3 Agile Architecture

M.M. Lankhorst, H.A. Proper

In this chapter, we will elaborate on the use of architecture in relation to agility. We start by clarifying what we mean by 'architecture' and 'enterprise architecture', since these notions are used in various ways within our field. Next, we explain what the roles of architecture can be: on the one hand, it can be used explicitly to design agile systems; on the other hand, it helps an organization to control agility and to keep a balance between stability and change. Finally, we outline how architecture processes fit within an agile context.

3.1 Introduction

The general opinion is that to manage the complexity of any large organization or system, you need architecture. But what exactly does 'architecture' mean? Even in building and construction, the term is ambiguous. It can denote the art and science of designing the built environment, a certain design style, as in 'gothic architecture', or it can refer to the design itself. The earliest use of the term 'architecture' in an IT context dates back to Amdahl, Blaauw and Brooks (1964), who define architecture (of the IBM S/360 mainframe system) as 'the conceptual structure and functional behavior as distinct from the organization of the data flow and controls, the logical design, and the physical implementation'.

Some say that, in the context of information systems, the term 'architecture' should be reserved solely to refer to a set of principles and constraints that should be applied to the design space. For instance, Dietz (2006) defines architecture as a 'normative restriction of design freedom', expressed in the form of principles governing the function and construction of systems. Most other definitions of architecture, however, refer to these structures themselves and not merely to the principles and constraints.

Many more definitions of enterprise, information and IT architecture have been proposed; for an overview, see e.g. Op 't Land et al. (2009) and Greefhorst & Proper (2011). The definition provided by TOGAF actually provides a dualistic view. The first one focused on the description of a system, and the second one on the structure and principles (The Open Group 2011):

1. A formal description of a system, or a detailed plan of the system at component level, to guide its implementation.
2. The structure of components, their inter-relationships, and the principles and guidelines governing their design and evolution over time.

Greefhorst and Proper (2011) also suggest to use a dualistic perspective on architecture by distinguishing *design principles* and *design instructions*. The design principles provide a declarative means to provide normative restrictions of design freedom, while design instructions (by way of e.g. ArchiMate models) provide a more imperative way to provide restrictions of design freedom.

Fehskens (2008) fields valid criticism at many of these definitions, for example that they are very IT-centric. He suggests to define architecture as: 'Those properties of an enterprise, its mission, and their environment, that are necessary and sufficient for the enterprise to be fit for purpose for its mission in that environment, so as to ensure continuous alignment of the enterprise's assets and capabilities with its mission and strategy.' Although this definition includes the goal or use of architectures – something missing from many other definitions – it runs the risk of encompassing 'everything' that is important in an enterprise. In our view, it is essential to focus on properties of the structure of an enterprise, as they can be *designed*, making it sensible to talk about their architecture in the first place.

In this book we do not provide our own definition of architecture. Rather, we will simply use the most commonly used definition from the ISO/IEC/IEEE FDIS 42010 standard (ISO/IEC/IEEE 2011):

> *Architecture*: fundamental concepts or properties of a system in its environment embodied in its elements, relationships, and in the principles of its design and evolution.

Although this definition also has its flaws, it is the most accepted in the architecture community. It is a refinement of the IEEE 1471 definition (IEEE Computer Society 2000), which itself was the result of an intense debate in the community but has now been used for over a decade. For example, the second definition of TOGAF has also been derived from this definition, and it is also used in the ArchiMate community (Lankhorst et al. 2009; The Open Group 2012).

Importantly, this definition also takes the perspective that an architecture is primarily a *conception* of a system, i.e., a mental construct. The standard therefore also distinguishes between architectures and architecture *descriptions*. Furthermore, the definition addresses a system *in its environment*; an architecture cannot be understood without looking at the system's context. Finally, it comprises *fundamental* concepts or properties; an architecture is not just the structure of physical components that make up a system, but remains invariant for different implementations. The focus on *fundamental* concepts or properties is made more specific by Fehskens' definition when he refers to 'The properties … necessary and sufficient for the enterprise to be fit for purpose for its mission in that envi-

ronment'. The ISO/IEC/IEEE definition also accommodates both the notion of a system's structure and the general underlying principles.

An architecture helps you to get an integrated view of a system that you are designing or studying. We can apply this notion of architecture at different abstraction or aggregation levels. We may talk about the architecture of a piece of software, an information system, an organization, an enterprise and even a network of enterprises. The ISO/IEC/IEEE standard takes no position on the question, 'What is a system?' Users of the standard are free to employ any system theory they choose. For example, 'system' could mean a software application, a subsystem, a service, a product line, a system of systems or an enterprise. Systems can be man-made or natural.

It should be clear that these notions of architecture and system do not prescribe any particular order or procedure in their design or development. *Any* system has an architecture, even if it was not consciously designed. In the creation of a system or service, sometimes the architecture will come first, but in many cases, the architecture, design, implementation and usage will co-evolve. This is particularly important when we want to combine architectural thinking and agile methods: Architecture does not equate to 'Big Design Up-Front', the agile movement's bugbear. This is where the overused comparison with architecture in building and construction fails: there, fully detailed plans and designs are needed before construction starts, because change during construction of a building is too difficult and costly. But in business and IT, changes to the artefacts are often easier.

Moreover, in this context the notion of architecture is used at different levels of granularity or scope and with different time scales: from the strategic, large-scale and longer-term 'zoning plan' level, which is (or should be) more principles-based, to the blueprint level of design for individual systems and services, which is more dominated by models. The 'Big Design Up-Front' problems that the agile movement aims to avoid, mainly arise when detailed models and blueprints are used at a wrong level of scale and scope. Regretfully, in practice it happens al too often that architects lose themselves in too detailed and specific 'designs', rather than focusing on those properties that are necessary and sufficient for the enterprise to be fit for purpose.

In this book, we focus on software-intensive socio-technical systems, which are a combination of business, organization and people aspects and the information systems and technology supporting these. Important in this respect is the notion of *enterprise architecture* (EA), where an enterprise is defined as 'the highest level (typically) of description of an organization and typically covers all missions and functions. An enterprise will often span multiple organizations.' (The Open Group 2011).

Fehskens (2008) observes how the interpretation of enterprise architecture has changed from enterprise-wide IT architecture to the architecture of the enterprise, and sees four kinds of use of the term:

- As a discipline or practice: 'I took a course on enterprise architecture.'

- As a process: ‘Enterprise architecture enables business transformation.’
- As applied to a class of things: ‘Every business should have an enterprise architecture.’
- As applied to a specific instance of a thing: ‘We used TOGAF as the basis for our enterprise architecture.’

Given the previous definition of ‘architecture’ as being concerned with *fundamental* properties of a system, enterprise architecture is first and foremost concerned with the backbone of the enterprise: those fundamental design decisions that influence the essence of its operations, facilitate agility, and control risk, and are determined by its business strategy and needs (see also Chap. 2). Thus, enterprise architecture serves several important purposes in IT management:

- It offers a holistic view on the enterprise, creating insight in the various dependencies between and within business and IT, facilitating management decisions by clarifying their effects;
- Based on this holistic view, it provides a backbone for coherent operation and alignment between business and IT;
- It provides a backbone for compliance with rules and regulations, both internally defined and externally required;
- It provides a backbone for the integration of an enterprise with its environment;
- It gives explicit variation points where change may be expected and accommodated.

Later in this chapter, we will describe in more detail how (enterprise) architecture activities can be embedded within agile organizations and projects.

In Chap. 1, we already introduced the prominent role of services in our approach and elsewhere. Modern enterprise architecture methods use services as a pivotal concept in bringing together various business, information and infrastructure aspects and domains. A prime example is the ArchiMate modelling standard for EA (The Open Group 2012; Lankhorst et al. 2009), also used elsewhere in this book. Service-oriented architecture (SOA) has become a prominent architectural style, not just in a technical sense (e.g. with Web services), but also as a way of structuring the business models, processes and organization of an enterprise.

3.2 Architecture to Manage Agility

To some it may seem that architecture is something static, confining everything within its rules and boundaries, and hampering agility and innovation. Many proponents of agile methods are opposed to the use of architecture, categorically classifying it as ‘Big Design Up-Front’ (BDUF). They argue that stakeholders cannot know what they really need and the problem will change anyway before the project is completed, so you cannot provide any useful designs up-front. Indeed, in

many cases stakeholders cannot formulate their requirements up-front and suffer from the IKIWISI syndrome ('I'll know it when I see it'). Moreover, the changing business environment makes stable requirements an illusion to begin with.
However, this is a misconception about the role of architecture. A well-defined architecture helps you in positioning new developments within the context of the existing processes, IT systems, and other assets of an organization, and in identifying necessary changes. Thus, good architectural practice helps an organization innovate and change by providing both stability and flexibility. The insights provided by an enterprise architecture are needed on the one hand in determining the needs and priorities for change from a business perspective, and on the other hand in assessing how the organization may benefit from technological and business innovations.

This also goes back to the distinction between process and system agility, as described in Chap. 2. To achieve a truly agile organization, we should not only use responsive, iterative and interactive processes, but also create organizational and technical systems that can easily be adapted to changing circumstances and requirements. In a competitive environment, an organization should focus its energy on being agile in those change options that differentiate it from its competitors or help it keep up with the market. Change in other aspects is simply a waste of time and energy, since it will not make the organization compete more effectively. Architecture provides the backbone for making such decisions; it forms the stable core of the enterprise and provides the variation points for agility.

Moreover, agility is not the only concern of an enterprise. Many trade-offs have to be made, of cost-efficiency versus flexibility, versus reliability, versus other '-ilities'. A well-designed architecture helps in making such trade-offs, analysing different change scenarios with respect to these different properties, and in assessing their impact across the enterprise.

Thus, architecture serves several important roles in fostering agility: First, it gives designers and developers the insight in a system and its environment they need for making changes. Second, it provides a way of designing organization-level agility, for example by employing specific architecture principles, defining standardized interfaces, creating reusable building blocks and using infrastructures that speed up development. Third, it helps in focusing design effort on those points of variability or uncertainty that are important from a business perspective: where do we expect future changes to occur, and how can we facilitate these?

The latter points are illustrated by Fig. 11. Simply put, a well-designed architecture and infrastructure is an up-front investment ('A' in the figure) that makes later changes easier, faster and cheaper (the coefficient 'b' in the graph is smaller than 'c'). Classical agile development states that you should avoid big design up-front, because you cannot know all. But when you do know which parts of your organizational and technical landscape can provide a stable infrastructure on which enterprise agility is founded, designing those parts up-front is certainly smart.

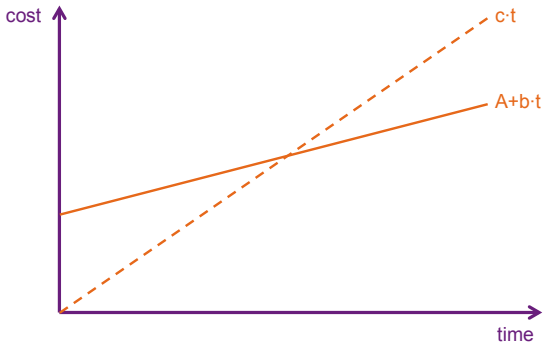

Fig. 11. Up-front investment vs. fully agile development.

3.2.1 Agility Aspects

When considering an enterprise, there are two important capabilities that we might architect, directly related to the different kinds of agility we identified in Chap. 2:

1. The *execution* system, i.e., the 'things' needed for 'business as usual'. Most architecture approaches implicitly focus on this capability, and this is where the notion of system agility is focused.
2. The *innovation* system, i.e., the capability to innovate or change. This ranges from social processes, to the system development process supported by methods and tools. This is targeted by the notion of process agility.

In an agile context (or agile part of the enterprise), we see that these two systems blend together: the critical design focus should shift from having an efficient execution capability to developing an effective combination of the execution and innovation systems. Designing the execution system in such a way that it lends itself to quick changes within giving boundaries and ambitions is a prerequisite for the innovation capability to be effective. We will address the innovation system in Chap. 6; here, we focus on the requirements on the execution system in an agile context.

In Chap. 2, we have outlined the five aspects making up a system's agility: making changes, deploying these changes, dealing with their effects, integrating a system with its environment and decoupling it from this environment (to be reused elsewhere). Many well-established architecture principles have been identified that positively influence the quality attributes comprising these agility aspects; for a broad overview of such principles, see (Greefhorst and Proper 2011, App. A).

Making changes The first aspect of system agility is the ease of making changes. This can be decomposed in customizability by the system's users, adaptability by

system management, analyzability by designers and changeability by developers. These are properties that have to be built in explicitly. In particular analyzability and changeability critically depend on a clear structure, i.e., the architecture of the system. Important and long-established architecture principles apply here, each related to the modularity of the system under concern:

- *Separation of concerns*: using layering and modularization to concentrate functionality in specific places and ensure that changes remain as local as possible;
- *Low coupling and high cohesion*: a low number of relations between subsystems and a high internal cohesion of each subsystem facilitate analysis and understanding, avoid changes propagating throughout the system, and hence greatly enhance the agility of a system;
- *Encapsulation*: if a system has clear interfaces and prevents the environment to depend on its internal implementation, this implementation can be changed without affecting that environment.

An important aspect related to the modularity of the system is the structure of the team. As Conway's law states, 'organizations which design systems [...] are constrained to produce designs which are copies of the communication structures of these organizations' (Conway 1968). In order for two system elements to be connected correctly, the designers and implementers of each element must communicate with each other. If a larger system or service development project requires multiple subteams, the team structure should therefore follow the high-level architecture of the system (and certainly not the other way around).

Deploying changes To deploy a system easily and quickly, qualities such as learnability, installability, testability and manageability need to be addressed. This is particularly important in iterative development processes, where a system and its parts have to be tested, installed and used many times over.

In agile development, testing occurs early and often, and not just at the end of development. Test-driven design is an important agile practice and easily testable systems are therefore important in facilitating such an agile process.

Management (including maintenance) which takes a large effort compared to the actual usage time is nonproductive. If changes to a system require a large management effort, this has a negative impact on agility.

Dealing with the effects of changes In agile development, systems are changed often and these changes may not always work out well. If something goes wrong during a change, the effects should be minimal an easily corrected, to minimize disruption of and risk to the day-to-day operations. The more often a system changes, the more important this becomes. To this end, the architecture of the system should include facilities for e.g. redundancy and recovery.

Systems that are fault-tolerant and can recover quickly and independently after a failure occurs can be tested and used more easily in uncertain or changing conditions.
and in circumstances that have not been predicted at its time of design. Further-

more, systems that have been designed with mechanisms to cope with possible bugs and errors can be deployed more easily in a 'half-finished' state, allowing for rapid testing in practice and short development cycles, thus promoting agility.

Integrating For a system, whether technical or organizational, to be put in place rapidly, it must be easy to connect it to its environment. This requires attention to interoperability and conformance to applicable standards. A system that is highly interoperable is easily connected to other systems. This makes it easier to use it in new or changing environments, hence enhancing the agility of the system and of the organization of which this system is a part.

To some, use of standards may seem limiting to agility, since you cannot freely choose your own solution to a design problem. However, the use of standardized and greatly facilitates the rapid development of new solutions from existing building blocks. Lego is a good example: you can build almost anything from these fixed blocks. Hence agile architectures consist of interoperable elements that are easily configured and combined. A useful architectural principle in this regard is *design by contract* (Meyer 1991), where precise and verifiable specifications of a service are used, for example with preconditions, postconditions and invariants.

Decoupling If you want to change a part of some system, ideally this change does not influence or depend upon any other parts. The more depedencies you have to deal with, the more difficult, risky and time-consuming a change will be. Therefore, a low level of coupling between system elements is an important property of agile systems.

Moreover, an important way of achieving development speed and simplicity is reuse. Reusing system elements requires that they have been designed in such a way that they can easily be decoupled from their environment, and conversely, that the environment does not depend unduly on specific implementation aspects of these elements. If a system or system element is easily replaceable, the organization using it is more agile, since it can respond more quickly when the need for replacement arises because of changing circumstances.

Building new systems from reusable components is an important way of improving agility. The Lego example above serves to illustrate this: because standardized, reusable building blocks are available, new systems can be built quickly and with a relatively low effort. Choosing the right elements that need to be reusable (but are themselves stable) is therefore highly important.

Designing a component for independence and reuse requires information on the potential contexts in which that component may be used in the future. A solid architecture backbone is indispensable to provide such a context. This does not mean that the architecture prescribes the design of each system element; merely that it clearly specifies the boundaries of these elements, i.e., the context, principles, standards and interfaces in which they must fit.

3.2.2 Operating Models

As we have stated before, flexibility does not come for free, and there are other concerns to be addressed as well. It should be an explicit choice where you want to be agile as an enterprise, and which aspects can be standardized or otherwise fixed. A clear set of business goals and drivers (see also Chap. 2) is essential in making such a choice.

Explicit strategic guidance is given by the *operating models* of Ross, Weil and Robertson (2006). As they show with numerous case studies, successful enterprises employ an operating model with clear choices on the levels of integration and standardization of business processes across the enterprise:

- *Diversification*: different business units are allowed to have their own business processes. Data is not integrated across the enterprise. Example: diversified conglomerates that operate in different markets, with different products.
- *Replication*: business processes are standardized and replicated across the organization, but data is local and not integrated. Example: business units in separate countries, serving different customers but using the same centrally defined business processes. Example: a fast food chain replicating its way of working through all its local branches.
- *Coordination*: data is shared and business processes are integrated across the enterprise, but not standardized. Example: a bank serving its clients by sharing customer and product data across the enterprise, but with local branches and advisers having autonomy in tailoring processes to their clients.
- *Unification*: global integration and standardization across the enterprise. Example: the integrated operations and supply chain of a chemicals manufacturing company.

In those operating models that prescribe standardized processes or data integration, project-level agility is bounded by these organization-level choices: a project may not be allowed to define its own business processes or data models, but must comply with company-wide standards. At the organization level, however, this may actually enhance agility: because the organization is explicit about its operational choices, timely decision making is facilitated and the type of response to changes in the environment may be known beforehand. Moreover, use of standardized processes or systems may help in quickly developing solutions to new requirements, as we have argued above.

In addition to the operating model, they provide a stage model of the architectural development of organizations:

1. *Business Silos*: every individual business unit has its own IT and does local optimization.
2. *Standardized Technology*: a common set of infrastructure services is provided centrally and efficiently.

3. *Optimized Core*: data and process standardization, as appropriate for the chosen operating model, are provided through shared business applications (e.g. ERP or CRM systems).
4. *Business Modularity*: loosely coupled IT-enabled business process components are managed and reused, preserving global standards and enabling local differences at the same time.
5. *Dynamic Venturing*: rapidly reconfigurable, self-contained modules are merged seamlessly and dynamically with those of business partners.

The level at which you can achieve agility is related to these stages. Organizations that are at the first stage can only do local optimization, which precludes a coherent agile response at the organization level if, for example, changing market demands or regulatory pressure require this. At stages 2 and 3, the standardization and optimization at the technology level facilitate a global response, but within the bounds of the current business- and organization-level structures. At stages 4 and 5, the business itself becomes adaptable, reconfigurable and fluidly integrated with a dynamic environment.

3.2.3 Standardization and Variation

You might think that standardization is bad for agility. Standardizing in the wrong way can indeed be very detrimental, but you can also use standardized functions to enhance the agility of an enterprise's execution system.

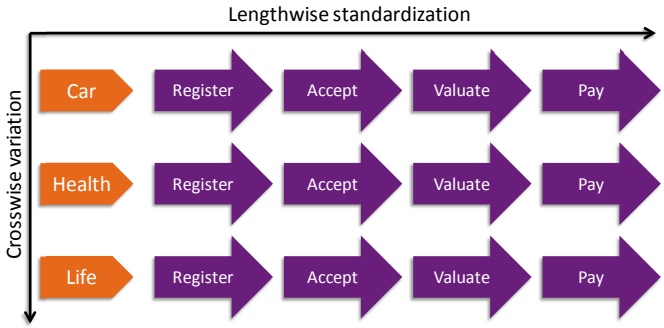

Fig. 12. Standardization vs. variation.

Importantly, you have to differentiate between 'lengthwise' and 'crosswise' standardization and variation (Govers & Südmeier 2011), as depicted in Fig. 12. Lengthwise standardization means standardizing the steps that are taken over the length of a certain type of process, across different products, services or customer segments. Lengthwise standardization can be good for agility. If all processes have the same series of steps, you can possibly reuse some steps in implementing

a new stream. For example, if AgiSurance starts selling an extreme sports insurance, it may perhaps reuse parts of the registration steps from its property insurance products, parts of the valuation from health insurance, et cetera. Thus, lengthwise standardization provides you with a stable architectural backbone for variation across the standardized parts.

Crosswise standardization means using the same implementation of a process step across these different streams (Fig. 13). This may harm your agility, however. By conflating all specific cases into a single process step – let alone a single IT service – such a step will often become unmanageable spaghetti. The result is much too big and complicated, and because it is based on knowledge from many different domains (in the example of AgiSurance, different types of insurance products), nobody has the combined expertise to really understand it. Moreover, everything becomes dependent on everything else; in the example, if a specific type of insurance requires a different kind of valuation, the decision tree for all types of insurance may be impacted.

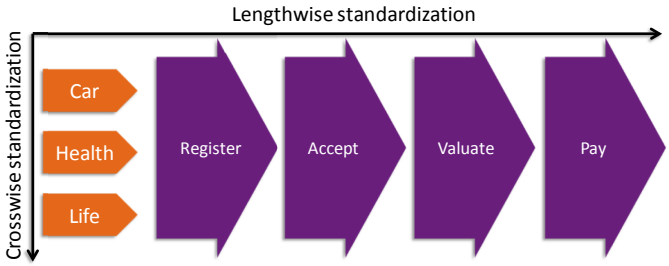

Fig. 13. Crosswise standardization.

Finally, change dynamics are often different for various types of products or customer segments. Take the example of travel insurance versus health insurance: in the Netherlands, the changes in health insurance are largely dictated by government regulation and have a yearly rhythm, where changes to processes and systems must be implemented within a limited time frame before January 1; for travel insurance, insurers may largely decide on their own when and what to change.

You should therefore factor out these types of decisions and not standardize them across different product types. This is one way of promoting the 'decoupling' aspect of system agility described in Sect. 2.3.3. You might do this by defining separate processes or services for each product or, perhaps even better, by treating decisions separately. In Chap. 4, we will show how decision models can be combined with process models to achieve this separation of concerns.

This view on standardization goes against the common wisdom on ERP implementations, where you often see crosswise standardization as well. This is one of the important causes for the lack of agility of many ERP implementations (Govers 2003).

In applying the operating models described in the previous section, you should also keep this in mind. In particular the Business Modularity and Dynamic Venturing stages highly depend on this type of structure and variability. The danger of over-standardizing at the Optimized Core stage should be avoided.

Example: AgiSurance claim handling process
Our example company AgiSurance has decided that each insurance claim has to go through four steps: registration of the claim, acceptance (e.g. check for completeness), valuation (to assess the amount to be paid), and payment. If AgiSurance would use crosswise standardization, it would use a single Valuate step, for example, which has to incorporate knowledge of all the different insurance products that the company sells, and uses large and complicated decision trees (or other means) to compute the valuation results in individual cases. This would not be a good idea.

3.2.4 Model-Based Development

Another important architectural approach that can provide more system agility is the use of *models* and model-driven tools to facilitate the development and change process. This improves both the innovation and the execution capabilities of the enterprise, because it shortens the path from 'business idea' to 'business execution' and it improves the business insight in the operational reality as well.

Importantly, we do not want to force IT-oriented models onto business stakeholders, but rather advocate the use of domain-specific concepts and languages to capture and communicate relevant business knowledge. These models can then be targeted to suitable IT infrastructure, either by transforming them to technology-oriented models or software code, or even by directly interpreting and executing these models. Such a model-based architecture facilitates the rapid development and deployment of new business services.

In Chap. 4, we will describe this model-based development approach in much more detail. If we compare such an approach to writing software code in languages like Java, PHP, C or Cobol, we see that much of this code is only intended as scaffolding to deliver the required functionality, but does not add business value itself. Writing code is also more error-prone, simply because there are more opportunities and places for mistakes. Moreover, a lot of the same code is written over and over again, compounding these problems even further.

Thus, model-based approaches can be both faster and more efficient in developing services. They are not without their own challenges, however. The up-front investment needed may be considerable, and a clear business case is needed to show when this will pay off, as we have already argued in Sect. 3.2 (see also Fig. 11). These approaches may also require extensive retraining of employees, both at

the business and the IT side of you organization. In Chaps. 4 and 8, we will go deeper into these challenges and ways to overcome them.

Example: AgiSurance's future model-based architecture
In insurances, the parameters of individual policies change quite often. In its IT infrastructure, AgiSurance has therefore chosen to move towards a business rule engine solution, which should support the latter type of changes and provide a high level of flexibility. In Fig. 14, we see an abstracted view of AgiSurance's vision on its future IT landscape, in which several engines for the interpretation of models play a central role.

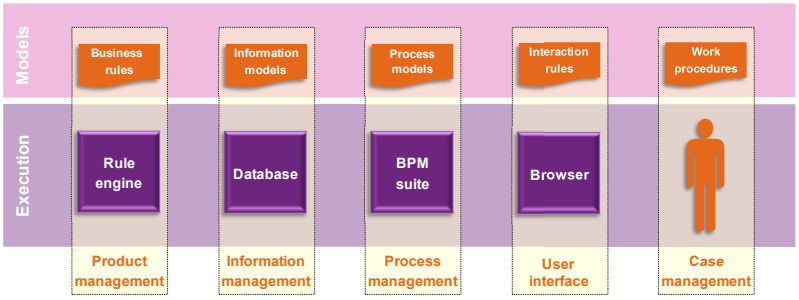

Fig. 14. AgiSurance's model-based architecture vision.

3.3 Architecture Processes in an Agile Context

To create a truly agile enterprise, you should design both your systems and your development processes with change in mind. This also holds for architecture products and processes. Thus, architecture should focus on the backbone of the enterprise and on guarding its essence. Moreover, architecture is invaluable in helping to take risky or high-impact design decisions.

An architecture will itself evolve over time. Architectural artefacts will therefore only have a temporary status. Architectures change because the environment changes and new technological opportunities arise, and because of new insights as to what is essential to the business. A good architecture process ensures that the various artefacts remain relevant and up-to-date, and avoids unnecessary waste.

3.3.1 A Risk-Driven Approach

Next to the organization's own strategy and maturity level, there are other important reasons for managing and controlling agility. In particular risk manage-

ment and compliance with external laws and regulations may limit the freedom of organizations. Risk management policies may imply that the costs and effects of changes are thoroughly investigated before a solution or change may be developed and deployed. In an agile process, the available resources and the delivery schedule are usually fixed, but the functionality is flexible and delivered in order of the priorities given by business stakeholders. Laws and regulations are not so flexible; they must be fully implemented at the date set by lawmakers or regulatory bodies, so the schedule and requirements are fixed.

Next to the operational and financial risks addressed by laws, regulations, and company policies, there is also the risk in the development process: taking wrong design decisions may be very costly. A common adage for architecture in an agile context is that it should be 'just in time, just enough' (e.g. Wagter et al. 2005). You should defer committing to a design choice until you absolutely need to, thereby increasing flexibility and raising your chances of success. The question then becomes: when do you need to take which architectural decisions? In particular, this concerns high-risk decisions, i.e., those that are both difficult and have a potentially high impact.

Taking a decision too early is dangerous, because you may need to do a lot of rework if that decision turns out to be wrong. Although some agile proponents say that it is easy to refactor your software (see the next section), it may not always be possible to easily turn back on an earlier decision. The aforementioned database structure may be an example of such a decision. Other examples are the use of existing infrastructure, performance or quality criteria, or the extensibility of your data model. Sometimes it may therefore be smart to have an analysis phase before starting an agile, iterative project. In that phase, you investigate the important risks, technical and otherwise, and you define the essence of the architecture. You can then use this as a project start architecture for the agile project, giving it sufficient guidance to avoid these risks.

There is also a lurking danger that the team starts with the simple stuff and postpones more difficult requirements and design decisions. For example, a Dutch social security institution saw its development of a new information system for a complicated new law fail, because (among other causes) it started with implementing the 'happy flow', tacked on some more difficult cases as exceptions to this flow, added the even more difficult cases, and so on, until the business process designs they used as the foundation for system development became completely unmanageable. This is a common hazard of incremental development processes. In prioritizing requirements, the common '80-20' rule ('let's focus on the 80% most common cases') is dangerous: perhaps these first 80% are easy to cover, but the 20% difficult cases may require a costly redesign of your solution.

A good heuristic in this respect is to take a *risk-driven approach*, where you address the most critical risks early in the project life cycle. If the necessary input for making a risky decision is not available yet, further analysis and investigation is needed. Perhaps you can build a so-called 'spike solution', a common technique in XP, to try out aspects of a design in more detail. If this is not possible and a de-

cision is too risky to take right now, you should postpone it until the necessary input is available and the risk of making a wrong choice is low.

3.3.2 Refactoring and Technical Debt

A very important technique to cope with changing architectures and designs in incremental development is *refactoring*. In software development, this is defined as a 'disciplined technique for restructuring an existing body of code, altering its internal structure without changing its external behavior' (Fowler 1999). By applying specific semantics-preserving transformations, you can change software code (or other designs, e.g. business processes, data models, business rules or even hardware designs) to accommodate new requirements, without altering its behaviour for existing cases. Before something is refactored, a solid suite of tests is created to ensure that the refactored result performs the same as the original.

Refactoring serves two main purposes: it improves maintainability and it increases extensibility. You should continuously improve the architecture, designs, models and code. If they start to 'smell', for example if new requirements seem progressively hard to implement, if the models are difficult to understand or explain, or if implementations nearly duplicate other code, it may already be too late.

In the previous example, if the social security institution had realized that their processes were gradually turning into a mess, it would have refactored them. For example, they could have separated the decision-making rules on client benefits from the generic workflow processes, keeping the latter stable and managing the complexity of the social security laws separately and explicitly.

Unfortunately, not every piece of design or implementation is amenable to refactoring. For example, if the architecture incorporates a legacy system or employs a commercial-off-the-shelf component, these may be fixed and unchangeable. Decisions on the use of such fixed elements are hence high-risk, and should be investigated early and thoroughly, as explained in the previous section.

Closely related is the role of architecture in avoiding what is sometimes called 'design debt' or 'technical debt'. Often, you need to take a temporary shortcut in your design, leading to an increase in complexity or a decrease in quality that you should resolve at some time in the future. Cunningham first drew the comparison between complexity and debt in a 1992 experience report (Cunningham 1992):

> Shipping first time code is like going into debt. A little debt speeds development so long as it is paid back promptly with a rewrite... The danger occurs when the debt is not repaid. Every minute spent on not-quite-right code counts as interest on that debt. Entire engineering organizations can be brought to a stand-still under the debt load of an unconsolidated implementation, object-oriented or otherwise.

Architecture is an important instrument to avoid that this debt gets out of hand. By explicitly providing the boundaries of the design space, it helps in providing a

'debt ceiling'. Temporarily, projects may be absolved from adhering to the architecture, but eventually their results must be brought back into the fold.

This is an important practice of the Dynamic Architecture (DyA) method (Wagter et al. 2005), for example, which has an explicitly controlled process for development without architecture. This process includes a management letter that stipulates how this deviation from the architecture is going to be resolved, for example by limiting the lifetime of the project result and/or starting development of a long-term solution in parallel.

3.3.3 An Agile Architecture Process

The process of enterprise architecting as described by Op 't Land et al. (2009) consists of three main activities:

1. *Create*: shaping the design of the desired situation, to address the goals and purposes as formulated by the enterprise, in cooperation with relevant stakeholders and within applicable constraints of time and resources.
2. *Apply*: using the architecture as a steering instrument to guide the development and evolution of the enterprise.
3. *Maintain*: keeping the architecture up to date and relevant, by monitoring the enterprise and its environment and responding accordingly.

This EA process may seem a bit linear and at odds with the interactive and iterative way of working in an agile environment. You may get the impression that the architects first do all the thinking, and then an agile team is only needed for implementing their ideas. However, this is not how things operate in an agile organization. These architecture activities are not strictly ordered, nor do they always precede the more detailed design and implementation activities. Rather, these three streams should run continuously and in parallel with development, keeping the architecture relevant and up-to-date at all times and interacting closely with the various development teams.

However, these are merely the architecture activities. How can we combine these with all the other work going on in an agile organization and its development processes for services? First of all, agile and other iterative projects do not assume the enterprise architecture as a given. In a changeable context, the architecture cannot be too detailed for the distant future, and must be amenable to adaptation if the need arises. This makes the aforementioned maintenance activities crucial in an agile context, and a constant awareness of the drivers for change is essential. The cycle between creation, use and maintenance of the architecture is also relatively short; enterprise architectures should not be detailed five-year plans, but provide the general direction in which an organization wants to move.

Second, each individual project will have its own activities for creating, applying and maintaining (parts of) the enterprise architecture, and of course for their

own project-level architectures. It is a good agile practice that architecture largely emerges from projects, and is not just invented from the top down. Only those aspects that really need to be decided at an enterprise-wide level (see also Sect. 3.2), should be given to the project at the start, for example in the form of a Project Start Architecture (Wagter et al. 2005). Conversely, the project-level architectures are an important source of information for the enterprise architecture.

This also holds for organizing the architecture work. An agile team is largely self-sufficient in its choice of methods and tools. However, for architecture products to be reusable in other contexts, a certain standardization of their contents will be needed, for example in the use of models and description languages. Moreover, if parts of the system's infrastructure rely on a specific model-based approach (see also Chap. 4), then this constrains the architecture as well. Individual projects will also uncover their own best practices for architectural (and other) work, and sharing these with other teams requires some level of organization beyond the project scale. In Chap. 6, we will go deeper into the various scales and cycles at which an agile way of working operates.