# Seizing and sizing SOA applications with COSMIC Function Points

Luca Santillo (luca.santillo@gmail.com)

## Abstract

*Service-oriented architecture (SOA) expresses a perspective of software architecture that defines the use of loosely coupled software services to support the requirements of business processes and software users. In a SOA environment, resources on a network (e.g. web services) are made available as independent services that can be accessed without knowledge of their underlying platform or programming language. When measuring the functional size of a distributed application in a SOA environment, we must face the "boundary positioning" issue. 1st generation FSM methods easily fail in providing a good answer to this question, since they lack a conceptual framework provided by new generation methods, as COSMIC Full Function Point 2.x and higher. Software applications are not monolithic (anymore). Layers, peer items, and functions between them, should be considered, to provide a more accurate answer to the sizing question, and hence to the software estimation problem.*

*This work aims to describe the boundary problem from the software measurement perspective, and to provide basic guidelines for the application of the COSMIC Function Point sizing method in a SOA environment.*

## 1. Introduction

Functional Size Measurement (FSM) methods aim to provide a technology-independent measure of the size of software systems. Despite (or, because of) such generality, some methods might lack relevant concepts or practices suitable for application in innovative approaches to software development. Service-oriented architecture (SOA) expresses a perspective of software architecture that defines the use of loosely coupled software services to support the requirements of business processes and software users. In a SOA environment, resources on a network are made available as independent services that can be accessed without knowledge of their underlying platform implementation or programming language.

When addressing the measurement of functional size of a distributed business application in a SOA environment, we face the issue that so many practitioners in function point analysis have to face in everyday counting activities – where is the boundary of "my" application to be positioned? In other words: what portions of the overall software environment being considered is inherently part of "the" software to be measured, and what portions are outside and/or independent of that? First generation FSM methods like IFPUG, NESMA, and MkII Function Point, would likely fail in providing a good answer to this question, since they all historically lack a conceptual framework that new generation methods, as COSMIC Full Function Point (2.x or higher versions), do provide. Software applications are not "monolithic", since software services can be separated and offered to more than one application, encouraging decoupling and reuse. Thus, concepts as "layers" and "peer items" within layers, and furthermore data and transactional functions between those, should be considered, depending on the measurement viewpoint and the purpose of the measurement exercise, to provide a more accurate answer to the starting sizing question, and hence to any estimation problem for cost, time or quality per software. The basic principle underlying such approach is that different software portions (services) are better estimated on their own, by separation, since they often show different productivity aspects to be included or considered. Aggregating the estimation results could and should of course be performed as a final step for

the overall estimation problem. This work aims to discuss the "boundary problem" from the software measurement and estimation perspective, and to provide some basic guidelines for the application of the COSMIC functional size measurement method in a SOA environment to address such issue.

## 2. SOA basic concepts and definitions

### 2.1. Software Architecture

Software architecture has emerged as a critical technology for successful system-building, since it structures the development effort, it allows nearly all of the required quality attributes as performance, security, etc., and it forms basis for whole families of systems. The software architecture of a program or computing system is the structure or structures of the system, which comprise software components, the "externally visible" properties of those components, and the relationships among them [1].

### 2.2. Service Oriented Architecture (SOA)

There is no widely-agreed upon definition of service-oriented architecture other than its literal translation that it is an architecture that relies on service-orientation as its fundamental design principle. Service-orientation describes an architecture that uses loosely coupled services to support the requirements of business processes and users. Resources on a (web) network in an SOA environment are made available as independent (web) services that can be accessed without knowledge of their underlying platform implementation [2]. The Organization for the Advancement of Structured Information Standards (OASIS) defines SOA as "a paradigm for organizing and utilizing distributed capabilities that may be under the control of different ownership domains. It provides a uniform means to offer, discover, interact with and use capabilities to produce desired effects consistent with measurable preconditions and expectations" [3].

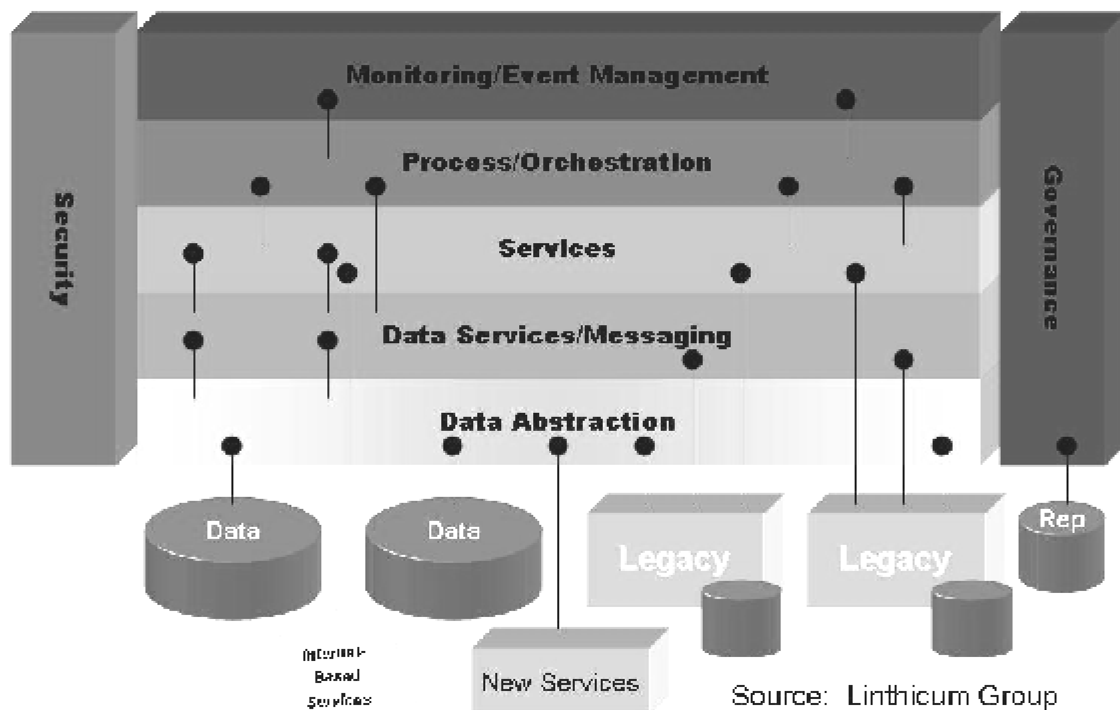

*Figure 1: SOA Meta-Model (source: www.linthicumgroup.com, license: Creative Commons 2.5, author: Dave Linthicum, 2006-04-19).*

Since a software architecture is not tied to a specific technology (note: neither do FSM methods, for our purposes), SOA may be implemented in practice using a wide range of technologies and protocols (e.g. REST, RPC, DCOM, CORBA, Web Services, etc., and any combination of those). The key is independent services with defined interfaces (described by each so-called "service contract") that can be called to perform their tasks in a standard way, without the service having foreknowledge of the calling application, and without the application having or needing knowledge of how the service actually performs its tasks. Such architecture promotes software reuse, at the macro (service) level rather than micro (object) level. It can also simplify interconnection to – and usage of – existing IT legacy applications, that is, software integration (Fig. 1). Thus, it is a common expectation that SOA can help businesses respond more quickly and cost-effectively to changing market conditions [4]. SOA also promotes the goal of separating users from the service implementations. Services can therefore be run on various distributed platforms and be accessed across networks.

## 2.3. SOA Principles

The following architectural principles for design and service definition focus on specific themes that influence the intrinsic behaviour of a system based on SOA [2, 5, 6, 7]:

- reusability: logic is divided into services with the intention of promoting reuse;
- service contract: services adhere to a communications agreement, as defined collectively by one or more service description documents – a service contract carries predefined information for identification, ownership, and description of the service and of its functional and non-functional requirements, operations and invocation methods;
- loose coupling: services maintain a relationship that minimizes dependencies and only requires that they maintain an awareness of each other;
- abstraction (i.e. separation of concerns, or information hiding): beyond what is described in the service contract, each service hides logic from the "outside world";
- composability: collections of services can be coordinated and assembled to form composite services;
- autonomy (or encapsulation): services have control over the logic they encapsulate;
- statelessness: services minimize retaining information specific to an activity (plus, service requests don't depend on each other);
- discoverability: services are designed to be outwardly descriptive so that they can be found and assessed via a service registry or similar discovery mechanism, for binding between the service "participants": the consumer and the producer or provider (Fig. 2).

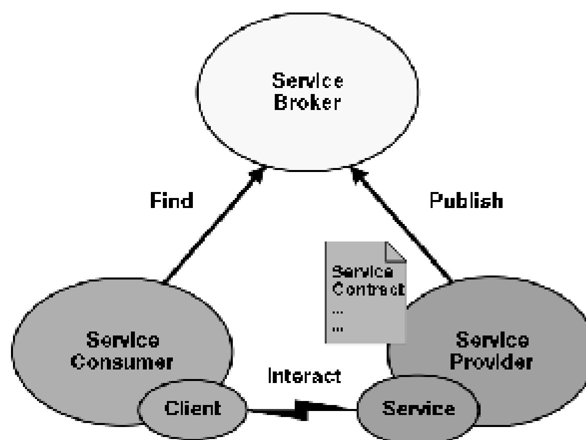


*Figure 2: Services participants and binding by a service registry ("broker") (source: [8]).*

## 2.4. Service types and levels

Two main types are usually identified for services:

- business services – designed to support well-defined business functionality based on the business model of the enterprise, and
- integration services – designed to expose functionality and data of existing applications as services.

Fig. 5 shows an example of service definition hierarchy, where business services on one layer might be composed of a number of collaborating or orchestrated services on a lower level [9].

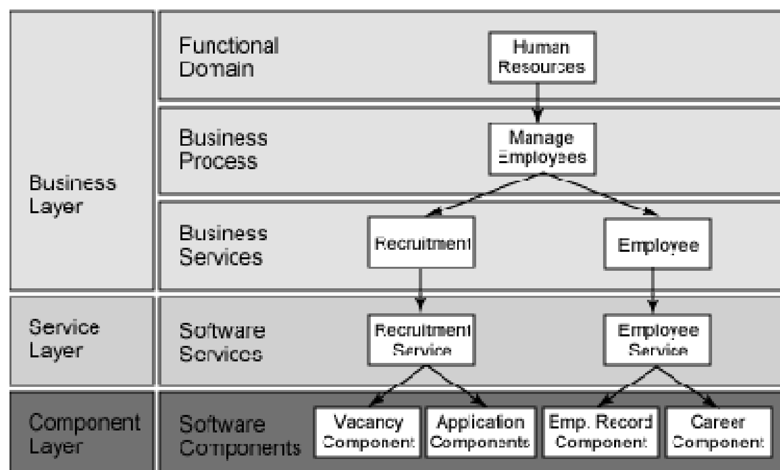


*Figure 5: service definition hierarchy (example) (source: [9]).*

Fig. 6 shows another example of architectural layering diagram, besides a layered architecture reference model, where services could be designed at the level of common business objects and/or foundation (domain specific layer) to serve for multiple business purposes at higher levels (application specific layer) [10]. It's worth noting that the layering concept is significant for service identification, as well as it is recognized in modern functional size measurement methods.

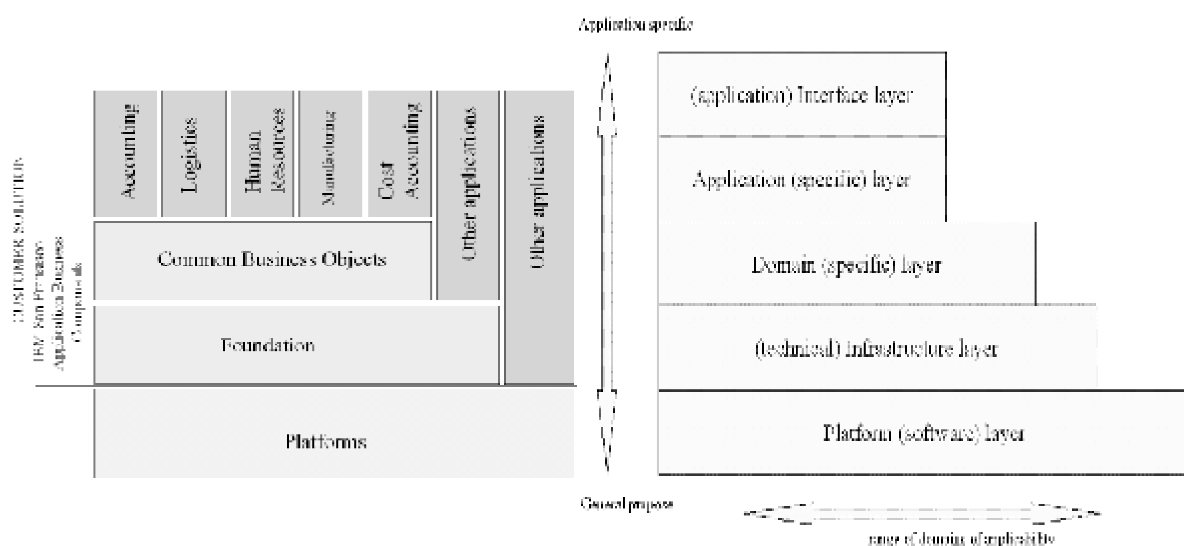


*Figure 6: layered architecture example and related reference model (source: [10]).*

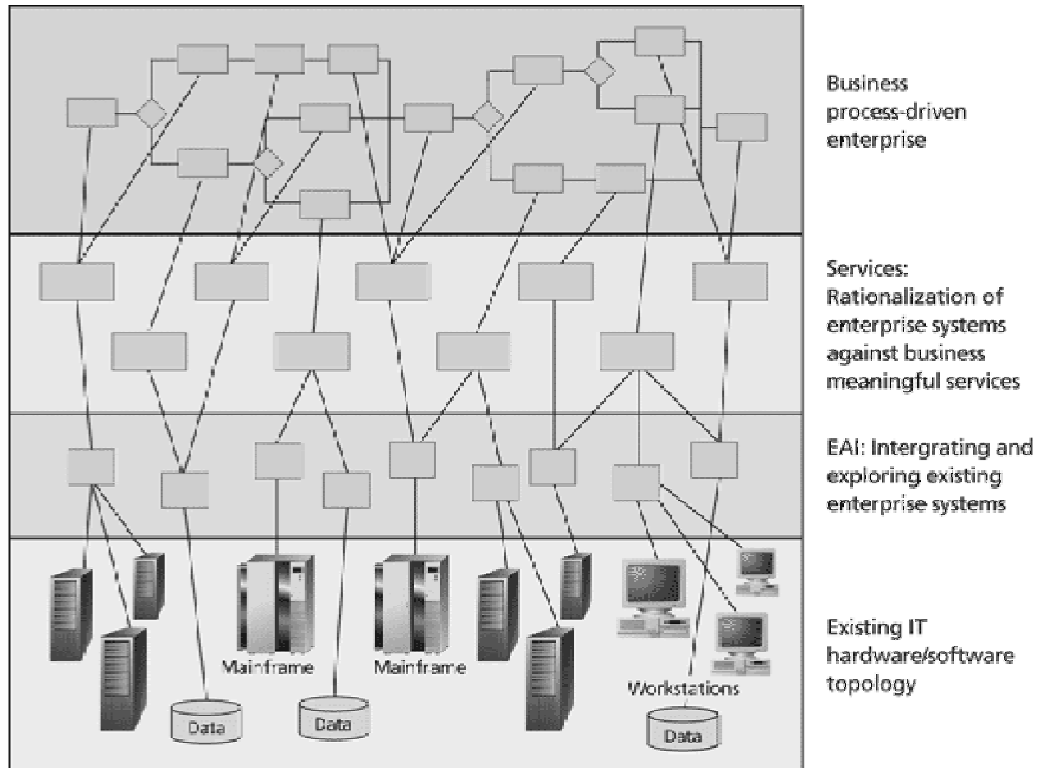


*Figure 3: SOA Top-Down Design's Architecture (source: [7]).*

## 2.5. SOA enterprise reference model and software integration

Top-down SOA design usually leads to the overall enterprise architecture (Fig. 3). At the top of this stack are business processes, defined by the business model, which use services for their execution. These services represent a rationalization layer for the application portfolio to the business model. In reality, at least initially, the bulk of these services will be implemented as wrappers for the existing applications. A typical service implementation exposes existing legacy system functionality by encapsulating it in its own implementation, which allows for extension of the legacy system functionality through the additional components without touching the existing legacy systems. A service implementation also makes it possible to increase the granularity of the service by combining the functionality of multiple legacy systems (or multiple interfaces of the same legacy system), implementing additional functionality, or aligning the legacy data with the enterprise data model (Fig. 4).

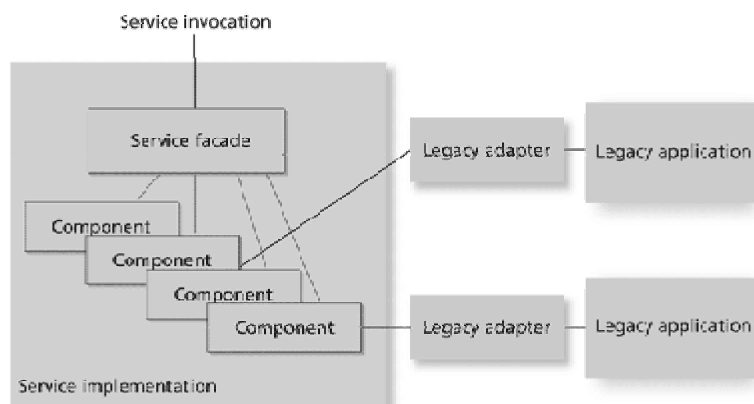


*Figure 4: legacy applications integration (source: [7]).*

### 2.6. Web Services

A particular case of service is represented by web services. A web service is a software component identified by a URI (Uniform Resource Identifier), whose public interfaces and bindings are defined and described using XML. Its definition can be discovered by other software systems. These systems may then interact with the Web service in a manner prescribed by its definition, using XML based messages conveyed by Internet protocols [11]. Figure 7 shows an example of several web services possibly involved by a single application from the user's perspective.

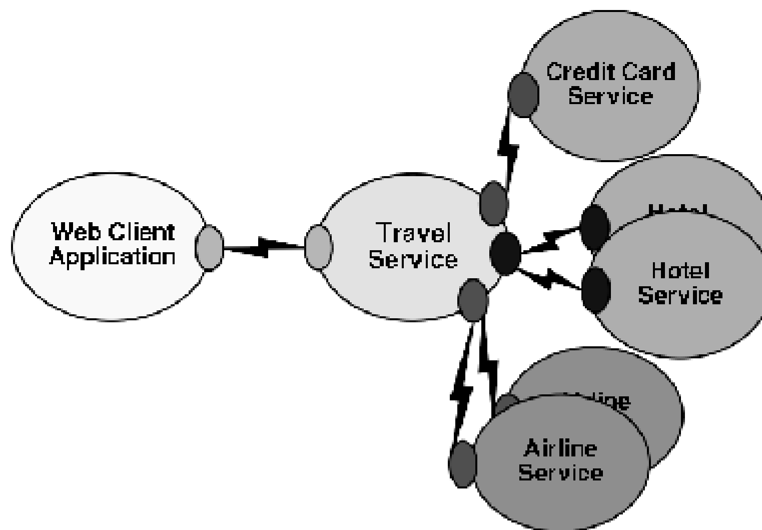


*Figure 7: An example of web services usage and combination (source: [11]).*

It's worth noting that although SOA is commonly implemented using web services, services can be made visible, support interaction, and generate effects through other implementation strategies. Web service-based technologies are concrete, based on specific protocols (as SOAP – originally "Simple Object Access Protocol", lately also "Service Oriented Architecture Protocol", WSDL – Web Services Description Language, and UDDI – Universal Description, Discovery, and Integration, XML, HTTP, etc. for instance), however the concepts of the underlying SOA reference model are valid in general. Thus, other examples, other than that shown in Fig. 7 can be found within an enterprise scope, for instance linking services owned (provided) by different functional areas, departments, or agencies (or physical/logical/infrastructural sites, nodes, or platforms).

## 3. Functional Size Measurement methods' features (short excerpt)

Since it's beyond our scope to describe the overall model and process of FSM methods, this section only recalls some features of 1st and 2nd generation methods, for the purposes of this work. IFPUG Function Point is taken as the main example of 1st generation method, due to its historical background and diffusion, while COSMIC Function Point (originally COSMIC Full Function Point) is the current new generation method in the practitioners' community. The reader can find exhaustive information about the cited methods respectively in [12] and [13] in the References.

### 3.1. IFPUG Function Point

An IFPUG Function Point measure is obtained by summing up logical data groups and elementary processes, classified respectively as Internal Logical Files, External Interface Files, and External Inputs, Outputs, or Inquiries, with respect to the "application boundary", which

separates the “system” being measured from the user domain (end user and/or other interfaced systems). It’s worth noting that the value scales provided by the IFPUG method to assign FP quantities to the identified functions are upper-limited, based on limited ranges of processed data elements and files. Thus, different interpretations regarding the aggregation opposite to the autonomy of elementary processes, for instance, easily lead to different numerical assignments for the same set of requirements. In other words, the IFPUG measure is not “associative” in a mathematical sense, or “the total is not equal to the sum of the parts”.

Although the latest release of the IFPUG method defines a counting purpose and a counting scope to be documented, no relevant example or case study about these concepts are provided in detail to date, such as they could be relevant for the purpose of SOA-like systems.

Also, the original IFPUG method provides a value adjustment factor (VAF) for taking into account several non-functional requirements for the final numerical assignment for the size of the system being measured. Nonetheless, such factor does not include any specific consideration for software reuse – literally, a function (service or service operation) provided several times to different systems is counted as many times, regardless of being designed and implemented only once or many times as well.

Finally, the IFPUG method could be denoted as a “near black box” approach to software measurement, since no internal process can be identified and measured according to the IFPUG counting practices, unless it is merged with another process that receive or send data from or to the end user or another system outside the overall system boundary.

### 3.2. COSMIC (Full) Function Point

COSMIC (Full) Function Point has been proposed as a modern FSM method to provide wider applicability and higher accuracy with respect to the functional features being measured than the IFPUG method. For our purposes, COSMIC method’s key concepts are the possibility of viewing the system being measured as composed by different linked layers (different levels of conceptual abstraction of the system functions), by possibly separated software peer items within each layers, and the capability to specify different measurement viewpoints (that is, user types other than the “end user”, or external system), based on different measurement purposes. Thus, the measurement scope is quite relevant to any COSMIC measurement exercise, and must be defined accurately.

It’s worth noting that, while logical data groups are not assigned any numerical value, the value scales provided by the COSMIC method to assign FP quantities to the identified functional processes (corresponding to IFPUG elementary processes) are NOT upper-limited. Thus, the COSMIC measure is more “associative” in a mathematical sense than the IFPUG one, that is, very complex processes (services or services operations) are assigned larger numbers than simple processes, in proportion to the amount of different Entry, Exit, Read and Write actions performed by each process.

The COSMIC methods does NOT provide any value adjustment factor, which would be considered a different measure than functional size. Nonetheless, the capability of decomposing the system being analyzed into layers and peer items within layers allows to consider some processes (services, or service operations) only once vs. several times, depending on the underlying software architecture and related choices of layers and peer items. Such capability renders “internal” processes, otherwise invisible to the end user, “visible” for specific measurement purposes.

Thus, finally, the COSMIC FSM method could be denoted as a “near white box” approach to software measurement. Each layer and peer item, when identified, has its own boundary, revealing otherwise internal processes to its own users (other layers or peer items).

## 4. Software partitioning and relevant measurement in a SOA environment

Several issues can be identified when measuring the functional size of a software system in a SOA environment. While the current IFPUG counting practices do not provide specific solutions to such issues, and could arguably lead to misleading results, some rational hints can be provided when measuring with the COSMIC approach, still being compliant with its standard concepts. (Such issues are mainly related to the further usage of the obtained size in an overall generic estimation model having the functional size as its main driver, but they can be expressed independently of any concrete estimation model).

Most of the proposed guidelines can be referred as an example to Fig. 8.

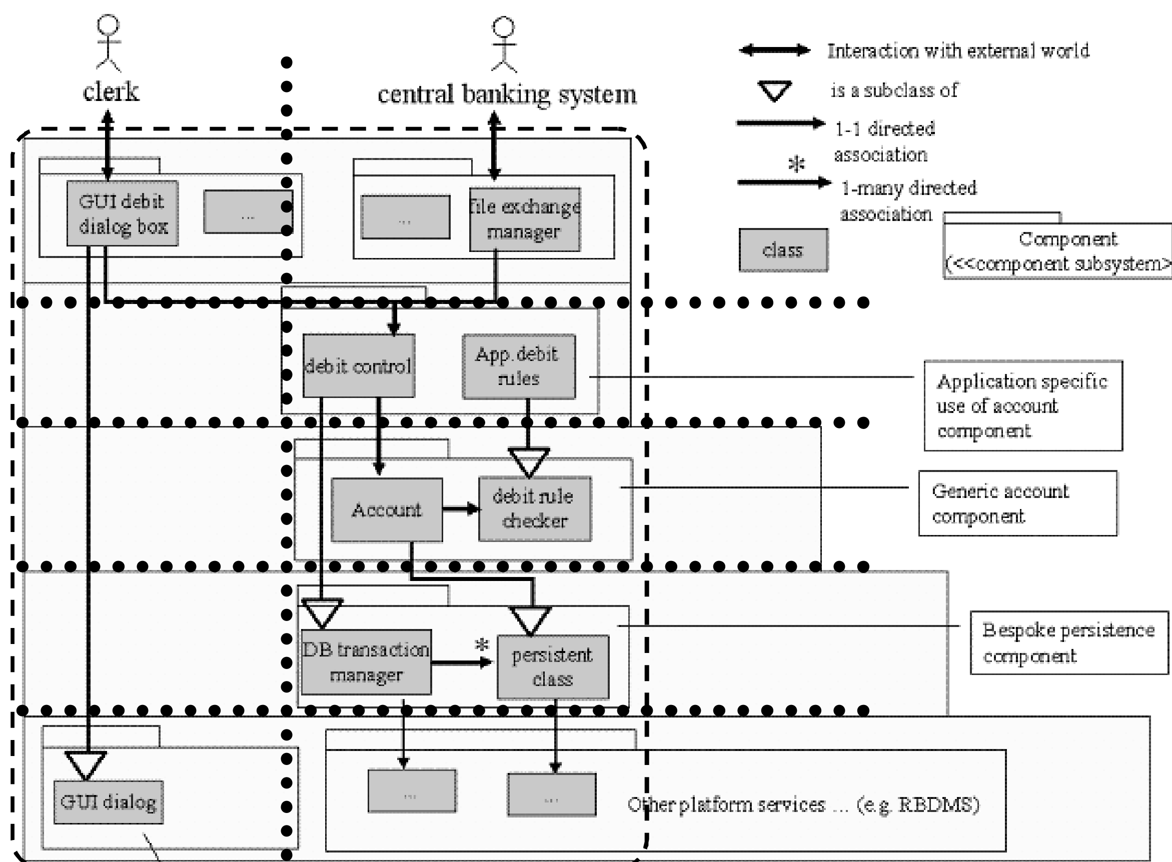


*Figure 8: layered architecture populated with an example (source: [10]). Thin dash line: IFPUG boundary; bold dotted lines: possible COSMIC partitioning.*

### 4.1. Monolithic applications/boundaries

The COSMIC method allows for an architecture-based partitioning of the overall software system being measured. In such a partition (several possibilities based on the specific measurement purposes and architecture being analyzed), any layer (and equivalently, any peer item, at the same abstraction level) is a collection of software "units" (that is, services, in a SOA environment) that may be invoked or accessed. A requirement is that each unit (service) have an interface by which its operations can be triggered or initiated or accessed.

The relation among layers or peer items is "allowed to use". If two software portions are related by this relation, then any unit of software (service client) in the first is allowed to use any unit of software (service provider) in the second. (Note: the "allowed to use" relation is usually anti-symmetric. That is, if (A,B) is in it then (B,A) probably should not be.)

A significant property of such layers view is cohesion: any layer (or peer item) provides a cohesive set of services, meaning that the services as a group would likely be useful (as a group) in some other context (and usually, such “other context” is identified by design).

Together, the layers and peer items within layers provide a complete partitioning of the software: every piece of software is allocated to exactly one layer and peer item within a layer (this ensures that no software service is counted twice).

### 4.2. Black vs. white box visibility, aka architectural structure recognizability

Several authors in the last decade recall “the well known failures of black box abstraction in component and distributed systems. These include the loss of rationale, invisibility to the client of critical implementation efficiency issues, and the inability of the client to specialize any behaviours that cross-cut the natural component breakdown of the system.” [14]. When the declared purpose of the measurement is to support design and implementation effort estimation for the overall system being analyzed, a white box approach provides fundamental details in order to improve and differentiate the estimation for different architectural choices.

It might be objected that architecture choices are more of a technical or quality feature, rather than functional aspects. While this might be true for the concrete implementation details of the architecture (standards, protocols, physical interface), it is still arguable that the reference model of a SOA approach is functionally different from other possible software architectures – at a first level of analysis, architecture is a matter of identifying and positioning separated functions in a consistent (and reusable) structure. Thus, a good FSM method should be able to take it into account. Deriving from the previous boundary topic, the partitioning capability of the COSMIC model implicitly provides such feature.

### 4.3. Shared data counting

The IFPUG practices [12] includes a section addressing several scenarios of “data shared between 2 or more applications”. In most of the scenarios, the IFPUG answer to these cases – based on the highest level “primary intent” of each case – is to identify an external logical data group, regardless of any possible way to accomplish the access or the feeding or the exchange of the data from one system to another. As an example, consider Fig. 9. In the IFPUG approach, the screen scraping tool (that might represent a specifically designed service) is considered as a matter of technical implementation of the higher level functional requirements of “accessing the legacy data”. Thus, in the IFPUG approach, such a service as the screen scraping tool, or equivalently many sorts of middleware goal-oriented service between any other two systems is simply not assigned any functional size.

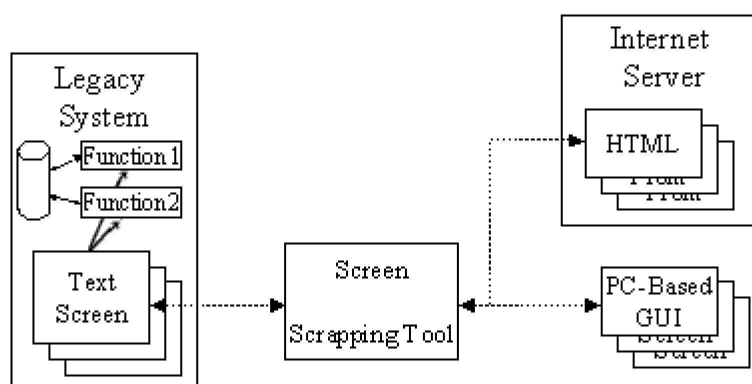


*Figure 9: legacy system wrapping using screen scraping (source: [15]).*

If the measurement purpose is to estimate any attribute related to the design and implementation of only such services, the measure would not simply exist for such a purpose. This situation is simply not acceptable from a practical perspective. The previously discussed topics, namely on partitioning boundaries in the COSMIC approach, would instead allow to identify the middleware services as autonomously and independently required and provided, thus they would be accountable.

## 4.4. Autonomous services (processes) AND Different software "channels"

The same partitioning capability of the COSMIC approach allows to solve another issue of the 1st generation FSM methods, (internal) autonomous processes. Referring to Fig. 10 as an example, what might be perceived simply as "one" set of functionality by the highest level perspective, is in fact provided by different "channels", all of which using the same underlying services, but possibly with differences due to the every channel.

If the measurement purpose is to estimate any attribute related to the design and implementation of the different channels features, and separately of such (internal) autonomous services, separate measures would not simply exist for such a purpose in the IFPUG common approach. Once again, the partitioning boundaries in the COSMIC approach, and its "near white box" feature, would instead allow to identify the inner services as autonomously and independently required and provided, thus they would be accountable.

One hint to acknowledge differentiation between software portions (being them channels, peer items, inner services, and the like), would likely be found in the (old) synchronous relation vs. (new) call-back/polling mechanism in a real world SOA environment. In such scenario, different channels do simply provide different functions, even when the starting situation and the final outcomes of the functions being compared might be the same.

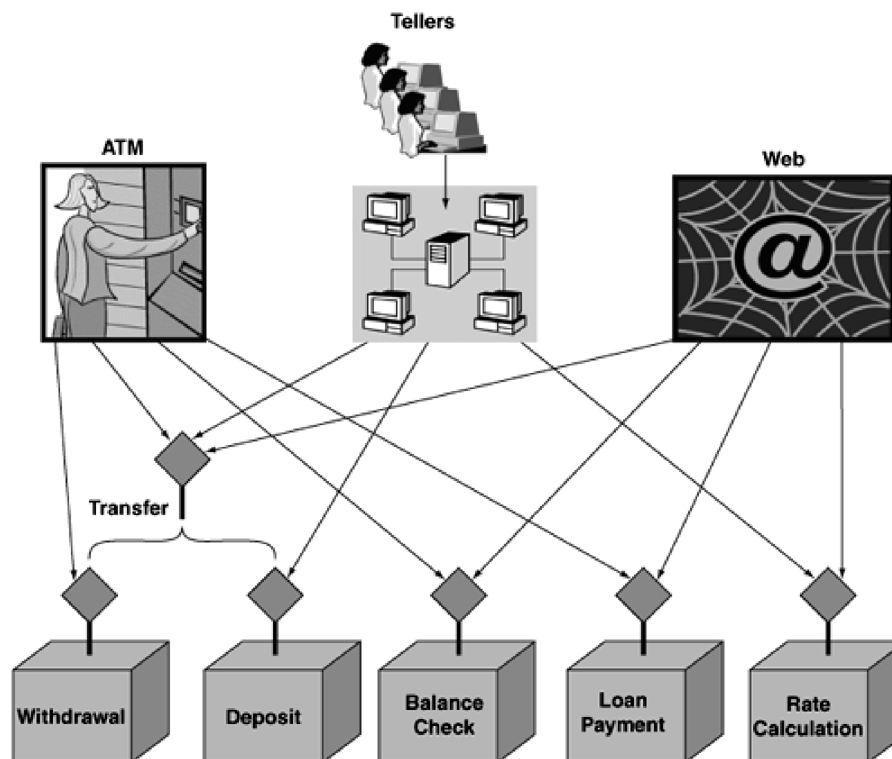


*Figure 10: accessing and composing services (source: [16]).*
*Several boundaries are possible (not shown).*

### 4.5. Services (processes) complexity

At a lower level of accuracy, still relevant for significant effort estimation purposes, lies the issue of having bounded vs. unbounded value scales for the measurement objects (be they data groups, functional processes, and the like). While simple range matrices in the IFPUG approach contribute to keep the method “simple and fast”, they are debated over the years since peculiar scenarios (simplest cases as well as particularly complex cases) are “stretched to fit into a 3-level complexity scale [17].

Even whenever any possible interpretation ambiguity has been solved about the exact amount and aggregation vs. separation of functional processes within a given software portion being measured, such simplification would lead to “same quantities” for “different (complex) software units”, in the IFPUG model.

Implicitly, the COSMIC model does provide open-range scales to take into account possibly high-complexity functions, such as a single service providing higher data exchanges or accesses (Entry, Exit, Read, or Write actions). This also would contribute to a more accurate numerical assignment to differentiate services in a SOA environment, as well as in any other software architecture type (and it would help mitigate possible misunderstanding in the starting aggregation or separation of functional processes, too).

## 5. Conclusions

SOA is a means of organizing solutions that promotes reuse, growth and interoperability. It is not itself a solution to domain problems but rather an organizing and delivery paradigm that enables one to get more value from use both of capabilities which are locally “owned” and those under the control of others. It also enables one to express solutions in a way that makes it easier to modify or evolve the identified solution or to try alternate solutions. When facing a functional size measurement exercise for a SOA-like software system – typically for estimation purposes –, the chosen measurement method should be able to differentiate functionality based on the architectural choices and aspects (other non-functional aspects should obviously be taken into account for the overall estimation problem, but those are beyond the scope of this work).

While first generation FSM methods like the IFPUG one might encounter severe application issues in such a task for SOA based software, the COSMIC (Full) Function Point modern framework is promising in doing that. In the author’s opinion, different answers to this sizing problem from the different generation methods are not simply a matter of degree, but lie in the intrinsic different framework of their models (the last discussed topics on complexity limited ranges could be easily solved, though). Field trials and practical experiences would be welcome to test such an approach, and would lead of course to a wider set of guidelines for practical application of COSMIC measurement for SOA-like systems.

# 6. References


[1] Clemens, P.C., “Software Architecture Documentation in Practice”, SEI Symposium, Pittsburgh, 2000.

[2] Wikipedia contributors, “Service-oriented architecture”, Wikipedia, The Free Encyclopedia, http://en.wikipedia.org/wiki/Service-oriented_architecture, accessed 28-February-2007.

[3] OASIS SOA Reference Model Technical Committee, “Reference Model for Service Oriented Architecture 1.0, Committee Specification 1”, 2 August 2006, Organization for the Advancement of Structured Information Standards, http://www.oasis-open.org/.

[4] Koch, C., “A New Blueprint For The Enterprise”, CIO Magazine, 01 March 2005.

[5] Erl, T., “Service-Oriented Architecture – Concepts, Technology, and Design”, Prentice Hall/PearsonPTR, 2006.

[6] SOA Practitioners, “SOA Practitioner's Guide – Part 2 – SOA Reference Architecture”, http://dev2dev.bea.com/soa/.

[7] Lublinsky, B, “SOA Design: Meeting in the Middle”, FTPOnline, August 20, 2004.

[8] Le Hégaret, L., “Introduction to Web Services”, W3C Talks, 2003, http://www.w3.org.

[9] Zimmermann, O., Krogdahl, P., Gee, C., “Elements of Service-Oriented Analysis and Design”, IBM developerWorks Library, 02 June 2004.

[10] Collins-Cope, M., Matthews, H, “A Reference Architecture for Component Based Development”, RATIO, http://www.ratio.co.uk/techlibrary.html.

[11] Haas, H., “Designing the architecture for Web services”, W3C Talks, 2003, http://www.w3.org.

[12] IFPUG, “Function Point Counting Practices Manual, Release 4.2”, International Function Point Users Group, 2000, http://www.ifpug.org/.

[13] Abran, A. e.a., “COSMIC-FFP Measurement Manual 2.2”, The Common Software Measurement International Consortium, Jan. 2003, http://www.lrgl.uqam.ca/cosmic-fpp/.

[14] Kiczales, G., “Beyond the Black Box: Open Implementation”, IEEE Software, January 1996.

[15] SEI, “Black-box Modernization of Information Systems”, Carnegie Mellon Software Engineering Institute, http://www.sei.cmu.edu/str/descriptions/blackbox_body.html.

[16] Newcomer, E., Lomow, G, “Understanding SOA with Web Services”, Addison Wesley Professional, 2004.

[17] Santillo, L., “Software complexity evaluation based on functional size components”, in: IWSM Procs, International Workshop on Software Measurement, Berlin, 2004.