Chapter 19
How Software Architecture can Frame, Constrain and Inspire System Requirements

Eoin Woods and Nick Rozanski

Abstract Historically a system's requirements and its architectural design have been viewed as having a simple relationship where the requirements drove the architecture and the architecture was designed in order to meet the requirements. In contrast, our experience is that a much more dynamic relationship can be achieved between these key activities within the system design lifecycle, that allows the architecture to *constrain* the requirements to an achievable set of possibilities, *frame* the requirements making their implications clearer, and *inspire* new requirements from the capabilities of the system's architecture. In this article, we describe this relationship, illustrate it with a case study drawn from our experience and present some lessons learned that we believe will be valuable for other software architects.

19.1 Introduction

Historically, we have tended to view a system's requirements and its architectural design as having a fairly simple relationship; the requirements drove the architecture and the architecture was designed in order to meet the requirements. However this is a rather linear relationship for two such key elements of the design process and we have found that it is desirable to strive for a much richer interaction between them.

This chapter captures the results of our experience in designing systems, through which we have found that rather than just passively defining a system structure to meet a set of requirements, it is much more fruitful to use an iterative process that combines architecture and requirements definition. We have found that this allows the architectural design to *constrain* the requirements to an achievable set of possibilities, *frame* the requirements making their implications clearer, and *inspire* new requirements from its capabilities.

Our experience has led us to believe that the key to achieving this positive interplay between requirements and architecture is to focus on resolving the forces inherent in the underlying *business drivers* that the system aims to meet. This process is part of a wider three-way interaction between requirements, architecture and project management. In this chapter we focus on the interaction between the requirements and architectural design processes, while touching on the relationship that both have with project management.

We start by examining the classical relationship between requirements and architectural design, before moving on to describe how to achieve a richer, more positive, interaction between requirements and architecture. We then illustrate the approach with a real case study from the retail sector. In so doing, we hope to show how architects need to look beyond the requirements that they are given and work creatively and collaboratively with requirements analysts and project managers in order to meet the system's business goals in the most effective way.

In this work, we focus on the architecture and requirements of *information systems*, as opposed to real-time or embedded systems. We have done this because that is the area in which we have both gained our architectural experience and applied the techniques we describe in the case study.

19.2 Requirements Versus Architecture

While both are very familiar concepts, given the number of interpretations that exist of the terms "system requirements" and "software architecture" it is worth briefly defining both in order to clearly set the scene for our discussion.

To avoid confusion over basic definitions, we use widely accepted standard definitions of both concepts, and they serve our purposes perfectly well.

Following the lead of the IEEE [7] we define systems requirements as being "*(1) a condition or capability needed by a user to solve a problem or achieve an objective; (2) a condition or capability that must be met or possessed by a system or system component to satisfy a contract, standard, specification, or other formally imposed document; or a documented representation of a condition or capability as in (1) or (2).*"

For software architecture, we use the standard definition from the ISO 42010 standard [8], which is "*the fundamental organization of a system embodied in its components, their relationships to each other, and to the environment, and the principles guiding its design and evolution.*"

So for the purposes of our discussion we view requirements as being the definition of the capabilities that the system must deliver, and the architecture of the system being its structure and organization that should allow the system to provide the capabilities described by its requirements.

Gathering requirements is a complicated, subtle and varied task and for large systems the primary responsibility for this usually lies with a specialist requirements analyst (or "requirements engineer" depending on the domain and terminology in force). The task is usually divided into understanding the *functions* that the system must provide (its functional requirements) and the *qualities* that it must exhibit while providing them (its non-functional requirements, such as security, scalability, usability and so on). There are many approaches for gathering requirements and the output of requirements-gathering activities varies widely [6, 17]. However, in our experience, requirements can be defined via formal written textual paragraphs, descriptions of scenarios that the system must be able to cope with, descriptions of typical usage of the system by one of its users, tables of measurements and limits that the system must meet or provide, user interface mock ups or descriptions, verbal discussions between individuals and so on. In reality the requirements of the system are usually found in a combination of sources –in, across and between the different requirements artefacts available.

For a large system, the design of the system's key implementation structures (its "architecture") is also a complicated and multi-faceted activity. It usually involves the design of a number of different but closely related aspects of the system including component structure, responsibility and interaction, concurrency design, information structure and flow, deployment approach and so on. One of the major difficulties with representing a system's architecture is this multi-faceted nature, and in response to this, most architectural description approaches today are based on the idea of using multiple views, each capturing a different aspect of the architecture [4, 11, 19]. Hence the output of an architectural design exercise is usually a set of views of the system, with supporting information explaining how architecture allows the system to meet its key requirements, particularly its non-functional requirements (or quality properties as they are often known).

Finally, it is worth drawing a distinction between both the requirements and architecture of the system and their *documented representations*. In informal discussion we often merge the two concepts and refer to "requirements" to mean both the actual capabilities that the system must provide and the written form that we capture them in. Even more likely is confusion as to whether the term "the architecture" refers to the actual architecture of the system or the architectural documentation that explains it. In both cases, we would point out (as we and others have elsewhere, such as [2]) that whether or not they are clearly defined and captured, all systems have requirements and an architecture. We would also suggest that whether the latter meets the needs of the former is one of the significant determinants of the success for most systems. For this discussion, we focus on the real-world processes for requirements capture, architecture design and the interplay between them.

19.3 The Classical Relationship

Recognition of the importance of the relationship between the requirements and the design of a system is not a recent insight and software development methods have been relating the two for a long time.

The classical "Waterfall" method for software development [18] places requirements analysis quite clearly at the start of the development process and then proceeds to system design, where design and architecture work would take place. There is a direct linkage between the activities, with the output of the requirements analysis processing being a (complete) set of specifications that the system must meet, which form the input to the design process. The design process is in turn a problem-solving activity to identify a complete design for the system that will allow it to meet that specification. The waterfall method is not really used in practice due to its rigidity and the resulting high cost of mistakes (as no validation of the system can be performed until it is all complete) but it does form a kind of cultural backdrop upon which other more sophisticated approaches are layered and compared. From our current perspective, the interesting thing about the waterfall approach is that although primitive, it does recognise the close relationship of requirements analysis and system architecture. The major limitation of the model is that it is a one-way relationship with the requirements being fed into the design process as its primary input, but with no feedback from the design to the requirements.

The well-known "spiral" model of software development [3] is one of the better-known early attempts at addressing the obvious limitations of the waterfall approach and has informed a number of later lifecycles such as Rational's RUP method [12]. The spiral model recognises that systems cannot be successfully delivered using a simple set of forward-looking activities but that an iterative approach, with each iteration of the system involving some requirements analysis, design, implementation and review, is a much more effective and lower-risk way to deliver a complicated system. The approach reorganises the waterfall process into a series of risk-driven, linked iterations (or "spirals"), each of which attempts to identify and address one or more areas of the system's design. The spiral model emphasises early feedback to the development team by way of reviews of all work products that are at the end of each iteration, including system prototypes that can be evaluated by the system's main stakeholders. Fundamentally the spiral model focuses on managing risk in the development process by ensuring that the main risks facing the project are addressed in order of priority via an iterative prototyping process and this approach is used to prioritise and guide all of the system design activities.

A well-known development of the spiral model is the "Twin Peaks" model of software development, as defined by Bashar Nuseibeh [15] which attempts to address some limitations of the spiral model by organising the development process so that the system's requirements and the system's architecture are developed in parallel. Rather than each iteration defining the requirements and then defining (or refining) the architecture to meet them, the Twin Peaks model suggests that the two should be developed alongside each other because "*candidate architectures can constrain designers from meeting particular requirements, and the choice of requirements can influence the architecture that designers select or develop.*" While the spiral model's approach to reducing risk is to feedback to the requirements process regularly, the Twin Peaks model's refinement of this is to make the feedback immediate during the development of the two. By running concurrent, interlinked requirements and architecture processes in this way the approach aims to address some particular concerns in the development lifecycle, in particular

"I will know it when I see it" (users not knowing what their requirements are until something is built), using large COTS components within systems and rapid requirements change.

Most recently, the emergence of Agile software development approaches [14] has provided yet another perspective on the classical relationship between requirements and design. Agile approaches stress the importance of constant communication, working closely with the system's end users (the "on-site customer") throughout the development process, and regular delivery of valuable working software to allow it to be used, its value assessed and for the "velocity" (productivity) of the development team to be measured in a simple and tangible way. An important difference to note between the Agile and spiral approaches is that the spiral model assumes that the early deliveries will be prototype software whereas an Agile approach encourages the software to be fully developed for a minimal feature set and delivered to production and used (so maximising the value that people get from it, as early as possible). In an agile project, requirements and design artefacts tend to be informal and lightweight, with short "user stories" taking the place of detailed requirements documentation and informal (often short-lived) sketches taking the place of more rigorous and lengthy design documentation. The interplay between requirements and design is quite informal in the Agile approach, with requirements certainly driving design choices as in other approaches, and the emphasis being on the design emerging from the process of adding functions to the system, rather than "upfront" design. Feedback from the design to the requirements is often implicit: a designer may realize the difficulty of adding a new feature (and so a major piece of re-design – refactoring – is required), or spot the ability to extend the system in a new way, given the system's potential capabilities, and suggest this to one of the customers who may decide to write a new "user story."

In summary, the last 20 years have seen significant advances in the approach taken to relating requirements and design, with the emphasis on having design work inform the requirements process as early as possible, rather than leaving this until the system is nearly complete. However the remaining problem that we see with all of these approaches is that architecture is implicitly seen as the servant of the requirements process. Our experience suggests that in fact it is better to treat these two activities as equal parts of the system definition process, where architecture is not simply constrained and driven by the system's requirements but has a more fundamental role in helping to scope, support and inspire them.

19.4 A Collaborative Relationship for Requirements and Architecture

We have found that the basis for defining a fruitful relationship between requirements and architecture needs to start with a consideration of the *business drivers* that cause the project to be undertaken in the first place. We consider the business drivers to be the underlying external forces acting on the project, and they capture the fundamental motivations and rationale for creating the system. Business drivers answer the fundamental “why” questions that underpin the project: why is developing the system going to benefit the organization? why has it chosen to focus its energies and investment in this area rather than elsewhere? what is changing in the wider environment that makes this system necessary or useful? Requirements capture and architectural design, on the other hand, tend to answer (in different ways) the “what” and “how” questions about the system.

It is widely accepted that business drivers provide context, scope and focus for the requirements process, however we have also found them to be an important input into architectural design, by allowing design principles to be identified and justified by reference to them. Of course the requirements are also a key input to the architectural design, defining the capabilities that the architecture will need to support, but the business drivers provide another dimension, which helps the architect to understand the wider goals that the architecture will be expected to support. These relationships are illustrated by the informal diagram in Fig. 19.1.

Having a shared underlying set of drivers gives the requirements and architecture activities a common context and helps to ensure that the two are compatible and mutually supportive (after all, if they are both trying to support the same business drivers then they should be in broad alignment). However, it is important to understand that while the same set of drivers informs both processes, they may be used in quite different ways.

Some business drivers will tend to influence the requirements work more directly, while others will tend to influence architectural design more. For example, in a retail environment the need to respond to expansion from a single region where the organisation has a dense footprint of stores into new geographical areas is likely to have an effect on both the requirements and the architecture. It is clear that such drivers could lead to new requirements in the area of legislative flexibility, logistics and distribution, the ability to have multiple concurrent merchandising strategies, information availability, scalability with respect to stores and sales etc. However, while these requirements would certainly influence the architecture, what is maybe less obvious is that the underlying business driver could directly influence the architecture

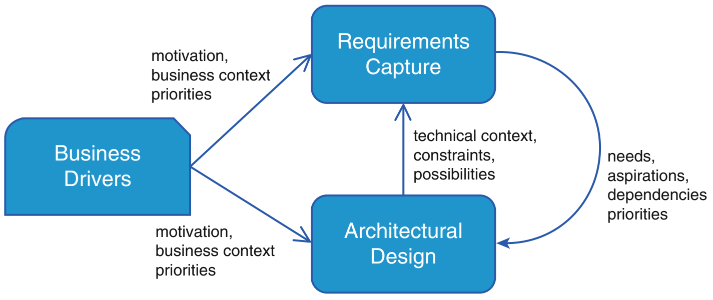
Requirements Capture
motivation, business context priorities
Business Drivers
technical context, constraints, possibilities
needs, aspirations, dependencies priorities
motivation, business context priorities
Architectural Design

Fig. 19.1 Business drivers, requirements and architecture

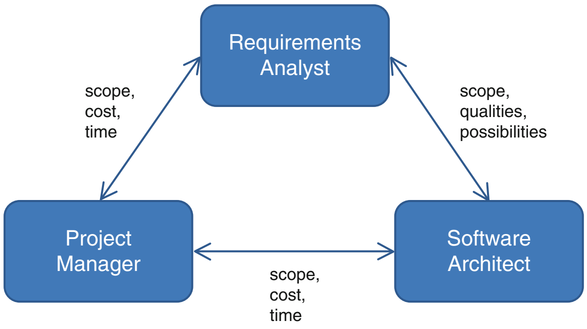

Fig. 19.2 The three decision makers

in other ways, such as needing to ensure that the system is able to cope with relatively high network latency between its components or the need to provide automated and/or remote management of certain system components (e.g. in-store servers). These architectural decisions and constraints then in turn influence the requirements that the system can meet and may also suggest completely new possibilities. For example, the ability to cope with high network latencies could both limit requirements, perhaps constraining user interface options, and also open up new possibilities, such as the ability for stores to be able to operate in offline mode, disconnected from the data center, while continuing to sell goods and accept payments.

The other non-technical dimension to bear in mind is that this new relationship between requirements and architecture will also have an effect on the decision-making in the project. Whereas traditionally, the project manager and requirements engineer/analyst took many of the decisions with respect to system scope and function, this now becomes a more creative three-way tension between project manager, requirements engineer and software architect as illustrated by the informal diagram in Fig. 19.2.

All three members of the project team are involved in the key decisions for the project and so there should be a significant amount of detailed interaction between them. The project manager is particularly interested in the impact of the architect's and requirements analyst's decisions on scope, cost and time, and the requirements analyst and architect negotiate and challenge each other on the system's scope, qualities and the possibilities offered by the emerging architecture.

19.5 The Interplay of Architecture and Requirements

As we said in the previous section, the relationship between requirements and architecture does not need to be a straightforward flow from requirements "down" to architecture. Of course, there is a definite flow from the requirements

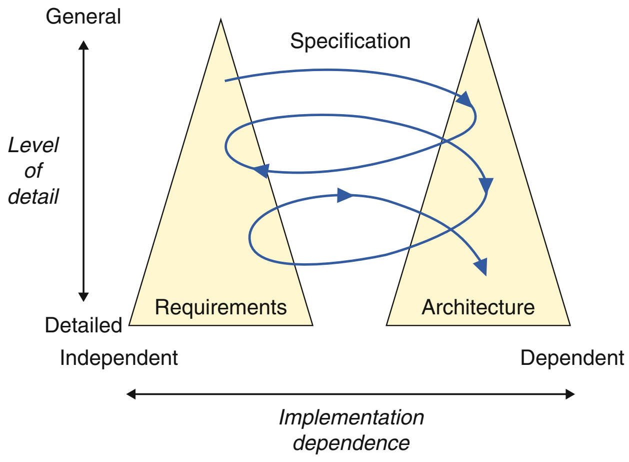

Fig. 19.3 Twin peaks (from [15])

analysis activity into the architectural design, and the requirements are one of the architect's primary inputs. But rather than being a simple one-way relationship, we would suggest that it is better to aim for a more intertwined relationship in the spirit of Twin Peaks, but developing this theme to the point where architecture frames, constrains and inspires the requirements as both are being developed. So we need to consider how this process works in more detail.

Bashar Nuseibeh's "Twin Peaks" model, as shown in Fig. 19.3, shows how the requirements definition and architectural design activities can be intertwined so that the two are developed in parallel. This allows the requirements to inform the architecture as they are gathered and the architecture to guide the requirements elicitation process as it is developed. The process of requirements analysis informing architectural design is widely accepted and well understood, as the requirements are a primary input to the architectural design process and an important part of architectural design is making sure that the requirements can be met. What is of interest to us here is how the architectural design process influences the requirements-gathering activity and we have found that there are three main ways in which this influence manifests itself.

19.5.1 The "Framing" Relationship

Starting with the simplest case, we can consider the situation where the architecture *frames* one or more requirements. This can be considered to be the classical case, where the requirements are identified and defined by the requirements analysis process and then addressed by the architecture. However, when the two are being developed in parallel then this has the advantage that the architecture provides context and boundaries for the requirements during their development rather than waiting for them to be completed. Such context provides the requirements analyst with insights into the difficulty, risk and cost of implementing the requirements and so helps them to balance these factors against the likely benefits that implementing the requirement would provide. If this sort of contextual information is not provided when eliciting requirements for a system then there is the distinct danger that "blue sky" requirements are specified without any reference to the difficulty of providing them. When the architecture is developed in parallel with the requirements, this allows the architect to challenge requirements that would be expensive to implement. In cases where they are found to be of high value, consider early modifications or extensions to the system's architecture to allow them to be achieved at lower cost or risk.

For example, while a "surface" style user interface might well allow new and innovative functions to be specified for a system, such devices are relatively immature, complicated to support, difficult to deploy and expensive to buy, so it would be reasonable for any requirements that require such interfaces to be challenged on the grounds of cost effectiveness and technical risk. The architecture doesn't prevent this requirement from being met, but investigating its implementation feasibility in parallel with defining the requirement allows its true costs and risks to be understood.

19.5.2 The "Constraining" Relationship

In other situations, the architect may realize that the implementation of a requirement is likely to be very expensive, risky or time-consuming to implement using any credible architecture that they can identify. In such cases, we say that the architecture *constrains* the requirements, forcing the requirements analyst to focus on addressing the underlying business drivers in a more practical way.

To take an extreme example, while it is certainly true that instant visibility of totally consistent information across the globe would provide a new set of capabilities for many systems, it is not possible to achieve this using today's information systems technology. It is therefore important that a requirements analyst respects this constraint and specifies a set of requirements that do not require such a facility in order to operate. In this case, understanding the implementation possibilities while the requirements are being developed allows a requirement to be highlighted as impossible to meet with a credible architecture, so allowing it to be removed or reworked early in the process.

19.5.3 The "Inspiring" Relationship

Finally, there are those situations where the architectural design process actually *inspires* new aspects of the emerging requirements, or "the solution drives the problem" as David Garlan observed [5]. However while Garlan was commenting on this being possible in the case of product families (where the possible solutions are already understood from the existing architecture of the product family), we have also seen this happen when designing new systems from scratch. As the architecture is developed, both from the requirements and the underlying business drivers, it is often the case that architectural mechanisms need to be introduced which can have many potential uses and could support many types of system function. While they have been introduced to meet one requirement, there is often no reason why they cannot then also be used to support another, which perhaps had not been considered by the requirements analyst or the system's users.

An example of this is an architectural design decision to deliver the user interface of the system via a web browser, which might be motivated by business drivers around geographical location, ease of access or low administrative overhead for the client devices. However, once this decision has been made, it opens up a number of new possibilities including user interface layer integration with other systems (e.g. via portals and mash ups), the delivery of the interface onto a much wider variety of devices than was previously envisaged (e.g. home computers as well as organisational ones) and accessing the interface from a wider variety of locations (e.g. Internet cafes when employees are travelling as well as office and home based locations). These possibilities can be fed back into the requirements process and can inspire new and exciting applications of the system. This effectively "creates" new requirements by extending the system's possible usage into new areas.

So as can be seen there is great potential for a rich and fruitful set of interactions between requirements analysis and architectural design, leading to a lot of design synergy, if the two can be performed in parallel, based on a set of underlying business principles.

In practice, achieving these valuable interactions between requirements and architectural design means that the requirements analysts and the architects must work closely together in order to make sure that each has good visibility and understanding of the other's work and that there is a constant flow of information and ideas between them.

As we said in the previous section, it is also important that the project manager is involved in this process. While we do not have space here to discuss the interaction with the project manager in detail, it is important to remember that the project manager is ultimately responsible for the cost, risk and schedule for the project. It is easy for the interplay between requirements and architecture to suggest many possibilities that the current project timescales, budget and risk appetite do not allow for, so it is important that the project manager is involved in order to ensure that sound prioritisation is used to decide what is achievable at the current time, and what needs to be recorded and deferred for future consideration.

19.6 Case Study

19.6.1 The Problem

A major clothing retailer was experiencing problems with stock accuracy of size-complex items in its stores, leading to lost sales and a negative customer perception.

A *size-complex* such as a men's suit is an expensive item which is sold in many different size permutations. A store will typically only have a small stock of each size permutation (for example, 44-Regular) on display or in its stockroom, since larger stock levels take up valuable store space and drive up costs in the supply chain.

Manual counting of size-complex items is laborious and error-prone. Even a small counting error can also lead to a *critical stock inaccuracy*, where the store believes it has some items of a particular size in stock but in fact has none. Critical inaccuracies lead to lost sales when customers cannot find the right size of an item they want to buy.

According to inventory management systems, the retailer's stock availability was around 85% for size-complex lines (that is, only 15% were sold out at any one time). However stock sampling indicated that real availability was as low as 45% for some lines, and that critical inaccuracy (where the line is sold out but the stock management system reports that there is stock available in store) was running as high as 15%. This was costing millions of pounds in lost sales, and also driving customer dissatisfaction up and customer conversion down (so customers were leaving stores without buying anything).

19.6.2 Project Goals

The goal of the project was to drive a 3–5% upturn in sales of size-complex lines by replacing error-prone and time-consuming manual stock counting with a more accurate and efficient automated system. By reducing the time taken to do a stock count from hours to a few minutes, the retailer expected to:

- Increase the accuracy of the stock count;
- Reduce the level of critical inaccuracy to near zero;
- Drive more effective replenishment;
- Provide timely and accurate information to head office management.

19.6.3 Constraints and Obstacles

The new system was subject to some significant constraints because of the environment into which it was to be used and deployed.

- The in-store equipment had to be simple to use by relatively unskilled staff with only brief training.
- The in-store equipment had to be robust and highly reliable. The cost of repair or replacement of units in stores was high and would eat away at much of the expected profits.
- The system had to be compatible with the infrastructure currently installed in stores. This was highly standardised for all stores and comprised: a store server running a fairly old but very stable version of Microsoft Windows; a wireless network, with some bandwidth and signal availability constraints in older stores because of their physical layout; and a low-bandwidth private stores WAN carrying mainly HTTP and IBM MQ traffic.
- The system had to be compatible with the infrastructure and systems which ran in Head Office and in partner organisations.
- The solution had to minimise the impact on the existing distribution channels and minimise any changes that needed to be made by suppliers and distributors.
- The solution had to minimise any increase in material or production costs.

Some further constraints emerged during the early stages of the project as described below.

19.6.4 Solution Evolution

The eventual architectural solution emerged over a number of iterations.

19.6.4.1 Initial Design

RFID (Radio Frequency Identification) was chosen as the underpinning technology for contactless data transfer. The initial concept was very simple: a passive (non-powered) RFID tag would be attached to each garment, and would store its UPC (universal product code, analogous to a barcode) which defined the garment's size, stroke etc.

A portable RFID reader would read the UPCs of all the garments in the area being counted, collate and filter the data, and send the resulting counts to the central stock management system in the form of a standard "stock correction" (Fig. 19.4).

This design illustrated the *framing* relationship between architecture and requirements. The retailer had some experience of RFID for stock-taking, but only at the pallet level, not for individual items.

19.6.4.2 First Iteration

Further investigation revealed that there were two types of RFID tag available for use, read-only (write once) and read-write (write many times). Read-write tags

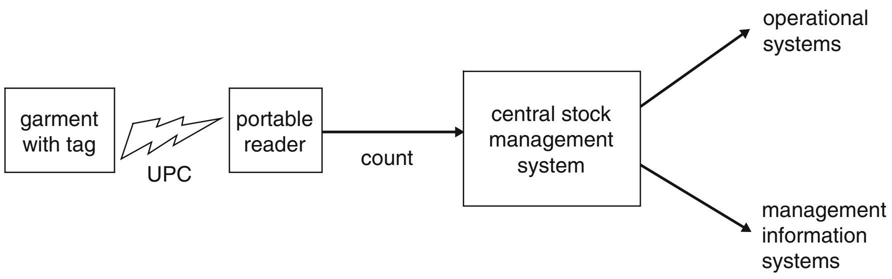
garment with tag
UPC
portable reader
count
central stock management system
operational systems
management information systems

Fig. 19.4 Initial design

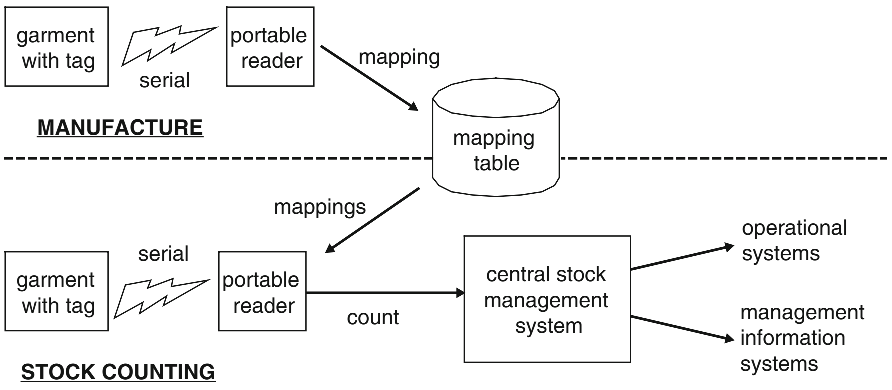

Fig. 19.5 First iteration

would be required for the solution above, since the UPC would need to be written to the tag when it was attached to the garment, rather than when the RFID tag was manufactured. However read-write tags were significantly more expensive and less reliable, so this approach was ruled out.

Ruling out read-write tags was a fairly significant change of direction, and was led primarily by cost and architectural concerns. However, since it had a significant impact on the production and logistics processes, the decision (which was led by architects) required the participation of a fairly wide range of business and IT stakeholders.

Since each RFID tag has a world-unique serial number, a second model was produced in which the serial number of a read-only tag would be used to derive the UPC of the garment. Once the tag was physically attached to the garment, the mapping between the tag's serial number and the garment's UPC would be written to a mapping table in a database in the retailer's data center (Fig 19.5).

There were again some significant implications to this approach, which required the architects to lead various discussions with store leaders, garment manufacturers, logistics partners and technology vendors. For example, it was necessary to develop a special scanning device for use by manufacturers. This would scan the RFID serial number using an RFID scanner, capture the garment's UPC from its label using a barcode scanner, and transmit the data reliably to the mapping system at the retailer's data center. Since manufacturers were often located in the Far East or other distant locations, the device had to be simple, reliable and resilient to network connectivity failures.

This iteration illustrated the *constraining* relationship between architecture and requirements. The immaturity of the read-write RFID technology, and the potential cost implications of this approach led to a solution that was more complex, and imposed some significant new requirements on garment manufacturers, but would be significantly more reliable and cheaper to operate.

19.6.4.3 Second Iteration

It was initially planned to derive the UPC of the counted garment at the time that the tag serial numbers were captured by the in-store reader. However the reader was a relatively low-power device, and did not have the processing or storage capacity to do this. An application was therefore required to run on the store server, which maintained a copy of the mapping data and performed the required collation before the counts were sent off (Fig. 19.6).

This iteration also illustrated the constraining relationship between architecture and requirements.

19.6.4.4 Third Iteration

The next consideration was product returns, an important value proposition for this retailer. If a customer were to return a product for resale, then any existing tag

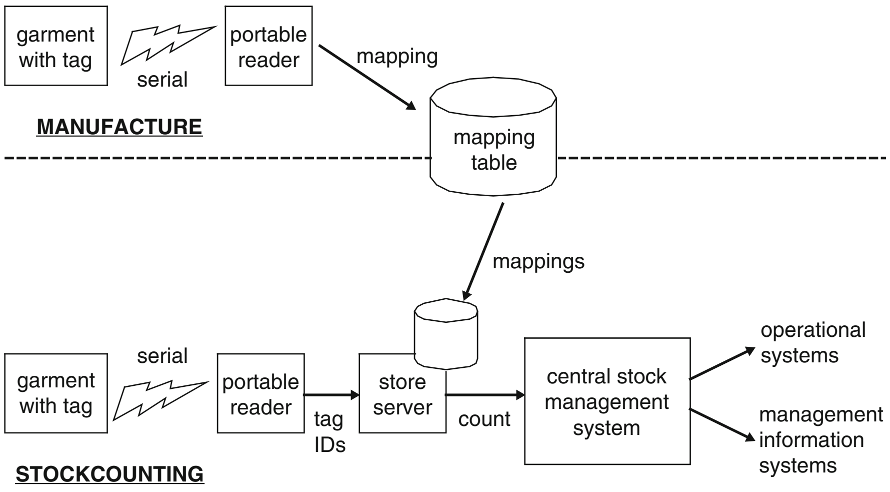

Fig. 19.6 Second iteration

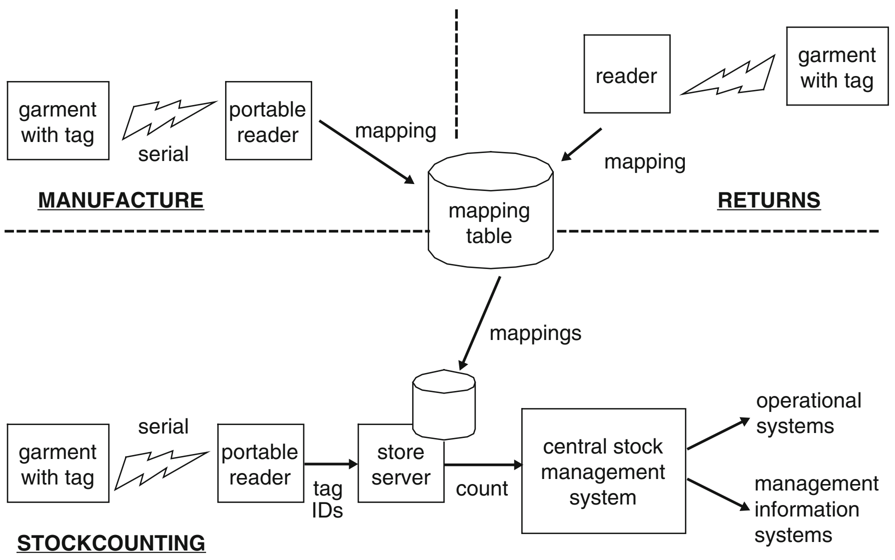

Fig. 19.7 Third iteration

would need to be removed, since it might have been damaged, a new tag attached, and the mapping table updated before the item was returned to the shop floor. This required a special tag reader on the shop floor, and also at the retailer's distribution centers.

This led to the third major iteration of the solution architecture as shown in Fig. 19.7.

This iteration illustrated the *inspiring* relationship between architecture and requirements. It was primarily the consideration of the architecture that prompted the addition of specific returns-processing capabilities to the solution, especially the provision of the specialized tag readers for this purpose.

19.6.4.5 Further Refinements

Discussions were also held with the team that managed the stock system. It already had the capability to enter stock corrections through a data entry screen, but an automated interface would need to be added and it was necessary to confirm that the system could deal with the expected volume of updates once the system was rolled out to all stores. This became an architectural and a scheduling dependency that was managed through discussions between the architects and project managers in both teams.

After surveying the marketplace it became clear that the reader would have to be custom-built. Manufacture was handed off to a third party but software architecture considerations drove aspects of the reader design. It needed to be portable, with its own battery, and had to shut down gracefully, without losing data, if the battery ran out. It also had to present a simple touch-screen user interface.

Another constraint which emerged in the early stages of the project was around customer privacy. The retailer was very keen to protect its brand and its reputation, and the use of RFID to tag clothing was becoming controversial. In particular there was a concern amongst privacy groups that retailers would be able to scan clothes when a customer entered a store, and use the information to identify the customer and track their movements.

To allay any customer concerns, the tag was designed so that it could be removed from the garment and discarded after purchase. The retailer also met with the privacy groups to explain that their fears were unfounded, but needed to be careful not to reveal too much detail of the solution before its launch. Architects were involved in the technical aspects of this and also supported the retailer's Marketing Department who produced a leaflet for customers to explain the tags.

19.6.5 Project Outcome

The system was very successful. The initial pilot showed a consistent uplift in sales for counted lines, which exceeded the project goals. It was popular with staff and business users, and there was no customer resistance to the tags since they could be removed once the item had been paid for.

There were some further lessons from the pilot that were incorporated into the solution. For example, the readers proved to have a significantly larger operating range than expected and care needed to be taken not to count stock that was in the stock room rather than on the shop floor.

19.7 Evaluation of Approach

We have found the iterative approach to be an effective way of highlighting, early in the software development lifecycle, areas where requirements are missing or unclear, and which may otherwise only have become apparent much later in the project. By proposing and evaluating an initial architecture, architecturally significant problems and questions can also be made visible, and the right answers determined, before there has been costly investment in developing software that may later have to be changed.

Using iteration and refinement allows stakeholders to focus on key parts of the solution, rather than the whole, and to develop the overall architecture in stages. It also ensures that the proposed architecture can be considered in the light of real-world constraints, such as time, budget, skills or location, as well as just "architectural correctness."

There are some weaknesses to this approach however. There will be a significant amount of uncertainty and change in the early stages of the lifecycle, and if ongoing changes to the requirements or architecture are not communicated to everyone concerned, then there is a significant risk that the wrong solution will be developed.

Also, many stakeholders are uncomfortable with this level of uncertainty. Users may insist that the requirements are correct as initially specified, and object to changes they view as being IT-driven rather than user-driven. Developers, on the other hand, may struggle to design a system whose requirements are fluid or unclear.

Finally, an iterative approach can be viewed by senior management as extending the project duration. Explaining that this approach is likely to lead to an earlier successful delivery can be difficult.

All of these problems require careful management, oversight and discipline on the part of the project manager, the architect and the requirements analyst.

19.8 Lessons for Architects

The intertwined, parallel approach to relating system requirements and architectural design is the result of our experience working as information system architects, during which time we have attempted to use the ideas of the software architecture research community as they have emerged. As a result of this experience, we have not only refined our ideas about how an architect should work in the early stages of the definition of a new system, but have also learned some useful lessons which we try to capture here to guide others who are attempting the same type of work.

The lessons that we have found to be valuable during our attempts to work collaboratively with requirements analysts and project managers are as follows.

- *Early Involvement* – it is important for you to be involved during the definition and validation of requirements and plans, so aim to get involved as early in the system definition process as possible, even if this involves changing normal ways of working. Offering to perform early reviews and performing some of the requirements capture or planning yourself (particularly around non-functional requirements) can be a useful way into this process.
- *Understand the Drivers* – work hard to elicit the underlying business drivers that have motivated the commissioning of the system or project you are involved in. Often these drivers will not be well-understood or explicitly captured and so understanding and capturing them will be valuable to everyone involved in the project. Once you have the drivers you can understand the motivations for the project and start to think about how design solutions can meet them.
- *Intertwined Requirements and Architecture* – we have found that there are real benefits to the "twin peaks" approach of intertwining parallel requirements gathering and architectural design activity. You should build a working relationship with the requirements analyst(s) that allows this to happen and then work

together with them in order to develop the requirements and the architecture simultaneously with reference to each other.

- *Work Collaboratively* – as well as working in parallel and referencing each other's work, this approach needs a collaborative mindset on the parts of the requirements analyst and the architect, so aim to establish this early and try to work in this way. Of course, "it takes two to tango" so you may not always be successful if the requirements analyst is unused to working in this way, but in many cases if the architect starts to work in an open and collaborative way, others are happy to do so too.
- *Challenge Where Needed* – as requirements start to emerge, do not be afraid to challenge them if you feel that they will be prohibitively expensive or risky to implement, or you spot fundamental incompatibilities between different sets of requirements that will cause severe implementation difficulties. It is exactly this sort of early feedback that is one of the most valuable outputs of this style of working.
- *Understand Costs and Risks*–during the early stages of a large project you are in a unique position to understand the costs and risks of implementing proposed requirements, as it is unlikely that a project manager or a requirements analyst will have a deep knowledge of the design possibilities for the system. You should work with the project manager to understand the costs and risks inherent in the emerging requirements, and then explain them to the other decision makers on the project to allow more informed decisions to be made.
- *Look for Opportunities* – the other dimension that you can bring to the project in its early stages is an understanding of opportunities provided by each of the candidate architectural designs for the system. Your slightly different perspective on the business drivers, away from their purely functional implications, allows a broader view that often results in an architecture that has capabilities within it that can be used in many different ways. By pointing these capabilities out to the requirements analyst, you may well inspire valuable new system capabilities, even if they cannot be immediately implemented.

In short, our experience suggests that rather than accepting a set of completed system requirements, architects need to get involved in projects early and working positively and collaboratively with the requirements analysts and project managers to shape the project in order to increase its chances of successful delivery, while making the most of the potential that its underlying architecture can offer.

19.9 Related Work

As already noted, many other people working in the software architecture field have explored the close relationship between requirements and software architecture.

The SEI's Architecture Trade-off Analysis Method – or ATAM – is a structured approach to assessing how well a system is likely to meet its requirements, based on the characteristics of its architecture [10]. In the approach, a set of key scenarios are identified and analysed to understand the risks and trade-offs inherent in the design and how the system will meet its requirements. When applied early in the lifecycle, ATAM can provide feedback into the requirements process.

A related SEI method is the Quality Attribute Workshop – or QAW – which is a method for identifying the critical quality attributes of a system (such as performance, security, availability and so on) from the business goals of the acquiring organisation [1]. The focus of QAW is identification of critical requirements rather than how the system will meet them, and so is often used as a precursor to ATAM reviews.

Global Analysis [16] is another technique used to relate requirements and software architecture by structuring the analysis of a range of factors that influence the form of software architectures (including organisational constraints, technical constraints and product requirements). The aim of the process is to identify a set of system-wide strategies that guide the software design to meet the constraints that it faces, so helping to bridge the gap between requirements and architectural design.

As well as architecture centric approaches, there have also been a number of novel attempts to relate the problem domain and the solution domain from the requirements engineering community.

One example is Michael Jackson's Problem Frames approach [9], which encourages the requirements analyst to consider the different *domains* of interest within the overall problem domain, how these domains are inter-related via *shared phenomena* and how they affect the system being built. We view techniques like problems frames as being very complimentary to the ideas we have developed here, as they encourage the requirements analyst to delve deeply into the problem, domain and uncover the key requirements that the architecture will need to meet.

Another technique from the requirements engineering community is the KAOS method, developed at the universities of Oregon and Louvain [13]. Like Problem Frames, KAOS encourages the requirements analyst to understand all of the problem domain, not just the part that interfaces with the system being built, and shows how to link requirements back to business goals. Again we view this as a complimentary approach as the use of a method like KAOS is likely to help the requirements analyst and architecture align their work more quickly than would otherwise be the case, as well as understand the problem domain more deeply.

19.10 Summary

In this chapter we have explained how our experience has led us to realise that a much richer relationship can exist between requirements gathering and architectural design than their classical relationship would suggest. Rather than passively accepting requirements into the design process, much better systems are created when the requirements analyst and architect work together to allow architecture to *constrain* the requirements to an achievable set of possibilities, *frame* the requirements making their implications clearer, and *inspire* new requirements from the capabilities of the system's architecture.

References

1. Barbacci M, Ellison R, Lattanze A, Stafford J, Weinstock C, Wood W (2003) Quality attribute workshops (QAWs) 3rd edn. Technical report, CMU/SEI-2003-TR-016, Software Engineering Institute, Carnegie Mellon University
2. Bass L, Clements P, Kazman R (2003) Software architecture in practice, 2nd edn. Prentice Hall, Englewood Cliffs
3. Boehm B (1986) A spiral model of software development and enhancement. ACM SIGSOFT Softw Eng Notes 21(5):61–72
4. Clements P, Bachmann F, Bass L, Garlan D, Ivers J, Little R, Nord R, Stafford J (2002) Documenting software architectures: views and beyond. Addison Wesley, Boston
5. Garlan D (1994) The role of software architecture in requirements engineering. In: Proceedings of the second international conference on requirements engineering. University of York, UK, 18–21 April 1994
6. Hull E, Jackson K, Dick J (2010) Requirements engineering. Springer Verlag, Berlin
7. Institution of Electrical and Electronic Engineers (1990) IEEE standard glossary of software engineering terminology. IEEE Standard 610.12-1990
8. International Standards Organisation (2007) Systems and software engineering – recommended practice for architectural description of software-intensive systems. ISO/IEC Standard 42010:2007
9. Jackson M (2001) Problem Frames. Addison Wesley, Wokingham
10. Kazman R, Klein M, Clements P (2000) ATAM: a method for architecture evaluation. Technical report, CMU/SEI-2000-TR-004, Software Engineering Institute, Carnegie Mellon University, Pittsburgh
11. Kruchten P (1995) The 4 + 1 view model of software architecture. IEEE Softw 12(6):42–50
12. Kruchten P (2003) The rational unified process: an introduction, 3rd edn. Addison-Wesley, Boston
13. van Lamsweerde A, Letier E (2002) From object orientation to goal orientation: a paradigm shift for requirements engineering. In: Proceedings of the radical innovations of software and systems engineering, Venice, Italy, 7–11 October 2002. Lecture notes in computer science, vol 2941. Springer, Heidelberg, pp 325–340
14. Larman C (2004) Agile and iterative development: a manager's guide. Addison-Wesley, Reading
15. Nuseibeh B (2001) Weaving together requirements and architectures. IEEE Comput 34 (3):115–119
16. Nord R, Soni D (2003) Experience with global analysis: a practical method for analyzing factors that influence software architectures. In: Proceedings of the second international software requirements to architectures workshop (STRAW). Portland, Oregon, 9 May 2003
17. Robertson S, Robertson J (2006) Mastering the requirements process, 2nd edn. Addison Wesley, Reading
18. Royce W (1970) Managing the development of large software systems. In: Proceedings of IEEE WESCON, vol 26. Los Alamitos, pp 1–9
19. Rozanski N, Woods E (2005) Software systems architecture: working with stakeholders using viewpoints and perspectives. Addison Wesley, Boston