Chapter 2
Architecting for Improved Evolvability

Pierre America, Piërre van de Laar, Gerrit Muller, Teade Punter, Nico van Rooijen, Joland Rutgers, and David Watts

Abstract We define evolvability as the ability of a product family to respond effectively to change, or in other words, to accommodate changing requirements with predictable, minimal effort and time. Despite this definition, evolvability is an elusive concept in system engineering: It is almost impossible to measure and it depends on many factors, of which architecture is only one. Nevertheless there are a number of things that architects can do to improve evolvability and in this chapter we give a few guidelines for this. We start with explaining the main causes of evolvability problems: lack of shared understanding, insufficient motivation to invest in architecture, and high expected effort and cost of architecture improvements. Then we describe how to address each of these causes. We present the following three recommendations: First, establishing a reference architecture helps fostering shared understanding in the developing organization. Second, a thorough long-term cost-benefit analysis of architectural investments may provide the required motivation. Finally, there are a number of measures that can be taken to deal with the high cost and effort of implementing architecture changes, such as simplicity, size reduction, supplier support, and incremental change. At the end of the chapter we give a number of examples.

Keywords Architecting · Evolvability

P. America (✉)
Philips Research, High Tech Campus 37, 5656 AE Eindhoven, The Netherlands
e-mail: Pierre.America@philips.com

P. van de Laar, T. Punter, and D. Watts
Embedded Systems Institute, Den Dolech 2, LG 0.10, 5612 AZ Eindhoven, The Netherlands
e-mail: Pierre.van.de.Laar@esi.nl; Teade.Punter@esi.nl; David.Watts@esi.nl

G. Muller
Buskerud University College, Frogs vei 41, 3603 Kongsberg, Norway
e-mail: Gerrit.Muller@gmail.com

N. van Rooijen and J. Rutgers
Philips Healthcare MRI, Veenpluis 4-6, 5684 PC Best, The Netherlands
e-mail: Nico.van.Rooijen@philips.com; Joland.Rutgers@philips.com

2.1 Introduction

In Chap. 1 we saw that complex software-intensive systems, such as MRI scanners, are constantly subject to changes in their requirements. We have also defined *evolvability* as the ability to respond effectively to change, in other words, the ability of a product family to accommodate changing requirements with predictable, minimal effort and time. In this chapter, we will write about a 'product' or 'system', where in reality we mean a family of systems, since nowadays it is extremely rare that a single system is developed with no variety at all. In Chap. 1 we also saw that evolvability is not a straightforward property of the system itself, but rather it depends on the system in its context, both during development and during use. This can be indicated by the acronym BAPO: business, architecture, processes and organization (van der Linden et al. 2007). Within this very broad area, the Darwin project, and therefore this book, focuses on the (system and software) architecture but takes its context of business, processes, and organization explicitly into account, just like architects should. Therefore the focus of this chapter can be characterized fairly well by the question: 'What can architects do to improve evolvability?'

Of course, the intended readers for this chapter are in the first place system and software architects, since the chapter directly addresses their evolvability concerns. However, the chapter may also be useful to managers, designers, researchers, and possibly others, because it may help them to understand evolvability better, even though the focus will be on the architects' concerns.

In this chapter, Sect. 2.2 identifies the most important causes of problems with evolvability. Section 2.3 presents a number of guidelines to improve evolvability. Then Sect. 2.4 provides some examples of applying these guidelines and finally Sect. 2.5 gives some concluding remarks.

2.2 Main Causes of Evolvability Problems

When analyzing the problem of evolvability, we found three main causes for it:

- Lack of shared understanding
- Insufficient motivation to invest in evolvability
- Large effort and high cost of architecture improvements

We will discuss these causes in the following subsections. Section 2.3 will then discuss guidelines for improvements.

2.2.1 Lack of Shared Understanding

By far the most important cause of evolvability problems is lack of shared understanding. Ideally, when dealing with complex systems, many different people should have a pretty good understanding of the following items:

- The current system: How does it work? How is it structured? What varieties exist? What are its limitations?
- The current requirements: What is really needed? What is currently not achieved?
- The current problems: What are the most important ones? Who is affected?
- The objectives of the organization: What do we want to achieve and when, regarding our products and possibly services?

The following people should share this understanding:

- Architects, designers, and developers
- Managers at various levels
- Marketing people, application specialists, researchers, consultants
- Other stakeholders, e.g., in manufacturing, service

When we talk about shared understanding we mean that many people have the same knowledge, even in the sense that they think of the various aspects of the system in the same terms and moreover that they know that the knowledge is shared among them: I know that you know...

Of course, it is not necessary that everyone knows everything about every system. Every person has a specific role in the organization and needs specialized knowledge to fulfill his role. Nevertheless it is important that people in the organization share a significant amount of knowledge explicitly. It is not enough that knowledge is in the heads of the specialists. Only when the knowledge is out in the open can it be analyzed, discussed, and checked for consistency with other items.

Lack of shared understanding typically leads to decisions that favor local optimizations, while at the system level they often cause problems. When these problems are detected in a later stage they are much more expensive to fix. This might be disastrous for the system, or even the business, as a whole.

2.2.2 Insufficient Motivation to Invest in Evolvability

Another source of evolvability problems is lack of motivation to invest in it. Evolvability can be improved by spending effort in different areas, including business, architecture, processes, and organization. When resources are limited, and they always are, difficult choices must be made regarding the allocation of those resources. In that context, investments in evolvability have to compete with investments aimed at other benefits, for example new products with improved functionality or quality aspects. In most of these cases, the investment in evolvability requires more time to pay off than the other investments. An improved product can be offered to customers as soon as it is ready, and it generates cash flow from that moment on. Evolvability, in itself, cannot be sold to customers; it can only lead to a shorter time-to-market and reduced development cost in the future, maybe years away. Therefore the investment in product improvement looks more attractive, even though in the long term the evolvability investment is more profitable.

2.2.3 Large Effort and High Cost of Architecture Improvements

A third reason for evolvability issues is the high effort and cost to implement architecture improvements. There are several areas where small investments can lead to a large improvement, but architecture is not one of them. This is in the nature of architecture. Recent definitions describe architecture as the set of most important technical decisions about the system, where "most important" refers not only to the impact of the decision, but also to the effort involved in changing it. *Defining* a new and improved architecture is already a considerable challenge, because it typically involves lengthy discussions among the organization's most experienced experts, feasibility studies, and several other measures to make sure the right decisions are taken. However, the bulk of the effort goes into *implementing* the new architecture. Changing the whole system to conform to the new architecture is often a lot of work. In principle, this could be done in a single project. This has the advantage that such a project typically accommodates a truly innovative mindset, focused on establishing the best possible architecture for the product family. However, such a radical architecture renovation project often results in a severe overrun of time and effort budgets, delaying the benefits of the new architecture until it is completely done. Especially this high risk makes such projects very unattractive.

The alternative would be to implement the new architecture incrementally over a number of projects, e.g., by updating a single component or subsystem at a time, in such a way that each increment results in a working system again, preferably already showing some advantages of the new architecture. Such an incremental approach is often more attractive (and sometimes it is the only option) because it delivers earlier results (albeit partial ones) and does not require large commitments. However, there are also disadvantages. First of all, in an incremental approach the total effort needed to achieve a complete architecture overhaul is typically much larger than by making the necessary changes in one go, because the incremental approach requires that for each modified subsystem the interfaces with the other subsystems must be managed such that the whole system keeps on working adequately. Another danger of incremental improvement is that the total process may take a very long time. This can cause loss of market share because of outdated products. Furthermore the new architecture may already be inadequate by the time it is fully implemented, just as there are gothic cathedrals that are never free of scaffolding because when renovation is done at one end it must start again at the other end. Finally, while doing all these relatively small steps, it is very difficult to develop and maintain a clear and sufficiently ambitious vision of the intended end result.

2.3 General Guidelines for Improving Evolvability

Here we will give a few guidelines that help to improve evolvability. We will address each cause of evolvability problems mentioned above in one of the next subsections.

2.3.1 How to Create Shared Understanding?

The most important guideline to address this issue is the following:

> Architects should produce and present views that enhance understanding of the system in its context, the problems that currently exist, and the objectives of the organization.

Here a *view* is a description of a certain aspect of the system that addresses a particular concern of a set of stakeholders. Such a view typically consists of one or more models, where each model is represented by a diagram, a table, or a piece of text. Together, a collection of views make up an architecture description (ISO/IEC 2007). We call this a *reference architecture* if it is especially aimed at fostering a shared understanding within a whole organization, see Chap. 7 and Muller and Hole (2007). This distinction is important, because the definition of views for a normal architecture looks the same. Whereas a normal architecture describes a concrete product, or product family, that has already been developed or is being developed, a reference architecture intends to describe the principles that are less specific for a particular product and therefore more constant over time. Typically it also shows more of the context of the system, its current problems, and the organization's objectives. One way to show these objectives could be to show a vision of what the system should look like in 10 years' time.

Creating a reference architecture is not easy. It requires at least one extra step of reflection and abstraction over a normal architecture description in order to identify and visualize the essential issues and principles relevant for the system and its stakeholders over a longer period of time, say 10 years. It requires a lot of practice and multiple iterations to come up with views of sufficient quality to share them within the organization for many years. Fortunately, when this has been achieved, further work is limited: The views are expected to remain valid for a long time and to require only minor updates.

Although authoring a good reference architecture is not easy, this does not mean that a reference architecture should consist of very complicated views. The real challenge is to choose the right views and the right abstraction level for each view. We recommend a number of views where each view illustrates a basic aspect of the system. The following list is a good starting point:

- **Functionality.** This is often forgotten, but it typically covers the main reason of existence for many systems. Such a view could illustrate the flow of data, work, or even physical goods through the system and its environment.
- **Space.** An illustration of the spatial layout of the system or a subsystem in its context. It may be useful to draw such a view on different scales, e.g., micrometers, millimeters, and meters.
- **Dynamics.** This illustrates the behavior of the system over time. Here as well, different time scale may be useful (e.g., microseconds, seconds, and hours).
- **Quality attributes.** For each system, a few quality attributes are critical, in the sense that they are not easy to keep within acceptable ranges. Each of these attributes deserves one or more views that illustrate how the attribute is managed.

- **Economics.** A representation of the various kinds of costs and benefits related to the system. This could cover the development and manufacturing of the system as well as its use.
- **Decisions.** Although decisions play a key role in architecting, they are rarely described explicitly in architectural views. Nevertheless, a good description of the main architectural decisions and their mutual relationships may be useful to capture the rationale of the system's design.

We can also formulate a few heuristics for creating a reference architecture:

1. Focus on functions instead of components: They are less likely to change.
2. Focus on information flow instead of control flow: Often the existing control flow is overly constrained, while information shows the essential constraints.
3. Make your drivers explicit, especially the key drivers (Muller 2010): These are extremely valuable to understand why certain decisions were taken.
4. Try out many views and visualizations: The first one that you imagine is probably not the best.
5. First get many details, then abstract from them. This way you can abstract from reality instead of squeezing reality into a previously imagined structure.

Creating the views is not enough. They should be shared among the whole organization. They should be easily accessible not only among the development department, but also, selectively, in other departments, such as marketing, application, manufacturing, and technical support. This sharing can be achieved by presentations by the architects with sufficient occasion for feedback from the audience. The ideal situation would be where the feeling of ownership of the reference architecture is not limited to the architects, but spread among the whole organization.

2.3.2 How to Motivate Investments in Evolvability?

The second guideline, aimed at providing proper motivation for evolvability investments is:

> Architects, together with other stakeholders, should do a thorough long-term analysis of the costs and benefits of evolvability improvements, considering especially how such improvements fit into the business strategy.

In order to ensure that investments in architecture renovation towards improved evolvability are made when they are necessary, it is important to take such decisions on the basis of a quantitative economic analysis instead of intuitive understanding. For this purpose we need good estimates of the benefits as well as the costs of all projects that compete for the manpower of the development department. Typically the costs are already estimated with reasonable accuracy, mostly based on a work breakdown structure. However, the value of a project is much more difficult to estimate, especially for evolvability improvements, which inherently pay off only in the long run. To avoid largely cost-driven decisions (DeMarco 2009), we must therefore pay much more attention to quantification of the benefits.

Ideally, we would like to do a thorough bottom-line analysis, specifying how much money the company will gain by doing each project. In principle, the value of a project that aims at improving evolvability can be estimated by Real Options analysis (Copeland and Antikarov 2003). The theory of options explains how value can be created by allowing decisions to be taken later in time, when there is less uncertainty, e.g., regarding market demand. It is good to understand this principle to grasp the economic value of evolvability. In practice, however, using the principles of Real Options to analyze quantitatively the value of architecture improvements towards evolvability turns out to be extremely difficult (Ivanovic and America 2008).

If a bottom-line analysis is impractical, at the very least we should get a quantitative image of how the proposed new architecture scores towards a number of criteria that are strategically important for the business. For this purpose it is important to get an explicit formulation of the business strategy and how it relates to the relevant aspects of the proposed, more evolvable architecture. Strategy maps (Kaplan and Norton 2004) can be a practical way of making these relationships explicit. Chapter 15 illustrates how this works. Even though this approach does not directly associate a specific amount of money with each project, it still allows decisions to be taken on the basis of clear criteria instead of pure intuition.

2.3.3 How to Reduce Implementation Effort and Cost of Architecture Improvements?

As we saw in Sect. 2.2.3, architecture improvements are often costly by nature and therefore there is no silver bullet to kill this problem. However, here are several guidelines that help in reducing the cost of improvements or at least make it easier to swallow.

2.3.3.1 Simplicity

The most powerful way to reduce the cost of change is simplicity. A simple system is easier to change because it has fewer and clearer internal dependencies (see Chaps. 3 – 6 for techniques to find and visualize such dependencies). Therefore the consequences of changes are easier to comprehend. In that way expensive problems in the late development phases are avoided. Achieving this simplicity, however, is not easy. It requires continuous, conscious effort. A small book by John Maeda (Maeda 2006) can provide some inspiration.

2.3.3.2 Size Reduction

The work involved in changing the architecture of a system is directly related to the size of the system description, e.g., circuit diagrams, program code, or CAD drawings, because the change must be effectuated on that system description. Therefore, we can limit the change effort by making the system description as concise as possible. There are several ways to do that:

- *Feature reduction:* By eliminating various features, in functionality or quality, it is often possible to simplify, reduce, or even remove the parts of the system that realize these features. It is clear that this requires careful consideration of how important each feature is for the customers.
- *Abstraction:* In the software area, it is well-known that programmer productivity in terms of lines of code per month, although widely varying among individuals, is about constant across programming languages (Boehm 1981; Prechelt 2000; Sommerville 2006). Therefore so-called high-level languages, which allow more concise programs by leaving more details to be handled by tools, can reduce the implementation effort needed for the same functionality. Undoubtedly this also works not only for software but also for other areas of system design. Moreover, it works for *modification* of systems as well as their initial design. However, raising this level of abstraction is a slow process in which the worldwide scientific community is involved. Currently, most effort towards higher abstraction levels is spent on model-based development. The challenge is to make adequate use of the results of that work.
- *Refactoring:* By making sure that duplication of functionality in the system is avoided, we can reduce the size of the system description. Care must be taken that this makes the system simpler, not more complex. The key here is in choosing the right concepts and interfaces.
- *Buying instead of making:* Whenever we can buy a component of the system instead of making it ourselves, the system description becomes much smaller. Of course, this increases the dependency on the supplied components and on the supplier. Especially the life cycle of the supplied component and its associated change management are important. Moreover, we must be careful not to make too many undesirable changes to the rest of the system just to make the supplied component fit in.

2.3.3.3 Supplier Support

Often an architectural change involves a transition to newer components or tools supplied by others. Even in cases where the transition is very drastic, for example, when changing to a different programming language, support by the supplier of the new components or tools can significantly reduce the effort. The supplier may provide tools or services to ease the transition, e.g., a translator from the old to the new programming language, or it may ensure that the new component or tool is backward compatible with the old one. The supplier typically provides this support because it is in its own interest that its clients make the transition. To make optimal use of supplier support, it is important to use components and tools in the way intended by the supplier, not too creatively, and to follow the supplier's developments closely enough, i.e., change before the component or tool becomes obsolete.

2.3.3.4 Incremental Change

Despite all the above measures, implementing a new architecture completely is typically a significant effort. We have seen in Sect. 2.2.3 that doing this implementation in one radical project is simply infeasible in many cases. On the other hand, incremental architecture renovation often struggles with a slow innovation rate and loss of direction. Therefore we recommend the following incremental approach:

1. Set a target date for realizing the new architecture well into the future (e.g., five years ahead).
2. Develop a description for the target architecture that is detailed enough to guide implementation and ambitious enough to cover the needs projected for the target date.
3. Make an incremental plan for implementing the target architecture and start carrying it out.
4. Track progress of the implementation rigorously.
5. Roughly every year, assess whether the target architecture still covers the needs projected for the target date. Update the architecture if necessary, but do not shift the target date.
6. When the target date is very close or has already passed, evaluate the architecture improvement activity and start over with a new long-term cycle.

2.4 Examples

In this section, we present a few examples that illustrate the recommendations in the previous section. These examples are all taken from the domain of MRI scanners. More examples can be found in the other chapters of this book.

2.4.1 System Cooling

Figure 2.1 shows a view illustrating the basic principles involved with the cooling of the system. Although this picture looks a bit simple, it does indicate the elements of the system that are most important when discussion power dissipation. We see, among others, the following facts, which therefore do not need to be described textually in an architecture document:

- The following devices generate so much heat that it must be transported out of the system by liquid cooling: RF amplifier, gradient amplifier, gradient coil, and electronics cabinet.
- A scan program determines the power dissipation of the RF amplifier and gradient amplifier, and indirectly the power fed into the RF transmit coil and the gradient coil.

- The various MRI scanners can have different numbers (and types, not shown in Fig. 2.1) of RF amplifiers, RF coils, and gradient amplifiers.
- The patient is most affected by the power radiated from the RF transmit coil.

Certain things are not indicated in Fig. 2.1, but could be considered:

- How much power is transmitted along the various arrows in the picture?
- Here we left this out because it depends on the system configuration and the scan program. You could decide to choose a particular configuration, e.g., the largest one, and program, e.g., the most demanding one, and add those numbers. Alternatively you could add the numbers for various configurations and scan programs separately in a table.
- How is heat transported, e.g., by air flow (patient), liquid flow (liquid cooling cabinet), or otherwise? We left this out because we want to focus on function and flow, see the guidelines in Sect. 2.3.1. However, when discussing a new cooling mechanism the means for heat transportation is crucial.

Figure 2.1 could be part of a reference architecture. It could be complemented by another diagram which provides more detail than this figure, including various implementation details and numerical estimates for several quantities, such as power, flow, and temperature. The advantages of Fig. 2.1 are that it is unlikely to change very often and that it does not overwhelm the reader by the level of detail.

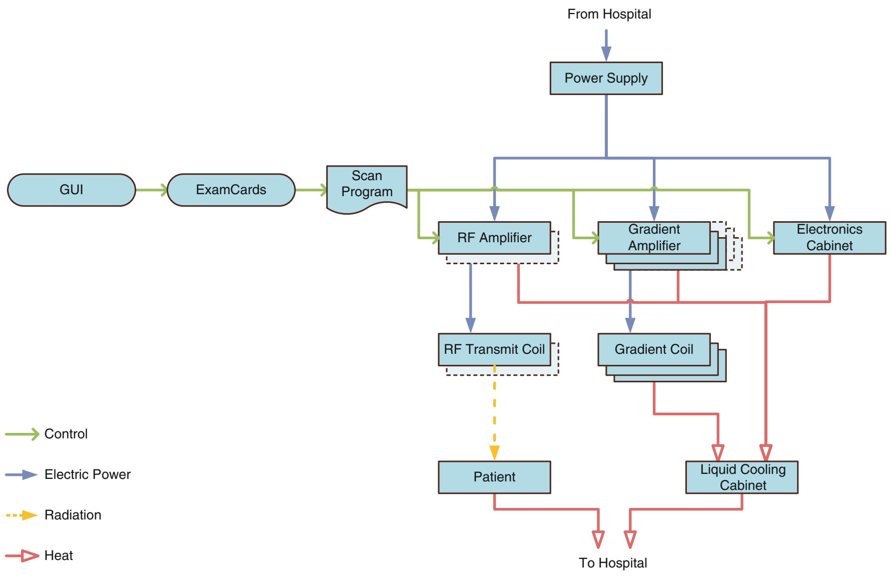


Fig. 2.1 Functional diagram for power and cooling

2.4.2 *Protection Against Hazardous Output: Heat*

We have found that in many circumstances A3 is a suitable format for an architecture view. This is described at length in Chap. 8. Here we provide a different example. Figure 2.2 shows such an A3-formatted view for a reference architecture. It describes the principles and mechanisms involved in preventing danger to the patient because of the heat generated by the MRI scanner. The original size of the figure conforms to the DIN A3 paper size ($297 \times 420\,\text{mm}^2$) and therefore it had to be reduced drastically to fit into this book. Of course it has become unreadable and serves only to give an impression of the overall layout. We will discuss a few elements from Fig. 2.2 separately. The reverse side (not shown here) of the A3 sheet represented in Fig. 2.2 is a suitable place for explanations of this kind.

The European Standard (IEC 2002) discusses multiple kinds of hazardous output of MRI equipment and the protection against them. This standard specifies limits that provide a sensible balance between risk and benefit for both patient and medical staff. This standard defines three levels of operating modes of MRI equipment: normal, first level controlled, and second level controlled. These levels must help the operator to decide whether or not a scan is in the interest of the patient, since the output of MRI equipment may cause undue physiological stress to the patient.

Heating of the patient is one of the hazards of MRI equipment. Adverse biological effects can be caused by temperature rises in tissue that exceed 1 °C (ICNIRP

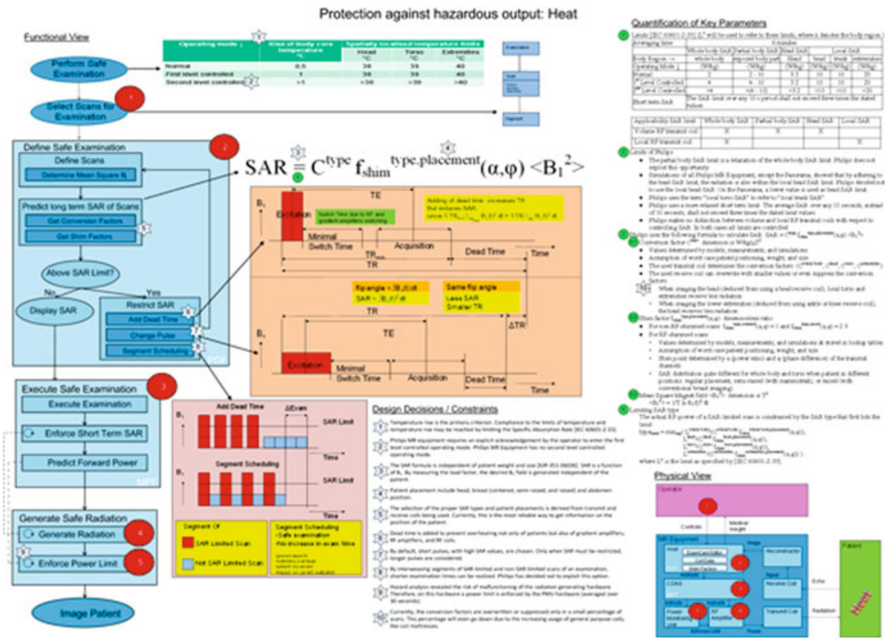


Fig. 2.2 Example of A3 reference architecture view (reduced in size)

Table 2.1 Maximum allowed temperatures (IEC 2002)

		Spatially localized temperature limits		
Operating mode	Rise of body core temperature °**C**	Head °**C**	Torso °**C**	Extremities °**C**
Normal	0.5	38	39	40
First level controlled	1	38	39	40
Second level controlled	>1	>38	>39	>40

Table 2.2 SAR limits (IEC 2002)

Averaging time	Six minutes					
	Whole body SAR	Partial body SAR	Head SAR	Local SAR		
Body region →	Whole body	Exposed body part	Head	Head	Trunk	Extremities
Operating mode ↓	(W/kg)	(W/kg)	(W/kg)	(W/kg)	(W/kg)	(W/kg)
Normal	2	2–10	3.2	10	10	20
First level controlled	4	4–10	3.2	10	10	20
Second level controlled	>4	>(4–10)	>3.2	>10	>10	>20
Short term SAR	The SAR limit over any 10 s period shall not exceed three times the stated values					

1998). Hence, the European Standard limits the temperature rise to the values given in Table 2.1. This is also visible at the top of Fig. 2.2.

The main cause of temperature rise is the exposure to electromagnetic fields with frequencies above about 100 kHz, such as produced by the RF (radio-frequency) chain of the MRI scanner. Therefore the European Standard mentions (only) one possible realization to reach compliance with the limits of the standard: limiting the Specific Absorption Rate (SAR). Available experimental evidence indicates that the exposure of resting humans for approximately 30 min to electric, magnetic, and electromagnetic fields producing a whole-body SAR of between 1 and 4 W/kg results in a body temperature increase of less than 1 °C (ICNIRP 1998). This leads to the SAR limits given in Table 2.2. This is also visible at the top right of Fig. 2.2.

Now that we understand the SAR limits as defined by the European Standard, we can look at Fig. 2.4, also visible at the left of Fig. 2.2, for the functional aspects of SAR control. Here the white numbers in red circles (gray in the printed version of this book) refer to Fig. 2.3, also visible at the bottom right of Fig. 2.2. It shows which part of the system, or the operator, is responsible for each part of the functionality. There are also numbers in small white stars, which refer to textual remarks visible at the bottom center of Fig. 2.2. Finally there are arrows, not shown in Fig. 2.4, but visible in Fig. 2.2, which point to adjacent diagrams that provide further detail. One example of these is shown in Fig. 2.5, which is also visible in the center of Fig. 2.2. This shows how the SAR could be reduced by using a wider but lower RF

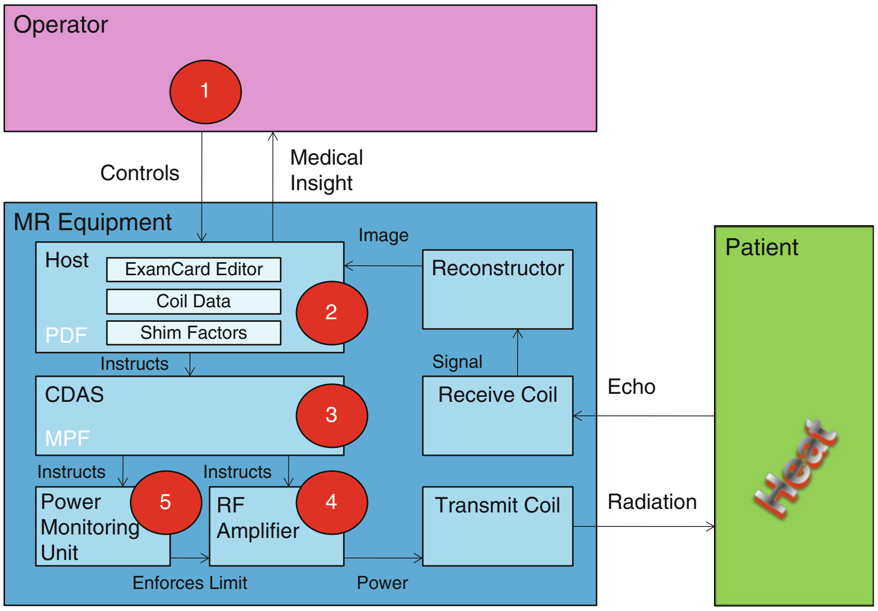


Fig. 2.3 Physical view

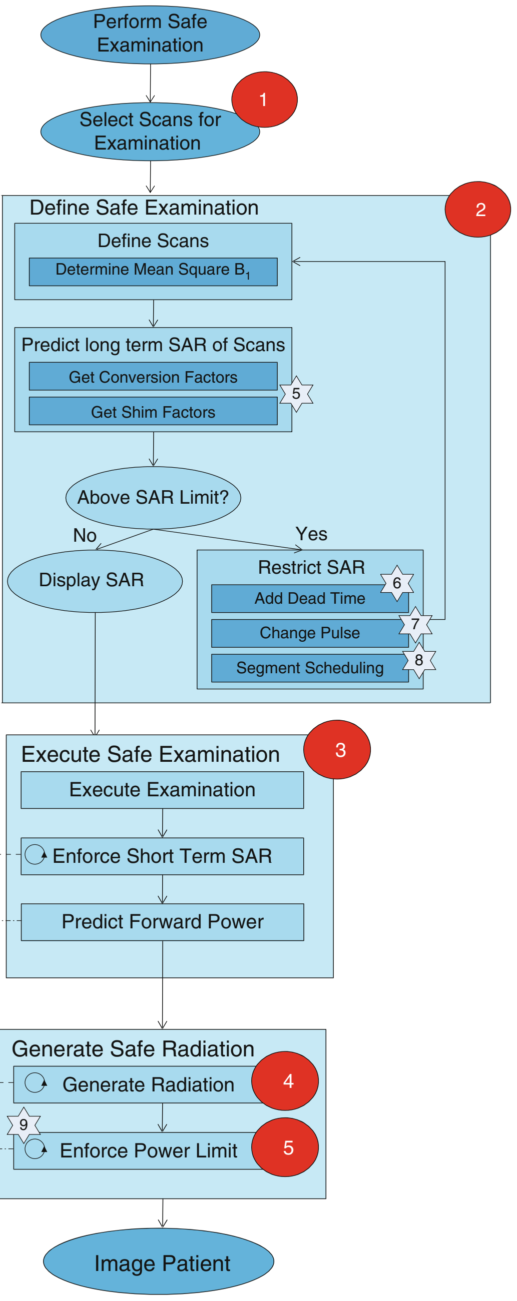
excitation pulse. Another such diagram is about rescheduling scan segments, which is discussed in more detail in Chap. 14.

We see that one single, two-sided, A3-sized sheet of paper can represent a thorough understanding of the principles and mechanisms involved in preventing the patient from being exposed to an unhealthy amount of heat during an MRI scan. This information is unlike to change rapidly, and therefore it is a valuable contribution to a reference architecture.

2.5 Conclusions

Developing evolvable systems is challenging, for various reasons. Nevertheless, as we have seen in Sect. 2.3, there are several things that architects can do to improve evolvability. First of all, they can actively foster shared understanding in the development organization, for example by writing down a reference architecture. Section 2.4 shows what the views in such a reference architecture could look like and Chaps. 7 and 8 provide much more information. Second, architects, together with others in the organization, can provide a financial estimate not only of the cost of architecture improvements, but especially of their value. In that way they can give a solid motivation for making the most promising improvements. Finally, there are several ways to mitigate the cost of such improvements, as Sect. 2.3.3 has shown.

Fig. 2.4 Functional view



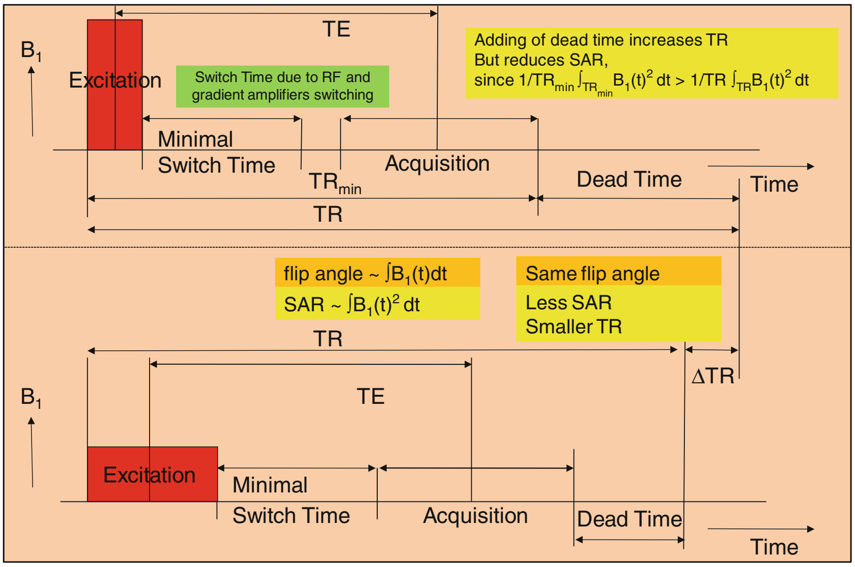


Fig. 2.5 Changing pulse shape

For more inspiration, the rest of this book describes a number of new ways to tackle that challenging evolvability issue. While some of them still need considerable work to reach maturity, many of them can already be applied now to achieve more evolvable systems.

References

Boehm B (1981) Software engineering economics. Prentice Hall, Englewood Cliffs, NJ

Copeland T, Antikarov V (2003) Real options, a practitioner's guide. TEXERE, New York

DeMarco T (2009) Software engineering: an idea whose time has come and gone? IEEE Softw 26(4):96–95

ICNIRP (1998) Guidelines for limiting exposure to time-varying electric, magnetic, and electromagnetic fields (up to 300 GHz). Health Phys 74(4):494–522

IEC (2002) Medical electrical equipment – Part 2–33: particular requirements for the safety of magnetic resonance equipment for medical diagnosis. Standard 60601-2-33, International Electrotechnical Commission

ISO/IEC (2007) Systems and software engineering – recommended practice for architectural description of software-intensive systems. Standard 42010 IEEE Std 1471-2000, International Electrotechnical Commission

Ivanovic A, America P (2008) Economics of architectural investments in industrial practice. In: Proceedings of SPLC –2nd international workshop on measurement and economics of software product lines, Limerick, Ireland, Lero Int. Science Centre, ISBN: 978-0-7695-3303-2

Kaplan RS, Norton DP (2004) Strategy maps: converting intangible assets into tangible outcomes. Harvard Business School, Boston, MA

Maeda J (2006) The laws of simplicity. MIT Press, Cambridge

Muller G (2010) Gaudí System Architecting. http://www.gaudisite.nl/. Accessed 28 May 2010

Muller G, Hole E (2007) Reference architectures; why, what and how. white paper resulting from system architecture forum meeting. http://www.architectingforum.org/whitepapers/SAF_WhitePaper_2007_4.pdf. Accessed 9 June 2010

Prechelt L (2000) An empirical comparison of seven programming languages. IEEE Comp 33(10):23–29

Sommerville I (2006) Software engineering. Addison Wesley, Reading, MA

van der Linden F, Schmid K, Rommes E (2007) Software product lines in action. Springer, Heidelberg