A Typology Based Approach to Assign Responsibilities to Software Layers

LEO PRUIJT, HU University of Applied Sciences, Utrecht, Netherlands
WIEBE WIERSEMA, HU University of Applied Sciences, Utrecht, Netherlands
SJAAK BRINKKEMPER, Utrecht University, Utrecht, Netherlands

In software architecture, the Layers pattern is commonly used. When this pattern is applied, the responsibilities of a software system are divided over a number of layers and the dependencies between the layers are limited. This may result in benefits like improved analyzability, reusability and portability of the system. However, many layered architectures are poorly designed and documented. This paper proposes a typology and a related approach to assign responsibilities to software layers. The Typology of Software Layer Responsibility (TSLR) gives an overview of responsibility types in the software of business information systems; it specifies and exemplifies these responsibilities and provides unambiguous naming. A complementary instrument, the Responsibility Trace Table (RTT), provides an overview of the TSLR-responsibilities assigned to the layers of a case-specific layered design. The instruments aid the design, documentation and review of layered software architectures. The application of the TSLR and RTT is demonstrated in three cases.

Keywords: software architecture, layers pattern, layered style, layers, responsibility, typology, classification

1. INTRODUCTION

The Layers pattern, or Layered style, is one of the most common patterns used in software architecture (Clements et al. 2010, Harrison & Avgeriou, 2008). The concept of layering can be traced back to the works by Dijkstra (1968) and Parnas (1972). Buschmann et al. described the Layers pattern extensively (1996). Avgeriou and Zdun (2005) have shown that layers are also described as patterns or styles by many other authors. For a concise definition we cite Larman (2005) who summarized the essential ideas of the Layers pattern as: *"Organize the large-scale logical structure of a system into discrete layers of distinct, related responsibilities, with a clean, cohesive separation of concerns such that the 'lower' layers are low-level and general services, and the higher layers are more application specific. Collaboration and coupling is from higher to lower layers; lower-to-higher layer coupling is avoided".* An example of a layered design, shown in Fig. 1, represents a strict layered design in which the following usage rules are respected: usage relationships are from top to bottom; neither back call nor skip call usage is allowed.

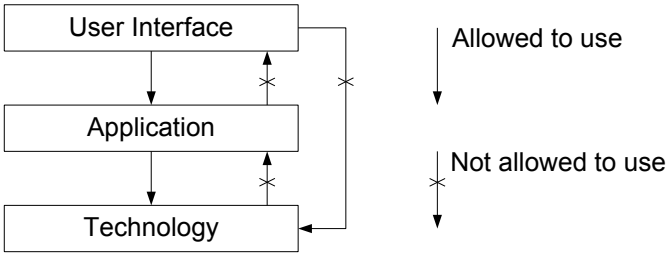

Fig. 1 Example of a strict layered design

Applying the Layers pattern may improve software qualities, like analyzability, reusability and portability of the system, but may also impose liabilities like a lower efficiency, or more rework when a change affects several layers. When a layered view of architecture is drawn up, a number of design decisions must be taken, preferably explicit and documented (Kruchten, Lago, & Vliet, 2006). For example, the following design questions should be answered:

- What software qualities are to be answered by the layered architecture?
- Which layers are distinguished?
- Which types of responsibility are assigned to each layer?
- Which usage relationships between the layers are allowed?

Unfortunately, the layered designs are often poorly defined and many violate the principles for which layers are designed (Clements & Nord 2000, Clements et al. 2010). In practice and in student projects, we encounter many layered designs describing or showing only the layers and the names of the layers,

without a specification of the contents, the communication rules, and a justification. Such an architectural product gives little guidance to the developers in achieving a desired software quality. For instance, another layered model with three layers (Presentation, Domain, and Technology), could represent the same-layered design, as the one shown in Fig.1, but it could be substantially different as well. The names of the layers do not clarify the exact responsibilities of the layers, e.g. where the control of the task is located, or where the status is maintained. Therefore, a specification of the responsibilities of the layers is needed.

We aimed our research on the analysis of the causes of these problems and on the provision of instruments to design layered architectures of high quality. One cause of the problems is described by Clements et al. (2010): "The layered view of architecture, shown with a layer diagram, often is poorly defined, and so often misunderstood". In addition, we hypothesize another cause, namely that the terminology regarding layered designs is not clear and sometimes downright contradictory. A uniform classification is lacking; different authors use varying and sometimes conflicting terms for layers, types of logic and responsibilities. A good example is the popular concept "application logic" or "application layer". Application logic is interpreted substantially diverse by different authors. Larman (2005) describes it as: “handles presentation layer requests; workflow; session state; window/page transition; …”, whereas Erl (2008) defines it as: “an automated implementation of business logic …” Even, the concept of "domain layer" has different meanings in different layered designs.

The starting point of our investigation was the observation that to answer the design question “*Which types of responsibility are assigned to each layer?”,* a uniform classification for the naming and characterization of types of logic and responsibilities in software layers could be useful. This perception resulted in the following research questions, which were leading in our study: 1) What types of responsibilities are distinguished in layered architectures; 2) How can these types of responsibility be named and defined unambiguously; 3) How can a typology of responsibilities be applied in practice?

To answer these research questions, we studied leading literature about software layers to get an overview of common types of responsibilities and the names given to them. Based on this literature, we constructed the Typology of Software Layer Responsibility (TSLR). The TSLR gives an overview of the different types of responsibility, gives them unambiguous names, specifies them and exemplifies them. To enhance the application of the TSLR in practice, we designed the Responsibility Trace Table (RTT). An RTT shows the assignment of TSLR-responsibilities to the layers of a case-specific architecture, without the need for extensive textual descriptions. The TSLR and RTT may be used to design and document a particular layered design and to assess the quality of existing layered designs. Finally, to evaluate and improve the typology and its related trace table, the instruments were reviewed by experts in the field of software architecture, applied in case studies, and used in training courses for bachelor students.

In this paper, we propose our typology in Section 2, and an approach to apply the TSLR for different practical purposes in Section 3. Next, Section 4 illustrates the application of the TSLR and RTT by means of three cases. Thereafter, Section 5 discusses our research approach, the artifacts and the limitations, and Section 6 presents the conclusions and an outlook to future work.

2. PROPOSED TYPOLOGY OF SOFTWARE LAYER RESPONSIBILITY

2.1 Fundamentals of the Typology

Responsibility in the context of software architecture is defined by Clements et al. (2010) as “a general statement about an architecture element and what it is expected to contribute to the architecture. This includes the actions that it performs, the knowledge it maintains, the decisions it makes or the role it plays in achieving the system’s overall quality attributes or functionality”. We adopt this definition and the notion from the same source, that a *layer* (as all software architecture modules) is characterized by its set of responsibilities.

A uniform classification for the naming and characterization of types of responsibilities in software layers should provide an answer to the following requirements:

- The Typology should identify the types of responsibilities distinguished in layered software designs.
- Each responsibility should have a unique and unambiguous name, which expresses the contents well.
- Each responsibility should be described properly and illustrated with an example, if needed.
- The responsibilities should be arranged in a classification schema.

The meta-model of the TSLR, as shown in Fig. 2, is based on the requirements and matches the structure of the proposed typology. The composite pattern (Gamma, Helm, Johnson, & Vissedes, 1995) is used to enable the hierarchical structure. The meta-model allows for possible future extensions in width and depth and may be used to provide tool support. Each *Responsibility* in the typology has a name and description. *CompoundResponsibilities* represent a set of responsibilities and *designCriteria* help to determine which system responsibilities are included in the set, or excluded from the set. *SingularResponsibilities* represent specific responsibilities in our typology, and they are illustrated by *examples*.

For reasons of comprehensibility, *designCriteria* and *examples* are modeled as attributes, although they may contain multiple values.

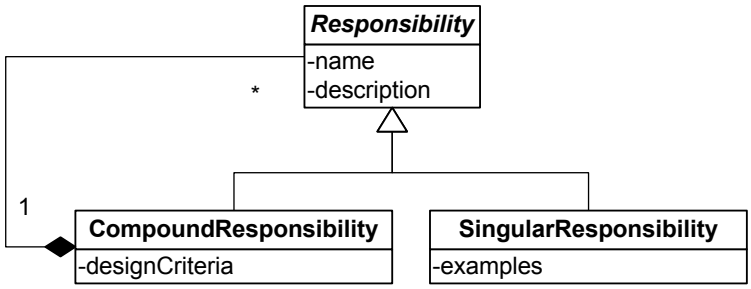

Fig. 2 TSLR meta-model

2.2 Typology of Software Layer Responsibility

The Typology of Software Layer Responsibility (TSLR) provides an overview of the distinct types of responsibility commonly found in the software of business information systems. It identifies and orders responsibilities, gives them unambiguous names and specifies them. The TSLR consists of a classification schema, as shown in Fig. 3, and a textual specification of the responsibilities.

Three abstraction levels of responsibility are distinguished:

(1) *System* represents all the logical responsibilities of the software.

(2) *Type of logic* represents a compound level of responsibility. The TSLR distinguishes five types of logic: Consumer Interface Logic, Task Specific Logic, Domain Generic Logic, Infrastructure Abstraction Logic and Infrastructure Logic.

(3) *Sub-responsibility* represents a singular responsibility. The TSLR distinguishes thirteen sub-responsibilities within the five types of logic.

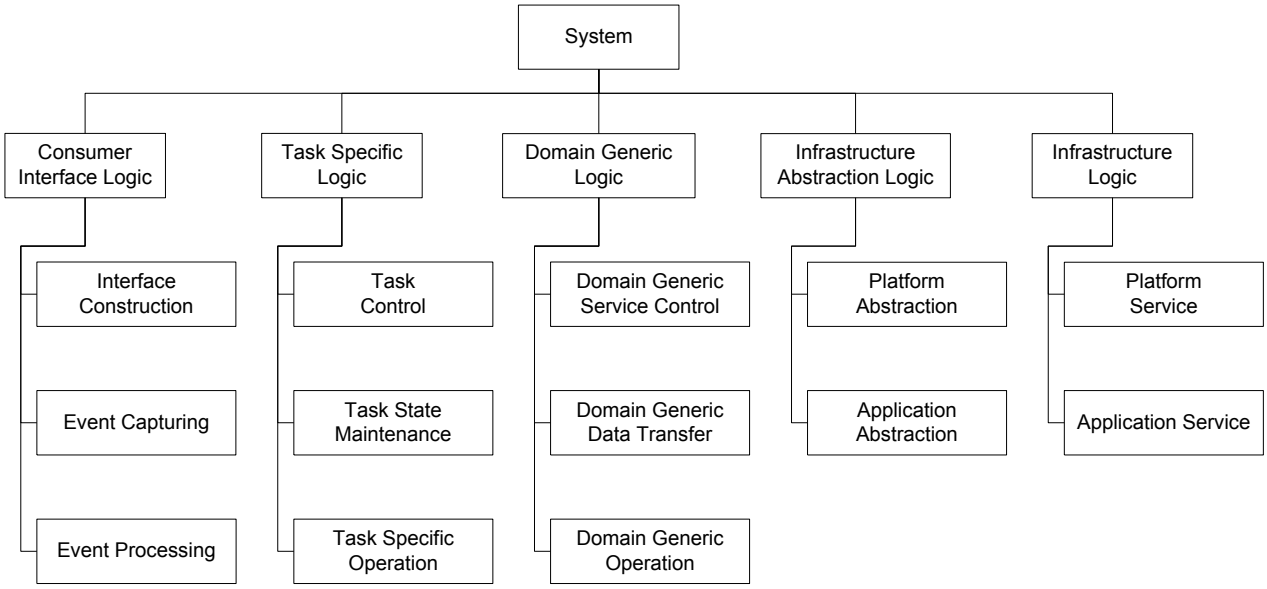

Fig. 3 TSLR classification schema

2.2.1 Scope of the TSLR

The types of logic within the TSLR and their sub-responsibilities are primarily useful in the context of business information systems, since most literature used as basis for the typology is focused on this type of systems. The typology may also be useful for embedded systems, control systems, games, et cetera, but extensions may be necessary.

The TSLR provides an overview of logical responsibilities. Logical means here "free of implementation choices", since the typology is not intended for a specific platform, orientation or deployment strategy. The layered style is a modular style (Clements et al., 2010) and does not focus on the runtime behavior or the allocation of software components.

2.3 Specification of the Responsibilities

In accordance with the meta-model, each responsibility has a name and description. For each type of logic, design criteria are specified. Design criteria help to classify (fragments of) responsibilities of a specific software application, when the TSLR is used in practice. Similarly, sub-responsibilities are illustrated with examples.

2.3.1 Consumer Interface Logic

Description: Consumer Interface Logic is responsible for establishing and maintaining communication with a consumer of a system service in a manner appropriate to the task of the consumer. The consumer can be an end user communicating via a user interface as well as an automated client system communicating via a protocol. Consequently, the interface and the events may have very different forms.

Design criteria: Responsibility is...

- Included, when it is specific to the interface of a task.
- Excluded, when it is reusable across different interfaces of a task.

Table 1 Sub-responsibilities of Consumer Interface Logic

SUB-RESPONSIBILITY	DESCRIPTION	EXAMPLES
Interface Construction	Providing an interface to the consumer with information and/or control appropriate to the task of the consumer.	• GUI building • Report (layout) • Speech interface • Service interface
Event Capturing	Capturing events from the consumer.	• Recognizing input from consumer: data, control, speech • Knowledge about when an event is captured
Event Processing	Processing events from the consumer as far as it concerns Consumer Interface Logic.	• Deciding what to do with the input (data, speech ...) • Format check on input data • Delegation

2.3.2 Task Specific Logic

Description: Task Specific Logic is responsible for the coordination of the task, the maintenance of the task state and the execution of functionality specific to the task. A task is a unit of work, to be performed as a whole, which provides the consumer with a result of value. A task is generally, in terms of Cockburn (1997), at the user-goal level. Task Specific Logic is potentially reusable across different interfaces of the task.

Design criteria: Responsibility is...

- Excluded, when it is specific to a task interface.
- Included, when it is specific to a task.
- Excluded when it is potentially broadly reusable.

Table 2 Sub-responsibilities of Task Specific Logic

SUB-RESPONSIBILITY	DESCRIPTION	EXAMPLES
Task Control	Coordination of the task. Deciding what needs to happen when an event takes place.	• Workflow, orchestration, page flow • Control of task specific sub-responsibilities • Delegation
Task State Maintenance	Tracking and maintaining the task state.	• Tracking which data is selected, inserted, or changed • Transaction state control
Task Specific Operation	Performing actions and transformations exclusive to the task.	• Conversion of data • Task specific constraints • Task specific transformations and computations, like calculating report totals, joining data, sorting data

2.3.3 Domain Generic Logic

Description: Domain Generic Logic is responsible for the coordination and the execution of functionality dealing with concepts, information and rules of the business. Domain Generic Logic has to do purely with the problem domain and is potentially reusable across the tasks.

Design criteria: Responsibility is...

- Excluded, when it is specific to a task.
- Included, when it is specific to the business.
- Excluded when it has knowledge of infrastructure that has to be abstracted.
- Excluded, when it is reusable across different business applications.

Table 3 Sub-responsibilities of Domain Generic Logic

SUB-RESPONSIBILITY	DESCRIPTION	EXAMPLES
Domain Generic Control	Coordination of the activities needed to handle requests.	• Control of domain generic sub-responsibilities • Delegation
Domain Generic Data Transfer	Retrieving and storing data.	• Selecting and sorting data • Storing new or changed data
Domain Generic Operation	Execution of domain generic actions and transformations.	• Generic constraints • Generic transformation rules • Maintaining entity state

2.3.4 Infrastructure Abstraction Logic

Description: Infrastructure Abstraction Logic is responsible for the translation of infrastructure independent requests into requests dependent on the infrastructure. Infrastructure Abstraction Logic is separated from other logics, when needed to meet quality requirements like portability, analyzability, reusability.

Design criteria: Responsibility is...

- Excluded, when it is specific to a domain or task.
- Included, when it has knowledge of infrastructure that has to be abstracted.
- Excluded, when it is part of an infrastructure platform or infrastructure application.

Table 4 Sub-responsibilities of Infrastructure Abstraction Logic

SUB-RESPONSIBILITY	DESCRIPTION	EXAMPLES
Platform Abstraction	Encapsulating functionality dependent on an application platform element.	• Adapter to a specific database • Functionality formatted to make use of a specific object relational mapping framework • Adapter a specific security framework
Application Abstraction	Encapsulating functionality dependent on an infrastructure application.	• Adapter to a specific electronic mail client • Adapter to a specific document editor

2.3.5 Infrastructure Logic

Description: Infrastructure Logic is responsible for broadly reusable functionality, non-specific to the business. It may be bought, but also self-built, e.g. utilities. Since there are a huge number of infrastructural services, the TSLR connects here with the TOGAF Technical Reference Model (TRM) (The Open Group, 2009). The TRM defines and exemplifies the concepts Infrastructure Application and Application Platform. Furthermore, it provides a typology of the services of the Application Platform.

Design criteria: Responsibility is...

Excluded, when it is specific to a business application.

Included, when it is reusable across different applications and/or businesses.

Table 5 Sub-responsibilities of Infrastructure Logic

SUB-RESPONSIBILITY	DESCRIPTION	EXAMPLES
Platform Service	Providing generic application support (by the technology components of hardware and software).	• Data interchange service • Data management service (DBMS, OODBMS, ORB …) • Network service • Operation System Service • Software engineering service (Programming language …) • Security service (Identification, Authentication …)
Application Service	Providing general-purpose business functionality (by Commercial Off-The-Shelf software).	• Electronic mail client • Document editing and presentation • Spreadsheets • Workflow management

2.4 Justification of the TSLR and Related Work

2.4.1 Founding Literature

The responsibilities and types of logic in the TSLR are distilled from leading literature in the field of software architecture and layers. We performed a literature study based on the search strings "software", "layer", "architecture", "responsibility" and their synonyms in various combinations. We found the most valuable sources to be books, well-established in the software architecture community; an experience matching with a systematic literature review conducted by Savolainen and Myllärniemi (2009).

The first category of sources consulted in the course of this investigation describes the basics of the Layers pattern and guidelines to design a layered architecture (e.g., Bass et al. 2012, Buschmann et al. 1996, Clements et al. 2010, Evans 2004, Fowler et al. 2003, Larman 2005, Shaw & D. Garlan 1996). Authors often refer to these sources, when the subject of layers is addressed. Evans (2004), Fowler et al. (2003) and Larman (2005) extensively describe layered designs suitable for business systems. Evans distinguishes four layers, Fowler three layers, while Larman specifies six common layers in an information system's logical architecture. The second category of sources describe a specific layered design, useful within the scope of this study (e.g., Allen & Frost 1997, Gorton 2006, MSDN 2009, Snoeck et al. 2000, The Open Group 2009).The third category of sources discusses layered designs for service oriented architectures (e.g., Erl 2008, Krafzig et al. 2005, Lankhorst 2009, Winter & Fischer 2007). Service layers cannot be compared straightforwardly with software layers, since they do not focus on the internal structuring of an application, but distinguish services at different levels of abstraction.

2.4.2 Design Decisions with Regard to the Types of Logic

The names of the types of logic within the TSLR are intended to be as clear and unambiguous as possible. Terms used in the founding literature, like application logic, business logic and even domain do not make clear what is meant by them, and the terms are used for quite different responsibilities (Larman, 2005). An interesting example of different definitions of “domain” and “business logic” by two authors in the same book makes clear that the responsibilities can differ considerably. Fowler describes domain as "logic that is the real point of the system”, “also referred to as business logic" (Fowler et al. 2003, p. 20). On the other hand, Stafford divides "business logic into two kinds: domain logic, having to do purely with the problem domain (such as strategies for calculating revenue recognition on a contract), and application logic, having to do with application responsibilities"(Fowler et al. 2003, p. 134). To prevent confusion, new, semantic-rich names are chosen within the TSLR.

Consumer Interface Logic is selected as name, because it is a semantically rich name, and it is not frequently used and burdened, like Presentation and User Interface, who are frequently used as names for layers not only given presentation responsibilities, but also given task specific responsibilities.

Task Specific Logic is business logic, exclusive for a task and not commonly reusable. It maps to the second kind of business logic in Stafford's description. The term "Task Specific Logic" is derived from the term task-centric service as used by Erl (2008).

Domain Generic Logic within the TSLR maps to the first kind of business logic in Stafford's description. The term is chosen to make clear that this type of business logic is broadly reusable. The distinction between Task Specific Logic and Domain Generic Logic is made primarily to enable reuse and analyzability. The distinction between these two types of logic is also described by Alan and Frost (1997) as the distinction between user and business service and by Larman (2005) as the distinction between the application layer and domain layer.

The distinction between Infrastructure Abstraction Logic and Infrastructure Logic also requires some explanation. Evans (2004) distinguishes one Infrastructure layer only. On the other hand, Larman (2005) distinguishes three layers containing broadly reusable logic (Business Infrastructure, Technical Services, Foundation), but the criteria used, are a bit vague. Within the TSLR, the *Infrastructure Abstraction Logic* is specific to the business application, while *Infrastructure Logic* is not specific and enables reuse across different business applications.

3. APPROACH TO APPLY THE TSLR WITH THE RESPONSIBILITY TRACE TABLE

The TSLR is intended to aid the design, documentation and review of layered software architectures. The Responsibility Trace Table (RTT) enhances the application of the TSLR in practice. In this section, the RTT is introduced and exemplified at first. Next, the application areas of the TSLR and RTT are discussed.

3.1 Responsibility Trace Table

An RTT shows the assignment of the TSLR-responsibilities to the software layers of a case-specific layered design. The types of logic with their sub-responsibilities are represented as columns and the software layers as rows. An X within an intersecting cell of a TSLR responsibility and a layer shows the assignment of the TSLR-responsibility to the layer. The advantage of the trace table is that it provides an overview of the responsibilities of the software layers, without the need for extensive textual descriptions, since the responsibilities are defined within the TSLR. An example of an RTT is included as Table 6. It shows the responsibilities of the three principal layers as described by Fowler et al. (2003) and shown in Fig. 4. The analysis of this layered design is discussed in the next section.

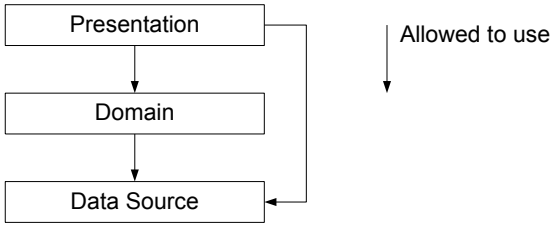

Fig. 4 Layered design based on the three principal layers(Fowler et al., 2003)

Table 6 Responsibility Trace Table linking Fowler's three principal layers to the responsibilities defined by the TSLR

Type of Logic →	Consumer Interface			Task Specific			Domain Generic			Infrastructure Abstraction		Infrastructure	
TSLR Responsibility → / Software Layer ↓	Interface Construction	Event Capturing	Event Processing	Task Control	Task State Maintenance	TS Operation	DG Service Control	DG Data Transfer	DG Operation	Platform Abstraction	Application Abstraction	Platform Service	Application Service
Presentation	X	X	X										
Domain							X	X	X				
Data Source										X	X		

3.2 Application areas

3.2.1 Design of Software Layers

During the design of layered software architectures, a number of design decisions, described before in the Introduction section, have to be taken. The TSLR and RTT aid the decision and documentation regarding the design question: *Which types of responsibility are assigned to each layer?*

The TSLR gives a complete overview of the assigned responsibilities per layer and can be used to consider alternatives and to decide on clear-cut separations of concerns per layer. The RTT shows the assignment of the TSLR-responsibilities to the application-specific layers, and this overview supports reasoning about the architecture. Finally, the documentation of the responsibilities of the layers of a system specific layered design may be prepared by the combined use of the TSLR and RTT. An RTT makes it easy to an architect to complement the graphical representation of the layered design with a definition of the responsibilities of the layers, without much documentation.

3.2.2 Analysis of Layered Designs

Another application area is the analysis of existing or proposed layered designs. TSLR and RTT are useful to gain a clear insight into the division of the responsibilities over the software layers. The RTT is very useful here, since it shows omissions and redundancies in the assignment of responsibilities to the layers within a software architecture. This helps to evaluate the quality of the layered design and the effectiveness in achieving the quality requirements.

3.2.3 Training

Finally, the TSLR and RTT may be helpful in the training of students, software engineers and architects. We used the TSLR and RTT in software architecture courses for third year bachelor students Computer Science. Drawing up or implementing a layered design requires knowledge of the different types of responsibilities. We use the TSLR and some assignments to let the students acquire this knowledge. Furthermore, we discuss the suitability of several layered designs to meet specified quality requirements, and we discuss proposals for layers in student projects. The visual character of the TSLR's classification schema, and the overview provided by an RTT, support the explanation and discussion of different design alternatives regarding the assignment of responsibilities to layers.

4. APPLICATIONS

Three cases are described below to illustrate the practical use of the TSLR and RTT.

4.1 Fowler's Three Principal Layers

To demonstrate the applicability of the TSLR and RTT as supporting tools for the *analysis of an existing layered design*, we use Fowler's "Three Principal Layers". This layered design, discussed in the previous section and shown in Fig. 4, serves well for this purpose, since it is extensively described (Fowler et al. 2003, pg. 19-22) and well known. The translation of the description of the three layers into TSLR responsibilities was fairly easy. The resulting Responsibility Trace Table, shown in Table 6, provides a good overview of the responsibilities per layer. Two observations are interesting to discuss.

The first observation is that the Task Specific Logic is not assigned to a layer, which should be regarded as an omission in the definition of a layered design. The description of the layers makes clear that the Presentation layer and Domain layer include all responsibilities of respectively Consumer Interface Logic and Domain Generic Logic from the TSLR. However, the definitions of Presentation and Domain do not make clear where the Task Specific Logic is allocated. In later chapters, it appears that Task Specific Logic may be included in both layers, Presentation and Domain, depending on the pattern chosen. The Application Controller Pattern assigns Task Specific Logic to the Presentation layer. In terms of the TSLR, an Application Controller contains Task Specific Logic, since its two main responsibilities are defined as: "deciding which domain logic to run and deciding the view with which display the response". The Service Layer Pattern is used to organize the Domain and assigns Task Specific Logic to the Domain layer. A service layer "encapsulates the application's business logic, controlling transactions and coordinating responses in the implementation of its operations". In terms of the TSLR a services layer contains Task Specific Logic, especially since it "typically includes logic that's particular to a single use case".

The second observation is that the name of the third Principle Layer, the Data Source layer, does represent only a part of its contents. A more general name should enhance the interpretability of this layer, since it is not only responsible for the communication with data sources in the infrastructure, but also for the communication with the rest of the infrastructure, like transaction monitors, other applications, and messaging systems.

4.2 Layered Design of HUSACCT

To demonstrate the applicability of the TSLR and RTT as supporting tools for the *design of a layered architecture*, we use the case of the development of HUSACCT (HU University Software Architecture Compliance Checking Tool). We have been working on HUSACCT for several years during a specialization semester "Advanced Software Engineering" for third year bachelor students. HUSACCT can be used to: 1) Define the intended modular architecture: layers, subsystems, components, external systems, and rules constraining their properties and relations (Pruijt, Köppe, & Brinkkemper, 2013); 2) Analyze the implemented architecture embedded in the source code (Java, C#); and 3) Validate the compliance between intended and implemented architecture.

Based on the requirements, the layers model was drawn up, shown in Fig. 5, and the responsibilities per layer were specified in a RTT, shown in Table 7. The layered design of HUSACT, combined with a domain model and a logical component model, served well to address the key requirement, divide the work over six teams, and identify and specify the required communication between the system's components.

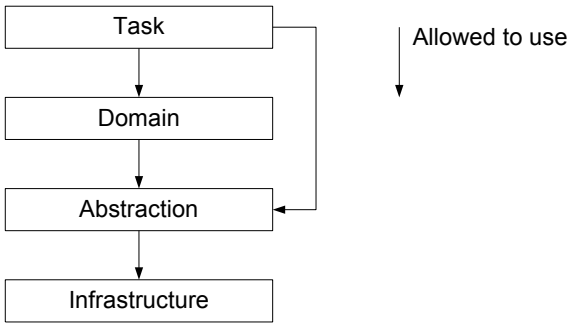

Fig.5 Layered design of HUSACCT

Table 7 Responsibility Trace Table of HUSACCT's Layered design

Type of Logic →	Consumer Interface			Task Specific			Domain Generic			Infrastructure Abstraction		Infrastructure	
TSLR Responsibility → Software Layer ↓	Interface Construction	Event Capturing	Event Processing	Task Control	Task State Maintenance	TS Operation	DG Service Control	DG Data Transfer	DG Operation	Platform Abstraction	Application Abstraction	Platform Service	Application Service
Task	X	X	X	X	X	X							
Domain							X	X	X				
Abstraction										X			
Infrastructure												X	

The Task layer includes two types of logic from the TSLR: Consumer Interface Logic, and Task Specific Logic. The restrictions of two strictly separated layers for these responsibilities seemed to impose more cons than pros. For reasons of analyzability, separate packages were created for these types of logic within the Task Layer, but the communication between these packages is not restricted by the default rules of a layered design. The abstraction layer was introduced to implement the analysis of source code as programming language independent as possible; since an important requirement was that the tool should be expandable with regard to the analysis of other programming languages. Since certain processes within the Task layer need direct access to abstracted infrastructural services, a skip call is allowed from the Task layer to these services of the Abstraction Layer. The layered design of HUSACCT illustrates that the number and names of system specific layers do not have to match the TSLR's types of logic.

4.3 Layered Architecture of a Large Software System

To demonstrate the applicability of the TSLR and RTT for large systems, we use a case of a complex, governmental administration system, aimed at a user base of approximately 6000 end users and distributed across 75 different physical locations. The system's layering schema, part of the well-documented software architecture, is shown in Fig. 6. The layers and their constituting components are described concisely below. The mapping of the layers and their components to the TSLR responsibilities was done in retrospect, and the result is visible in the Responsibility Trace Table, shown in Table 8.

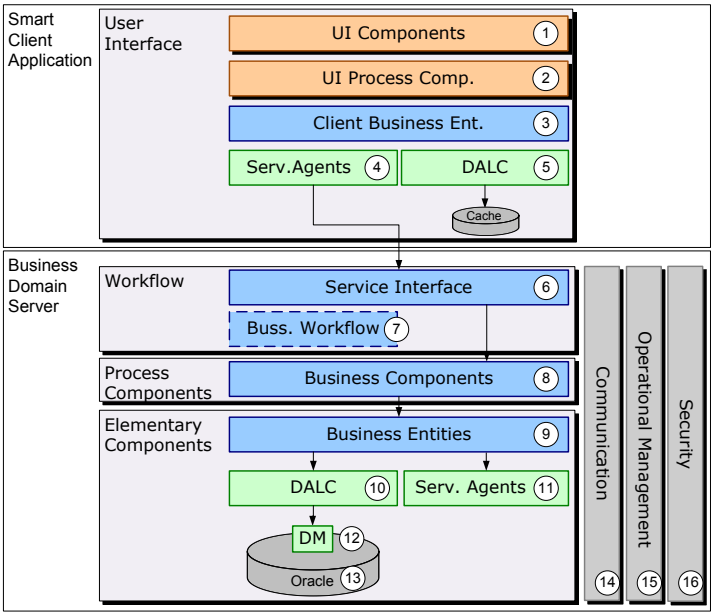

Fig. 6 Layering schema of the case system

The system's architecture was based primarily on the Microsoft .Net reference architecture for .Net version 1.1. However, it deviated from Microsoft practices in the following manner: a) the system made heavily use of object orientation to handle the large quantity of business rules in a classic OO style; b) the system was split up in a Smart Client application and a Business Domain Server application.

The Smart Client is the implementation of the *User Interface layer*, responsible for capturing input and calling the Business Domain Server through web services. It was designed to be user-friendly while having only a minimal amount of business knowledge. The User Interface layer contains five types of components. User Interface Components (1) are responsible for showing data to the users, for collecting and syntactical validating data entered by the user and for interpreting events. The UI Process Components (2) are responsible for coordination of the user process and management of the process state. Client business entities (3) are the “less intelligent” cousin classes of the Business Entities found in the domain server. They typically have very little domain knowledge and are used to enforce required fields, field formats, restrict list values etc. The business domain server provides an xml structure, which states what fields are required and what list selections are appropriate for the given state of the object being viewed. UI Service Agents (4) have the responsibility of making available data to the system and can be seen as a courier used to handle the conversation with the Domain Server. User Interface Data Access Logic Components (5) are responsible for providing access to the data cache in this layer. This cache is used to minimize bandwidth usage and overall response time.

The Business Domain Server is responsible for processing the requests from the client and other channels, while maintaining integrity and security. It is organized in three layers and three cross cutting concerns. The *Workflow layer* is responsible for coordinating workflow in a future release of the system. In the current release, it only had the responsibility to provide a facade to access the underlying domain functionality. Service Interface Components (6) provide the services that the application offers in a simple, secure manner and hide the underlying system implementation. The service interfaces are implemented with web services. The Business Workflow component (7) is included to enable the integration of an external workflow application in a future release of the system. The *Process Components layer* is

responsible for coordinating the processing of single business events that happened during the workflow. This layer contains two types of Business Components (8). Task Controllers are responsible for controlling the underlying Process Controllers and persisting (or undoing at a failure) all activities as one transaction. Process Controllers are reusable business activities and are responsible for coordinating the data transformations. The *Elementary Components layer* contains Business Entities (9), responsible for maintaining the integrity of the information, and Data Access Logic Components (10), EC Service Agents (11) and Data Mappers (12), responsible for storage and retrieval of the information in the Oracle database (13) or in external systems.
The *Crosscutting Concerns* (MSDN, 2009) Communication (14), Operational Management (15) and Security (16) were handled by services of the application platform infrastructure. Only access to the security library was abstracted by means of service interface classes.

Table 8 Responsibility Trace Table of the case system's principal layers and components

Type of Logic →		Consumer Interface			Task Specific			Domain Generic			Infrastructure Abstraction		Infrastructure	
TSLR Responsibility → ↓ Software Layer / Component		Interface Construction	Event Capturing	Event Processing	Task Control	Task State Maintenance	TS Operation	DG Service Control	DG Data Transfer	DG Operation	Platform Abstraction	Application Abstraction	Platform Service	Application Service
User Interface	1	X	X	X										
	2				X	X	X							
	3								X	X				
	4										X			
	5										X			
Workflow	6	X	X	X										
	7											X		
Process Comp.	8				X	X	X	X						
Elementary Comp.	9								X	X				
	10										X			
	11										X			
	12								X		X			
	13												X	
Crosscutting Conc.	14												X	
	15												X	
	16										X			

5. DISCUSSION

Since the research was intended to deliver an artifact relevant to the professional practice, our study can be characterized as design-science research (Peffers & Tuunanen, 2008). Based on a practical problem, we defined research questions, studied leading literature about software layers, designed instruments in line with this literature, and evaluated the instruments.

To evaluate the typology and its related trace table, the instruments were reviewed by experts, applied in practical cases, and used in training courses for bachelor students. Five experts in the field of software architecture reviewed our proposals on completeness and accuracy. In their responses, they provided useful feedback, which was used for improvements. Some names were discussed and changed, descriptions were improved and examples added. A lot of discussion focused on an unambiguous name for the first type of logic in the TSLR and resulted in "Consumer Interface Logic". In addition to the expert review, the completeness and accuracy of the TSLR was evaluated by means of a case study of the software architecture of a large software system. The outcome of the evaluation was positive; no responsibilities were found missing in the TSLR, and the arrangement and naming of the responsibilities appeared accurate. The mapping of the system's responsibilities on these of the typology required several

iterations in which the architect's knowledge of the TSLR was deepened, as well as the researcher's knowledge of the particular software architecture. In this process, the trace table proved to be a valuable instrument. The visual overview supported architectural reasoning and helped to recognize omissions and redundancies in the initial versions of the system's RTT.

There are some limitations to our research so far. One important limitation is that our research focused on responsibilities of the software of business information systems. Therefore, other types of systems, like embedded systems and games, might contain responsibilities not included in the TSLR. Another limitation has to do with the completeness of the typology. Despite our extensive literature study and validation activities, we cannot ensure that all types of responsibilities, common in business information systems, are represented in the TSLR. However, future additions and evolution are taken into account; the meta-model of the typology enables extensions in width and depth. Finally, the typology could be viewed and used as a layered model. However, *the typology is not intended to be a template for layered designs, with layers exactly matching the types of logic of the typology.* Layered designs in practice should be designed to meet the specific requirements of the system. The number and names of the required layers may vary, the responsibilities per layer may vary, and a layer may contain sub-responsibilities from different types of logic within the TSLR.

6. CONCLUSIONS

In this paper, we proposed two novel instruments to support software architects in their task to design layered architectures of high quality: the Typology of Software Layer Responsibility (TSLR) and the complementary Responsibility Trace Table (RTT). These instruments, together with some illustrations of their practical use, provide answers to the research questions, which formed the basis of this study. We started with the observation that the terminology regarding layered designs is not clear and sometimes contradictory. We finished with a proposed typology and a trace table to aid the practical use of the typology.

The TSLR provides an overview of the distinct types of responsibility commonly found in the software of business information systems. The TSLR may be used when a layered design is drawn up and when an existing layered design is analyzed or reviewed. Furthermore, it is useful in training courses to discuss and exercise the different possibilities to divide responsibilities over the layers and their impact on the quality characteristics of the software system. The TSLR responsibilities are distilled from leading literature on layers in the field of software architecture. The TSLR separates and groups the responsibilities, gives them unambiguous names, specifies them and exemplifies them. At the level of infrastructural responsibilities a connection is established to the TOGAF Technical Reference Model (The Open Group, 2009), which classifies a huge number of infrastructural services.

The Responsibility Trace Table (RTT) shows the assignment of the TSLR responsibilities to the different software layers. The RTT is an instrument to complement a system's graphical representation of the layered design with a specification of the responsibilities of the layers. In addition, the RTT may be used to assess and enhance the quality of a layered design, since it shows omissions and redundancies in the assignment of the responsibilities.

To illustrate the application of the instruments three cases were presented: a design case, a review case, and a complex case of al large governmental software system. These cases were also used to evaluate the completeness, accuracy, and applicability of the instruments. Furthermore, experts in the field of software architecture conducted a review, and the instruments were used in training courses for bachelor students.

Further research may be aimed on the applicability and scope of the TSLR and RTT. At first, it will be interesting to study the effectiveness of the TSLR and RTT when practitioners and students apply these instruments. Next, to enlarge the field of application of the TSLR, literature and case studies are needed on the responsibilities of other types of software systems (other than business information systems). Finally, it will be interesting to study the applicability of the instruments in the context of other software architecture patterns.

ACKNOWLEDGEMENTS

The authors want to thank Rik Bos, Christian Köppe, Maurice Driessen, Arnoud ten Hoedt, Jurriaan van Reijsen, Erik van der Starre, Fenne Verhoeven and Johan Versendaal for their valuable contributions.

REFERENCES

Allen, P., & Frost, S. (1997). *Component Based Development for Enterprise Systems: Applying the Select Approach*. Cambridge University Press.

Avgeriou, P., & Zdun, U. (2005). Architectural Patterns Revisited — A Pattern Language. *10th European Conf. Pattern Languages of Programs (EuroPLoP)* (pp. 431–470).

Bass, L., Clements, P., & Kazman, R. (2012). *Software Architecture in Practice* (Third Edit., p. 624). Addison-Wesley.

Buschmann, F., Meunier, R., Rohnert, H., & Sommerlad, P. (1996). *Pattern-Oriented Software Architecture: A System of Patterns, Volume 1* (p. 476). John Wiley & Sons.

Clements, P., Bachmann, F., Bass, L., Garlan, D., Merson, P., Ivers, J., Little, R., et al. (2010). *Documenting Software Architectures: Views and Beyond* (p. 537). Pearson Education.

Clements, P., & Nord, R. (2000). Documenting a Layered Software Architecture. *Fourth International Software Architecture Workshop Limerick* (pp. 121–124).

Cockburn, A. (1997). Structuring Use Cases with Goals. *Journal of Object-Oriented Programming*, (Sep.-Oct. (part I) and Nov.-Dec. (part II)).

Dijkstra, E. W. (1968). The structure of the "THE"-multiprogramming system. *Communications of the ACM*, *11*(5), 341–346.

Erl, T. (2008). *SOA Design Patterns*. Prentice Hall.

Evans, E. (2004). *Domain-Driven Design: Tackling Complexity in the Heart of Software* (p. 529). Addison-Wesley Professional.

Fowler, M., Rice, D., Foemmel, M., Hieatt, E., Mee, M., & Stafford, R. (2003). *Patterns of enterprise application architecture*. Addison-Wesley, Boston, MA, USA.

Gamma, E., Helm, R., Johnson, R., & Vissedes, J. (1995). *Design Patterns: Elements of Reusable Object-Oriented Software*. Pearson Education.

Gorton, I. (2006). *Essential Software Architecture*. Springer Berlin Heidelberg.

Harrison, N. B., & Avgeriou, P. (2008). Analysis of Architecture Pattern Usage in Legacy System Architecture Documentation. *Seventh Working IEEE/IFIP Conference on Software Architecture (WICSA 2008)* (pp. 147–156). IEEE Comput. Soc.

Krafzig, D., Banke, K., & Slama, D. (2005). *Service-Oriented Architecture Best Practices*. Prentice-Hall.

Kruchten, P., Lago, P., & Vliet, H. Van. (2006). Building up and reasoning about architectural knowledge. *Quality of Software Architectures*, 43–58.

Lankhorst, M. (2009). *Enterprise Architecture at Work*. Springer, Berlin.

Larman, C. (2005). *Applying UML And Patterns* (p. 703). Prentice Hall PTR.

MSDN. (2009). *Microsoft Application Architecture Guide, 2nd ed.* Microsoft Corporation.

Parnas, D. L. (1972). On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, *15*(12), 1053–1058.

Peffers, K., & Tuunanen, T. (2008). A design science research methodology for information systems research. *Journal of Management Information Systems*, *4*, 45–78.

Pruijt, L., Köppe, C., & Brinkkemper, S. (2013). Architecture Compliance Checking of Semantically Rich Modular Architectures: A Comparison of Tool Support. *29th International Conference ofn Software Maintenance* (p. 220-229). IEEE Computer Society Press.

Savolainen, J., & Myllarniemi, V. (2009). Layered architecture revisited—Comparison of research and practice. *WICSA/ECSA* (pp. 317–320).

Shaw, M., & Garlan, D. (1996). *Software Architecture: Perspectives on an Emerging Discipline*. Prentice Hall.

Snoeck, M., Poelmans, S., & Dedene, G. (2000). A layered software specification architecture. *Conceptual Modeling—ER 2000*.

The Open Group. (2009). The Open Group Architecture Framework: Version 9, Enterprise Edition.

Winter, R., & Fischer, R. (2007). Essential layers, artifacts, and dependencies of enterprise architecture. *Journal of Enterprise Architecture*, (May), 1–12.