

[▶ subscribe](#)[▶ contact us](#)[▶ submit an article](#)[▶ rational.com](#)[▶ issue contents](#)[▶ archives](#)[▶ mission statement](#)[▶ editorial staff](#)

Thoughts on Functional Decomposition

by [Murray Cantor](#)

Principal Consultant
Rational Strategic Services
Organization
IBM Software Group

It is common knowledge in the Rational field organization that functional decomposition is something to avoid, and a great deal of practical experience reinforces our belief that it leads to poorly designed, low-quality systems. However, some organizations find this practice useful and seem unwilling to part from it. We have worked to understand exactly what the term functional decomposition means to different organizations and how they translate it into practice. In this article I will share what we have learned and what Rational recommends.



What Is Functional Decomposition?

I am not aware of any official definition of *functional decomposition*. Like many system engineering terms, it is used to describe several activities, including the following three, which are not entirely disjointed. Depending on the context, functional decomposition can refer to:

1. **Adding more detail to a general requirement.** "In order to do A, the system must do X, Y, and Z." For example, if your requirement is "A simple application that prints out student report cards," you might add, "In order to print out student report cards (A), the system must be able to access student records (Z)." This explains the requirement more fully.
2. **Organizing requirements, particularly use cases, into packages.** This is often done on larger systems. For example, a project to build a comprehensive system for a ship may have so many requirements that it makes sense to sort them into packages: navigation capabilities, performance requirements, endurance under extreme conditions, and so on. This type of organization helps the

team deal with the multitude of requirements.

- 3. Determining subsystem requirements.** In some instances the team defines subsystems by the system requirements they enable. For example, a bank account management system may have one subsystem dedicated to online customer transactions and another subsystem for in-branch transactions. These subsystems maintain their separate databases.

There is little harm in the first two uses. They are simply ways to better understand and manage requirements. But the third use is a problem. *Using functional decomposition methods to derive the system architecture and set subsystem requirements puts the system at risk.*

Architecture Should Come First

Teams use functional decomposition to define architecture in two different ways:

- **Requirement allocation.** This involves adding details to requirements (as noted in item 1 in the above list), and then assigning requirements to subsystems (as noted in item 3 in the above list).
- **Requirement derivation.** This involves determining subsystems first, and then deriving their requirements.

Let's explore how organizations pursue each of these approaches.

Requirement Allocation

Often organizations perform activities 1 and then 3 from our list above: *They define subsystems by grouping decomposed requirements.* For example, if subsystem 1 may be specified by meeting requirements X, Y, and Z, the customer considers requirements X, Y, and Z as "allocated" to subsystem 1.¹ This approach to finding subsystems typically leads to poor architectures. *It provides no mechanism for determining whether the subsystems can provide common, underlying services that could be reused across a range of requirements.* Typically, such common services (e.g., business-rule parsers, event handlers, etc.) do not emerge when you add detail to system requirements, but rather when you perform some flavor of object analysis. *The outcome of using functional requirement allocation to create an architecture -- rather than starting with a defined architecture -- is subsystems with duplicate code. This leads to unnecessarily large, complex systems that are costly to develop and operate, and difficult to maintain.*

Here is an actual example: One image satellite ground support system that is currently being fielded was built with a functional decomposition architecture. The system requirements included the ability to plan missions, control the satellites, and process the collected data for analysis. Accordingly, the developer built three subsystems: mission planning, command and control, and data processing. Each of these subsystems was

given to an independent team for development. During the project, each team independently discovered the need for a database with the satellite's orbital history (the satellites can, to some extent, be steered to different orbits as needed). So each team built its own separate database, using separate formats. But the information needs to be consistent for the overall system to operate correctly, and now, the effort required to maintain these three databases is excessive and could easily have been avoided had the team done some kind of object analysis, including a study of the enterprise data architecture.

Requirement Derivation

In my work, I have seen organizations perform what they refer to as "functional decomposition" by determining their subsystems first and then deriving requirements. Usually, the team decides up front that they will use a certain type of architecture (e.g., standard three-tier database application). Sometimes they are also carrying a logical decomposition around in their heads but not recording it in a shared document. Quite literally, they "have a design in mind." In any case, these teams decompose the requirements to align with their instincts about what the system architecture should be; then, they allocate those *decomposed requirements to the intended architectural elements*. Sometimes they follow this with a synthesis step to combine similar requirements.

Although teams often refer to this practice as "functional decomposition," it is really a form of *requirement derivation*, and may not result in the same problems as functional allocation. The main difficulty I have seen with this approach is that the team does not document the architecture explicitly. Instead, they proceed with their work based on the architecture that is implied by the way requirements are allocated. But *unless the architecture is very simple and the team very small, this lack of documentation for the architecture leads to poor understanding of the system* and therefore to many missteps along the development path. There is great value in explicitly documenting the logical architecture in UML.

An explicit, well-documented architecture also makes requirement derivation a repeatable practice by introducing limits and discipline to the process. For example, consider an automobile differential, which allows the driving wheels to run at different speeds when turning, so the automobile can maintain traction while in a curve and thereby hold the road. The differential's requirements are based on (a) supporting the basic requirements for an automobile, such as the need to maintain traction on curves, to be of a certain overall weight and volume, to be easy to maintain, and so on; (b) an architectural decision to design the automobile with a drive train consisting of a single engine connected to a drive shaft, which is connected to a differential that is connected to two rear axles, which in turn are connected to the rear wheel hubs. Since there are other possible architectural solutions to the basic set of requirements for an automobile -- solutions that do not require a differential -- the differential's requirements must be derived from the chosen architecture and cannot be discovered merely by doing a functional allocation.

Some Final Thoughts

Decades of experience have shown that good system architectures -- those that are maintainable, extendable, cost effective, and so on -- *result from finding subsystems with internal services that:*

- *Are reusable.*
- *Enable those subsystems to collaborate in order to meet system requirements.*

Generally, each subsystem can play a role in meeting more than one system requirement. This is the essence of logical decomposition. You can find a workflow for deriving requirements for subsystems in Unified Modeling Language–based design models in the whitepaper "[Rational Unified Process ® for Systems Engineering, TP 165](#)" and in the RUP plug-in for RUP SE.² As in the use-case flowdown method, *these requirements are "derived" from studying collaborations in the design models.* That means the subsystem requirements will have a many-to-many relationship with the system requirements.

To help Rational customers understand this concept, I explain that it is optimal for subsystems to provide services in an "m-n relationship." That is, each subsystem requirement may be derived from many system requirements, and each system requirement can result in many subsystem requirements. And subsystem requirements are related to architectural decisions, as we saw in the automobile example above. In fact, they can even think about this approach to subsystems as a form of functional decomposition --with derived subsystem requirements. To quote Nelson Mandela, "If you want to make peace with your enemy, you have to work with your enemy. Then he becomes your partner."

Notes

¹ A more detailed description of this process and its shortfalls can be found in *Practical Software Requirements* by Benjamin Kovitz (Manning, 1999).

² Available at <http://www.rational.net/> (authorization required).



For more information on the products or services discussed in this article, please click [here](#) and follow the instructions provided. Thank you!

