of service granularity, we propose a classification of service granularity types that reflects three different interpretations: firstly, *functionality granularity* refers to how much functionality is offered by a service. Secondly, *data granularity* reflects the amount of data that is exchanged with a service. Finally, the *business value granularity* of a service indicates to which extent the service provides added business value. For each of these types, we describe the impact of granularity on a set of architectural concerns, such as performance, reusability and flexibility.

It should be noticed that we define different granularity types only by looking at the *interface* of the service. In other words, we describe granularity from the point of view of a consumer, although we assess the impact for both the consumer and the provider. In section 4 we briefly describe service granularity from the *realisation* viewpoint, which inspects the implementation of the service. Furthermore we indicate the differences between the interface and realisation perspectives.

The classification of service granularity is schematically represented in Figure 1. Concerning data granularity, a distinction is made between data that is sent to the service (*input data granularity*) and data that is returned by the service (*output data granularity*). For functionality granularity, we distinguish between the amount of functionality that is always offered when calling the service (*default functionality granularity*) and the functionality that can optionally be offered (*parameterised functionality granularity*).

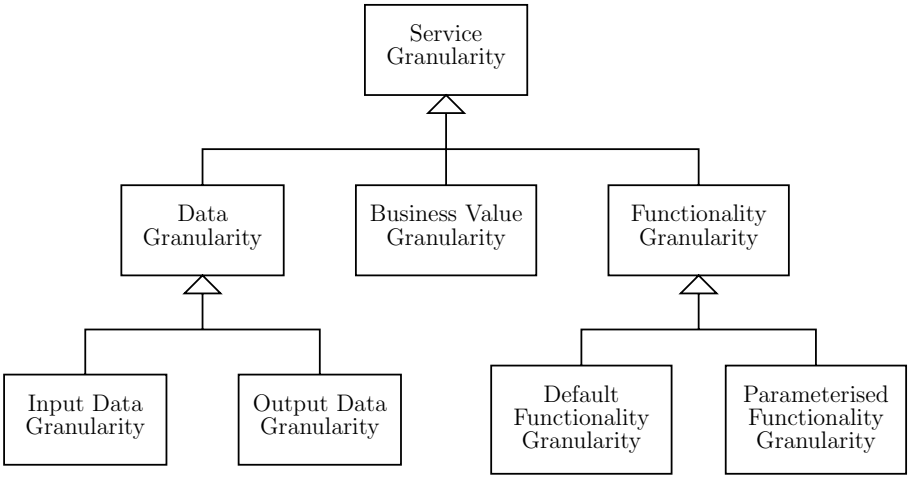

Fig. 1. Classification of service granularity types

Table 1 gives an overview of the architectural impact of coarse-grained services for each of the five granularity types. In the last column we indicate whether the impact is advantageous or disadvantageous for the consumer and the provider. In the following sections we go into more detail.

Table 1. Architectural impact of *coarse-grained* services

granularity type	**architectural impact of coarseness**	**party involved**
Input Data Granularity	less communication overhead better transactional support data possibly outdated no state lost better scalability no coordination required	p+,c+ p+ p- p+ p+ c+
Output Data Granularity	less communication overhead higher reusability	p+,c+ c+
Default Functionality Granularity	higher reuse efficiency lower reusability stability problems	c+ c- c-
Parameterised Functionality Granularity	higher reuse efficiency higher reusability no stability problems difficult implementation	c+ c+ c+ p-
Business Value Granularity	clear architecture control points consumer needs satisfied	p+ c+

Legend: p = provider, c = consumer,
+ = possitive impact, - = negative impact

3.1 Input Data Granularity

The input data granularity of a service reflects how much data is passed on to that service by a service consumer. A coarse-grained service requires one or more business objects as parameters while a fine-grained service has few or even no input parameters. Not only the number of parameters influences granularity but also their type. For example, the (data) granularity of insurance contract is bigger than that of zip code, hence when used as input parameter, they influence the input data granularity of the service accordingly. In general, a data element is coarser-grained if it is composed of other data elements and if the datatypes of its attributes are other data elements instead of primitive datatypes.

Example. With respect to input data granularity, the service ValidateContract (Contract c) is coarser-grained than the service ValidateAddress (Address a).[1]

Discussion. It is generally recommended to create coarse-grained services of this type for several reasons: Firstly, if the business objects are transferred by value, the communication overhead is reduced since the number of network transfers is decreased. Especially in the case of Web services, this overhead is high since asynchronous messaging requires multiple queuing operations and numerous XML transformations [16]. Moreover, if a service has to update multiple data elements in one transaction, it is best to pass all data at the same time, since this approach makes compensation mechanisms unnecessary. On the other hand, the input data of a coarse-grained service may be outdated if it was collected during previous service calls (i.e. not in the same transaction). Therefore the input data should be validated by the service.

It is common practice to make a service document-based, i.e. to include the entire execution context in the input message of a service, which makes the service coarse-grained. Since the provider service itself does not maintain state in this case, it is called stateless [17]. Statelessness is generally considered as a desired property for many reasons: firstly, the call of a service (operation) does not depend on previous calls, which eliminates the risk of losing state between different calls. Secondly, statelessness ensures higher scalability since more provider instances can be added if demand is high. Finally, the consumer is relieved from coordinating several fine-grained services if all data can be sent at once.

3.2 Output Data Granularity

The output data granularity of a service indicates how much data is returned to the service consumer. A coarse-grained service returns one or more (references to) business objects while a fine-grained service rather returns nothing or a few attributes. The above-mentioned remark about granularity of data elements also applies to output data granularity.

Example. With respect to output data granularity, the service Client SearchCustomer() is coarser-grained than the service Date SearchBirthDate().

Discussion. Generally it is beneficial to create services that are coarse-grained with respect to output data: similarly as for input data granularity, the number of consequent calls can be kept small if much data is returned by value. Secondly a coarse-grained service of this type doesn't hamper reuse since the superfluous part can simply be discarded by the service consumer. Although in this case

[1] All examples follow the format OutputParameterType ServiceName (InputParameterType name), whereby both the input and output parameter are only specified if they influence the corresponding level of granularity. Although all service examples are represented as a single conceptual operation, their interface might consist of multiple operations that can be invoked.

some network bandwidth might be wasted, this generally doesn't pose any severe problems, certainly not for intra-enterprise service interactions.

It is possible to make the output data granularity more dynamic by specifying a list of data elements that should be returned. However, this increases the amount of input data and may decrease the comprehensibility of the service. Alternatively it is possible to develop multiple services with different output data granularities, whereby a coarser-grained service is composed of the finer-grained services. These services are called *multi-grained* in [1, chap. 2].

3.3 Default Functionality Granularity

The default functionality granularity of a service indicates how much functionality is offered in any case, i.e. the amount of functionality that cannot be adjusted by setting some parameters. A service that performs CRUDS (create, read, update, delete, search) functionality is finer-grained than a service that also executes logic. Moreover, services that aggregate (e.g. orchestrate) other services are typically coarser-grained than their constituents. For example a service that supports a business process is coarser-grained that a service that executes a single activity of that process.

Example. With respect to default functionality granularity, the service HandleClaimProcess() is coarser-grained than the service IdentifyCustomer().

Discussion. This definition of service granularity is usually implied since it directly reflects the amount of work that is performed by the service. As we already discussed earlier, business people prefer to use and reuse services that correspond to units of work as they are used to handle them. These units of work are typically coarser-grained than the units that are processed in a software program.

The architectural consequences of coarse-grained services are similar to those of coarse-grained components, which were discussed in section 2. Firstly, the reuse efficiency is high because of the large contribution that is made by the service. Secondly, the reusability of coarse-grained services is low since the service can only be used to solve specific problems. For example the service HandleClaimProcess() will only be used in the claims domain, whereas IdentifyCustomer() may be used in multiple domains. Finally, chances are high that a change to some of the many functionalities in a coarse-grained service will cause changes to its interface. In other words, the service is unstable since it has limited capabilities to adapt to changes. The latter two arguments may be valid reasons to limit the granularity of a service.

3.4 Parameterised Functionality Granularity

The parameterised functionality granularity of a service defines the amount of functionality that optionally *can* be offered by a service. A coarse-grained (fine-grained) service offers many (a few) facilities to let the consumer configure the desired functionality, e.g. by means of input parameters. Not only the number of parameters, but also their type defines the coarseness of the service. For example

the parameter may be a boolean which represents a binary choice, or it may as well be a structured file that is being interpreted by the service. With other things being the same, the former case will yield a service with a smaller parameterised functionality granularity than the latter.

Example. With respect to parameterised functionality granularity, the service HandleProcess (Process aProcess) is coarser-grained than the service WriteCredit (boolean alsoValidate).

Discussion. Since a coarse-grained service of this type makes the service rather generic, it can easily be used in different contexts. Indeed, each different combination of input parameters yields a different behaviour of the service and therefore the service is highly reusable. Schmelzer [18] argues that, if we push this line of reasoning to the extreme, we would create a service DoSomething() that fulfils every possible need. He continues that, despite the apparent advantages of this service construction method, it has a major drawback in that it shifts the problem to the implementation of the service. Additionally, the usage tends to become more complex to the consumers as well, as they need to understand how the – often complicated – parameterisation mechanism works.

Whereas a small-grained service obviously is not reuse efficient, the consumer can control the reuse efficiency of coarse-grained services through parameter setting. For example, the service HandleProcess (Process aProcess) is reuse efficient if a complex process description is provided as input, while a straightforward process with only a few activities as input will limit the contribution of the service. Finally, a coarse-grained service is typically protective to changes (or stable) since these changes can be absorbed through configuration.

3.5 Business Value Granularity

Business value granularity measures the appropriateness of a service for the business. In other words, this type of granularity indicates the value being attached to a service. The analysis of value creation is an essential part of business modelling techniques, such as the e^3-*value* approach [19] or the i^* framework [20]. In most general terms, those approaches capture value exchanges or the extent to which the creation of value (i.e. the execution of services in our case) contributes to the goals and visions of an organisation. The extent to which a service *directly* contributes to a high-level business goal can therefore be seen as a metric for business value granularity. As an example, consider the goal-oriented derivation of services as proposed by Rolland et al. [21]. More specifically, each service realises the fulfilment of an *intention* or *goal* by following a particular *strategy*. A goal can be seen as a state to be reached while a strategy represents an approach to reach a particular state. Because the resulting services have close ties to business goals, they have high levels of business value granularity by construction.

Example. With respect to business value granularity, the service ConcludeInsuranceAgreement() is coarser-grained than the service AddClient(), which is coarser-grained than the service ValidateAccountNumber().

Discussion. The business value granularity obviously is an important indicator for business people since it gives an overview of which services should receive most attention. Dreyfus and Iyer argue that, given the complexity of architecture and limited organisational resources to implement and modify the architecture, it is indispensable to choose a subset of systems that are deemed important because of their influence on the emergence of the architecture [22]. These systems support the business goals of the enterprise and are denoted as *architecture control points (ACP)*. With respect to business value granularity, coarse-grained services and their implementing systems are the ACPs of an organisation. Services with high business value are beneficial to their consumers as well since they are more likely to satisfy the needs of those consumers. On the contrary, the composition of multiple fine-grained services with respect to business value generally causes more overhead for the consumer. Therefore companies tend to bundle multiple services into one package with increased business value granularity. We refer the reader to the work of Baida for more information about service bundling [23].

One could argue that high levels of functionality granularity automatically imply high levels of business value granularity. For example, a service that supports insurance claim handling consists of many process steps (i.e. it has high functionality granularity) and that service is highly valued in the insurance domain (i.e. it has high business value granularity). However, other examples indicate a negative relationship between these two types of granularity. Firstly, consider a service that consolidates accounting data from different information systems once a month, in batch mode. As this service executes multiple steps (data retrieval, comparison, cleansing, etc.) it has a high functionality granularity. On the other hand, the business value granularity is low since it merely corrects (or even just reports on) inconsistencies between data sources. As a second example, consider an accurate and zero-latency currency conversion service that is being used inside the company as well by external clients. Although the service has a low functionality granularity, its business value granularity is high because of its high Quality of Service (QoS) and level of reuse.

4 Interface Versus Realisation View on Granularity

In the discussions of the different service granularity types we only took the interface viewpoint into account. In other words, only the externally visible properties of a service were considered during the evaluation of the influence of service granularity on both the consumer and the provider. However, this viewpoint does not reveal all architectural consequences. Indeed, granularity can also be discussed by looking at how the service is realised in the information system(s). This viewpoint is therefore of particular interest to the service provider. In what follows, we briefly discuss the differences between the interface and realisation view on the three types of granularity. By means of a few examples, we will show that both views on granularity are not always in accordance with each other.

- **Data granularity:** Many industrial consortia have proposed sets of standardised messages that can be exchanged between different parties. For example, the ACORD (Association for Cooperative Operations Research and Development) standards define messages for the insurance and related financial services industries; likewise SWIFT (Society for Worldwide Interbank Financial Telecommunication) defines messages that are exchanged between banks and other financial institutions. These messages are typically very extended since they ought to cover all data that may be relevant during a particular transaction. Since services in these particular domains may (and should) rely on standards for their data exchange, these services are coarse-grained with respect to (input and output) data granularity. Although a lot of data is exchanged, this does not imply that all data is effectively being used during the service execution. Hence from the interface viewpoint the service is coarse-grained while from the realisation point of view it may be fine-grained.
- **Functionality granularity:** We argued that an orchestration service is coarser-grained that its constituents with respect to default functionality granularity. In fact, the granularity of the former is the sum of the granularities of the orchestrated services plus the granularity of the coordination logic. From the realisation point of view however, the orchestration service only implements the coordination logic. Therefore the service can be implemented without much effort, although it is coarse-grained from the interface viewpoint.
- **Business value granularity:** The difference between the interface and realisation viewpoint is particularly relevant to business value granularity. Suppose that a provider wants to determine how much business value is attached to the services that are delivered by ICT infrastructure components. For example, consider a database management system (DBMS) that delivers data storage, data retrieval and transaction processing services. From an interface point of view, these services are fine-grained with respect to business value granularity, since they do not directly contribute to high-level business goals. Suppose that from a realisation viewpoint, not much business value would be attached to these services either. This would imply that ICT could just as well reimplement the data services for each business case that would require these services. Obviously this inefficient approach would repeatedly generate pointless ICT costs. Therefore the business should appreciate the use of a DBMS that is proven to be reliable, reusable and high-performing. In other words, from a realisation viewpoint, the business value granularity of ICT infrastructure components is high.

  Note that the distinction between the two viewpoints on business value granularity has far-reaching consequences for the interrelation between business and ICT. From the interface viewpoint, business would only be interested in the fulfilment of their requirements towards ICT without considering the approach adopted by ICT. From the implementation viewpoint though, business would appreciate the optimisation strategies that are chosen by ICT, such as the construction of reusable and flexible infrastructures. In this

case costs should be distributed among all consumers that (will) use these infrastructures. This may not be a straightforward task if not all consumers are known in advance.

5 Evaluation and Future Work

The results of this work are currently being validated at KBC Bank & Insurance Group, one of the top three bankinsurers in Belgium with a key position in Central-Europe. To have control over granularity is one of the major concerns in their migration to SOA. The validation of this work consists of two parts: firstly, the presented classification is in general adopted by KBC. This means that the impact of each service under development is verified with respect to each type of granularity. Moreover, to the best of our knowledge, our classification covers all aspects of granularity that are discussed in the existing literature. The second part of validation considers each type of the granularity in more detail. Whereas the validation of functionality and business value granularity are left for future work, we already focused on data granularity.

In general it can be observed that current services research mainly focuses on the issue of flexibility because services generally represent "units of functionality" that need to be coordinated. Therefore, too little attention is paid to the data perspective on services. To alleviate this problem, we elaborated guidelines to optimise the input data granularity of services. This resulted in the active-passive hybrid data collection pattern [24], which distributes the responsibility of collecting data across the service consumer and provider. The decisions are mainly based on the properties of the data to be collected, such as their availability, visibility and accessibility.

As part of future research, we will propose concrete metrics for all granularity types in two different contexts. Firstly, we will define metrics in an event-driven SOA that is based on the MERODE methodology [25]. Although the architecture is object based (as a possible service implementation) and therefore of limited use on enterprise level, all models and concepts are formally defined, which allows inferring formal metrics as well. Secondly, we will extend our approach in the context of BECO [26]. BECO itself is an extension to MERODE that defines the enactment of business processes by means of an event-based coordination of components. This approach allows incorporating enterprise-class concerns, such as the integration of legacy and the treatment of business processes as first-class citizens.

Finally, we perform research on rules to determine which granularity levels are appropriate in a particular context. At the ICT department of KBC, all projects are firstly analysed in the 'work preparation' stage before they are effectively being implemented in the 'work execution' stage. It is obvious that the concerns of the people in the two stages are different, and yet, the same service concept is used by both. For example during work preparation, the problem is firstly assigned to a particular service domain, such as the claims domain. Subsequently,

the architects have to delineate the relevant services that will be implemented in the scope of the project. Now the services should be defined at such a level of granularity that changes to the existing service portfolio can be assessed. For example, the introduction of a service for claim handling will affect other domains such as accounting, payments, etc. Finally, to enable work execution, the services must be decomposed into even more fine-grained services. For example, the service for claim handling will rely on some backend services that contain business logic, some services that maintain process state, some services that generate user interfaces, etc. We will verify how the proposed granularity types can be used to derive appropriate granularity levels in a given context.

6 Conclusion

In this paper we attempted to structure the discussion of service granularity. Although the importance of coarse-grained services is often stated, we argued that enterprise architects nowadays have to deal with a broad spectrum of possible service granularity levels for different granularity types. From an interface perspective, we distinguished between data granularity, functionality granularity and business value granularity. By means of some extreme values for each granularity type we discussed the impact on architectural concerns such as reusability, reuse efficiency, stability, performance, etc. Although the interface perspective reveals several consequences of granularity for both consumer and provider, the provider will also be interested in the realisation view on granularity. By means of some examples, we showed that both views are not always in accordance with each other. Finally we presented some preliminary ideas of how granularity may assist in alleviating some architectural issues as we currently encounter them at KBC, such as the data issues around services and a granularity-driven delineation of services.

Acknowledgements

This work was funded by the KBC-Vlekho-K.U.Leuven research chair on 'Service and Component Based Development' sponsored by KBC Bank & Insurance Group.

References

1. McGovern, J., Tyagi, S., Stevens, M., Mathew, S.: Java Web Services Architecture. Morgan Kaufmann, San Diego (2003)
2. Hanson, J.: Coarse-grained interfaces enable service composition in soa (August 2003), http://articles.techrepublic.com.com/5100-22-5064520.html
3. Fellner, K.J., Turowski, K.: Classification framework for business components. In: Proceedings of the 33rd Annual Hawaii International Conference on System Sciences (HICSS-33). IEEE Computer Society, Maui (2000)
4. Herzum, P., Sims, O.: Business Components Factory: A Comprehensive Overview of Component-Based Development for the Enterprise. John Wiley & Sons, Inc., New York (2000)

5. Goedertier, S., Haesen, R., Vanthienen, J.: EM-BrA^2CE v0.1: A vocabulary and execution model for declarative business process modeling. FETEW Research Report KBI_0728, K.U.Leuven (2007)
6. Mili, H., Mili, A., Yacoub, S., Addy, E.: Reuse-Based Software Engineering: Techniques, Organizations, and Controls. John Wiley & Sons, Chichester (2002)
7. Wang, Z., Xu, X., Zhan, D.: A survey of business component identification methods and related techniques. International Journal of Information Technology 2, 229–238 (2005)
8. Wang, Z., Zhan, D.C., Xu, X.F.: STCIM: a dynamic granularity oriented and stability based component identification method. ACM SIGSOFT Software Engineering Notes 31(3), 1–14 (2006)
9. Vitharana, P., Jain, H., Zahedi, F.: Strategy-based design of reusable business components. IEEE Transactions on Systems, Man and Cybernetics, Part C: Applications and Reviews 34(4), 460–474 (2004)
10. Sims, O.: Developing the architectural framework for SOA - part 2-service granularity and dependency management. CBDI Forum Journal (June 2005)
11. Erradi, A., Anand, S., Kulkarni, N.: SOAF: An architectural framework for service definition and realization. In: Proceedings of the IEEE International Conference on Services Computing (SCC 2006), pp. 151–158. IEEE Computer Society, Washington, DC (2006)
12. Artus, D.J.: SOA realization: Service design principles. IBM Developer Works (February 2006), `http://www-128.ibm.com/developerworks/webservices/library/ws-soa-design/`
13. Wang, Z., Xu, X., Zhan, D.: Normal forms and normalized design method for business service. In: ICEBE 2005: Proceedings of the IEEE International Conference on e-Business Engineering, pp. 79–86. IEEE Computer Society, Washington, DC (2005)
14. Foody, D.: Getting web service granularity right (August 2005), `http://www.soa-zone.com/index.php?/archives/11-Getting-web-service-granularity-right.html`
15. Wilkes, L., Veryard, R.: Service-oriented architecture: Considerations for agile systems (April 2004), `http://msdn2.microsoft.com/en-us/library/aa480028.aspx`
16. Bussler, C.: The fractal nature of web services. IEEE Computer 40(3), 93–95 (2007)
17. Foster, I., Frey, J., Graham, S., Tuecke, S., Czajkowski, K., Ferguson, D., Leymann, F., Nally, M., Sedukhin, I., Snelling, D., Storey, T., Vambenepe, W., Weerawarana, S.: Modeling stateful resources with web services (March 2004)
18. Schmelzer, R.: Solving the service granularity challenge (March 2006), `http://www.zapthink.com/report.html?id=ZAPFLASH-200639`
19. Gordijn, J., Akkermans, H.: Value based requirements engineering: exploring innovative e-commerce ideas. Requirements Engineering Journal 8(2), 114–134 (2003)
20. Yu, E.S.K.: Towards modeling and reasoning support for early-phase requirements engineering. In: Proceedings of the 3rd IEEE International Symposium on Requirements Engineering (RE 1997), pp. 226–235. IEEE Computer Society, Annapolis (1997)
21. Rolland, C., Kaabi, R.S., Kraïem, N.: On ISOA: Intentional Services Oriented Architecture. In: Krogstie, J., Opdahl, A., Sindre, G. (eds.) CAiSE 2007 and WES 2007. LNCS, vol. 4495, pp. 158–172. Springer, Heidelberg (2007)
22. Dreyfus, D., Iyer, B.: Enterprise architecture: A social network perspective. In: Proceedings of the 39th Hawaii International International Conference on Systems Science (HICSS-39), January 2006. IEEE Computer Society Press, Kauai (2006)

23. Baida, Z.: Software-aided Service Bundling - Intelligent Methods & Tools for Graphical Service Modeling. PhD thesis, Vrije Universiteit, Amsterdam, The Netherlands (2006)
24. Haesen, R., De Rore, L., Snoeck, M., Lemahieu, W., Poelmans, S.: Active-passive hybrid data collection. In: Proceedings of the 11th European Conference on Pattern Languages of Programs (EuroPLoP 2006), Irsee, Germany, Universitaetsverlag Konstanz, pp. 565–577 (2006)
25. Snoeck, M.: Object-Oriented Enterprise Modelling with Merode. Leuven University Press (1999)
26. Lemahieu, W., Snoeck, M., Goethals, F., De Backer, M., Haesen, R., Vandenbulcke, J., Dedene, G.: Coordinating cots applications via a business event layer. IEEE Software 22(4), 28–35 (2005)