A Systematic Method for Software Architecture Design

FRITZ SOLMS, University of Pretoria, Private Bag X20, 0028 Hatfield, Republic of South Africa
LOEK CLEOPHAS, Stellenbosch University, Private Bag X1, 7602 Matieland, Republic of South Africa

This paper presents the Systematic Method for Architecture Design (SyMAD) which extends the Attribute Driven Design (ADD) method by (i) enforcing, at any level of granularity, a separation between application design addressing functional requirements and architecture design addressing non-functional requirements, (ii) making it explicit that non-functional requirements at any particular level of granularity may give rise to both functional and non-functional requirements at the next lower level of granularity and that these are, in turn, addressed through application and architecture design at that lower level, (iii) enforcing that at any level of granularity the infrastructure of any component is constrained by a single structural pattern, (iv) introducing concepts and constraints within which application components are to be realized, and (v) explicitly including the identification of reuse candidates and the customization of reused architectural components.

The separation of concerns enforced by SyMAD allows the architecture and application designs of any component to evolve independently. This leads to improved reuse of both architectural and application components. A case study is presented in order to illustrate the method and to expose the differences between SyMAD and other software architecture design approaches.

1. INTRODUCTION

Catastrophic failures of large software projects are often caused by the failure of the software architecture to address the non-functional requirements (NFRs) [Fowler 2003]. Re-architecting is generally very expensive and difficult since it may impact much of the developed code and may require investment in different technologies and frameworks [Khadka et al. 2013; Solms 2015].

One way of reducing architectural risk of software systems is to base the software architecture on a reference architecture. A reference architecture (RA) is a template architecture which aims to address the NFRs for a particular software domain [Lloyd and Galambos 1999]. Examples of such RAs include AUTOSAR [Kindel and Friedrich 2009] and Java-EE which address common architectural concerns for typical automotive and enterprise systems respectively. However, there are many domains for which there are no proven RAs. Furthermore, whilst using a RA may reduce architectural risk, it also reduces the opportunity for innovation.

A need for software architecture design thus remains. Various software architecture design methods have been introduced to guide software architects and to reduce the risk of software architecture failure. However, many of the existing methods do not cleanly separate between application design—addressing functional requirements (FRs)—and software architecture design—addressing NFRs—despite the benefits of such a separation.

Model-driven development approaches like OMG’s Model Driven Architecture (MDA) [Frankel 2003] do envision technology-neutral design of application functionality. Here application functionality refers to the externally visible functionality addressing the FRs of system users. In these approaches one incrementally refines a requirements model; first by introducing technology neutral design artifacts—yielding a Platform Independent Model (PIM)—and subsequently with technology and target architecture information—yielding a Platform Specific Model (PSM). However, current MDA implementations typically hard-code the software architecture and technologies into the model transformations used to generate a PSM from the PIM [Hou et al. 2006; Ameller and Franch 2010; Edwards and Gruner 2013]. This approach makes it hard to reason about and modify the software architecture. In contrast, to make such reasoning and modification easier, we want to have the architectural model as a first class entity,
next to the technology-neutral application design model (the PIM). We therefore want to have repeatable engineering processes for both technology neutral application design and for application functionality neutral software architecture design.

We do have methods for technology neutral application design [Almeida et al. 2003; Almeida et al. 2005; Solms and Loubser 2009; Solms et al. 2011]. In this paper, we introduce the Systematic Method for Architecture Design (SyMAD), which aims to provide a repeatable engineering process for designing a software architecture model that is decoupled from application components. Here an application component refers to a system component concerned with addressing the primary functional requirements for the system. The envisaged benefits of such an approach include simplification due to separation of concerns, the ability of software architecture and application design to evolve independently and increased reuse of both architectural and application components.

2. MOTIVATION AND PROBLEM STATEMENT

Even though software architecture from the perspective taken in this paper, is concerned with the NFRs for the system, it still needs to provide functionality to implement tactics chosen to address these NFRs. For example, connection pooling could be one of the tactics chosen to address performance and scalability requirements. Even though these are NFRs, the software architecture still needs to provide the functionality to retrieve a connection from the connection pool and to return it to the pool. Note that no user specified corresponding FRs but that they arose from choosing specific tactics to address NFRs. To this end Poort and de With [Poort and de With 2004] distinguish between primary FRs which are requirements for functionality required by the user and secondary FRs induced by NFRs. From the perspective taken in this paper, application design is thus concerned with addressing primary FRs whilst software architecture design is concerned with addressing NFRs and consequently the secondary FRs induced by architectural decisions made to address NFRs.

Separating architecture design and application design provides a range of benefits [Frankel 2003; Wagelaar and Jonckers 2005], including:

(1) A separation of concerns, which results in a simplification of both the architecture-neutral application design and the architecture design.

(2) The same application design can be realized within different architectures. This reduces the cost of providing application functionality while addressing different NFRs (e.g., different integration, scalability requirements, performance, or security requirements).

(3) Symmetrically, different software applications can be realized (and potentially even deployed) within the same software architecture.

As mentioned in the introduction, MDA does envision technology-neutral design of application functionality, but current MDA implementations typically hard-code the architecture and technologies into the model transformations used to get from PIM to PSM. This was done, for example, in a tool [Edwards and Gruner 2013] for generating Java-EE implementation mappings of an architecture and technology neutral PIM based on the Use-Case, Responsibility Driven Analysis and Design method (UR-DAD) [Solms and Loubser 2009; Solms et al. 2011]. This exemplifies how current MDA implementations do not treat architectural models as first class entities.

In contrast, we want to have the Architectural Model (AM) not only separated from the technology-neutral application design model (the PIM), but as an explicit, first class entity itself. Eventually, we want to develop a domain specific language for specifying processable AMs, such that we are able to generate a software artifact using both
A Systematic Method for Software Architecture Design

Fig. 1. A model driven approach generating a software system with both a technology and architecture independent application model (PIM) and an architecture model as inputs. Note that both application and architecture analysis and design are done across levels of granularity.

a PIM and an AM as input. A processable AM is an architecture model which can be processed by a software system. Figure 1 depicts our approach.

For the process from FRs to an explicit and processable technology-neutral application design model (the PIM), a method such as URDAD [Solms and Loubser 2009] can be used.

For the non-functional side, we need a repeatable engineering process to obtain an architectural design, as well as a language for representing explicit, processable architectural design models.

A systematic method for software architecture design, and the models resulting from its application, should satisfy a number of requirements, in line with the above definition of software architecture, standard practices and in order to support MDA:

1. They should distinguish between architectural and application components.
2. They should be based on a component decomposition.
3. For any architectural component one should explicitly specify the architectural tactics which are to be used to address the quality requirements for that component. Here an architectural component refers to a component of the software architecture specification addressing the NFRs for the software system.
4. One should constrain the infrastructure of every architectural component through the selection of structural patterns.
5. Levels of granularity should be effectively managed such that, at any level of granularity, the software infrastructure is constrained by a single structural pattern. If multiple structural patterns are used at a particular level of granularity, each pattern will constrain the infrastructure for a domain (sub-set) of the components.

ACM Transactions on Software Engineering and Methodology, Vol. ?, No. ?, Article ??, Pub. date: August 2015.
for that level of granularity. This would leave the infrastructure between such do-

mains un-specified. Instead, modeling each of these domains as a separate compo-
nent projects out a new level of granularity for which the infrastructure can be once
again constrained by a structural pattern.

(6) For every architectural component which hosts application components, the con-
cepts and constraints within which the application components are to be designed
should be specified.

(7) The resulting architectural model should be explicit.

(8) The resulting architectural model should be processable.

In the current paper, we focus on the method and process, and hence the latter two
requirements fall outside our current focus.

3. RELATED WORK

We consider existing work in software architecture design methods, to see whether
and how they address the requirements we listed in the previous section. The expo-
sition will highlight that no existing method satisfies all of those requirements. This
motivates the introduction of our software architecture design method in the following
section.

Various software architecture design methods have been introduced to guide soft-
ware architects and to reduce the risk of software architecture failure. Some meth-
ods assume that software architecture is a high-level abstraction of a software sys-
tem [Zhang and Goddard 2005; Ran 2001; Fowler 2003]. This assumption is inconsis-
tent with the widely accepted premise that software architecture needs to address the
NFRs [Alebrahim et al. 2011], as it is not feasible to address the NFRs for a system
whilst remaining within a high-level abstraction of the system. In other methods, the
architecture is specified across levels of granularity [Bachmann and Bass 2001; Bass
et al. 2002; Alebrahim et al. 2011]. However, most of these methods do not cleanly sepa-
rate between application design—addressing primary FRs—and software architecture
design—addressing NFRs. In fact, most methods require as input both the primary
FRs as well as the NFRs.

The current approaches taken in software architecture design methods can be
largely categorised into the following, not necessarily mutually exclusive, groups:

(1) variations of attribute-driven design (ADD) which are based on a component
decomposition of a software system, specifying component interfaces and architec-
tural patterns and tactics across levels of granularity,

(2) aspect-oriented (AO) architecture design methods which enrich a normal
component-connector based architecture with aspects addressing cross-cutting con-
cerns around NFRs,

(3) model-driven (MD) architecture design methods which start with a require-
ments model which is incrementally enriched with design decisions and technology
information,

(4) problem-frame (PF) based approaches where functional problem frames are
enriched with quality requirements, and

(5) decision-oriented (DO) design methods which use decision trees and archi-
tectural decision models to guide software architects in the process of designing a
software architecture.

All of the current software architecture design methods discussed include at least
aspects of the high-level application design as part of the software architecture design.
3.1. Attribute-Driven Design

Attribute-Driven Design (ADD) [Bachmann and Bass 2001; Bass et al. 2002] is one of the more widely adopted software architecture design methods. The method is extensible and has been widely used as a base approach for a number of refined methods including the model-driven approach of Perovich et al. [Perovich et al. 2009].

ADD provides a high-level software architecture design method which can be used to guide experienced software architects in designing a conceptual software architecture based on quality attribute requirements. The inputs to the method are the FRs in the form of use cases and a set of prioritized quality requirements in the form of testable quality attribute scenarios [Bachmann et al. 2005; Wojcik et al. 2006]. The latter are specified as quantified stimulus-artifact-response scenarios with associated response measures.

ADD is based on a recursive decomposition of the software system into components, starting with the system as a whole (see Figure 3). Each iteration commences with the selection of the component to decompose: the quality attribute impact on the software architecture is analyzed to generate a priority-impact mapping from which the architectural drivers are chosen based on both priority and impact. In the choose design concepts step, software architects choose architectural patterns and tactics to satisfy the architectural drivers and make acceptable trade-off decisions. During the instantiate architectural components step one identifies architectural responsibilities from both the chosen tactics and the FRs and assigns these responsibilities to architectural components. During the interface definition step of ADD the services contracts for the components are specified. This includes the interface specification for the component with the services offered by the components; the inputs, outputs, pre- and post-conditions for each service, and the quality requirements.

![Fig. 3. A UML activity diagram depicting the ADD process.](image)

ADD bases the instantiation of “architectural” components on both chosen tactics and FRs and the resultant architecture specification thus contains both architectural components addressing NFRs and application components addressing FRs and any new use case which is added to the system would require additions to the software architecture itself. The resultant software architecture can thus not be reused for different applications. Furthermore, although ADD does allow for component reuse, it does not provide any explicit guidance around such reuse.

3.2. Model-driven software architecture design

[Matinlassi 2005] proposed Quality-Driven Software Architecture Model Transformations (QAMTs) as a basis for tool support to software architects. These tools are aimed at guiding software architects on pattern selection and model transformation.
Perovich et al. [Perovich et al. 2009] have developed an MDE-based specialization of ADD [Bachmann et al. 2005]. In this approach architectural decisions are encoded within model transformations and models are incrementally enriched through model transformations which are selected to address specific architectural concerns.

Tekinerdogan et al. [Tekinerdogan et al. 2007] point out that many non-functional concerns can be viewed as cross-cutting concerns which can be addressed by aspects. The authors show that aspects can be added to a model by applying corresponding concern transformations.

A major disadvantage of these methods is that they do not yield a separate architecture model which focuses purely on the NFRs. Furthermore, architectural decisions like the choice of architectural tactics or patterns are generally not directly encoded within these models, making it hard to reason about the software architecture. In cases where application design is done in an architecture and technology-neutral way, the software architecture is commonly encoded within model transformations [Edwards and Gruner 2013].

### 3.3. Enriching functional designs with tactics addressing quality requirements

One of the approaches taken is to enrich an application design addressing FRs based on use cases with interceptors addressing quality requirements [de Bruin and van Vliet 2003]. Within this approach interceptors are used to facilitate both pre- and post-refinement which typically involves the application of architectural tactics addressing quality concerns.

Schmidt et al [Schmidt and Wentzlaff 2006] have introduced an approach which is based on an enrichment of problem frames with quality attribute requirements. It has been refined by Alebrahim et al [Alebrahim et al. 2011]. The approach effectively adds architectural design decisions, addressing quality requirements, to an application design, addressing FRs. The method starts with Problem Diagrams which capture the FRs and which are then annotated with quality requirements—resulting in Quality Problem Diagrams. Design alternatives are chosen by selecting and combining architectural styles. FRs are then decomposed, leading to splitting of problem diagrams; while quality requirements are decomposed into more fine-grained ones. This is followed by the definition of an initial architecture by defining the style components (e.g. the layers of a layered architecture). So-called solution mechanisms (tactics) are then selected to address the refined quality requirements, resulting in Concretized Quality Problem Diagrams. These are further refined and concretized into an implementation architecture of connectors and components (some of which implement solution mechanisms addressing quality requirements). Unfortunately, this approach tightly couples architecture design and application design.

### 3.4. Aspect-oriented software architecture design methods

The approach of enriching functional designs with plugins or interceptors addressing quality requirements naturally leads to the field of aspect-oriented software architectures [Tong et al. 2011; Pinto et al. 2011; Pinto et al. 2012]. These approaches are attractive as many NFRs can be viewed as cross-cutting concerns, i.e. concerns which cut across components.

In Aspect-Oriented Software Architecture Design (AOSAD) [Jing et al. 2008; Tong et al. 2011] one concurrently designs a component-connector based software architecture and a set of aspects, i.e. of components addressing the cross-cutting concerns. One then identifies the point-cuts which are used to determine the join points at which the aspects are applied. AOSAD thus provides an aspect-oriented enrichment of a traditional component-connector based software architecture design.
Examples of NFRs which can be partially or fully handled within aspects include security (authentication, encryption, . . . ) and scalability (load balancing). There are, however, architectural concerns which are not readily modeled as cross-cutting concerns. For example, providing a persistence environment or a process execution environment are more naturally modeled as architectural components. In addition, AOSAD does not differentiate between application and architectural components. It also does not include the specification of concepts and constraints for application components. However, AO software architecture design methods provide one of the most attractive and practical ways to specify architectural tactics addressing cross-cutting concerns.

3.5. Decision-oriented approaches to software architecture design

One of the recent trends within the software architecture community is that of viewing architectural design decisions as first-class entities of an architectural description [Perry and Wolf 1992; Bosch 2004; Tyree and Akerman 2005; Harrison et al. 2007]. Within these approaches, feature models are commonly used to show architectural optionality and the architectural decisions made, i.e. design alternatives are expressed through optionality nodes in a feature tree and design decisions are represented by decision nodes. One of the aims of decision-oriented approaches is to retain the decision rationale within architecture descriptions and simplify the process of making trade-off decisions [Gilson and Englebert 2011]. Whilst these methods are useful to guide architects through choosing between alternative options around architectural decisions, and are able to retain the rationale behind architectural decisions, the method typically remains at a relatively high level. Furthermore the process of documenting software architecture decisions has been found to be quite tedious with limited pay-off [Zimmermann et al. 2008].

Zimmermann et al. [Zimmermann et al. 2008] combine architectural decision models with pattern languages. They propose an architectural pattern- and decision-based design method (ArchPad) which aims to simplify the decision making process around selecting architectural patterns. On the one side, this approach exposes the pattern alternatives to software architects, simplifying the selection process. On the other hand, the approach simplifies the process of documenting architectural decisions as these are based on architectural patterns which have been specified through a pattern language.

3.6. Separating architectural from application functionality

Poort and de With [Poort and de With 2004] proposed a Non-Functional Decomposition (NFD) method which separates primary FRs from supplementary requirements which include NFRs, implementation requirements (which correspond to the architectural constraints in this paper) and secondary FRs. Secondary FRs are strictly speaking not requirements for the system, but requirements which arise due to making certain architectural decisions. The authors thus take a similar approach to what is taken in this paper, separating application design addressing (primary) FRs from software architecture design addressing NFRs and architectural constraints with secondary FRs arising from selecting certain tactics to address NFRs. In order to formulate software architecture design as an optimization problem, they introduce the concept of conflicts in supplementary requirements which are not caused by conflicting requirements but arise from limitations of the solution domain. In particular these conflicts arise from having selected specific architectural tactics for their positive impact on some quality attributes whilst having a negative impacting on others — this is illustrated in Table I. The optimization process they propose is a two-phase optimization process which first aims to partition the system such as to remove conflicts in secondary requirements, isolating components or modules for which such conflicts cannot be removed through
partitioning. For the latter they perform a second optimization phase which aims to select structural solutions (patterns) and functional solutions (tactics) such as to minimize the trade-off decisions to be made. Note that the components arising from NFD typically need to address both primary FRs as well as supplementary requirements. There is thus no separation between architectural component addressing NFRs and application components addressing (primary) FRs.

3.7. Architecture design methods and agile development

A lot of work has been done on integrating software architecture design methods and practices within higher-level software development methods in general and agile methods in particular [Nord and Tomayko 2006; Madison 2010; Miyachi 2011; Poort 2014]. This study focuses solely on a software architecture design method, not on how to integrate software architecture design within higher level software development methods.

3.8. Shortcomings of existing methods

Although the discussion of related work has been in terms of a rather coarse grouping of architecture design methods, none of the methods in the groups satisfy all of the requirements we listed for a software architecture design method. In particular, none of the methods explicitly specify the concepts and constraints within which application components are to be specified nor do any enforce a clean separation between architectural and application components. Also, none of the methods fix the level of granularity by enforcing that, at any level of granularity, only a single pattern may be used to constrain the infrastructure. This motivates the introduction of our software architecture design method SyMAD in Section 4. ADD however satisfies many of the requirements we posited, and its flexibility and extendibility allow us to define SyMAD largely as a refinement of ADD.

4. THE SYMAD METHOD

SyMAD extends the high-level ADD process depicted in Figure 3. As in ADD one selects first the system as a whole as the context component, i.e. as the component whose software architecture is to be designed. Similar to ADD, a requirements phase is followed by a design phase which includes selecting patterns and tactics as well as identifying architectural responsibilities and assigning them to the next lower level components before repeating the process for one of those lower level components. SyMAD (illustrated in Figure 4) refines ADD by (1) adding explicit support for reuse, (2) enforcing that the infrastructure at any level of granularity is fixed by a single architectural pattern, thereby fixing the level of granularity, (3) explicitly distinguishing between architectural and application components and adding (4) support for the software architecture specifying concepts and constraints for application components. Note that SyMAD focuses solely on a system perspective of software architecture and that it does not aim to address any of the many other responsibilities which a software architect needs to address like that of interfacing with strategic management of the organization to align the software architecture with the overall organizational architecture and vision.

The SyMAD process, illustrated in Figure 4, starts with selecting the software architecture for the system as a whole as the architectural component for the first level of granularity. After having specified the architectural requirements for this component one checks for reuse, i.e. whether a specification and/or an implementation of an architectural component satisfying the architectural requirements is available or not. If a suitable architectural component is available, it is selected and potentially configured to address the quality requirements. Otherwise one designs the software architecture for the required component. This is done by selecting tactics to address quality require-
A Systematic Method for Software Architecture Design

Fig. 4. Overview of the SyMAD process resulting in a Responsibility View (RV), a Quality Requirements View (QVR), a Constraints View (CV), a Tactics View (TV), a Responsibility Allocation View (RAV), a Structural View (SV), and an Application Concepts and Constraints View (ACCV).

ments, distributing the architectural responsibilities across lower level architectural components, and designing the infrastructure and, if the component hosts application components, the concepts and constraints within which application functionality is to be developed. The process is then repeated for the lower-level architectural components until the software architecture can be assembled from available components.

The architectural requirements as well as the design decisions are currently captured within IEEE 42010 [Emery and Hilliard 2009] compliant views, with the semantics of each view specified in a view point. The concrete syntax currently makes use of language artifacts from both the Unified Modeling Language (UML) as well as the UML extensions introduced by AO-ADL [Pinto et al. 2011]. Future work will consider the refinement of the semantics required for SyMAD, a rigorous assessment of
ADLs for their support of the required semantics and the formalization of a concrete syntax.

4.1. Architectural requirements for selected component
The architectural requirements include the specification of architectural responsibilities, quality requirements, and architectural constraints.

4.1.1. Architectural responsibilities. An architectural responsibility may arise from the need of addressing infrastructural concerns, i.e. concerns other than that of providing certain application functionality. Typical examples of architectural responsibilities are those of providing access and integration channels for humans and/or between systems, providing an execution environment (e.g. application server, rules engine, business process execution engine, ...), within which application logic is deployed and executed, managing system resources (e.g. thread management, memory management, connection management, ...), providing a persistence infrastructure, and performing specific activities to address quality requirements (e.g. authorization for security, caching for performance, or load balancing for scalability).

Since these architectural responsibilities will be assigned to lower level architectural components, the activity of identifying them fixes the next level of granularity. To this end one needs to check whether multiple architectural responsibilities cannot be abstracted into a single, higher level responsibility—i.e. that one does not skip levels of granularity.

4.1.2. Quality requirements. The core architectural requirements are the quality requirements. A typical list of quality requirements includes performance, scalability, reliability, flexibility, accessibility/integrability, security, auditability, maintainability, cost and usability. The specification of these quality requirements should be such that they are testable. Many of these quality requirements can be quantitatively bounded. For example, one could specify lower bounds on the performance overheads which the architecture may introduce.

The quality requirements need to be prioritized to enable the software architecture team to make appropriate quality attribute trade-off decisions.

4.1.3. Architectural constraints. The client may, optionally, specify a set of architectural constraints which the resultant architecture for the component must adhere to. Examples of such constraints are technologies (programming languages, frameworks, integration technologies, ...) which must be or may not be used, and hardware infrastructure within which the software system must be deployed.

4.1.4. Requirements propagation. When traversing to the next lower level of granularity, one needs to assess which of the quality requirements need to be propagated to the next level of granularity, which can be relaxed and which need to be refined. For example, some security requirements may have been fully addressed at the previous level of granularity and may no longer apply to the lower level architectural components whilst certain scalability requirements need to be propagated and certain performance requirements need to be refined for the lower level components. Similarly the software architecture constraints are typically propagated downwards across levels of granularity.

4.2. Checking for architectural component reuse
Once the requirements for an architectural component are known, one checks for reuse opportunities, i.e. whether there is already an architectural component specified which largely fulfills the architectural requirements. Examples of architectural components which are commonly re-used include application servers, databases, connectors and
adapters to other systems, authentication or authorization frameworks and encryptors.

The re-use can be at the design level or at the concrete (implemented) component level. At the design level, there may be one or more reference architectures which are able to address the quality requirements, architectural constraints, and also some or all of the architectural responsibilities. There may also be implemented architectural components which address the requirements. Examples are frameworks which implement reference architectures (e.g. ServiceMix implementing a SOA), frameworks which do not (e.g. Python Django), or lower level components which fully address the architectural requirements (e.g. a particular database or a service request broker). Such components could be sourced from the open source community, from vendors, or from other internal projects.

SyMAD requires that, even at the first level of granularity, one checks for reuse. It might be that there is already a complete software architecture available which largely or fully addresses the NFRs for the system. This is not uncommon. For example, in ongoing enterprise systems development one often deploys new functionality and often even new applications within an existing software architecture which addresses the NFRs. Alternatively, one might be able to customize a reference architecture or framework to satisfy the architectural constraints and quality requirements and address a subset of the architectural responsibilities.

For example, an insurer might already have an enterprise systems architecture which hosts some of its enterprise applications, but which also satisfies the architectural requirements for some new application. In this case the architecture design can be short-cut after verifying that an existing software architecture satisfies all architectural requirements.

In order to be able to reuse an architectural component it must satisfy the architectural constraints and quality requirements and either address the architectural responsibilities or have the option to plug in lower level components which do. If all architectural responsibilities are addressed by the architectural component being reused, no lower levels of granularity need to be designed.

When there are multiple architectural components which meet the requirements, one needs to choose between them. This choice can be guided by the prioritized list of quality requirements, as well as the degree to which each of these are satisfied in the respective architectural components. One of the advantages of the strict separation of concerns of application and software architecture design is that it is quite likely that software architecture components which are independent of application functionality can be reused across different software systems. Conversely different application components could be reused across clients which require the same application functionality within very different software architectures addressing different quality and integration requirements.

4.3. Design of architectural component
The architecture design phase for each level of granularity (see Figure 4) includes

- the specification of architectural tactics to concretely address quality requirements,
- the assignment of architectural responsibilities to architectural components,
- specifying the infrastructure between architectural components (usually by specifying an architectural pattern for that level of granularity),
- in case the architectural component hosts application components, the specification of concepts, constraints and tactics within which the application functionality is to be specified.
4.3.1. Selecting architectural tactics to address quality requirements. Once the infrastructure for the architectural component has been specified, the tactics through which the quality requirements are concretely addressed are selected. The method used to select the architectural tactics used to address the quality requirements is not specified by SyMAD. SyMAD can be viewed as being based on the template method pattern [Gamma et al. 1995], i.e. as a high-level process where the details of how the individual steps within the process are performed are left to the method application. One would typically base the tactics selection on insights, methods, and tools provided by one of the many studies which focus on how architectural tactics can be selected based on quality requirements and architectural trade-off decisions [Poort and de With 2004; Kim et al. 2009; Keuler and Webel 2009; Harrison and Avgeriou 2010; Mirakhorli et al. 2013]. It is understood that selecting an architectural tactic to address a quality requirement may negatively impact on other quality attributes. Table I illustrates how different quality attributes are positively and negatively affected by the selection of an architectural tactic. Note that whilst the table is loosely based on the above studies and on insights obtained from software architecture analysis projects, the table is purely illustrative, i.e. it is neither authoritative nor exhaustive.

Most of the architectural tactics can be applied as aspects, i.e. through a form of interception [Pinto et al. 2012]. For example, thread pooling, caching, connection pooling, scheduling, . . . , encryption, integrity checking, . . . , can all be implemented through aspects. Others can be implemented by introducing additional architectural components addressing additional architectural responsibilities (e.g. resource monitoring and interface/contract repository) or through choosing specific infrastructural patterns (e.g. grids and clusters).

4.3.2. Responsibility allocation. We fixed the next lower level of granularity when we identified architectural responsibilities during the requirements phase for this level of granularity (see Section 4.1.1). In addition we added the responsibilities of having to implement the selected architectural tactics. Here we allocate these responsibilities to lower level components, i.e. we require components which address said responsibilities. The detailed architectural requirements for these components will be elicited and specified as we zoom into each such component.

4.3.3. Constraining infrastructure. SyMAD requires that the infrastructure of any architectural component at any level of granularity is constrained by a single architectural pattern which is best aligned with the most important quality requirement for that component. If more than a single pattern is required for a component, then the component can be decomposed into separate regions (sub-components), each with a structure constrained by the respective pattern. The infrastructure for connecting these different regions covered by the different patterns would still have to be specified by yet another pattern. That would be the (single) pattern for the component itself and the other patterns would be the patterns governing the infrastructure of sub-components of that component. So whilst we are not disputing the benefit of using multiple patterns within a component, we argue that this can be used to decompose the component into sub-components, the structure of each being governed by a separate pattern and which are tied together by a higher level pattern which is the pattern for the component which was decomposed.

As with tactics selection (see section 4.3.1), SyMAD does not prescribe the method to be used to select the architectural pattern used to constrain the infrastructure of the component. One would typically base a pattern selection on insights and tools provided by one of the many studies which focus on how architectural patterns and tactics can be selected based on quality requirements and architectural trade-off decisions [Harrison and Avgeriou 2007; Zimmermann et al. 2008; Harrison and Avgeriou 2010]. It
### Table I. Impact of different tactics on quality attributes. A solid upwards/downwards pointing triangle represents a strong positive/negative impact on a quality attribute and an empty triangle represents a moderate effect on the quality attribute. If a tactic has no symbol assigned for a quality attribute, that tactic has no significant effect on that quality attribute.

<table>
<thead>
<tr>
<th>Goal</th>
<th>Tactic</th>
<th>Performance</th>
<th>Reliability</th>
<th>Security</th>
<th>Auditability</th>
<th>Integrability</th>
<th>Flexibility</th>
<th>Reusability</th>
<th>Maintainability</th>
<th>Testability</th>
<th>Affordability</th>
<th>Usability</th>
<th>Deployability</th>
</tr>
</thead>
<tbody>
<tr>
<td>Resource reuse</td>
<td>Thread pooling</td>
<td>△</td>
<td>▲</td>
<td>▽</td>
<td>▲</td>
<td>△</td>
<td>▲</td>
<td>△</td>
<td>△</td>
<td>△</td>
<td>△</td>
<td>△</td>
<td>△</td>
</tr>
<tr>
<td></td>
<td>Caching</td>
<td>△</td>
<td>▲</td>
<td>▽</td>
<td>▲</td>
<td>△</td>
<td>▲</td>
<td>△</td>
<td>△</td>
<td>△</td>
<td>△</td>
<td>△</td>
<td>△</td>
</tr>
<tr>
<td></td>
<td>Connection pooling</td>
<td>△</td>
<td>▲</td>
<td>▽</td>
<td>▲</td>
<td>△</td>
<td>▲</td>
<td>△</td>
<td>△</td>
<td>△</td>
<td>△</td>
<td>△</td>
<td>△</td>
</tr>
<tr>
<td></td>
<td>Virtualization</td>
<td>△</td>
<td>▲</td>
<td>▽</td>
<td>▲</td>
<td>△</td>
<td>▲</td>
<td>△</td>
<td>△</td>
<td>△</td>
<td>△</td>
<td>△</td>
<td>△</td>
</tr>
<tr>
<td>Spread load across time</td>
<td>Queueing</td>
<td>△</td>
<td>▲</td>
<td>▽</td>
<td>▲</td>
<td>△</td>
<td>▲</td>
<td>△</td>
<td>△</td>
<td>△</td>
<td>△</td>
<td>△</td>
<td>△</td>
</tr>
<tr>
<td></td>
<td>Scheduling</td>
<td>△</td>
<td>▲</td>
<td>▽</td>
<td>▲</td>
<td>△</td>
<td>▲</td>
<td>△</td>
<td>△</td>
<td>△</td>
<td>△</td>
<td>△</td>
<td>△</td>
</tr>
<tr>
<td></td>
<td>Preprocessing</td>
<td>△</td>
<td>▲</td>
<td>▽</td>
<td>▲</td>
<td>△</td>
<td>▲</td>
<td>△</td>
<td>△</td>
<td>△</td>
<td>△</td>
<td>△</td>
<td>△</td>
</tr>
<tr>
<td>Spread load across resources</td>
<td>Grids</td>
<td>△</td>
<td>▲</td>
<td>▽</td>
<td>▲</td>
<td>△</td>
<td>▲</td>
<td>△</td>
<td>△</td>
<td>△</td>
<td>△</td>
<td>△</td>
<td>△</td>
</tr>
<tr>
<td></td>
<td>Clusters</td>
<td>△</td>
<td>▲</td>
<td>▽</td>
<td>▲</td>
<td>△</td>
<td>▲</td>
<td>△</td>
<td>△</td>
<td>△</td>
<td>△</td>
<td>△</td>
<td>△</td>
</tr>
<tr>
<td></td>
<td>Load balancing</td>
<td>△</td>
<td>▲</td>
<td>▽</td>
<td>▲</td>
<td>△</td>
<td>▲</td>
<td>△</td>
<td>△</td>
<td>△</td>
<td>△</td>
<td>△</td>
<td>△</td>
</tr>
<tr>
<td></td>
<td>Paging</td>
<td>△</td>
<td>▲</td>
<td>▽</td>
<td>▲</td>
<td>△</td>
<td>▲</td>
<td>△</td>
<td>△</td>
<td>△</td>
<td>△</td>
<td>△</td>
<td>△</td>
</tr>
<tr>
<td>Reduce communications load</td>
<td>Compression</td>
<td>△</td>
<td>▲</td>
<td>▽</td>
<td>▲</td>
<td>△</td>
<td>▲</td>
<td>△</td>
<td>△</td>
<td>△</td>
<td>△</td>
<td>△</td>
<td>△</td>
</tr>
<tr>
<td></td>
<td>Batch</td>
<td>△</td>
<td>▲</td>
<td>▽</td>
<td>▲</td>
<td>△</td>
<td>▲</td>
<td>△</td>
<td>△</td>
<td>△</td>
<td>△</td>
<td>△</td>
<td>△</td>
</tr>
<tr>
<td></td>
<td>Course grained services</td>
<td>△</td>
<td>▲</td>
<td>▽</td>
<td>▲</td>
<td>△</td>
<td>▲</td>
<td>△</td>
<td>△</td>
<td>△</td>
<td>△</td>
<td>△</td>
<td>△</td>
</tr>
<tr>
<td>Optimization</td>
<td>Dynamic code optimization</td>
<td>△</td>
<td>▲</td>
<td>▽</td>
<td>▲</td>
<td>△</td>
<td>▲</td>
<td>△</td>
<td>△</td>
<td>△</td>
<td>△</td>
<td>△</td>
<td>△</td>
</tr>
<tr>
<td></td>
<td>Dynamic query optimization</td>
<td>△</td>
<td>▲</td>
<td>▽</td>
<td>▲</td>
<td>△</td>
<td>▲</td>
<td>△</td>
<td>△</td>
<td>△</td>
<td>△</td>
<td>△</td>
<td>△</td>
</tr>
<tr>
<td></td>
<td>Database indexing</td>
<td>△</td>
<td>▲</td>
<td>▽</td>
<td>▲</td>
<td>△</td>
<td>▲</td>
<td>△</td>
<td>△</td>
<td>△</td>
<td>△</td>
<td>△</td>
<td>△</td>
</tr>
<tr>
<td>Fault detection</td>
<td>Resource monitoring</td>
<td>△</td>
<td>▲</td>
<td>▽</td>
<td>▲</td>
<td>△</td>
<td>▲</td>
<td>△</td>
<td>△</td>
<td>△</td>
<td>△</td>
<td>△</td>
<td>△</td>
</tr>
<tr>
<td></td>
<td>Checkpoint evaluation</td>
<td>△</td>
<td>▲</td>
<td>▽</td>
<td>▲</td>
<td>△</td>
<td>▲</td>
<td>△</td>
<td>△</td>
<td>△</td>
<td>△</td>
<td>△</td>
<td>△</td>
</tr>
<tr>
<td></td>
<td>Deadlock detection</td>
<td>△</td>
<td>▲</td>
<td>▽</td>
<td>▲</td>
<td>△</td>
<td>▲</td>
<td>△</td>
<td>△</td>
<td>△</td>
<td>△</td>
<td>△</td>
<td>△</td>
</tr>
<tr>
<td>Fault prevention</td>
<td>Active redundancy</td>
<td>△</td>
<td>▲</td>
<td>▽</td>
<td>▲</td>
<td>△</td>
<td>▲</td>
<td>△</td>
<td>△</td>
<td>△</td>
<td>△</td>
<td>△</td>
<td>△</td>
</tr>
<tr>
<td></td>
<td>Removal of faulty components</td>
<td>△</td>
<td>▲</td>
<td>▽</td>
<td>▲</td>
<td>△</td>
<td>▲</td>
<td>△</td>
<td>△</td>
<td>△</td>
<td>△</td>
<td>△</td>
<td>△</td>
</tr>
<tr>
<td></td>
<td>Persistent messaging</td>
<td>△</td>
<td>▲</td>
<td>▽</td>
<td>▲</td>
<td>△</td>
<td>▲</td>
<td>△</td>
<td>△</td>
<td>△</td>
<td>△</td>
<td>△</td>
<td>△</td>
</tr>
<tr>
<td>Fault recovery</td>
<td>Passive redundancy</td>
<td>△</td>
<td>▲</td>
<td>▽</td>
<td>▲</td>
<td>△</td>
<td>▲</td>
<td>△</td>
<td>△</td>
<td>△</td>
<td>△</td>
<td>△</td>
<td>△</td>
</tr>
<tr>
<td></td>
<td>Checkpoint rollback</td>
<td>△</td>
<td>▲</td>
<td>▽</td>
<td>▲</td>
<td>△</td>
<td>▲</td>
<td>△</td>
<td>△</td>
<td>△</td>
<td>△</td>
<td>△</td>
<td>△</td>
</tr>
<tr>
<td></td>
<td>Reset to valid state</td>
<td>△</td>
<td>▲</td>
<td>▽</td>
<td>▲</td>
<td>△</td>
<td>▲</td>
<td>△</td>
<td>△</td>
<td>△</td>
<td>△</td>
<td>△</td>
<td>△</td>
</tr>
<tr>
<td>Extensibility tactics</td>
<td>Interception points/layers</td>
<td>△</td>
<td>▲</td>
<td>▽</td>
<td>▲</td>
<td>△</td>
<td>▲</td>
<td>△</td>
<td>△</td>
<td>△</td>
<td>△</td>
<td>△</td>
<td>△</td>
</tr>
<tr>
<td></td>
<td>Support aspects</td>
<td>△</td>
<td>▲</td>
<td>▽</td>
<td>▲</td>
<td>△</td>
<td>▲</td>
<td>△</td>
<td>△</td>
<td>△</td>
<td>△</td>
<td>△</td>
<td>△</td>
</tr>
<tr>
<td></td>
<td>Contracts based decoupling</td>
<td>△</td>
<td>▲</td>
<td>▽</td>
<td>▲</td>
<td>△</td>
<td>▲</td>
<td>△</td>
<td>△</td>
<td>△</td>
<td>△</td>
<td>△</td>
<td>△</td>
</tr>
<tr>
<td>Service/</td>
<td>Naming service</td>
<td>△</td>
<td>▲</td>
<td>▽</td>
<td>▲</td>
<td>△</td>
<td>▲</td>
<td>△</td>
<td>△</td>
<td>△</td>
<td>△</td>
<td>△</td>
<td>△</td>
</tr>
<tr>
<td>component publication</td>
<td>Trader service</td>
<td>△</td>
<td>▲</td>
<td>▽</td>
<td>▲</td>
<td>△</td>
<td>▲</td>
<td>△</td>
<td>△</td>
<td>△</td>
<td>△</td>
<td>△</td>
<td>△</td>
</tr>
<tr>
<td></td>
<td>Interface/contract repository</td>
<td>△</td>
<td>▲</td>
<td>▽</td>
<td>▲</td>
<td>△</td>
<td>▲</td>
<td>△</td>
<td>△</td>
<td>△</td>
<td>△</td>
<td>△</td>
<td>△</td>
</tr>
<tr>
<td>Security tactics</td>
<td>Encryption</td>
<td>△</td>
<td>▲</td>
<td>▽</td>
<td>▲</td>
<td>△</td>
<td>▲</td>
<td>△</td>
<td>△</td>
<td>△</td>
<td>△</td>
<td>△</td>
<td>△</td>
</tr>
<tr>
<td></td>
<td>Message integrity</td>
<td>△</td>
<td>▲</td>
<td>▽</td>
<td>▲</td>
<td>△</td>
<td>▲</td>
<td>△</td>
<td>△</td>
<td>△</td>
<td>△</td>
<td>△</td>
<td>△</td>
</tr>
<tr>
<td></td>
<td>Authorization</td>
<td>△</td>
<td>▲</td>
<td>▽</td>
<td>▲</td>
<td>△</td>
<td>▲</td>
<td>△</td>
<td>△</td>
<td>△</td>
<td>△</td>
<td>△</td>
<td>△</td>
</tr>
<tr>
<td></td>
<td>Authentication</td>
<td>△</td>
<td>▲</td>
<td>▽</td>
<td>▲</td>
<td>△</td>
<td>▲</td>
<td>△</td>
<td>△</td>
<td>△</td>
<td>△</td>
<td>△</td>
<td>△</td>
</tr>
<tr>
<td></td>
<td>Request/connection dropping</td>
<td>△</td>
<td>▲</td>
<td>▽</td>
<td>▲</td>
<td>△</td>
<td>▲</td>
<td>△</td>
<td>△</td>
<td>△</td>
<td>△</td>
<td>△</td>
<td>△</td>
</tr>
<tr>
<td></td>
<td>Minimize accessibility</td>
<td>△</td>
<td>▲</td>
<td>▽</td>
<td>▲</td>
<td>△</td>
<td>▲</td>
<td>△</td>
<td>△</td>
<td>△</td>
<td>△</td>
<td>△</td>
<td>△</td>
</tr>
<tr>
<td></td>
<td>Signature scanning</td>
<td>△</td>
<td>▲</td>
<td>▽</td>
<td>▲</td>
<td>△</td>
<td>▲</td>
<td>△</td>
<td>△</td>
<td>△</td>
<td>△</td>
<td>△</td>
<td>△</td>
</tr>
</tbody>
</table>

is understood that selecting an architectural pattern to constrain the infrastructure of a component based on the quality requirements for that component may make it more difficult to achieve other quality attributes. Table II shows that different quality attributes are positively and negatively affected by the selection of an architectural pattern. Note that just like Table I, whilst this table is loosely based on the above studies and on insights obtained from software architecture analysis projects, it is purely illustrative, i.e. it is neither authoritative nor exhaustive.

ACM Transactions on Software Engineering and Methodology, Vol. ?, No. ?, Article ??, Pub. date: August 2015.
Table II. Impact of architectural patterns on quality attributes.

<table>
<thead>
<tr>
<th>Goal</th>
<th>Pattern</th>
<th>Performance</th>
<th>Scalability</th>
<th>Reliability</th>
<th>Security</th>
<th>Availability</th>
<th>Integrability</th>
<th>Flexibility</th>
<th>Reusability</th>
<th>Maintainability</th>
<th>Testability</th>
<th>Affordability</th>
<th>Usability</th>
<th>Deployability</th>
</tr>
</thead>
<tbody>
<tr>
<td>Layering</td>
<td>▽</td>
<td>△</td>
<td>△</td>
<td>△</td>
<td>△</td>
<td>△</td>
<td>△</td>
<td>△</td>
<td>△</td>
<td>△</td>
<td>△</td>
<td>△</td>
<td>△</td>
<td>△</td>
</tr>
<tr>
<td>Pipes and filters</td>
<td>△</td>
<td>△</td>
<td>△</td>
<td>△</td>
<td>△</td>
<td>△</td>
<td>△</td>
<td>△</td>
<td>△</td>
<td>△</td>
<td>△</td>
<td>△</td>
<td>△</td>
<td>△</td>
</tr>
<tr>
<td>Microkernel</td>
<td>▼</td>
<td>▲</td>
<td>▲</td>
<td>▲</td>
<td>▲</td>
<td>▲</td>
<td>▲</td>
<td>▲</td>
<td>▲</td>
<td>▲</td>
<td>▲</td>
<td>▲</td>
<td>▲</td>
<td>▲</td>
</tr>
<tr>
<td>Blackboard</td>
<td>▼</td>
<td>▲</td>
<td>▲</td>
<td>▲</td>
<td>▲</td>
<td>▲</td>
<td>▲</td>
<td>▲</td>
<td>▲</td>
<td>▲</td>
<td>▲</td>
<td>▲</td>
<td>▲</td>
<td>▲</td>
</tr>
<tr>
<td>Component Graph</td>
<td>▼</td>
<td>△</td>
<td>△</td>
<td>△</td>
<td>△</td>
<td>△</td>
<td>△</td>
<td>△</td>
<td>△</td>
<td>△</td>
<td>△</td>
<td>△</td>
<td>△</td>
<td>△</td>
</tr>
</tbody>
</table>

For example, if security and flexible access channels are the central quality requirements for the component whose architecture is to be designed, one may want to constrain the infrastructure through layering. On the other hand, if integrability is central, one may prefer to choose the microkernel pattern as the structural pattern for that component.

4.3.4. Introducing concepts, constraints and tactics for application components. Some architectural components are responsible for hosting application components. Whilst the architecture specification does not include the specification of the application components themselves, it often does determine the concepts and constraints within which these application components are to be specified.

For example, SOAs introduce the concept of a service which needs to be stateless, discoverable and self-healing. Furthermore, in most SOA-based architectures the pipes and filters pattern is used to assemble higher level services (application functionality) from lower level services. Here the services represent the filters and message queues represent the pipes. In addition, SOA introduces the constraint that service contracts are published. In order to improve flexibility and maintainability, SOA introduces runtime service provider lookup and versioning as tactics.

The Java-EE reference architecture, on the other hand, provides the concepts of stateful and stateless application components and higher level application components are developed using the controller pattern. Processes are thus not assembled through pipes and filters, but every component is a controller which uses lower level components to realize its services. Java-EE introduces the application constraints of having to publish component contracts in the form of local and/or remote interfaces. Additionally Java-EE introduces a range of tactics for application components including interception and dependency injection.

Similarly, AUTOSAR, rules engines and complex event processing frameworks all introduce concepts, constraints, and tactics for application components.

Commonly used concepts for components providing application functionality include

— objects, which maintain state and provide methods which may alter the environment,
— services, which may alter the environment, but do not maintain state across service invocations, and
— functions, which only get an input and produce an output without maintaining state and without affecting the environment.

The FRs for such application components can be encapsulated in a contract, i.e. a component, service, or function contract. Application functionality operates on application data. The latter may be contained within objects providing application function-
ality, but is often factored out into class-based data objects (e.g. entities in Java-EE, Microsoft.Net, . . . ), self-describing data objects (e.g. XML or JSON based data objects as used in web services and service-oriented architectures) or semantic data structures [Peckham and Maryanski 1988; Tan and Maciejowski 1991].

Communication between application components is commonly constrained through architectural patterns. Note that in the preceding section we applied architectural patterns to constrain the infrastructure between architectural components. Examples of patterns which are also used for constraining the infrastructure between application components are pipes and filters (e.g. in service-oriented architectures), controller (e.g. in Java-EE), microkernel (e.g. in AUTOSAR and service-oriented architectures), and blackboard (e.g. in space-based architectures).

In order to address quality requirements around the application functionality itself, a software architecture may specify a number of tactics which are applied to application components deployed within it. A software architecture specification may provide the infrastructure for application component interception (as is, for example, provided by Java-EE) in order to improve flexibility and extensibility, or may provide application component versioning in order to improve maintainability (i.e. to have an architecture within which different versions of the same application component are concurrently active).

Table III. Application component concepts, constraints and tactics and their impact on quality attributes.

<table>
<thead>
<tr>
<th>Application component concept</th>
<th>Quality Attributes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Application object concept</td>
<td>Performance</td>
</tr>
<tr>
<td>Service</td>
<td>Scalability</td>
</tr>
<tr>
<td>Function</td>
<td>Reliability</td>
</tr>
<tr>
<td>Contract</td>
<td>Auditable</td>
</tr>
<tr>
<td>Class-based data object</td>
<td>Integrity</td>
</tr>
<tr>
<td>Self-describing data objects</td>
<td>Flexibility</td>
</tr>
<tr>
<td>Semantic data</td>
<td>Reusability</td>
</tr>
<tr>
<td>Application component</td>
<td>Maintainability</td>
</tr>
<tr>
<td>Infrastructure constraints</td>
<td>Testability</td>
</tr>
<tr>
<td>Tactics for application</td>
<td>Affordability</td>
</tr>
<tr>
<td>components</td>
<td>Usability</td>
</tr>
<tr>
<td>Component interception</td>
<td>Deployability</td>
</tr>
<tr>
<td>Component versioning</td>
<td></td>
</tr>
<tr>
<td>Run-time service provider</td>
<td></td>
</tr>
<tr>
<td>lookup</td>
<td></td>
</tr>
<tr>
<td>Dependency injection</td>
<td></td>
</tr>
<tr>
<td>Exception/error communication</td>
<td></td>
</tr>
<tr>
<td>Self-testing components</td>
<td></td>
</tr>
<tr>
<td>Adding course grained services</td>
<td></td>
</tr>
</tbody>
</table>

Table III shows the impact of application component concepts, constraints, and tactics on quality attributes. Whilst the table is not exhaustive, such a table can provide useful guidance in making the appropriate quality attribute trade-offs. For example, using functions (without side effects) improves performance, auditability, flexibility, reusability and maintainability. However, developing within a purely functional paradigm is more difficult than using services which may modify the environment or
data objects which may maintain state. Further guidance on making appropriate quality attribute trade-off decisions can be obtained from [Mirakhorli et al. 2013] and [Har- rison and Avgeriou 2010].

4.4. Transition to lower level of granularity
Following the SyMAD method, one performs a component decomposition of the software architecture. Once the architecture of a component at a level of granularity has been designed, we select one of the lower levels of architectural components as our new context component. Whilst that component was introduced to address NFRs for the higher level component, it will, once again, have FRs and NFRs which are addressed through application and software architecture design respectively. Note, however, that these lower level FRs arise from addressing higher-level NFRs. As mentioned before, Poort and de With [Poort and de With 2004] use the term “secondary FRs” for lower level FRs arising from addressing higher-level NFRs. Thus, whilst the primary FRs for the component are addressed through application design, this still falls under the scope of software architecture from the perspective of the design of the higher level component.

For example, in order to address the scalability requirements for some system, say a computer game, we might introduce a thread pool. The thread pool does not address any of the FRs of the game and hence is purely an architectural component of the game—its design thus fully falls within the realm of the software architecture design for the game. Now, for our thread pool we have once again FRs like that of providing a thread as well as NFRs like how quickly a thread must be provided (performance) and how many threads must be managed (scalability). From the perspective of the higher level component (e.g. the game) these lower level FRs are secondary FRs arising from implementing the thread pooling tactic to address the NFRs of the higher level component. However, from the perspective of the lower level component, these are now primary FRs. The architecture of the thread pool must address these NFRs and the application design for the thread pool addresses the secondary FRs. That application design for the thread pool is, however, part of the architecture design of the game.

4.5. Comparison of SyMAD and ADD
SyMAD is based on ADD in the sense that it also is based on a component decomposition approach, applying architectural patterns and tactics to the components in order to satisfy quality requirements. The main difference between SyMAD and ADD is that the former separates the concerns of addressing primary FRs and NFRs across application and software architecture design whilst ADD uses both FRs and NFRs as inputs. The consequence is that the outputs of the two methods are also significantly different in that we generate an application model as well as a software architecture model with the latter being devoid of any application functionality. Other more minor differences include:

- SyMAD enforces that a single structural pattern or style is applied to constrain the infrastructure of a particular component. ADD does not introduce such a constraint.
- As ADD does not differentiate between application and software architecture design, it does not lead to a software architecture model which introduces concepts and constraints within which application components are to be developed.
- SyMAD includes explicit support for reuse of architectural components.
- In ADD the full requirements process for the next lower level of granularity is completed prior to selecting a particular lower level component and doing the design of that component (waterfall effect). SyMAD follows a more agile approach where a particular architectural component is selected and the requirements and design
processes for that component are executed. One of the advantages of this approach is that the component may be architected by a different team with a better understanding of that component’s domain.

— SyMAD supports an aspect-oriented approach to applying architectural tactics to components.

5. CASE STUDY: A SIMPLIFIED BANKING SYSTEM

As a case study we consider the design of a software architecture for a simplified banking system. The case study illustrates

— the identification of architectural responsibilities and the allocation of these responsibilities to architectural components,
— the reuse of architectural components,
— the selection of architectural patterns and strategies,
— how one introduces concepts and constraints for application components,
— the traversal across levels of granularity and the architectural design of lower level components, and
— the interplay between software architecture and application design.

5.1. System overview

The case study subject is a regular banking system used across the African continent by individual and corporate clients for transactional banking, loans and deposits. The banking services need thus be remotely accessible by humans and client systems over, at times, low-bandwidth Internet connections.

In addition to standard banking processes, the system must facilitate the maintenance of customer information and the generation of regulatory reports which need to be fed through to the industry regulator.

5.2. First level of granularity

The software architecture for the system as a whole is selected as the context component for which we need to elicit the architectural requirements, check for reuse and potentially design the software architecture.

5.2.1. Software architecture requirements. The requirements for the first level of granularity include the specification of the high-level architectural responsibilities, the quality requirements, and the architectural constraints for the banking system.

Architectural responsibilities. The first level granularity architectural responsibilities that were identified from the requirements are shown in Figure 5. It includes the responsibilities of providing various access and integration channels as well as the responsibilities of hosting business processes and data, and that of providing reporting services. Note that these are all infrastructural responsibilities which are not specific to the actual banking functionality.

Fig. 5. Architectural responsibilities for the first level of granularity as shown in a Responsibilities View (RV).
**Quality requirements.** Figure 6 shows the quality requirements for the system as a whole. Security, scalability, reliability and auditability are regarded by the client as the most important quality attributes followed by performance and maintainability.

![Quality Requirements View (QRV)](image)

**Architectural constraints.** In order to reduce the long-term maintenance risk the client requires that no third-party architectural elements of the system should be vendor locked, i.e. all elements should be either standards compliant and hence pluggable or available and actively maintained by the open source community.

5.2.2. *Check for reuse.* In SyMAD one already checks for reuse at the first level of granularity. It is not only possible but even quite common that the new application has NFRs similar to other applications developed for the organization and that the new application can be developed for the same software architecture within which the current applications have been developed or one for which some minor architectural modifications were made (e.g. to support another integration channel). Furthermore, there are a number of reference architectures which provide architectural templates for application domains and for most of these reference architectures there are implementing frameworks available which one can use as an architectural component addressing the majority of the NFRs for applications from that domain. These reference architectures generally facilitate the plugging in of different lower level components to address specific architectural requirements for an instance architecture. For example, within the Java-EE reference architecture one can plug in different persistence mappers and different Java Connector Architecture (JCA) providers to address specific persistence and integration requirements.

**Identifying candidate reference architectures and frameworks.** The system is an enterprise system with typical architectural responsibilities and quality requirements for such systems. There are a range of reference architectures and frameworks which are widely used for such systems. Examples of reference architectures for enterprise systems include Java-EE [Fabrice 2013] and SOA [Erl 2005]. For both of these there are a range of implementing frameworks (e.g. RedHat WildFly, IBM Websphere, Apache ServiceMix, and Mule). In addition there are a range of frameworks targeting enterprise systems which are not based on a reference architecture. Enterprise application
frameworks not based on a reference architecture include the *Spring Framework* and *Microsoft.Net* [Gillespie 2010].

In order to reduce the bulk of the analysis we will confine ourselves to the comparative assessment of only three of the above, *Java-EE*, *Microsoft.Net*, and SOA. We need to assess these candidate reference architectures and frameworks for the extent to which they are able to satisfy the architectural constraints and requirements for our banking system. In particular, only reference architectures and frameworks within which it is possible to meet the quality requirements and the architectural constraints can be considered. In addition one needs to assess to which extent the architectural responsibilities are addressed and how one would plug in additional architectural components in order to address any architectural responsibilities not addressed by the chosen architectural component.

*Assess against constraints.* Both reference architectures, *Java-EE* and SOA, are based on a set of community-managed public standards and there exist a range of framework implementations both as proprietary vendor products as well as in the open source community. However, Java-EE is a more detailed and specific standard supporting full portability of applications across Java-EE compliant application servers. The SOA standards generally only guarantee interoperability, not application portability. The *Microsoft.Net* framework, on the other hand, is a framework using Microsoft-specific technologies which are only partially based on public community-managed standards and which have only recently been partially released as open source. At this stage there is still only one viable application server for Enterprise-level *Microsoft.Net* applications which is the one from *Microsoft* itself. Using *Microsoft.Net* would thus result in vendor lock-in (at least in the short to medium term).

*Assess alignment of quality requirements.* We first compare from a quality attributes perspective the two reference architectures, SOA and Java-EE.

As seen from Figure 7, the SOA reference architecture is built around an enterprise services bus (ESB) which represents the microkernel of the microkernel architectural pattern. The primary quality attribute targeted by the reference architecture is *integrability* followed by flexibility (time to market). Integrability is further improved by...
using standard text-based protocols (either SOAP/HTTP or REST/HTTP) and having services contracts published in a public standard (either the Web Services Description Language (WSDL) or the Web Application Description Language (WADL)) service registries. Flexibility and “time-to-market” are one of the strengths of SOA-based software architectures as higher level services are easily “orchestrated” from published lower level services sourced across systems connected to the services bus. Services are stateless and process state is maintained within the messages which are transferred across all services employed in the process specification. Queueing is used to improve reliability and scalability. Scalability is also strengthened by SOA frameworks supporting clustering and thread pooling. Auditability is provided through the ability of logging all messages communicated over the services bus. Performance is the quality attribute which is largely traded off for integrability and flexibility with the tactics chosen to support the latter introducing significant performance overhead.

At the first level of granularity the Java-EE reference architecture is based on the layered architectural pattern with an access/presentation layer containing the component model, and is used with stateless and stateful application components. Communication is done either via native method calls or via an efficient binary protocol.
Component contracts are specified in CORBA's Interface Definition Language (IDL). Process state is commonly maintained within the session and not transferred with messages exchanged between components.

The reference architecture employs a range of tactics to address scalability including general resource pooling—such as thread, object, and connection pooling—, database caching, and clustering. Integrability is supported through supporting CORBA and Web Services standards and through providing a connector architecture within which connectors for non-standards based integration are developed. The reference architecture has good security support with authentication, encryption and role-based authorization at the business logic layer all done through interception. Reliability is largely addressed through clustering, transactions, and support for messaging. Auditable can be implemented through global interceptors which log all exchanged messages. The dominant quality attributes are scalability, reliability, security, and maintainability. Flexibility and integrability have, to some extent, been traded off for these, though Java-EE still addresses these quality requirements by being components-based and by supporting public integration standards as well as a standard infrastructure (JCA) for integration with proprietary technologies.

Table IV shows the result of the verification of the realizability of the quality requirements within architectures based respectively on the SOA and Java-EE reference architectures. Provided clustering and redundancy of all architectural components are enforced, a Java-EE based instance architecture was found to be able to address all quality requirements. On the other hand, basing the architecture of the banking system on the SOA reference architecture would put performance and maintainability quality requirements at risk. Also, there is no way to address entity change logging within SOA, but this quality requirement could be addressed at the lower level component used by services containers to access persistent storage providers.
Assess addressing of architectural responsibilities. In order to address the architectural responsibilities specified in the requirements (see Section 5.2.1), we require that the chosen reference architecture must provide architectural components addressing these responsibilities. Figure 9 depicts the required architectural components. The web broker needs to provide web-page access, generates the service requests from the provided information and renders the return values back onto the web. Asynchronous system integration to humans is usually done via emails. An email adapter needs to render requests and responses originating from the banking system to an email and, for a request, it needs to provide a web link for humans to provide their responses. A web-services broker maps SOAP-based web services requests onto actual method calls and asynchronous system access is done through message queues. The remainder of the required components shown in Figure 9 are self-explanatory.

![Fig. 9. Architectural components required to address architectural responsibilities are modeled using the UML notation for required interfaces.](image)

We need to assess now to what extent the two candidate reference architectures provide specifications for the required components. In the case of SOA, both web services and messaging brokers are provided, but human access needs to be addressed through other components which would be plugged into the reference architecture (e.g. using a JSF-supporting web container or a Django front-end). SOA also has good support for business process execution engines which are plugged into the ESB—the most commonly used engines are BPEL and jBPM engines. However, even though some work has been done on providing SOA with a persistence infrastructure [Krizevnik and Ju-ric 2010], no standard is thus far widely adopted or supported.

![Fig. 10. A UML diagram using realization relationships to specify the concrete components which realize the required components within Java-EE reference architecture.](image)
Java-EE, on the other hand, does provide specifications for most of the required components and implementing frameworks provide implementations for these specifications. Figure 10 shows the mapping of the abstract components onto the components as specified in the Java-EE reference architecture as well as the components which are not provided by the reference architecture. These components are selected as the subjects for the next lower level of granularity where the requirements specification for them is done. We will then either source existing architectural components which meet these requirements or design appropriate architectural components. For the scope of this paper it is sufficient to do the next level granularity for only one of these components, the email adapter. For the remaining components we have included some results from the next level of granularity by showing which components can be sourced and which need to be designed.

Component selection. Both SOA and Java-EE satisfy the architectural constraints, but the quality attribute trade-offs made in Java-EE are better aligned with the quality requirements for the banking system, with Java-EE having advantages on scalability, performance, security and maintainability. SOA is stronger when it comes to integrability, but this is not a central quality requirement for this case study. Furthermore, Java-EE addresses more of the architectural responsibilities. The analysis thus leads us to select the Java-EE reference architecture for the system as a whole.

We could go a step further and also select a particular implementing framework (e.g. RedHat WildFly or IBM Websphere Application Server).

5.2.3. Configuring the architectural component. We have chosen the Java-EE reference architecture for the architectural component of the first level of granularity providing the infrastructure within which the NFRs are addressed and within which application logic will be deployed and executed. The reference architecture provides some freedom around how architectural responsibilities are addressed, how the infrastructure is configured, which architectural tactics are used to address quality requirements, and which concepts and constraints are introduced for application development. These are specified when configuring the reference architecture.

When configuring the re-used architectural component at the current level of granularity, we need to map the architectural responsibilities onto components of the re-used reference architecture and configure the infrastructure, architectural tactics and application concepts and constraints of the re-used component.

Configure architectural tactics. In our chosen reference architecture, some architectural tactics are always used, whilst others are optional. For example, resource pooling (including thread, object and connection pooling) is always applied in Java-EE. However, to address the scalability and reliability requirements, we choose to configure the architecture with clustering and replication to a disaster recovery site.

As mentioned previously, these tactics are often cross cutting concerns which are applied across a subset of the architectural components. Hence they are naturally modeled as aspects. For this reason we choose the AO-ADL [Pinto et al. 2011] to document the application of architectural tactics. In addition to providing connections between architectural components, Figure 11 shows the application of tactics to the system as a whole. In AO-ADL aspectual roles are represented by solid arcs to connectors represented by largish circles (e.g. C1).

Mapping architectural responsibilities onto architectural components. During the process of assessing the Java-EE reference architecture against the potential alternatives, we already performed the mapping of architectural responsibilities onto the components specified in the Java-EE reference architecture (recall Figure 10).
Fig. 11. An AO-ADL diagram showing the architectural tactics applied to the system as a whole. In AO-ADL round circles represent connectors which can be connected to other regular components via UML associations as well as to aspectual roles via lines with arcs. Aspectual roles are represented by interfaces which are ultimately realized through aspectual components.

Configuring the infrastructure. Being a reference architecture, Java-EE provides a template software architecture which needs to be configured. The Java-EE configuration for this level of granularity is shown in Figure 12.

Fig. 12. Configuration of the infrastructure of the Java-EE reference architecture for the banking system.

For our designed software architecture we choose a JSF/Facelets based presentation layer (instead of Servlets/JSP). The reason for this choice is that the JSF life-cycle supports short-cut processing facilitating partial page processing and rendering for lightweight dynamic Ajax based web front-ends. This is particularly important for the bank as the bank has clients who interface with the system over low-bandwidth Internet connections. Furthermore, we constrain message driven beans to be pure messaging
adapters devoid of any application logic, i.e. they demarshall messages received from incoming queues or topics, map them onto requests made to stateless session beans containing the application logic and marshall the return values received from these beans back onto a message which is put onto a response queue. Hence, in our designed architecture the message driven bean based router is part of the access layer, even though it is technically deployed within the EJB application server. The reasons for this constraint are (i) to comply with the architectural design decision that the application logic is specified using (stateless) services as application logic concepts – see Configuring application concepts and constraints below, and (ii) to separate the different technical access channels from the application logic so that the former are cleanly part of the software architecture (devoid of application logic) and the latter are clean application components which can be reused across access channels.

Persistence is done through JPA based persistence (i.e. using an object-relational mapper, object cache and queries and relationships specified within object graphs and not tables). The Email adapter, JCA-based regulator adapter and JPA-based persistence adapters are all part of an infrastructure layer, as depicted in Figure 12.

Configuring application concepts and constraints. Since the banking domain is naturally a services domain which will benefit from services contracts and service reuse, the architectural decision was made that application logic should reside in stateless services and that all domain objects should be represented by entities which hold state but are a devoid of any application (business) logic. This simplifies reuse and testability as no state is maintained within these services across processing separate service requests. This architectural decision can be realized within the Java-EE reference architecture by requiring that all application logic must reside within stateless session beans as opposed to stateful session beans and message-driven beans. The methods of these beans represent services in the above sense which can be conveniently published as SOAP-based and REST-full web services. An added benefit of the above decision is that it leads to more efficient bean passivation and activation and hence to more efficient thread sharing.

5.3. Lower levels of granularity

Some of the architectural components are not specified by Java-EE. In particular, the database, reporting engine and email adapter are not specified by the reference architecture and also not provided by Java-EE based frameworks. We assigned these responsibilities to lower level architectural components, but the requirements, reuse assessment and design of these components is done as we traverse to lower levels of granularity.

Some of the lower level architectural components may be sourced (re-used) and need not be designed. However, we still need to base the reuse selection on architectural requirements for that component. For example, it is unlikely that one will design a new database for a typical enterprise system. However, the choice of the persistence architecture used will depend on, for example, the performance, scalability, flexibility and integrability requirements. Common choices include relational databases, key-value stores, document stores, graph/object stores, .... Concrete designs and realizations of these stores employ different tactics to achieve quality requirements like scalability (e.g. eventual consistency). For a banking system it is likely that integrability requirements will result in a selection of a relational database management system like PostgreSQL or Oracle RDBMS. We thus select/reuse a relational database as architectural component to address the persistence responsibility and do not need to design that component.
The second architectural component of the banking system which is not specified within the Java-EE reference architecture is the reporting engine. We would specify the architectural requirements for the reporting engine and then check for re-use. It is likely that one of the standard reporting engines (e.g. JasperReports or Pentaho [Books LLC 2010]) would fulfill the requirements for the reporting engine. The chosen engine would have to be configured to meet the architecture requirements of the banking system. This process would be similar to that of configuring the Java-EE reference architecture.

Assume that there are, however, no standard email adapters which map service requests which happen to be made to a human being onto an email with a link to a web page which captures the response message. This component would have to be architected and its functionality needs to be designed.

5.4. Email Adapter

The system submits asynchronous messages and requests to external parties in a way which is independent of whether the messages are processed by external systems or humans. The responsibility of the email adapter is to map asynchronous messages and requests onto an email for asynchronous human consumption and, in the case of requests, provide a response capture facility which enables humans to submit a response through a web page whose content is demarshalled, with the demarshalled message being provided asynchronously back to the system.

From the perspective of the banking system, the email adapter is purely an infrastructural/architectural component—it is just an adapter and does not provide any application functionality for banking. However, at this lower level of granularity we have both NFRs and FRs and hence will have to do both architecture and application design for this component. Without going into much detail on the side of application design, this example also illustrates how both software architecture and application design are done across levels of granularity.

5.4.1. Architecture design for the email adapter. The quality attributes of a lower level component need to support the qualities of the architecture at the higher level. Furthermore, the lower level architectural components (e.g. the email adapter) are deployed within the architecture of the parent component (the banking system). For example, the processes of the email adapter will be executed within the process execution engine (the EJB container) of the parent component. In this context, lower level components benefit from the architectural tactics which have been applied to the parent component (e.g. resource pooling in the context of thread and connection pooling).

Software architecture requirements for the email adapter. The software architecture requirements for the lower level component are constrained by the quality requirements and architectural constraints of the parent component.

Figure 13 depicts the architectural responsibilities which need to be addressed by the infrastructure within which the application logic for the email adapter is to be developed.

![Fig. 13. The architectural responsibilities of the email adapter.](Image)

The architecture of the email adapter needs to provide the infrastructure to integrate with the banking system's asynchronous input and output channels (the mes-
sage queues) on the one side and the mail server and human beings on the other side. In particular, it delivers asynchronous messages to humans via email and captures the asynchronous responses from humans via the web. In addition, the email adapter also requires an execution environment within which its processes can be executed.

The quality requirements for the email adapter are directly derived from the quality requirements of the higher level component—the banking system as a whole. To do this, each higher level quality requirement was analyzed for its relevance at this lower level of granularity. Those quality requirements which were relevant were quantified in such a way that the quality requirements for the email adapter support those for the banking system as a whole. The resultant quality requirements for the email adapter are shown in Figure 14.

![Scalability](scalability.png)
![Audibility](audibility.png)
![Security](security.png)

Fig. 14. The quality requirements for the email adapter.

The software architecture of the lower level component must be designed within the architectural constraints of the parent component. Hence no third-party elements of the email adapter may be vendor locked.

**Check for reuse.** Assume we cannot source an architectural component satisfying the requirements for an email adapter. Of course, we may still reuse lower level components within the email adapter—the opportunity for such reuse will be investigated as we traverse to lower levels of granularity.

**Architecture Design.** We thus enter the software architecture design phase illustrated in Figure 4. We start with selecting architectural tactics through which we address the quality requirements for the email adapter. We then allocate the architectural responsibilities identified during the requirements phase to architectural components of the email adapter before designing the infrastructure between these components. Finally we specify the concepts and constraints within which the application logic for the email adapter is to be specified.

Architectural tactics are selected to concretely address the quality requirements specified previously. Table I can be used to assist with the selection of appropriate tactics, i.e. tactics which strengthen the sought quality attribute without having too negative an impact on other required quality attributes.

**Confidentiality** will be achieved through encryption of both the email-based requests and the HTTP responses (using HTTPS).

Some scalability tactics are inherited from the system as a whole, i.e. the email adapter already achieves certain levels of scalability through the clustering and load balancing applied to the banking system as a whole. In addition to those tactics, the email adapter will use thread and connection pooling as managed through the higher level EJB-container as well as caching of the response capture pages as done through the web container. Finally we also use queueing of requests to be sent by the adapter and responses captured by the adapter to further spread load over time and hence improve scalability.

In order to address Reliability, we choose persistent queueing of both request and response messages with message removal from queue not at message delivery, but after acknowledgement of successful processing of the message. The reliability of the
email adapter also benefits from the reliability tactics introduced for the system as a whole, including the clustering, replication, and the usage of a disaster recovery site.

Finally, to address the auditability requirements we perform logging of all messages as they are put onto the request and response queues as well as all emails sent and all captured user responses.

We need to assign the architectural responsibilities listed in the requirements specification (see Section 5.4.1) as well as the responsibilities of realizing the tactics to required architectural components.

Figure 16 shows how the architectural responsibilities are assigned to required architectural components (represented by required interfaces in UML). We also preempt the mapping of those required components onto the aspects of the architecture of the email adapter which are provided by virtue of having chosen the Java-EE reference architecture.
architecture at the previous level of granularity (that mapping would only be done as we go to the next lower level of granularity for the required architectural components). The remaining components will need to either be externally sourced (e.g. an email encryptor) or designed at the lower level of granularity (e.g. the logging interceptor).

5.4.2. Email adapter application design. The email adapter represents an asynchronous requests adapter to humans, i.e. it receives an asynchronous request which is to be processed by a human, generates a web page through which the response for the request can be captured as well as a response adapter which maps the HTTP post message received through that web page onto the response message which is put on a response queue.

For the application design one can use any architecture- and technology-neutral design method. This will introduce application components for the email adapter which need to be deployed in the software architecture of the email adapter which is, in turn, part of the architecture for the banking system. For this example we only sketch the first level granularity application design of the email adapter using the URDAD method (mentioned in Section 1).

Services Contract. Figure 17 depicts the services contract for the first level of granularity for the email adapter.

![Fig. 17. The services contract for the email adapter.](image)

Functional requirements. In URDAD, FRs is taken literally, i.e. as requirements for functions/services. Figure 18 depicts the required functions and the allocation of these to services contracts.

![Fig. 18. The FRs and responsibility allocation for the processRequest use case. FRs are allocated to abstract roles represented by required interfaces.](image)

Process design. During the first-level granularity process design phase for the email adapter we assemble a process across the functions/services specified during the FRs phase. Figure 19 depicts a process design for the email adapter.
5.4.3. Lower levels of granularity. Both the application and software architecture design for the email adapter will have to be taken through lower levels of granularity until all application and architectural components are assembled from available components. At any level of granularity we may have both software architecture design addressing the NFRs for that level of granularity and application design addressing the FRs for that level of granularity.

6. REFLECTIONS

The case study demonstrates (i) the feasibility of separating the concerns of addressing primary FRs and NFRs, leading to a software architecture specification which is devoid of application functionality, (ii) the reuse of architectural components, even at the first level of granularity, to address the bulk of the architectural requirements, augmented by lower level architectural components to address architectural concerns not covered by the higher level component, (iii) how secondary FRs arising from selecting tactics to address NFRs become primary FRs for lower level components. The application design used to address the primary FRs for the lower level component is part of the software architecture design for the higher level component (the system as a whole) as it does not provide any application functionality. At any level of granularity, we have both application and software architecture design. (iv) How one can use a single architectural pattern to specify the infrastructure of any architectural component at any level of granularity, with the infrastructure of each of the sub-components governed by a separate pattern. (v) How the bulk of the architectural tactics chosen to address the quality requirements for the system are addressed by architectural components which we were able to apply as aspects, and the usefulness of AO-ADL to document the software architecture.

Weaknesses and limitations of the method include (i) that the method cannot be applied to evolve an existing architecture where the separation between application and architectural components has not been made, (ii) SyMAD addresses purely system aspects of architecting and not any people or team aspects, and (iii) separating architecture and application design leads to a scenario where some quality requirements (particularly performance) can no longer be absolutely guaranteed. For example, whilst the architecture design may ensure that the infrastructural overheads like
those introduced through marshalling and de-marshalling, load-balancing, encryption and decryption and so on are no more than some specified limit, the algorithms realizing the application functionality might still be designed in an inefficient way incurring unnecessary computational overheads and impacting negatively on the performance quality.

On the other side, the benefits of separating the concerns of application and software architecture design include (i) the simplification introduced by being able to address these concerns independently, (ii) improved reuse as the architectural components are devoid of application functionality and can be reused across different software systems providing different application functionality, and conversely, the application components being reusable in different software architectures addressing different NFRs, and (iii) having a separate software architecture and design models which reasoned about and modified independently.

Outside this study, the SyMAD method has been applied to designing the software architecture for an educational discussion board which was implemented by the final year undergraduate computer science students at the University of Pretoria. Having developed separate software architecture and application design models made it easier to change from an initial architecture design based on the Java-EE reference architecture using a relational database for persistence to a node.js based software architecture with a more dynamic and richer JavaScript based presentation layer and a Mongo DB document store for persistence. Furthermore, we found that we were able to port many of the architectural decisions—including the majority of the architectural patterns and tactics—from the original Java-EE based software architecture specification to the final node.js based software architecture. Examples of tactics which were ported include dependency injection for decoupling and simplified testability, caching for scalability and performance, the inclusion of aspect-oriented frameworks for addressing cross cutting concerns like logging and security. On the negative side we found that the concepts and constraints which we introduced for the Java-EE software architecture did not port well into the asynchronous nature of a node.js based software architecture.

The separation of application and software architecture design has also been applied to a software architecture recovery method, SyMAR [Solms 2015]. It was found that the client did find the resultant architecture description useful and felt that the separation of concerns simplified the identification of architectural concerns and the ability to introduce architectural decisions addressing those concerns [Solms 2015].

7. CONCLUSIONS AND FUTURE WORK

This paper introduced the Systematic Method for Architecture Design (SyMAD). The method's description and the case study exemplifying SyMAD's application show how the method addresses the first six of the eight requirements for such a method, as posited in the introduction. SyMAD enforces a clean separation between software architecture design addressing NFRs and application design addressing FRs, by using component decomposition and distinguishing between architectural and application components, addressing the first and second requirement. For the third requirement, each architectural component has explicitly specified tactics aimed at addressing its quality requirements. Every architectural component has its infrastructure constrained by some structural pattern, and a single such pattern is used per level of granularity of the design, addressing the fourth and fifth requirements. Finally, the sixth requirement is addressed by the use of concepts and constraints to be satisfied by application components to be designed in a particular architectural component.

Additionally the method systematically checks for component reuse and the availability of reference architectures addressing the architectural concerns. The case study
illustrates both the application of the architecture design methodology as well as the differences with other architecture design methods.

Benefits of the approach include the clean separation of concerns between architecture and application design, generating a software architecture within which different applications providing different user functionality can be deployed. When used in conjunction with a technology and architecture neutral application design method like the Use-Case, Responsibility Driven Analysis and Design method (URDAD), application design and architecture design can vary independently. Technology and architecture neutral application designs can be mapped onto implementations within different software architectures satisfying different NFRs.

Eventually we want to be able to generate, from an URDAD application design model and an architecture design model resulting from a SyMAD process, a software system which provides certain application (user) functionality within a software architecture addressing specific NFRs. This would allow different clients to map application functionality onto a software architecture addressing the NFRs.

Future empirical work will assess the adoption rate and challenges of applying SyMAD to the software architecture design of industrial software systems, as well as improved support for capturing an architecture model. To facilitate the latter, we would like to specify the semantics required for a SyMAD architecture model either in the form of a metamodel or as an ontology. This would enable us to assess the extent to which current ADLs (in particular AO-ADL) support the required semantics or could be extended to do so. The resulting language would support the specification of explicit, processable models, thereby satisfying the last two requirements posited in the introduction.

REFERENCES


