THE ‘UNDESIRABLES’ OF SOFTWARE ARCHITECTURE

BY PANKAJ KAMTHAN

1. INTRODUCTION

The desirables of building architecture design—according to Marcus **Vitruvius** Pollio, a Roman author, architect, and civil engineer in the 1st Century BC—are:

- Firmitas (Strength)
- Utilitas (Utility)
- Venustas (Beauty)

This document explores the issue of the quality of software architecture, and it does so **indirectly** by pointing out certain ‘undesirables’ that could make the software architecture **unstable, unsupportive, and ugly**.

2. MOTIVATION FOR STUDYING ‘UNDESIRABLES’

The life of a software architect is a long (and sometimes painful) **succession** of **suboptimal decisions** made partly in the dark.
— Philippe Kruchten

In aiming for **‘high-quality’ software architecture**, it is important not only to commit to means (such as experiential knowledge) that can lead to such an aim, but also to **avoid, minimize, or eliminate** certain ‘undesirables’.

It is said that **every cloud has a silver lining**. Therefore, there is interest in studying such 'undesirables' in software engineering, in general [Mäntylä, 2003; Luo, 2010], and in software architecture, in particular [Brown, Malveau, McCormick, Mowbray, 1998; Lippert, Roock, 2006, Chapter 3; Garcia, Popescu, Edwards, Medvidović, 2009a; Garcia, Popescu, Edwards, Medvidović, 2009b; Im, McGregor, 2010; Mistrik, Bahsoon, Eeles, Roshandel, Stal, 2014].

2.1. ARCHITECTURAL DEBT

It has been pointed out that the presence of 'undesirables' in software architecture is a kind of **technical debt** [Tom, Aurum, Vidgen, 2013].

3. 'UNDESIRABLES' IN SOFTWARE ARCHITECTURE VIEWS

There are certain **view-specific** issues that can arise [Rozanski, Woods, 2005]. (The issues are about **non-functional requirements**.) These issues could be manifestations of the presence of a number of things including, but **not limited** to, **smells and/or anti-patterns**.

The collection of issues included in this document can serve as a **checklist** during a review of the software architecture description.

A possible **resolution of an issue** can vary on a number of **dimensions**:

- A resolution of an issue may be **trivial or non-trivial**.
- A resolution of an issue may be **total, or only partial**.
- A resolution of an issue may require **compromises**.

3.1. ISSUES (AND THEIR RESOLUTIONS) IN FUNCTIONAL VIEW

There are a number of potential information issues, including **Poorly Defined Interfaces, Poorly Understood Responsibilities, Infrastructure Modeled as Functional Elements, Overloaded View, Diagrams Without Element Definitions, Difficulty in Reconciling the Needs of Multiple Stakeholders, Inappropriate Level of Detail, 'God' Element, and Too Many Dependencies**, a subset of which is covered in the following.

ISSUE: NO INTERFACES OR POOR INTERFACES

- There is an element that is **isolated**.
- There are no interfaces between elements or, if they exist, they are not defined or poorly defined.

RESOLUTION

- There should not be an isolated element. The Functional View should be represented by a **connected graph**.
- The interfaces between elements must be clear, and clearly defined, in the sense that their **semantics** must be clear. The definition of an element should not be considered as ‘complete’ until the interfaces have been defined.

ISSUE: NO RESPONSIBILITY OR POORLY UNDERSTOOD RESPONSIBILITIES

The purpose of an element is not known, or the purpose is unclear.

RESOLUTION

There must be an **explicit reason** for each element to exist. In particular, what each element **does** must be clear.

ISSUE: INFRASTRUCTURE MODELED AS FUNCTIONAL ELEMENTS

There is an element that models infrastructure.

RESOLUTION

The infrastructure should be **hidden (abstracted)** in a Functional View. (The infrastructure should be modeled in the Deployment View.)

ISSUE: OVERLOADED VIEW

(The Functional View is significant and **almost always** a **mandatory view**.) The view appears to be **all** the views rather than a major view.

In other words, the view has information from **other possible views**.

Figure 1 presents an example of an overloaded view. The diagram, potentially, as a whole, **may not be useful to any stakeholder**.

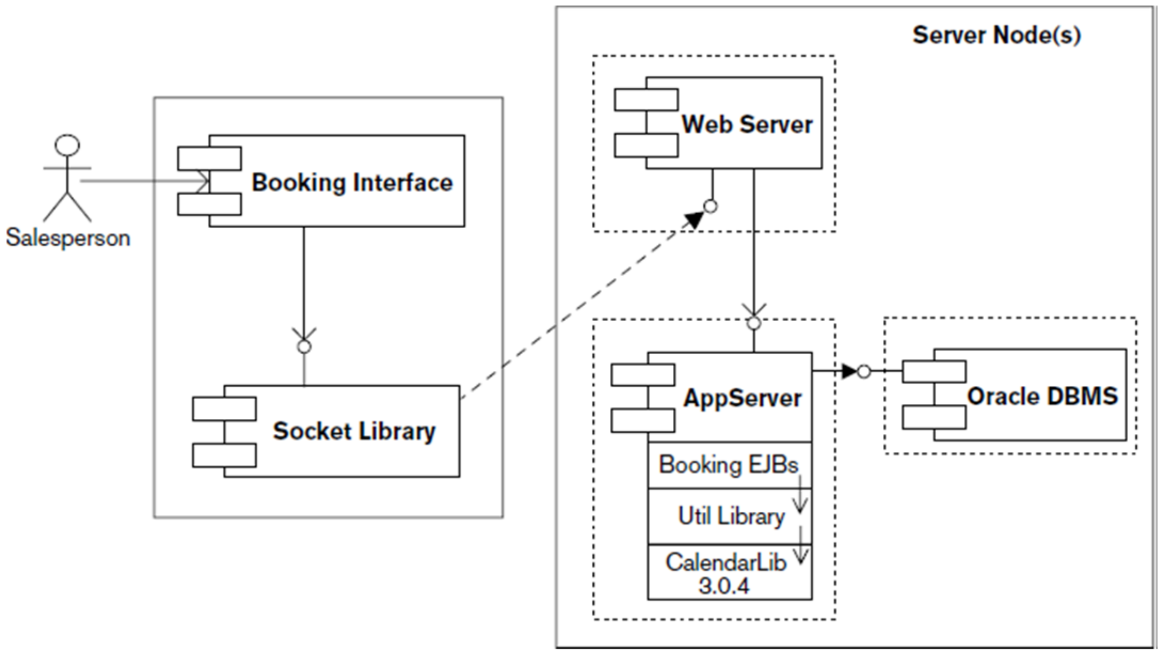

Figure 1. An overloaded view. (Source: [Rozanski, Woods, 2005].)

There are **non-standard, even erroneous**, UML extensions in the diagram in Figure 1. (For example, there is a dashed line from the Socket Library box to the Web Server box, the dotted lines within the Server Node(s) box, and so on.)

As a result, the **semantics** of the diagram is **unclear** (except, perhaps, to the software architect). For example, the dotted lines may mean operating system processes, **or** something else.

RESOLUTION

A view should be focused (and appear cohesive). It should 'show' one thing, and no more.

If there are fundamentally **different kinds of concerns**, then it is better to allocate them to **different kinds of views**.

ISSUE: DIFFICULTY IN RECONCILING THE NEEDS OF MULTIPLE STAKEHOLDERS

The description of the view ignores the needs of certain (non-technical) stakeholders.

RESOLUTION

The description of the view should use an appropriate **combination of text and graphics**.

If the graphics **cannot** be understood on its own, then it should be **annotated** by text. (This is the purpose of UML Note, figure caption, and so on.)

ISSUE: INAPPROPRIATE LEVEL OF DETAIL

If you need more than 3 levels of indentation, you're screwed anyway, and should fix your program.
— Linux Kernel Coding Style

There are more than **three levels** of detail.

RESOLUTION

It depends on the context but, assuming about ten elements at the top, the levels of detail should not exceed three. In any case, there must be a **distinction** between **macro- and micro-architecture**.

ISSUE: (PRESENCE OF) 'GOD' ELEMENTS

There is a 'God' element. (This is **not unique** to Functional View.)

The presence of a 'God' element leads to systems that can be **difficult to understand, can be difficult to develop, and/or can be difficult to maintain**. (The systems can be difficult to maintain, as they lack cohesion and are resistant to change.)

Figure 2 illustrates functional organization that has a 'God' element. In this situation, the Customer Management system element appears to exhibit the major characteristic of a 'God' element: nearly all inter-element interactions involve it.

It is likely that the **Customer Management** element contains too much of the system's functionality and has dependencies with too many of the system's elements.

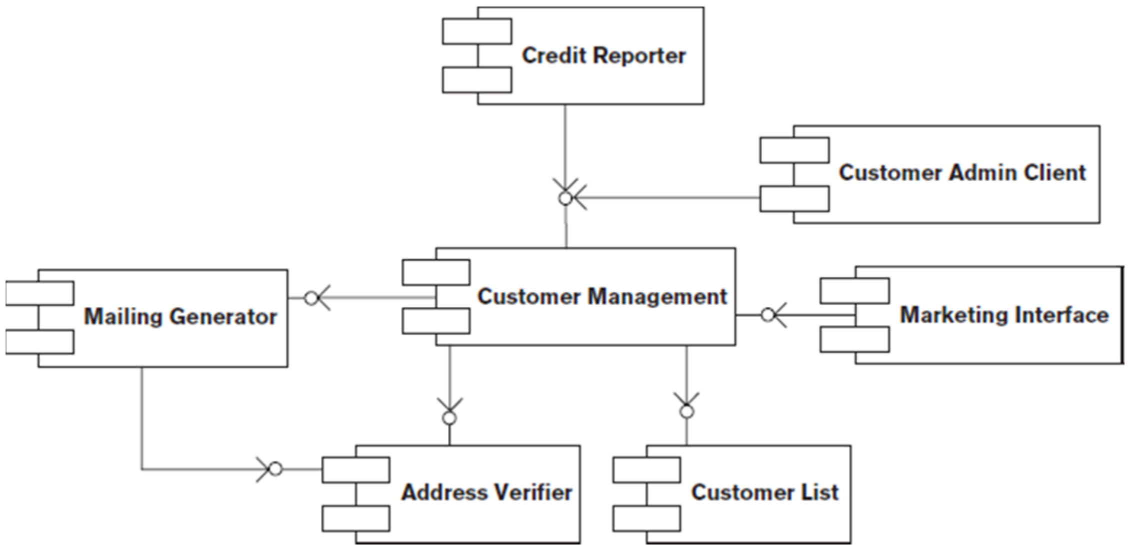

Figure 2. The presence of 'God' element. (Source: [Rozanski, Woods, 2005].)

RESOLUTION

It is better to **repartition** the system into a set of elements with more **evenly distributed functionality**.

It is suggested that **no more than 50%** of the system's responsibilities should be concentrated in **less than 25%** of the elements.

3.2. ISSUES (AND THEIR RESOLUTIONS) IN INFORMATION VIEW

There are a number of potential information issues, including **Data Incompatibilities, Poor Data Quality, Unavoidable Multiple Updaters, Key Matching Deficiencies, Poor Information Latency, Interface Complexity, and Inadequate Volumetrics**, a subset of which is covered in the following.

ISSUE: DATA INCOMPATIBILITIES

There is data incompatibility due to integrating multiple systems, or using a legacy system.

For example, **different character encodings** are used, **different currencies** are used, **different coding for countries** is used, and so on.

RESOLUTION

There may not be a straightforward resolution, especially if the systems are based on different **business models**.

ISSUE: INTERFACE COMPLEXITY

If two systems need to transfer information between themselves, one **bidirectional** interface needs to be developed.

The number of such interfaces, as illustrated in Figure 3, increases **geometrically** with the number of systems involved, leading to **overwhelming structural complexity**.

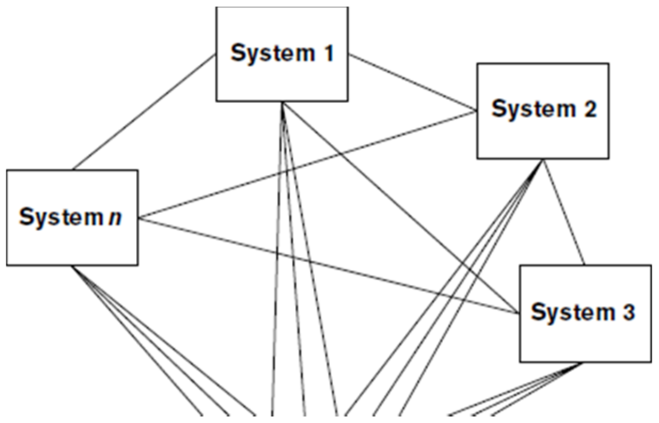

Figure 3. There are multiple systems, requiring multiple interfaces. (Source: [Rozanski, Woods, 2005].)

RESOLUTION

An **integration hub**, which is a kind of architectural style, could be used. (The integration hub is the basis for, and more general than, say, PUBLISH AND SUBSCRIBE pattern.)

In an integration hub, there are adapters (**between** the hub and systems, one per system) that perform **system-specific** transactions.

The **advantage** is that if a system changes, then often **only** the adapter for that system needs to be modified. This **supports maintenance**. Furthermore, specialized code for routing, resilience, and so on, has to be implemented only once, in the hub. This helps **avoid redundancy**.

However, a **disadvantage** is that the hub can **constrain throughput**, and is a **single point of failure**[1].

REMARKS

Burj Al Arab and **Hong Kong's Ocean Airport** have been featured in the BBC documentary series called **Engineering Connections**. They provide examples of how single points of failures can be avoided in the constructions.

3.3. ISSUES (AND THEIR RESOLUTIONS) IN CONCURRENCY VIEW

There are a number of potential concurrency issues, including **Modeling of the Wrong Concurrency, Resource Contention, Deadlock, and Race Conditions**, a subset of which is covered in the following.

ISSUE: MODELING OF THE WRONG CONCURRENCY

The focus is on the **specifics** of the **internal concurrency**, and **state** of each element.

For example, in a client-server environment, the attention is on how individual threads in a server will be allocated, used, and freed, along with how they all will be coordinated.

[1] A **single point of failure (SPOF)** is "a part of a system that, if it fails, will stop the entire system from working". For example, a router is a SPOF for the communication network between computers. Therefore, SPOFs are undesirable in any hardware or software system with a goal of high availability. [Wikipedia]

RESOLUTION

The focus should be on **architecturally significant concurrency**, that is, the overall concurrency structure, the mapping of elements to that structure, and the system-level states.

ISSUE: RESOURCE CONTENTION

There is point of resource contention[2].

The phenomenon of resource contention usually manifests itself as excessive activity in small, specific parts of the system (colloquially known as **hot spots**).

RESOLUTION

In theory, there should **not** be any resource contention points.

However, in practice, if one resource contention point is eliminated, another will emerge. Therefore, tackling resource contention is normally a process of **reducing** the contention to an acceptable level.

There are a number of suggestions:

- If possible, **reduce the amount of time the locks are by using locking methods**, such as optimistic concurrency control (or **optimistic locking**).
- If possible, **reduce the amount of concurrency** needed **around** resource contention points.
- If possible, **eliminate shared resources**.

3.4. ISSUES (AND THEIR RESOLUTIONS) IN DEVELOPMENT VIEW

There are a number of potential development issues, including **Uneven Focus**.

[2] A **resource contention** is "a conflict over access to a shared resource such as random access memory, disk storage, cache memory, internal busses or external network devices". For example, resource contention can occue if multiple processes attempt to use a shared resource. [Wikipedia.]

ISSUE: UNEVEN FOCUS

The focus on elements is unbalanced.

For example, the patterns to be used for **network request handling** are discussed in **detail**, but the **initialization processing** required of each element is **considered barely**, if at all.

RESOLUTION

There is a tendency by people to focus on aspects that they understand and find interesting. However, these aspects may only be a **subset** of the overall software architecture, and the complement (of that subset) that is ignored may be as significant.

There should be a **balance** in treating each architecturally significant aspect. It is not expected that a software architect is an expert in all of these aspects. In such a case, **domain experts** (also known as **subject matter experts**) can be sought.

3.5. ISSUES (AND THEIR RESOLUTIONS) IN DEPLOYMENT VIEW

There are a number of potential deployment issues, including **Unclear or Inaccurate Dependencies, Unproven Technology, Lack of Specialist Technical Knowledge, and Late Consideration of the Deployment Environment**, a subset of which is covered in the following.

ISSUE: UNCLEAR OR INACCURATE DEPENDENCIES

I've finally learned what "upward compatible" means. It means we get to keep all our old mistakes.
— Dennie van Tassel

There are many explicit and implicit **dependencies** on the **run-time environment**. These dependencies have not been checked.

The dependency statements, if they exist, are **vague**. The dependency statements are, for example, of the type **"Need Oracle"**, or something similar.

The (**undesirable**) result is the following. If, for example, the database server needs a utility library, then it is being pointed to a **wrong version** of utility library, and fails to start.

RESOLUTION

There is potential for run-time problems if dependencies have not been met.

The dependency statements (between elements of the system and the run-time environment) must be **precise enough to allow deployment** of the system. To do that, which versions are required, whether any optional parts of the products are needed, whether any patches are required, and so on, need to be **specified precisely**.

There should be commitment to the use of **existing, proven combinations of technologies** where the dependencies are well-understood.

(There is, at some point, need to perform **compatibility testing** to ensure the correctness of dependencies.)

ISSUE: UNPROVEN TECHNOLOGY

There is use of technology that is new or untested. There are a number of inevitable consequences of such commitment:

- **Risk.** In general, novelty or unfamiliarity is considered a **risk** in software engineering [Warkentin, Moore, Bekkering, Johnston, 2005].

- **Irreversibility.** There is non-trivial issue of **irreversibility** of the effect of any (including new) technology. For example, it is **not always straightforward,** to **discard or convert source code** based on a programming language in which **serious flaws** have been discovered, especially with the **passage of time**, and as **personnel get trained** and the **code base increases**.

RESOLUTION

(Disclaimer: There is **nothing inherently wrong** with the use of new technology. Furthermore, at some point in time, people do use new technologies. Finally, what is new today becomes old tomorrow. However, new, by definition, means that it lacks history of successes and failures.)

If commitment to a new technology is unavoidable, then there are a number of possibilities:

- It can be helpful to **solicit advice** of people who have used the technology previously, and the experience is **sufficient** enough to form a **dependable** conclusion.

- It can be helpful to have access to **independent, third-party, reviews** (**not advertisements**) of the new technology.

- It can be helpful to try (test) the new technology out, say, during prototyping, before making a definite commitment for use in the system being developed.

3.6. ISSUES (AND THEIR RESOLUTIONS) IN OPERATIONAL VIEW

There are a number of potential operational issues, including **Lack of Engagement with the Operational Staff, Lack of Backout Planning, Lack of Migration Planning, Insufficient Migration Window, Missing Management Tools, Lack of Integration into the Production Environment, and Inadequate Backup Models**, a subset of which is covered in the following.

ISSUE: LACK OF BACKOUT PLANNING

There is inherent assumption of a **perfect rollout**. There is **no graceful recovery mechanism** to deal with adverse situations, such as **failed upgrades**.

RESOLUTION

There should be a backout plan that specifies when the system should be backed out of its production environment.

4. 'UNDESIRABLES' IN SOFTWARE ARCHITECTURE

There are a number of closely related terms as per **'undesirables'** in software architecture. They are to be avoided, or, if they exist, either minimized or eliminated.

FAULT

Definition [Fault] [IEEE, 2009]. A manifestation of an error in software.

If there is an error in the software system, it **must** be present somewhere and **may** surface at some time during execution. This, along with the previous definition, leads to the following:

Definition [Software Architecture Fault]. A manifestation of an error in software architecture.

As an analogy, a **crack in the ceiling** is a building architecture fault. (The **extent** of the crack that, for example, leads to leakage of rain water, or even collapse of the ceiling, is a different matter.)

SMELL

The notion of smell was introduced in [Brown, Malveau, McCormick, Mowbray, 1998].

Definition [Software Architecture Smell]. An architectural decision, that impacts one or more quality attributes of the software system, such that (1) the **impact is negative**, and (2) the **impact is significant**.

It is **assumed** in the definition that the quality attribute(s) is(are) among the architectural **concerns**.

A smell is an **indicator or symptom** of an issue, **not the issue itself**. As an analogy, ‘bad’ smell does **not** automatically imply that the food is inedible.

A smell may be present in the software architecture due to a number of reasons, including the following:

- **The Case of Single Pattern.** The solution of a pattern is applied in a context for which the solution is inappropriate.

- **The Case of Multiple Patterns.** The solutions of multiple patterns are combined, leading to undesirable effects.

REMARKS

There are **collections of architectural smells** [Lippert, Roock, 2006, Chapter 3; Garcia, Popescu, Edwards, Medvidović, 2009a; Garcia, Popescu, Edwards, Medvidović, 2009b; Babar, Brown, Mistrik, 2014, Chapter 3].

ANTI-PATTERN

The notion of anti-pattern was introduced in [Brown, Malveau, McCormick, Mowbray, 1998].

Definition [Software Architecture Anti-Pattern]. A possibly the worst solution (as depicted in Figure 4) to a recurring architectural problem in a particular architectural context.

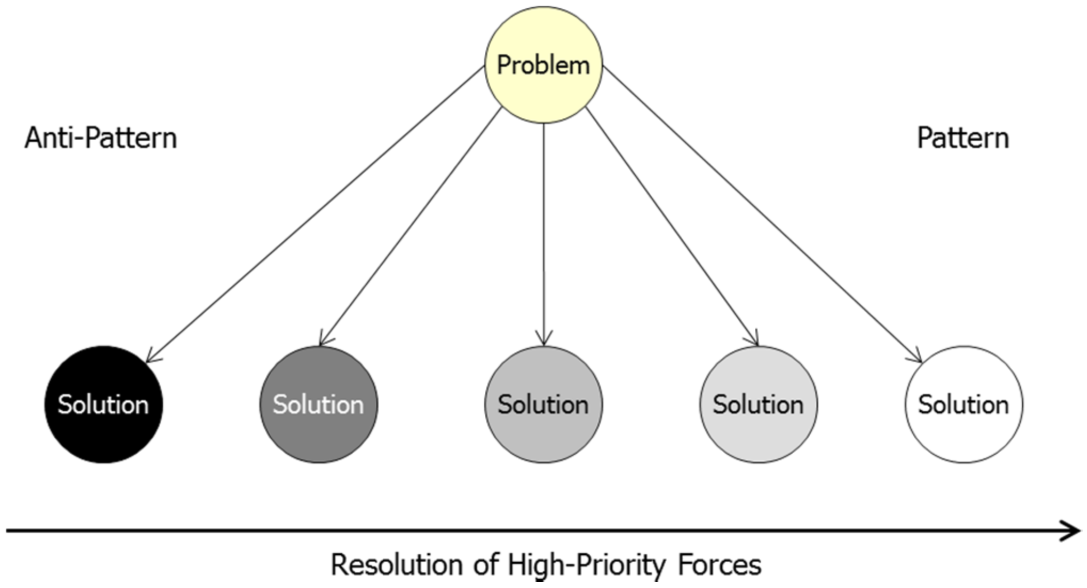

Figure 4. The relationship between a pattern and an anti-pattern.

REMARKS

- There are a number of widely-known anti-patterns for software architectures, such as BIG BALL OF MUD [Fairbanks, 2010, Section 14.7; Foote, Yoder, 1997], STOVEPIPE, VENDOR LOCK-IN [Brown, Malveau, McCormick, Mowbray, 1998], and so on.

- There are **collections** of anti-patterns for software architectures of general software systems[3,4] and for specific software systems[5].

4.1. FAULT VERSUS SMELL VERSUS ANTI-PATTERN

The terms fault, smell, and anti-pattern are related, but **not synonymous**.

4.1.1. FAULT VERSUS (SMELL AND ANTI-PATTERN)

Fault	Smell/Anti-Pattern
A software architecture fault is about a fault in the functionality of the software system (as specified by its functional requirements). The presence of a software architecture fault **cannot be justified**.	A software architecture smell, or a software architecture anti-pattern, is **not a fault** in the functionality of the software system. The use of a software architecture smell or a software architecture anti-pattern **may be justified**. For example, **software cloning**, although considered a bad smell, can be justified [Rahman, Bird, Devanbu, 2012]. For example, even if a solution suggested by a software architecture anti-pattern is **not technically acceptable**, the other solutions may **not be economically viable**.

4.1.2. SMELL VERSUS ANTI-PATTERN

A software architecture smell is **always technical** in nature. It pertains only to the technical aspects of the software architecture.

[3] URL: http://www.antipatterns.com/arch_cat.htm .
[4] URL: http://sourcemaking.com/antipatterns/software-architecture-antipatterns .
[5] URL: http://www.ibm.com/developerworks/webservices/library/ws-antipatterns/ .

For example, there are smells for **components and connectors**, such as **Connector Envy** [Garcia, Popescu, Edwards, Medvidović, 2009a; Garcia, Popescu, Edwards, Medvidović, 2009b].

The scope of software architecture anti-pattern is **broader** than that of software architecture smell. A software architecture anti-pattern **may or may not be technical in nature**.

A software architecture smell is a **possibility** of a problem (and, therefore, has **probability** on an interval, that is, on [0, 1]). A software architecture anti-pattern is a **certainty** of a problem (and, therefore, has **probability** on a point, that is, 1).

For example, WOLF TICKET[6] is a software architecture anti-pattern, and is essentially non-technical in nature. Furthermore, the description of an anti-pattern is structured; however, the description of a smell need not be associated with any (known) structure.

REMARKS

The similarities and differences between source code smells and source code anti-patterns have been investigated in [Luo, 2010, Section 2.2.4].

ACKNOWLEDGEMENT

The inclusion of images from external sources is only for non-commercial educational purposes, and their use is hereby acknowledged.

[6] "A technology is **assumed** to have positive qualities due to its open systems packaging or claimed standards compliance. [However,] few standards have test suites (< 6%) and few products are actually tested for conformance."

REFERENCES

[Babar, Brown, Mistrik, 2014] Agile Software Architecture: Aligning Agile Processes and Software Architectures. By M. A. Babar, A. W. Brown, I. Mistrik (Editors). Morgan Kaufmann. 2014.

[Brown, Malveau, McCormick, Mowbray, 1998] AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis. By W. J. Brown, R. C. Malveau, H. W. McCormick, T. J. Mowbray. John Wiley and Sons. 1998.

[Fairbanks, 2010] Just Enough Software Architecture: A Risk-Driven Approach. By G. H. Fairbanks. Marshall and Brainerd. 2010.

[Foote, Yoder, 1997] Big Ball of Mud. By B. Foote, J. Yoder. The Fourth Conference on Pattern Languages of Programs (PLoP 1997). Monticello, U.S.A. September 3-5, 1997.

[Garcia, Popescu, Edwards, Medvidović, 2009a] Identifying Architectural Bad Smells. By J. Garcia, D. Popescu, G. Edwards, N. Medvidović. The 2009 European Conference on Software Maintenance and Reengineering (CSMR 2009). Kaiserslautern, Germany. March 24-27, 2009.

[Garcia, Popescu, Edwards, Medvidović, 2009b] Toward a Catalogue of Architectural Bad Smells. By J. Garcia, D. Popescu, G. Edwards, N. Medvidović. The Fifth International Conference on the Quality of Software Architectures (QoSA 2009). East Stroudsburg, U.S.A. June 24-26, 2009.

[IEEE, 2009] IEEE Standard 1044-2009. IEEE Standard Classification for Software Anomalies. IEEE Computer Society. 2009.

[Im, McGregor, 2010] Locating Defects in Software Architectures through Debugging. By K. Im, J. D. McGregor. The Nineteenth International Conference on Software Engineering and Data Engineering (SEDE 2010). San Francisco, U.S.A. June 16-18, 2010.

[Lippert, Roock, 2006] Refactoring in Large Software Projects. By M. Lippert, S. Roock. John Wiley and Sons. 2006.

[Luo, 2010] Improving Software Quality Using An Ontology-based Approach. By Y. Luo. Ph.D. Thesis. Louisiana State University. Baton Rouge, U.S.A. 2010.

[Mäntylä, 2003] Bad Smells in Software - A Taxonomy and an Empirical Study. By M. Mäntylä. M.Sc. Thesis. Helsinki University of Technology. 2003.

[Mistrik, Bahsoon, Eeles, Roshandel, Stal, 2014] Relating System Quality and Software Architecture. By I. Mistrik, R. Bahsoon, P. Eeles, R. Roshandel, M. Stal (Editors). Morgan Kaufmann. 2014.

[Rahman, Bird, Devanbu, 2012] Clones: What is that Smell? By F. Rahman, C. Bird, P. Devanbu. Empirical Software Engineering. Volume 17. Issue 4-5. 2012. Pages 503-530.

[Rozanski, Woods, 2005] Software Systems Architecture: Working With Stakeholders Using Viewpoints and Perspectives. By N. Rozanski, E. Woods. Addison-Wesley. 2005.

[Tom, Aurum, Vidgen, 2013] An Exploration of Technical Debt. By E. Tom, A. Aurum, R. Vidgen. Journal of Systems and Software. Volume 86. Issue 6. 2013. Pages 1498-1516.

[Warkentin, Moore, Bekkering, Johnston, 2005] Analysis of Systems Development Project Risks: An Integrative Framework. By M. Warkentin, R. S. Moore, E. Bekkering, A. C. Johnston. ACM SIGMIS Database. Volume 40. Issue 2. 2009. Pages 8-27.