SAS vs. NSAS: Analysis and Comparison of Self-Adaptive Systems and Non-Self-Adaptive Systems based on Smells and Patterns

Claudia Raibulet[a], Francesca Arcelli Fontana[b] and Simone Carettoni
DISCo-Dipartimento di Informatica, Sistemistica e Comunicazione, Universita' Degli Studi di Milano - Bicocca, Viale Sarca, 336, Edificio 14, Milan, Italy

Keywords: Self-Adaptive Systems, Non-Self-Adaptive Systems, Software Quality, Architectural Smells, Code Smells, Design Patterns.

Abstract: Self-Adaptive Systems are usually built of a managed part, implementing their functionality, and a managing part, implementing their self-adaptation. The complexity of self-adaptive systems results also from the existence of the managing part and the interaction between the managed and the managing parts. The non-self-adaptive systems may be seen as the managed part of self-adaptive systems. The self-adaptive systems are evaluated based on their performances resulted from the self-adaptation. However, self-adaptive systems are software systems, hence, also their software quality is equally important. Our analysis compares the internal quality of self-adaptive and non-self-adaptive systems by considering code smells, architectural smells, and GoF's design patterns. This comparison provides an insight to the health of the self-adaptive systems with respect to the non-self-adaptive systems (the last being considered as a quality reference).

1 INTRODUCTION

Software systems able to manage changes and uncertainties during their execution (1) based on knowledge about their environment and about themselves, and (2) using mechanisms to reason about this knowledge and to use it properly, are considered as self-adaptive (de Lemos et al., 2013; Weyns, 2018). Self-adaptive systems (SAS) have usually bigger dimensions than non-self-adaptive systems (NSAS), when two versions of the same system, one with and one without self-adaptive mechanisms, are considered. Further, SAS may have more complex structures than equivalent NSAS (i.e., implementing similar functionalities without being self-adaptive). Self-adaptation may be performed at various abstraction levels and through different mechanisms (de Lemos et al., 2013). Therefore, its understanding and evaluation is an important and challenging task (Kaddoum et al., 2010).

The quality of software systems may be evaluated from various points of view and in various ways considering quality attributes, software metrics, design patterns, and anti-patterns. Architectural smells (Arcelli Fontana et al., 2017; Martini et al., 2018) and code smells (Fowler, 1999) are emerging as useful indicators about the internal quality of software systems. These smells identify recurring design and architectural problems which may lead to high maintenance and evolution costs if not addressed timely. There are several works on smells detection (Chatzigeorgiou and Manakos, 2010; Peters and Zaidman, 2012). Different works have also considered the impact of design patterns on software quality, by considering metrics and code smells (Walter and Alkhaeir, 2016). Only few works analyzed architectural smells for software quality aims (Martini et al., 2018).

For SAS evaluation there have been proposed various mechanisms including quality attributes, software metrics, and design patterns (Raibulet and Arcelli Fontana, 2017). Some of them are applied for NSAS evaluation concerning performance, dependability, robustness, security, complexity. Many approaches introduce novel mechanisms focused on the specificity of self-adaptivity, i.e., degree of autonomy, time for adaptivity, adaptability of services, support for detecting anomalous system behavior. Further, one of the first catalogs identifying 12 adaptation-oriented design patterns which capture the adaptation expertise is presented in (Ramirez and Cheng, 2010).

In this context, the contribution of this paper concerns the (1) identification of architectural and code

[a] https://orcid.org/0000-0002-7194-3159
[b] https://orcid.org/0000-0002-1195-530X

Raibulet, C., Fontana, F. and Carettoni, S.
SAS vs. NSAS: Analysis and Comparison of Self-Adaptive Systems and Non-Self-Adaptive Systems based on Smells and Patterns.
DOI: 10.5220/0009513504900497
In *Proceedings of the 15th International Conference on Evaluation of Novel Approaches to Software Engineering (ENASE 2020)*, pages 490-497
ISBN: 978-989-758-421-3

smells, as well as the GoF's design patterns in six SAS and NSAS examples, and (2) analysis of SAS in front of NSAS to observe possible variations (i.e., due to the presence of the self-adaptive part) in the occurrences of the quality issues. All SAS and NSAS are written in Java freely available and/or open source.

The rest of the paper is organized as follows. Section 2 presents our study design. Results are shown in Section 3. Section 4 addresses the threats to validity. Section 5 presents the conclusions and future work.

2 STUDY DESIGN

2.1 Context

In this paper, we analyze SAS by identifying code and architectural smells, and design patterns. We also compare SAS with NSAS, which are considered here as a quality reference. To achieve this objective, we propose the following research questions (RQ):

RQ1: How can SAS be evaluated in terms of code smells?

RQ2: How can SAS be evaluated in terms of architectural smells?

RQ3: How can SAS be evaluated in terms of design patterns?

RQ4: Which is the difference in terms of code smells and architectural smells occurrence/presence in SAS and NSAS?

RQ5: Which is the difference in terms of design patterns occurrence/presence in SAS and NSAS?

The answer to RQ1 and RQ2 may be exploited by developers/maintainers to have an idea of the most common code smells and architectural smells in SAS and focus their attention on the refactoring of these smells. The answer to RQ3 helps to understand some of the design decisions (due to the intent of the design patterns) and to further reuse or extend SAS.

The answer to RQ4 may indicate hints on the prevalence of some smells with respect to other according to SAS and NSAS. A developer can pay particular attentions to the most frequent ones. Some smells may be generated by the intrinsic nature of a category of systems, such as SAS; thus, such smells may be considered as false positive. In this case, developers may consider to implement filters to remove the false positive smells in the detection process.

The answer to RQ5 provides an overview of the design patterns commonly used by SAS and NSAS. This result may be used to address further design issues in SAS already experienced in NSAS. In case of differences, this result is a valuable indicator of the specific features and design solutions for SAS.

2.2 Analyzed Systems

For the analysis of the quality of SAS in front of NSAS we have chosen 6 examples from each category. All examples are available on Web sites and repositories which collect systems and make them available for usage, empirical studies, results comparison, or other research activities.

SAS considered in this paper are available at the *Exemplars* web site (i.e., https://www.hpi.uni-potsdam.de/giese/public/selfadapt/exemplars/):

- Adasim is a simulator for the Automated Traffic Routing Problem (ATRP). Adaptivity addresses scalability and environmental changes.
- DeltaIoT enables the evaluation and comparison of self-adaptive mechanisms for Internet of Things (IoT). It addresses performance and energy consumption issues.
- Intelligent Ensembles (IE) is a framework for dynamic cooperation groups (e.g.,, smart cyber-physical systems, smart mobility, smart cities).
- Lotus (Lotus@Runtime) uses models@runtime to monitor the execution traces and to verify reachability properties of SAS at runtime.
- Rainbow supports the development of SAS through architectural mechanisms, i.e., feedback control loops based on the MAPE-K (monitor, analyze, plan, execute, using knowledge) steps.
- Tele Assistance System (TAS) is a service-based health-care application for distance assistance. Self-adaptivity is used for uncertainties generated by third-party services (e.g., service failure).

A summary of SAS considering the architectural model used, the adaptive mechanisms, the application domain, the year of their publication, and the number of Java classes is shown in Table 1.

NSAS considered in this paper are available in the *QualitasCorpus*[1] and in the *MavenRepository*[2] sites:

- Apache ANT is a Java library which provides a series of automatisms that allow the programmers to compile, test, and run Java applications.
- Apache PDFBOX is an open source library that can be used to create, render, print, split, merge, edit, and extract text and metadata from PDF files.
- Cobertura calculates the percentage of Java code that can be covered by test implementation. It is based on JScoverage.

[1] http://qualitascorpus.com/
[2] https://mvnrepository.com/

- JHotDraw is a Java GUI framework for graphic design and object manipulation projects.
- ProGuard is a software that optimizes, reduces, and obscures Java code.
- Sunflow is a rendering system for the synthesis of photorealistic images through the implementation of global illumination algorithms.

A summary of NSAS considering their type and main objective, the number of their Java classes, and the year of the last version is shown in Table 2.

2.3 Collected Data

We present the data we collected on the SAS and NSAS described in Section 2.2. The data concerns code smells, architectural smells, and design patterns.

2.3.1 Code Smells

Code smells are indicators of possible problems at the code or design level (e.g., large classes or methods) (Fowler, 1999). They provide hints on parts of code which may be characterized by a poor quality, and may lead to negative effects on the maintenance and evolution of the software. The definition and details of the code smells considered in our study are available on the plug-in[3] Web site for their detection. They are briefly introduced as follows:

- *AntiSingleton (AS)*: a class that provides mutable class variables, which consequently could be used as global variables.
- *BaseClassKnowsDerivedClass (BCKDC)*: a class that invokes or has at least binary-class relationship pointing to one of its subclasses.
- *BaseClassShouldBeAbstract (BCSBA)*: a class with many subclasses without being abstract.
- *Blob (B)*: a large controller class that depends on data stored in surrounding data classes. A large class declares many fields and methods with a low cohesion.
- *ClassDataSouldBePrivate (CDSBP)*: a class that exposes its fields, thus violating the principle of encapsulation.
- *ComplexClass (CC)*: a class that has (at least) one large and complex method, in terms of cyclomatic complexity and lines of code.
- *FunctionalDecomposition (FD)*: a main class, i.e., a class with a procedural name, such as Compute or Display, in which inheritance and polymorphism are scarcely used, that is associated with small classes, which declare many private fields and implement only few methods.
- *LargeClass (LC)*: a class that has grown too large in term of lines of code.
- *LazyClass (LzzC)*: a class that has few fields and methods.
- *LongMethod (LM)*: a class that has (at least) a very long method in terms of lines of code.
- *LongParameterList (LPL)*: a class that has (at least) one method with a too long list of parameters in comparison to the average number of parameters per methods in the system.
- *ManyFieldAttributesButNotComplex (MFABNC)*: a class that declares many attributes but which is not complex and, hence, more likely to be a kind of data class holding values without providing behaviour.
- *MessageChains (MC)*: a class that uses a long chain of method invocations to implement (at least) one of its functionality.
- *RefusedPatternBequest (RPB)*: a class that redefines inherited methods using empty bodies, thus breaking polymorphism.
- *SpaghettiCode (SC)*: a class with no structure, declaring long methods with no parameters, and using global variables.
- *SpeculativeGenerality (SG)*: a class that is defined as abstract having few children, which do not make use of its methods.
- *SwissArmyKnife (SAK)*: a complex class that offers a high number of services, e.g., a complex class implementing a high number of interfaces.
- *TraditionBreaker (TB)*: a class that inherits from a large parent class but that provides little behaviour and without subclasses.

2.3.2 Architectural Smells

An architectural smell results from a common architectural decision, intentional or not, that negatively impacts the internal software quality (Garcia et al., 2009; Macia et al., 2012). The architectural smells considered in our study are:

- *Unstable Dependency (UD)*: describes a subsystem (component) that depends on other subsystems less stable than itself. UD is detected on packages.

[3]https://github.com/davidetaibi/sonarqube-anti-patterns-code-smells

Table 1: SAS Systems.

SAS	Architectural Model	Adaptive Mechanisms	Application Domain	Year	No. of Classes
Adasim	Agent-based	Parameter-based	Automated Traffic Routing	2012	64
DeltaIoT	IoT four tier	Parameter-based	Smart University Campus	2017	82
IE	Cyber-Physical	Architecture-based Parameter-based	Smart Cities/Mobility	2017	825
Lotus	Component-based	Parameter-based	Travel Planner	2017	51
Rainbow	Component-and-connector	Architecture-based	News Web Site	2005	1707
TAS	Service-based	Service dynamic composition	Healthcare	2012	765

Table 2: NSAS Systems.

NSAS	Type	Objective	Year	No. of Classes
ANT	Java library	Build Java and non-Java applications (e.g., compile, test, run)	2018	962
PDFBOX	Java library	Create and manipulate PDF files	2018	388
Cobertura	Java tool	Calculate the percentage of code accessed by tests	2018	172
JHotDraw	Java GUI framework	Create technical and structural graphics	2018	346
ProGuard	Java tool	Optimize, reduce, and obscure code	2005	707
Sunflow	Photo rendering system	Synthesize photo-realistic images through global illumination algorithms	2017	209

- *Hub-Like Dependency (HL)*: arises when an abstraction has (outgoing and ingoing) dependencies with a large number of other abstractions (Suryanarayana et al., 2015). HL is detected on classes.
- *Cyclic Dependency (CD)*: refers to a subsystem involved in a chain of relations that breaks the acyclic nature of a subsystem dependency structure. CD is detected on classes.

We have considered these architectural smells since they represent critical problems related to dependency issues. Components highly coupled and with a high number of dependencies cost more to maintain and hence can be considered more critical.

2.3.3 Design Patterns

Design patterns may provide significant hints on the development and quality of a system by capturing indications about the design decisions due to their semantic. They provide enhanced and verified solutions to common design problems (Gamma et al., 1994). Their detection may be very useful for the understanding, maintaining, and evolving a system (Arcelli Fontana et al., 2011; Arcelli Fontana et al., 2013). Here, we have considered the following 13 GoF's design patterns (Gamma et al., 1994):

- *Creational*: Factory Method, Prototype, Singleton.
- *Structural*: Bridge, Composite, Decorator, Object Adapter, Command, Proxy.
- *Behavioral*: Chain of Responsibility, Observer, State-Strategy (they are considered together because of their identical structure; only their behavior is different), Template Method, Visitor.

State and Strategy have identical structures with different behaviors; in this study they are considered as a single pattern.

2.4 Tools

To collect the data described above we employed the following tools: (1) SonarQube[4] and an external plug-in[5] to collect the code smells (Raibulet and Arcelli Fontana, 2018; Lenarduzzi et al., 2019), (2) Arcan[6] to collect architectural smells (Arcelli Fontana et al., 2017), and (3) DPDT (Design Pattern Detection Tool)[7] to detect patterns (Tsantalis et al., 2006).

[4]https://www.sonarqube.org

[5]https://github.com/davidetaibi/sonarqube-anti-patterns-code-smells

[6]https://http://essere.disco.unimib.it/wiki/arcan

[7]https://users.encs.concordia.ca/~nikolaos/pattern_detection.html

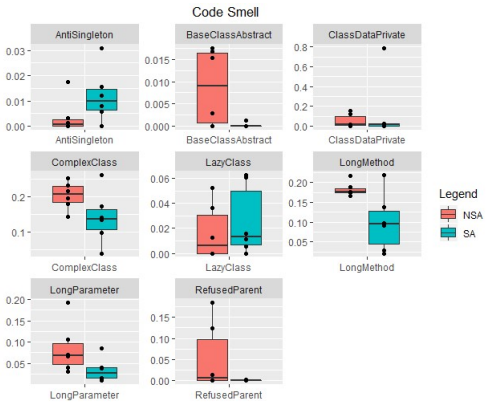

Figure 1: Box-Plot Results of Code Smells.

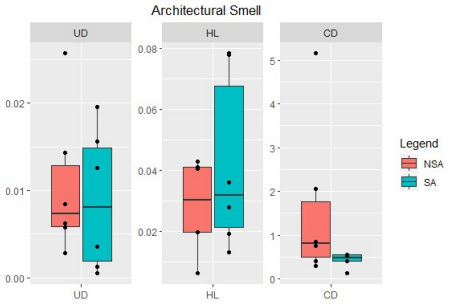

Figure 2: Box-Plot Results of Architectural Smells.

3 RESULTS

The results concerning the collected data about SAS and NSAS shown in the tables in this section, have been normalized based on the size, i.e., number of classes or packages of each system so that the various observations and comparisons can be made easily.

3.1 Results on Code Smell Detection

The code smells revealed in SAS and NSAS are summarized in Table 3. 10 out of 18 code smells have not been detected in SAS. The most detected code smells in SAS are: ComplexClass, LongMethod, and LongParameterList. Also AntiSingleton and LazyClass are present in almost all SAS (except Lotus). The less present code smells are: BaseClassShouldBeAbstract and RefusedParentBequest (detected only in IE).

Looking at NSAS, there is one code smell present, i.e., SwissArmyKnife in addition to the ones detected in SAS. Also in NSAS, the most present code smells are: ComplexClass, LongMethod, and LongParameterList. The less present code smells are: AntiSingleton, LazyClass, and RefusedParentBequest.

Figure 1 shows the box-plots associated to the values in Table 3. AntiSingleton, LazyClass, and LongMethod occur more frequently in SAS than in NSAS. BaseClassShouldBeAbstract, ClassDataShouldBePrivate, and RefusedParentBequest occur seldom in SAS, while are frequent in NSAS.

3.2 Results on Architectural Smell Detection

The architectural smells detected in SAS and NSAS are summarized in Table 4. All are present in all SAS and NSAS. From the results in Table 4 and Figure 2, the most interesting data concern the disparity between the number of CD present in SAS and NSAS. The high number of CD is harmful: a single change or bug in a package/class can have a strong effect on many others parts of the project, due to the cyclic dependencies.

UD and HL are more present in SAS than NSAS. Their greater presence can be traced back to the intrinsic nature of SAS. UD, in fact, describes a subsystem that depends on other subsystems that are less stable than itself. The presence of instability is presumably due to the fact that SAS must adapt and change their behavior based on changing internal or external system variables, this means that the various classes and packages of the system have to be able to maintain the highest possible degree of flexibility. Also the number of HD smells is a little higher because SAS operate in dynamic environments and interact with other components; they need to be able to analyze and adapt to the changes in the surrounding environment simultaneously to ensure the quality of the provided services.

3.3 Results on Design Pattern Detection

The creational design patterns detected in the SAS and NSAS are summarized in Table 5. There are three SAS with no creational patterns: Adasim, DeltaIoT, and Lotus. The most detected pattern is Singleton. This may be mostly due to the implementation of the MAPE-K managers. The less used pattern is Prototype. Looking at NSAS, most of them implement at least a couple of creational patterns. Also in NSAS, the most detected pattern is Singleton, while Prototype occurs only in JHotDraw.

The structural design patterns detected in the SAS and NSAS are summarized in Table 6. The most detected structural pattern in all SAS is Adapter. Composite occurs in Lotus and Rainbow, while Proxy has been detected in Rainbow. Looking at NSAS, the most detected pattern is Adapter, while the less detected are Composite and Proxy.

The behavioral design patterns detected in SAS

Table 3: Code Smells Detected in SAS and NSAS.

SAS NAME	Adasim	DeltaIoT	IE	Lotus	Rainbow	TAS
AntiSingleton	0.0310	0.0121	0.0157	0	0.0058	0.0078
BaseClassShouldBeAbstract	0	0	0.0012	0	0	0
ClassDataShouldBePrivate	0	0.0243	0.0787	0	0.0117	0.0156
ComplexClass	0.1718	0.1341	0.1393	0.0980	0.0386	0.0261
LazyClass	0.0625	0.0609	0.0157	0	0.0011	0.0052
LongMethod	0.2187	0.0975	0.0921	0.1372	0.0287	0.0209
LongParameterList	0.0156	0.0853	0.0387	0.0392	0.0093	0.0156
RefusedParentBequest	0	0	0.0012	0	0	0
NSAS NAME	ANT	PDFBOX	Cobertura	JHotDraw	ProGuard	Sunflow
AntiSingleton	0.0031	0	0.0174	0	0.0014	0
BaseClassShouldBeAbstract	0.0176	0.0154	0	0.0028	0.0169	0
ClassDataShouldBePrivate	0.0093	0	0.1511	0.0056	0.1244	0.0191
ComplexClass	0.1434	0.2164	0.2325	0.1797	0.2531	0.1961
LazyClass	0.0363	0.0128	0.0523	0	0	0
LongMethod	0.1663	0.2164	0.1744	0.1882	0.1753	0.1770
LongParamaterList	0.0301	0.1932	0.0406	0.0702	0.0664	0.1052
RefusedParentBequest	0.1850	0.0128	0	0	0.1244	0
SwissArmyKnife	0.0010	0	0	0	0.0990	0

Table 4: Architectural Smell Detected in SAS and NSAS.

SAS NAME	Adasim	DeltaIoT	IE	Lotus	Rainbow	TAS
UD	0.0156	0.0126	0.0036	0.0196	0.0005	0.0013
HL	0.0781	0.0366	0.0278	0.0784	0.0193	0.0130
CD	0.5312	0.4024	0.4048	0.5490	0.5530	0.1241
NSAS NAME	ANT	PDFBOX	Cobertura	JHotDraw	ProGuard	Sunflow
UD	0.0062	0.0257	0.0058	0.7556	0.4031	5.1483
HL	0.0062	0.0412	0.0406	0.0196	0.0198	0.0430
CD	0.8492	2.0644	0.2965	0.7556	0.4031	5.1483

and NSAS are summarized in Table 7. There are three SAS (i.e., Adasim, DeltaIoT, and Lotus) implementing only one behavioral pattern, i.e., State-Strategy. The most detected behavioral pattern is State-Strategy, while the less used are Chain of Responsibility and Visitor. Looking at NSAS, the most detected behavioral patterns are State-Strategy and Template Method, while the less detected ones are Chain of Responsibility and Visitor.

Figure 3 shows the box-plots of the detection of patterns grouped in categories. The use of creational patterns is similar, while SAS implement less structural and behavioral patterns than NSAS.

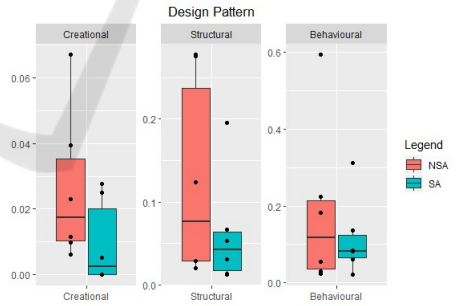

Figure 3: Box-Plot Results for Design Patterns.

4 THREATS TO VALIDITY

Different threats can be considered in this study. First, we analyzed a small number of projects made available by the ICSE-SEAMS community. We work on extending the number of analyzed SAS as new exemplars are added every year (Raibulet et al., 2020).

According to the results validity of the tools, we used tools largely spread, e.g., SonarQube, or tools for which papers on the validity of their results have been published, e.g., DPDT (Tsantalis et al., 2006) and Arcan (Arcelli Fontana et al., 2017; Martini et al., 2018). For the detection of the design patterns, we have used DPDT, which detects 13 out of the 24 GoF's patterns. To detect all the GoF's patterns we need various tools, a single tool recognizes only a subset of patterns.

Table 5: Creational Design Patterns Detected in SAS and NSAS.

SAS NAME	Adasim	DeltaIoT	IE	Lotus	Rainbow	TAS
Factory Method	0	0	0.0060	0	0.0052	0.0013
Prototype	0	0	0	0	0.0005	0
Singleton	0	0	0.0218	0	0.0193	0.0039
NSAS NAME	ANT	PDFBOX	Cobertura	JHotDraw	ProGuard	Sunflow
Factory Method	0	0.0051	0	0.0084	0.0070	0.1913
Prototype	0	0	0	0.2528	0	0
Singleton	0.0062	0.0180	0.0116	0.0056	0.0028	0.0478

Table 6: Structural Design Pattern Detected in SAS and NSAS.

SAS NAME	Adasim	DeltaIoT	IE	Lotus	Rainbow	TAS
Object Adapter	0.0156	0.0121	0.0484	0.1764	0.0527	0.0052
Bridge	0.0156	0	0.0012	0	0.0046	0.0013
Composite	0	0	0	0.0196	0.0005	0
Decorator	0	0	0.0036	0	0.0076	0.0065
Proxy	0	0	0	0	0.0011	0
NSAS NAME	ANT	PDFBOX	Cobertura	JHotDraw	ProGuard	Sunflow
Object Adapter	0.0166	0.0154	0.0290	0.0730	0.1145	0.2344
Bridge	0.0010	0	0	0.0365	0.0014	0.0047
Composite	0	0	0	0.0056	0.0099	0
Decorator	0.0010	0	0	0.0117	0.1131	0.0239
Proxy	0	0.0051	0	0	0.0248	0.0143

5 CONCLUDING REMARKS

In this paper we analyzed which code and architectural smells, and design patterns may be detected in six SAS and outlined possible differences of the presence of these issues in SAS and NSAS. We provide below the answers to the RQ defined in Section 2.

***RQ1*:** Less than half of the considered code smells have been detected. In our opinion this is a positive result. The most present code smells are: ComplexClass, LongMethod, and LongParameterList. A possible explanation of this presence is that some classes have management roles for the steps of the MAPE-K loop and have more complex and long methods. The less present code smells are BaseClassKnowsDerivedClass and RefusedParentBequest.

***RQ2*:** All the considered architectural smells have been found in SAS, in particular UD and HD. Probably some of these smells are present for some specific features of SAS; thus, for the detection of these smells in SAS, it may be useful to implement filters to remove possible false positive instances.

***RQ3*:** All the design patterns have been detected in SAS and all SAS implement pattern. In particular, Object Adapter and State-Strategy patterns are present in all the analyzed SAS. The two patterns suit well with self-adaptivity because they enable structural and behavioral modifications at runtime.

***RQ4*:** The detected code smells is similar in SAS and NSAS. In NSAS, we detected also the SwissArmyKnife smell. The presence of ComplexClass, LongMethod, and LongParameterList should be avoided in both SAS and NSAS. Further, architectural smells are present both in SAS and NSAS: more CD in NSAS, and more UD and HD in SAS. False positive instances may be present in SAS, but they may not represent real quality issues.

***RQ5*:** The results on pattern detection are similar in SAS and NSAS. The most used patterns are Singleton, Adapter, and State-Strategy; the less applied patterns are Prototype, Composite, Proxy, Chain of Responsibility, and Visitor. However, the number of instances of patterns detected in SAS and NSAS differs (also because of the dimension of SAS): there are almost four times more structural, and three times more beavioural instances in NSAS than in SAS.

In the future, we aim to extend our analysis to a larger set of systems and to consider other software quality metrics. We plan to analyze how the refactoring of the different kinds of smells may impact on a set of software quality metrics. A future work will concern the identification of smells or anti-patterns, as well as design patterns specific to SAS. The automate detection of self-adaptive related issues is not yet sup-

Table 7: Behavioural Design Patterns Detected in SAS and NSAS.

SAS NAME	Adasim	DeltaIoT	IE	Lotus	Rainbow	TAS
Chain of Responsibility	0	0	0	0	0	0.0039
Observer	0	0	0.0763	0	0.0082	0
State-Strategy	0.0312	0.0609	0.0084	0.1372	0.0500	0.0013
Template Method	0	0	0	0	0.0222	0.0156
Visitor	0	0	0	0	0.0005	0
NSAS NAME	ANT	PDFBOX	Cobertura	JHotDraw	ProGuard	Sunflow
Chain of Responsibility	0	0	0	0	0.0028	0.0095
Observer	0.0031	0	0	0.0056	0.1032	0
State-Strategy	0.0155	0.0128	0.0232	0.1432	0.1060	0.1722
Template Method	0.0051	0.0154	0.0058	0.0337	0.0198	0.0813
Visitor	0	0.0257	0	0	0.0919	0.3301

ported by tools, as far as concerns our knowledge.

REFERENCES

Arcelli Fontana, F., Maggioni, S., and Raibulet, C. (2011). Understanding the relevance of micro-structures for design patterns detection. *Journal of Systems and Software*, 84(12):2334–2347.

Arcelli Fontana, F., Maggioni, S., and Raibulet, C. (2013). Design patterns: a survey on their micro-structures. *Journal of Software: Evolution and Process*, 25(1):27–52.

Arcelli Fontana, F., Pigazzini, I., Roveda, R., Tamburri, D. A., Zanoni, M., and Nitto, E. D. (2017). Arcan: A tool for architectural smells detection. In *Intl Conf on Software Architecture Workshops, Sweden, April 5-7, 2017*, pages 282–285.

Chatzigeorgiou, A. and Manakos, A. (2010). Investigating the Evolution of Bad Smells in Object-Oriented Code. In *2010 Seventh Int'l Conf. the Quality of Information and Communications Tech*, pages 106–115. IEEE.

de Lemos, R., Giese, H., Müller, H. A., and Shaw, M., editors (2013). *Software Engineering for Self-Adaptive Systems II - Intl Seminar, Dagstuhl Castle, Germany, October 24-29, 2010 Revised Selected and Invited Papers*, volume 7475 of *LNCS*. Springer.

Fowler, M. (1999). *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, Boston, MA, USA.

Gamma, E., Helm, R., Johnson, R. E., and Vlissides, J. M. (1994). *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley.

Garcia, J., Popescu, D., Edwards, G., and Medvidovic, N. (2009). Identifying architectural bad smells. In *Conference on Software Maintenance and Reengineering*, pages 255–258, Germany. IEEE.

Kaddoum, E., Raibulet, C., Georgé, J., Picard, G., and Gleizes, M. P. (2010). Criteria for the evaluation of self-* systems. In *2010 ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems, South Africa, May 3-4, 2010*, pages 29–38.

Lenarduzzi, V., Lomio, F., Taibi, D., and Huttunen, H. (2019). On the fault proneness of sonarqube technical debt violations: A comparison of eight machine learning techniques. *CoRR*, abs/1907.00376.

Macia, I., Arcoverde, R., Cirilo, E., Garcia, A., and von Staa, A. (2012). Supporting the identification of architecturally-relevant code anomalies. In *Proc. 28th IEEE Int'l Conf. Software Maintenance (ICSM 2012)*, pages 662–665, Trento, Italy. IEEE.

Martini, A., Arcelli Fontana, F., Biaggi, A., and Roveda, R. (2018). Identifying and prioritizing architectural debt through architectural smells: a case study in a large software company. In *European Conf. on Software Architecture, Spain*, pages 320–335.

Peters, R. and Zaidman, A. (2012). Evaluating the Lifespan of Code Smells using Software Repository Mining. In *2012 16th European Conf. Softw. Maintenance and ReEng.*, pages 411–416. IEEE.

Raibulet, C. and Arcelli Fontana, F. (2017). Evaluation of self-adaptive systems: a women perspective. In *11th European Conf on Software Architecture, UK, September 11-15, 2017*, pages 23–30.

Raibulet, C. and Arcelli Fontana, F. (2018). Collaborative and teamwork software development in an undergraduate software engineering course. *Journal of Systems and Software*, 144:409–422.

Raibulet, C., Arcelli Fontana, F., and Carettoni, S. (2020). A preliminary analysis and comparison of self-adaptive systems according to different issues. *Software Quality Journal*, In press.

Ramirez, A. J. and Cheng, B. H. C. (2010). Design patterns for developing dynamically adaptive systems. In *ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems, South Africa*, pages 49–58.

Suryanarayana, G., Samarthyam, G., and Sharma, T. (2015). *Refactoring for Software Design Smells*. Morgan Kaufmann, 1 edition.

Tsantalis, N., Chatzigeorgiou, A., Stephanides, G., and Halkidis, S. T. (2006). Design pattern detection using similarity scoring. *IEEE Transaction on Software Engineering*, 32(11):896–909.

Walter, B. and Alkhaeir, T. (2016). The relationship between design patterns and code smells: An exploratory study. *Information and Software Technology*, 74:127 – 142.

Weyns, D. (2018). Software engineering of self-adaptive systems: An organized tour and future challenges. *Elsevier*, 19(1-–12):888–896.