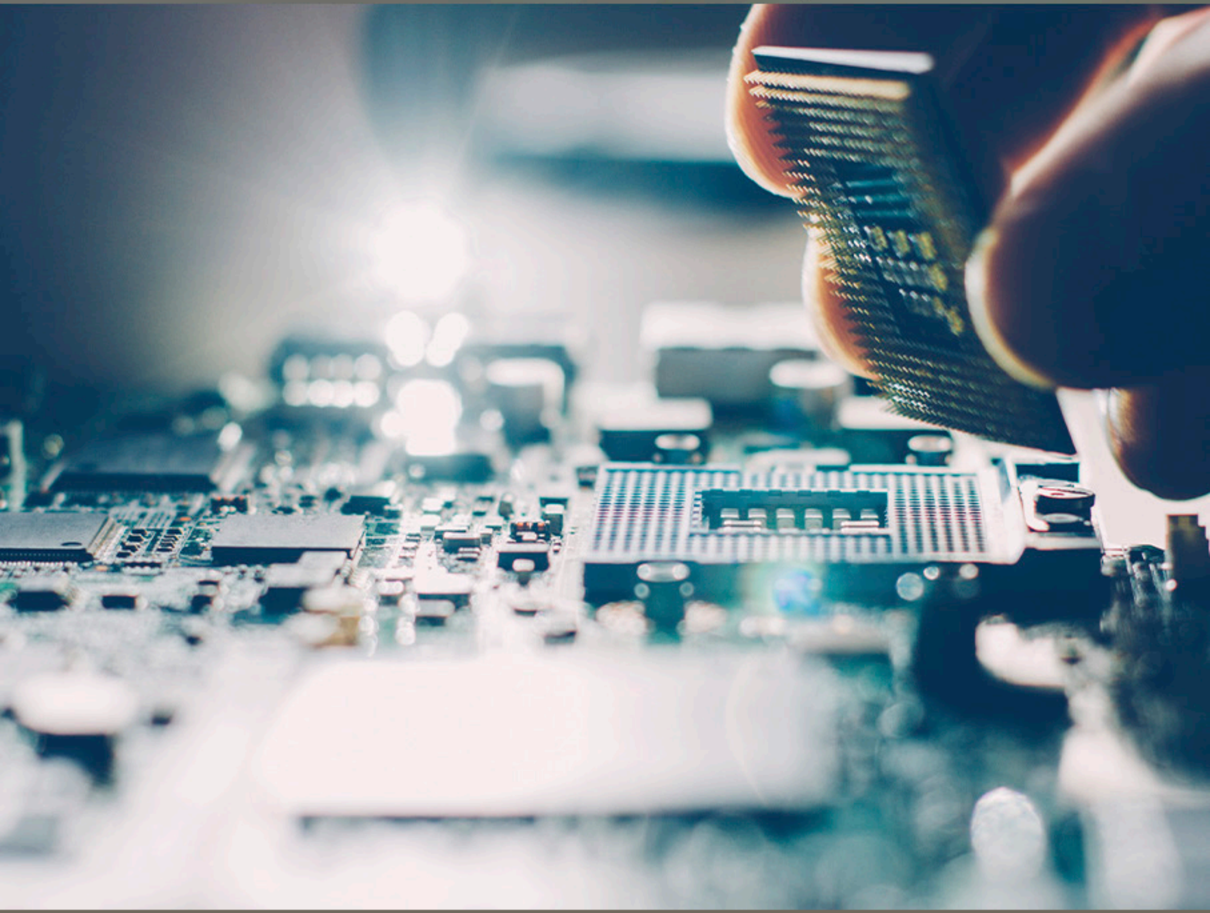
COLEÇÃO

DESAFIOS DAS ENGENHARIAS:

ENGENHARIA DE COMPUTAÇÃO



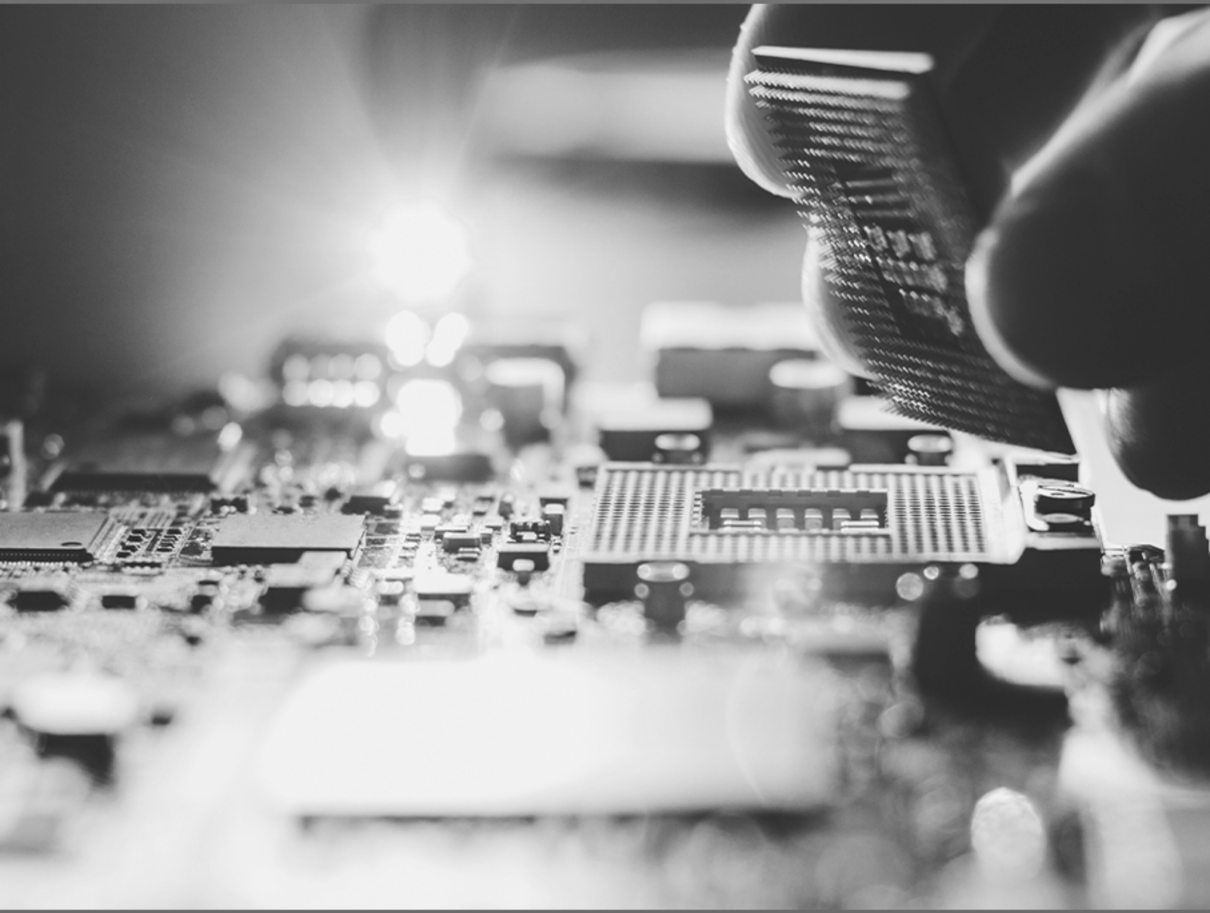
ERNANE ROSA MARTINS
(ORGANIZADOR)

Atena
Editora
Ano 2021

COLEÇÃO

DESAFIOS DAS ENGENHARIAS:

ENGENHARIA DE COMPUTAÇÃO



ERNANE ROSA MARTINS
(ORGANIZADOR)

Atena
Editora
Ano 2021

Editora chefe
Profª Drª Antonella Carvalho de Oliveira
Assistentes editoriais
Natalia Oliveira
Flávia Roberta Barão
Bibliotecária
Janaina Ramos
Projeto gráfico
Natália Sandrini de Azevedo
Camila Alves de Cremo
Luiza Alves Batista
Maria Alice Pinheiro
Imagens da capa
iStock
Edição de arte
Luiza Alves Batista
Revisão
Os autores

Atena
Editora
Ano 2021

Ciências Exatas e da Terra e Engenharias

Coleção desafios das engenharias: engenharia de computação

Diagramação:	Camila Alves de Cremo
Correção:	Maiara Ferreira
Indexação:	Gabriel Motomu Teshima
Revisão:	Os autores
Organizador:	Ernane Rosa Martins

Dados Internacionais de Catalogação na Publicação (CIP)

C691 Coleção desafios das engenharias: engenharia de computação / Organizador Ernane Rosa Martins. – Ponta Grossa - PR: Atena, 2021.

Formato: PDF
Requisitos de sistema: Adobe Acrobat Reader
Modo de acesso: World Wide Web
Inclui bibliografia
ISBN 978-65-5983-387-0
DOI: https://doi.org/10.22533/at.ed.870211808

1. Engenharia da computação. I. Martins, Ernane Rosa (Organizador). II. Título.

CDD 621.39

Elaborado por Bibliotecária Janaina Ramos – CRB-8/9166

Atena Editora
Ponta Grossa – Paraná – Brasil
Telefone: +55 (42) 3323-5493
www.atenaeditora.com.br
contato@atenaeditora.com.br

Atena Editora
Ano 2021

DECLARAÇÃO DOS AUTORES

Atena Editora
Ano 2021

DECLARAÇÃO DA EDITORA

Atena
Editora
Ano 2021

APRESENTAÇÃO

A Engenharia de Computação tem como definição ser o ramo da engenharia que se caracteriza pelo projeto, desenvolvimento e implementação de sistemas, equipamentos e dispositivos computacionais, segundo uma visão integrada de hardware e software, apoiando-se em uma sólida base matemática e conhecimentos de fenômenos físicos. O objetivo é a aplicação das tecnologias de computação na solução de problemas de Engenharia.

Deste modo, este livro, aborda diversos aspectos tecnológicos computacionais, tais como: o desenvolvimento de um jogo de RPG acessível em LIBRAS; uma reflexão quanto à necessidade de aplicação de supressores de surto como proteção de transformadores devido a eventos transitórios em manobras de disjuntores; um algoritmo para geração de contorno 2D envolvendo regiões irregulares; avaliação da influência das tensões residuais e imperfeições geométricas iniciais em colunas de aço submetidas à flexão em torno do eixo de menor inércia; os esforços em estruturas laminares, de características de geometria e carregamentos diversos através da implementação computacional de um elemento finito sólido hexaédrico de 8 nós programado com uma linguagem computacional de alto nível; uma análise computacional realizada através do programa SAP2000; a estabilidade e as vibrações de anéis e tubulações apoiados em uma fundação elástica de Pasternak; um controlador neural para dois elos de um robô manipulador de três graus de liberdade (3 GDL); uma ferramenta de autoria para livros relacionados a área da educação; um aplicativo com propósito de aumentar a taxa de reciclagem e minimizar os danos ambientais devido ao descarte incorreto de resíduos na natureza; a conscientização de crianças e adolescentes sobre as ocorrências de bullying; uma aplicação web interativa, de fácil utilização e interface amigável, por meio do pacote Shiny, destinada aos tópicos de intervalo de confiança e dimensionamento de amostra para o parâmetro proporção; segmentar e detectar, por meio de redes neurais convolutivas, as pás dos raspadores de escória em panelas de ferro gusa do Reator Kambara de uma siderúrgica; integrar a Biblioteca Digital de Artigos (IFPublica) e a Plataforma de Digital de Inscrição e Administração de Projetos (PDIAP), por meio de adaptações nos dois projetos, para impedir erros humanos e automatizar o processo de cadastro de artigos do PDIAP na base de dados do IFPublica.

Assim, espero que a presente obra venha a se tornar um guia aos estudantes e profissionais da área de Engenharia de Computação, auxiliando-os em diversos assuntos relevantes da área, fornecendo a estes novos conhecimentos para poderem atender as necessidades informacionais, computacionais e de automação das organizações de uma forma geral. Por fim, agradeço aos autores por suas contribuições na construção desta importante obra e desejo muito sucesso a todos os nossos leitores.

Ernane Rosa Martins

SUMÁRIO

CAPÍTULO 19

DESIGN PATTERNS FOR SOFTWARE EVOLUTION REQUIREMENTS

Data de aceite: 02/08/2021
Data de submissão: 06/07/2021

Anna Grimán Padua
Process and Systems Department, Simón Bolívar University
Caracas - Venezuela
https://orcid.org/0000-0002-9092-6952

Manuel Capel Tuñón
Software Engineering Department, College of Informatics and Telecommunications, University of Granada
Granada - Spain
https://orcid.org/0000-0003-2449-4394

Eladio Garví
Software Engineering Department, College of Informatics and Telecommunications, University of Granada
Granada - Spain
https://orcid.org/0000-0002-9267-1711

ABSTRACT: The Software Engineering term known as Software Evolution can be understood in two senses. First, as the changes experienced by software over the development cycle; second, software changes while the system is in service. In both cases, software architectures should lead, support and ease any modifications, reconfiguration or adaptation to a changing environment, of software systems. We present here the analysis of several design and architectural patterns for sustaining software systems evolution according to two perspectives: one connected with maintainability and the other with dynamicity of changes of any software design. We also study weaknesses and benefits, aligned with ISO 25010 standard characteristics, of each selected pattern in such way that they can easily be analyzed as candidate in a context of architectural improvement. At moment, it is widely acknowledged that design and architectural patterns should be used for software development focused on quality. The analysis conducted on patterns facilitates software evolution by easing software changes throughout software development phases and further maintenance of the final product.

KEYWORDS: Maintainability, Architectural Patterns, Design Patterns, Software Quality Standard, Dynamic Reconfiguration.

PADRÕES DE DESIGN PARA REQUISITOS DE EVOLUÇÃO DE SOFTWARE

RESUMO: O termo engenharia de software conhecido como Evolução de Software pode ser entendido em dois sentidos. Primeiro, como as mudanças experimentadas pelo software ao longo do ciclo de desenvolvimento; segundo, o software muda enquanto o sistema está em serviço. Em ambos os casos, as arquiteturas de software devem liderar, apoiar e facilitar quaisquer modificações, reconfiguração ou adaptação a um ambiente em mudança, de sistemas de software. Apresentamos aqui a análise de diversos padrões de design e arquitetônicos para sustentar a evolução dos sistemas de software de acordo com duas perspectivas, uma conectada à

manutenção e outra com dinâmica de mudanças de qualquer design de software. Também estudamos fraquezas e benefícios, alinhados com as características padrão ISO 25010, de cada padrão selecionado de forma que possam ser facilmente analisados como candidatos em um contexto de melhoria arquitetônica. No momento, é amplamente reconhecido que o design e os padrões arquitetônicos devem ser usados para o desenvolvimento de software focado na qualidade. A análise conduzida em padrões facilita a evolução do software, facilitando as mudanças de software ao longo das fases de desenvolvimento de software e posterior manutenção do produto final.
PALAVRAS-CHAVE: Manutenção de software, Padrões Arquitetônicos, Padrões de Design, Padrão de Qualidade de Software, Reconfiguração Dinâmica.

1 I INTRODUCTION

Over the last two decades and after publishing seminal works as Bass et al (2003), Software Architecture (SA) has increasingly created interest. Researchers and ICT professionals have come to acknowledge the paramount importance of SA in software's quality characteristics (Boch J., 2000; Grimán et al., 2006; Grimán et al., 2006b). On the other part, some selected methods and techniques have been defined to early assess such characteristics through an architectural view (Kazman et al., 1996).

Design Patterns, Architectural Styles and Patterns are particular cases of architectural mechanisms to be considered in software development. Firstly proposed in the seminal work conducted by Gamma et al. (1996), these architectural mechanisms have experienced a great use since then due to Object-Oriented Programming and concurrent and distributed systems dissemination. Some studies have been carried out until now to assess the effect of design patterns to software quality, mainly through empirical methods (Gommathy, 2008; Harrison & Avgeriou, 2007; Bode & Riebish, 2010; Ozkaya et al, 2007), but their conclusions with regard to quality characteristics show different and controversial results. Moreover, we only found few articles in the literature of SA regarding the importance of design patterns for successful software evolution (Lung et al, 1997).

The main objective of this paper is an initial proposal for reviewing the most popular GoF design patterns to determine their usefulness in evolutionary software and dynamic architectures. Our baseline here is to analyze each pattern, within a set of great use in industry nowadays, regarding quality characteristics according to ISO 25010 standard (ISO/IEC, 2011).

Since our study is deeply based on design and architectural patterns, we needed to identify a significant subset of GoF patterns, which analysis could give us a good foundation to predict quality characteristics of many software applications that normally make use of these patterns.

Section 2 is devoted to introduce some previous background on the impact that patterns have in the quality of evolutive software and then, in Section 3, the selection of the set of patterns for our study is explained.

Then, we will present our approach and show that is capable of identifying the important benefits and “sensibility points” that can propitiate some software quality characteristics and inhibit others at the same time.

In Section 4 a set of Design Patterns are presented in order to analyze their main properties and to conclude different degrees of maintainability for software systems that use each one of these patterns. Section 5 presents a second set of patterns, in this case architectural patterns, in order to study their distinct support degree of dynamic software architecture reconfiguration. Finally, the future work and conclusions are presented in Section 6.

2 | BACKGROUND

The basic idea behind this research is that the desired quality of any software piece can be firstly attained by the level of concordance between selected architectural decisions and the functional and supplementary requirements of the system being modeled. Among the diverse architectural decisions that software architects can make, the usage of certain design patterns and styles is of interest for this investigation because these are capable of providing software quality.

On other hand, we will name evolutionary software the software that goes through different versions over the development process. Different versions can be also understood as successive versions of the same software, which perform changes correspondent to the product specification either by adding or removing requirements. The latter fact brings into focus the necessity of establishing the quality characteristics associated to software evolution. In order to achieve this goal, we will use the acknowledged ISO 25010 quality standard.

Our investigation is based on a previous study conducted by Grimán et al (2006), that we have adapted to the alignments established by the standard above.

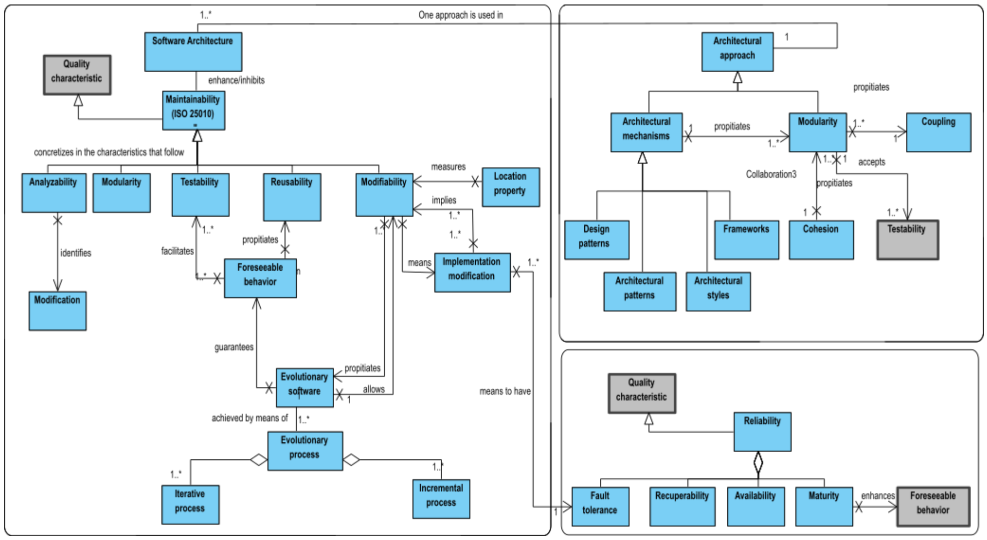


Fig. 1. Ontology for Maintainability as defined by ISO 25010 Model. Adapted from Grimán et al (2006)

In figure 1 we can see the ISO 25010based ontology of “evolutionary software”, where we can observe that evolutionary requirements of software can be expressed in terms of *Maintainability* and *Reliability* characteristics. This ontology allow us to establish metrics, which are inspired on the subcharacteristics of software quality related to software evolution and disclosed in the contributed work (Grimán et al, 2006).

In order to analyze the level of concordance between architectural decisions and system requirements, we firstly need to establish the quality characteristics associated to the evolution of a software product. In order to achieve that objective, we use the acknowledged ISO 25010 quality standard quality (ISO/IEC, 2011).

We needed a set of patterns from which we can derive the quality attributes that each one enhances or inhibits, then we will translate these into ISO 25010 quality characteristics. In doing so, we will need to follow a critical analysis process in order to assess quality characteristics of each selected pattern: a) to select a relevant set of design and architectural patterns; b) to identify the Maintainability subcharacteristics in which we want to focus on our valuation; c) to determine which design or architectural patterns can enhance and which ones can punish this subcharacteristic; and d) to recommend the usage of the well-assessed patterns.

3 | PATTERNS SELECTION

We were more interested in patterns whose scope will be objects, not a class, since the software that we are interested to maintain and make evolve is based on complex structures that need collaborative relations between objects. Among the set of 23 GoF patterns (Gamma et al, 1996) in Table 1, we selected the emphasized ones (blue) to conduct our study.

By purpose		Creational	Structural	Behavioral
By scope	Class	Factory method	Adapter (class)	Interpreter Template method
	Object	Abstract factory Builder Prototype Singleton	Adapter (object) Bridge Composite Decorator Façade Flyweight Proxy	Chain of responsibility Command Iterator Mediator Memento Observer State Strategy Visitor

Table 1. GoF Patterns (Gamma et al, 1996).

In current applications, objects get created and deleted dynamically in accordance with the real world, which is plenty of ubiquitous devices filled up with business logic and intelligence. *Factory method*, *Singleton*, *Adapter* (class), *Interpreter* and *Template method* are therefore not selected for analysis of quality characteristics either.

Secondly, Maintainability of any software architecture is transparent of those design patterns that only temporally store the state of an object or a software application (*Memento*), or those that only traverse a structure of objects (*Iterator*) or make easier representation and traversal of complex object hierarchies by recursive searches like DFS (*Composite*).

We did not include *Abstract Factory* (AF) design pattern as it leads to an appreciable code redundancy, since this pattern induces duplication of code and then causes further problems to build and later maintenance of software. FACTORY-classes and their descendants need to be changed if new products are needed in the catalogue, thus AF pattern shows lack of flexibility with respect to new inclusions of concrete classes' objects. Moreover, any future changes to be performed, or any error or fault correction that might arise, will impact to all the variants of the same code. As consequence, the use of AF turns the code complex and difficult to analyze or change. Thus, we decided not to assess the patterns: *Abstract Factory*, nor other creational patterns like *Builder*, *Prototype* and *Singleton* as a first approach of our study.

In our experience, when using *Observer* pattern error traceability becomes difficult if not impossible to carry out. This may seriously inhibit the attributes of Modifiability and Testability of any software. Thus, we can affirm that Testability and Recoverability sub-characteristic are affected. Testing any software that uses *Observer* with guarantee becomes quite complex. Thus, it is not considered for analysis in the first part of the study presented here.

On the other hand, patterns that encapsulate application's behavior, at an execution instant, by storing the internal state of the application objects make possible to change the behavior at run time in a flexible way (*State*, *Strategy*) without resorting to unnecessarily populate the code with conditional statements. The pattern *Strategy* therefore support dynamic reconfiguration at run time and greatly improve the maintainability of any software and thus.

In Sections 4-5, we present in a more systematic way the results of the analysis of the whole set of selected GoF design patterns.

Dynamic Architectural Reconfiguration is an important evolutionary aspect in software development nowadays, especially, for the requirements of self-adaptability, multicore load balancing and resilience to failures that collaborative applications present today.

The following patterns allow us to run time architecture reconfiguration: *Decorator*, *Strategy*, *Broker*, and *Bridge*. Additionally, we found some architectural patterns, which support a dynamic reconfiguration from a higher level of abstraction (Schmidt et al, 1999). We decided to incorporate a subset of these patterns with the purpose of observing changes in the maintainability metrics responses when introducing high-level architectural decisions. We selected the following patterns: *Component Configurator* (CC), *Interceptor*, *Extension Interface*, and *Reflection*. Thus, we conducted a theoretical and critical analysis of these 8 patterns in order to better complete our evaluation of patterns for evolutionary software.

4 | PATTERNS PROPITIATING HIGHER MAINTAINABILITY LEVELS

In this section, we present the analysis of a subset of the selected patterns. This first approach included 7 design patterns. Since this kind of patterns provide very detail information, we were able to analyze the impact of different low-level architectural decisions. A theoretical and critical analysis that includes a description of every pattern, their components and application, as well as the pros and cons of their use in terms of the quality characteristics according to ISO 25010 standard (ISO/IEC, 2011) has been carried in our study (see Tables 2-3). For each pattern and ISO characteristic, we identify whether a particular sub-characteristic of the produced software gets (*Impact*) enhanced (+) or inhibited (). Characteristic assessment has been carried by application of appropriate metrics based on the Maintainability ontology described in Figure 1 (Section 2).

For space reasons, following we briefly present the analysis for a single design pattern: **Wrapper-façade (WF)** pattern**.**

WF encapsulates functions and data provided by nonobject oriented existing APIs inside more concise, robust, portable, easytomaintain and cohesive interfaces.

Our aim when using this pattern is to avoid the deployment of conditional compilation directives and direct programming of APIs that make the code difficult to understand, debug, transfer and evolve. By the creation of a class named WF that encapsulates spare functions and data throughout nonobject oriented coded APIs, conditional compilation, scripts and other inclusions in the object-oriented base code become unnecessary. Thus, the methods defined in a WF interface are called instead of macro expansion.

After analyzing these properties, in Table 2 we can see a compilation of pattern's attributes and ISO quality characteristics that seem to get enhanced or inhibited if we use the pattern WF to design one part of any software. Notice that by using the pattern WF, the following quality characteristics and subcharacteristics were enhanced in the final software product: Maintainability/ modifiability, modularity, reusability; Performance/ resource usage; Compatibility/ coexistence, interoperability; Usability/ appropriateness, learnability, user error protection, user interface aesthetics; Functional suitability/ completeness, correctness, appropriateness.

Characteristic	Sub-characteristic	Impact	Metric
Maintainability	Testability	-	Dynamicity
	Modifiability	+	Cohesiveness
	Reusability	+	Reusable components existence
	Modularity	+ +	Cohesiveness Complexity (low)
	Analysability	+ -	Inheritance relationships and generic types Dynamicity
Performance	Response time	-	Contextual consistency
	Resource usage	+	Encapsulation
Portability	Flexibility	-	Compatibility
	Adaptability	-	Environment dependencies
Security	Integrity (*)	+ -	Cohesiveness Contextual consistency
Reliability	Maturity	+ + + - -	Encapsulation Complexity Cohesiveness Conformance to standards External dependencies
Compatibility	Co-existence	+	Coupling (low)
	Interoperability	+	Unified interfaces
Usability	Appropriateness	+	Simplicity
	Learnability	+	Complexity (low)
	Operability	-	Deployability
	User error protection	+	Cohesiveness of interfaces
	User interface aesthetics	+	Avoids APIs direct programming
	Accessibility	-	Flexibility

Functional suitability	Completeness	+	Analyzability
	Correctness	+	Compositionality of interfaces
	Appropriateness	+	Traceability

Table 2. Quality characteristics of Wrapper-Façade design pattern.

As a conclusion of the analysis presented in Table 2, we can affirm that software maintainability gets enhanced because the WF pattern creates cohesive and reusable components that can be interconnected inside container-components through inheritance relationships and generic types. Moreover, encapsulation and cohesiveness propitiate application stability since changes are mainly bounded to component interfaces and not to component implementation.

We can identify (Table 2) several ISO 25010 sub-characteristics to be connected to pattern WF. The majority of them are (complete or partially) enhanced by pattern WF and 6 of them get completely inhibited, i.e., Testability, Accessibility, Operability, Flexibility, Adaptability and Response Time.

Following, Table 3 summarizes the analysis results for the rest of the GoF selected patterns. Notice that those patterns promoting dynamic reconfiguration of software were included as a part of the analysis presented in section 5.

Pattern	Characteristic	Sub-characteristic	Impact	Metric
Adapter	Maintainability	Modifiability	+ -	Functional extension Interfaces comprehension and integrity
		Reusability	+	Reusable components
		Analysability	- -	Complexity Unexpected changes
	Performance Efficiency	Response time	+	Deployment
		Resource usage	+	Class hierarchies reduction
	Security	Integrity	+ -	Flexibility Reconfiguration
	Reliability	Fault tolerance	- -	Adaptor's inheritance Difficult error handling of ADAPTEES
		Maturity	+ +	Calls handling Encapsulation

Chain of Respon.	Maintainability	Modifiability	+	Unified interfaces
		Reusability	+	Low coupling
		Analysability	-	Dynamicity
	Performance	Response time	-	Response predictability
		Resource usage	+	Encapsulation
	Security	Integrity	+	Flexibility
			-	Reconfiguration
	Reliability	Fault tolerance	-	Lost calls
			+	Scalability
		Maturity	+	Coupling
			+	Accuracy
Mediator	Maintainability	Modifiability	+	Encapsulation
		Reusability	+	Cohesiveness
		Analysability	+	Coupling (low)
		Testability	-	Complexity (increasing)
	Performance	Resource usage	+	Encapsulation
	Portability	Interoperability	+	Contextual independence
	Security	Integrity	+	Consistency (internal, external)
Visitor	Maintainability	Modifiability	+	Encapsulation
		Analysability	+	Dynamicity
		Testability	+	Occurred events monitoring
	Security	Integrity	+	Consistency
	Performance	Resource usage	+	Abstract visiting draws code economy
		Response time	-	Class hierarchies extension
	Reliability	Fault tolerance	+	Error handling centralization
	Portability	Adaptability	+	Interoperability
Facade	Maintainability	Modifiability	+	Simplicity
		Analysability	+	Interfaces unification
		Testability	+	Encapsulation
	Portability	Adaptability	+	Interoperability
	Performance	Resource usage	+	Coupling (low)
		Response time	+	Simpler communications
	Security	Integrity	+	Encapsulation
	Reliability	Maturity	+	Encapsulation
Command	Maintainability	Modifiability	+	Unified interfaces
		Analysability	+	Auditing (logs)
			+	Simplicity
		Testability	+	Traceability
	Performance	Resource usage	+	Encapsulation

Table 3. Quality characteristics associated to the selected design patterns.

Based on the analysis presented in Table 3, we can affirm that software maintainability is total or partially supported by the whole set of selected patterns. At the same time, the following quality characteristics are promoted by these patterns: Performance, Security, Reliability, and Portability. On the other hand, we can observe in Table 3 that certain sub-characteristics (such as Analyzability and Fault Tolerance) got completely inhibited because of the usage of Adapter pattern.

5 I PATTERNS FOR DYNAMIC ARCHITECTURES

As a rationale for Dynamic Configuration analysis we assessed those services provided on the server side, such as scheduling mechanisms for clients and servers.

For the study of this particular requirement of software evolution we included a different subset of the patterns selected in section 3 that facilitate run time architecture reconfiguration: *Decorator*, *Strategy*, *Broker*, *Bridge*, *Component Configurator*, *Interceptor*, *Extension Interface*, and *Reflection*. Since this subset includes both design and architectural patterns, we were able to analyze the impact of architectural decisions presenting different levels of abstraction. The fundamental pros and cons of these patterns regarding ISO 25010 characteristics are shown in Table 4. For space reasons, we only present a brief analysis of **Extension Interface** pattern**.**

This pattern aims at multiple interfaces exportation by only one component, so that modification or extension of a component functionality will not break the source client code or prevent the definition of interfaces with a high level of complexity.

The fundamental insight we must obtain from this pattern is separation of concerns when programming clients so that they can access components through separate interfaces, one interface per component role instead of mixing all the roles in the same interface or component implementation. Even though *Extension Interface* is an effective mechanism, application timeliness can be compromised because it is complex and prone to experience locks. The interesting characteristic of this pattern lies in the use of interfaces and the implementation of abstract methods more than the sub-classing mechanism. Thus, development of factorybased software components is promoted with this pattern.

Pattern	Characteristic	Sub-characteristic	Impact	Metric
Comp. Configu-rator	Maintainability	Testability	+	Independence
		Modifiability	+	Dynamicity
		Reusability	+	Cohesiveness
		Modularity	+	Functional extension
		Analysability	-	Comprehensibility
	Performance	Response time	-	Complexity
		Resource usage	+	Low coupling
	Portability	Adaptability	-	Standardization
	Security	Integrity	-	Vulnerability
	Reliability	Maturity	-	Integration
Intercep-tor	Maintainability	Modifiability	+	Encapsulation
		Reusability	+ + + + +	Extensibility Flexibility Coupling Dynamicity Complexity
		Analysability	- - -	Extensibility Flexibility Complexity
	Performance	Response time	-	Complexity
		Resource usage	+	Monitoring control
	Reliability	Fault tolerance	- +	Defective timeliness Monitoring control
		Maturity	-	Defective timeliness
Ext. Interface	Maintainability	Modifiability	+	Cohesion
		Reusability	+ + +	Polymorphism Coupling Dynamicity
		Analysability	-	Complexity
	Performance	Response time	-	Complexity
		Resource usage	-	Complexity
	Compatibility	Interoperability	-	Complexity
			+	Extensibility
	Reliability	Fault tolerance	-	Locks
			+	Monitoring control
		Maturity	-	Timeliness lock
Decorator	Maintainability	Modifiability	+ + +	Flexibility Extensibility Dynamicity
		Analysability	+	Flexibility
			-	Complexity
		Testability	-	Complexity
	Performance	Resource usage	+	Complexity

Strategy	Maintainability	Modifiability	+	Extensibility
		Reusability	+ +	Simplicity Dynamicity
		Analysability	-	Complexity
		Testability	+ +	Extensibility Simplicity
	Performance	Resource usage	-	Complexity
	Functional suitability	Completeness	+	Extensibility
	Compatibility	Interoperability	-	Complexity
Broker	Maintainability	Modifiability	+	Flexibility
		Analysability	-	Simplicity
		Testability	-	Hiding
	Compatibility	Interoperability	-	Transparency
	Performance	Resource usage	-	Optimization
		Capacity	+	Scalability
		Response time	-	Complexity
	Reliability	Fault tolerance	-	Hiding
	Portability	Adaptability	+	Multi-platform
Reflection	Maintainability	Modifiability	+ +	Extensibility Dynamicity
		Analysability	-	Complexity
		Testability	-	Complexity
	Functional suitability	Correctness	+	Accuracy
	Performance	Resource usage	-	Complexity
	Reliability	Maturity	-	Complexity
		Fault tolerance	-	Failure biased

Table 4. Quality characteristics of the selected dynamic patterns.

From the analysis presented in Table 4, we can affirm that dynamic architecture partially promoted by the whole set of selected patterns. Notice that the sub-characteristic Analyzability got inhibited in all cases. On the other hand, the following quality characteristics are partially promoted by these patterns: Performance, Reliability, Compatibility, Functional Suitability and Portability.

6 I CONCLUSIONS AND FUTURE WORK

The analyzed patterns benefit software evolution from two aspects, according to discussed approaches in this paper, Maintainability and Dynamicity. Worth to mention is the variety of different mechanisms, common to design and architectural patterns, such as coupling, encapsulation, inheritance, and complexity, among others, which are used to guarantee such aspects. We also observed in our analysis that these characteristics and subcharacteristics are rarely achieved without punishing others, like Performance Efficiency or Reliability. However, according to the results obtained in this research, we can affirm that the usage of patterns is an effective way for constructing software if evolution requirements have to be satisfied. We recommend, during the software design phase, to incorporate design and architectural patterns within a rigorous analysis that considers the benefits and sensibility points (tradeoffs) exposed here.

We can see that these presented patterns are not excluding alternatives, but it is advisable to combine several of them, thus causing a greater heterogeneity in the target software architecture and then it makes sense to perform architectural assessments that take into account the effect of applying a set of these patterns.

Finally, we recognize the limitations of the theoretical approach followed in this research; however, the obtained results allow us to establish a preliminary set of hypothesis for an empirical evaluation of the affected characteristics and sub-characteristics. Currently, we are working in the preparation of different experiments within academic as well as industrial settings to implement and measure these results using Java applications.

REFERENCES

BASS, L. *et al*. **Software Architecture in Practice**. 1. ed. Addison-Wesley, 2003.

BODE, S.; RIEBISH, M. Impact evaluation for qualityoriented architectural decisions regarding evolvability. Lecture Notes on Computer Science, 6285, pp. 182-197, 2010. Article presented *in* ECSA 2010, Copenhagen, Denmark.

BOSCH, J. **Design and Use of Software Architecture**. England: ACM Press, 2000.

GAMMA, E. *et al.* **Design Patterns: elements of reusable objectoriented software**. Addison-Wesley, 1994.

GOMATHY, C.K. Software pattern quality comportment in service-oriented architectures. **European Scientific Journal**, v.10, n. 92014, March 2014, pp.:412-423.

GRIMAN, A. *et al.* Towards a Maintainability Evaluation in Software Architectures. *In*: 8th International Conference on Enterprise Information Systems ICEIS , 2006.

GRIMAN, A *et al.* Characteristic analysis for architectural evaluation methods. **Journal of Systems and Software**, v. 79, n. 6, 2006, pp.: 871-888.

HARRISON N.; AVGERIOU P. Leveraging architecture patterns to satisfy quality attributes. Lecture Notes on Computer Science, 4758, pp. 263-270, 2007. Article presented in ECSA 2007, Aranjuez, Spain.

ISO - International Organization for Standardization. **ISO/IEC 25010:2011 (en):** Systems and software engineering - Systems and software Quality Requirements and Evaluation (SQuaRE) - System and software quality models. Geneva: ISO, 2011.

KAZMAN, R. *et al*. ScenarioBased Analysis of Software Architecture, **IEEE Software**, V. 13, n.6, Nov. 1996.

LUNG, C.H. *et al.* An Approach to Software Architecture Analysis for Evolution and Reusability. *In:* SEI Digital Library. USA, 1996. Available on: https://resources.sei.cmu.edu/library/asset-view.cfm?assetid=29765. Retrieved on: 12 jan 2021.

OZKAYA, I. *et al*. **QualityAttributeBased Economic Valuation of Architectural Patterns**. 2007. Software Engineering Institute. DOI 10.1184/R1/6582686.v1. Available on: https://resources.sei.cmu.edu/library/asset-view.cfm?assetid=8307. Retrieved on: 12 jan 2021.

SCHMIDT, D. C. *et al*. **Patterns for Concurrent and Networked Objects**. Chichester, UK: Wiley, 2000.

ÍNDICE REMISSIVO

COLEÇÃO

DESAFIOS DAS ENGENHARIAS:

ENGENHARIA DE COMPUTAÇÃO

Atena Editora

Ano 2021

COLEÇÃO

DESAFIOS DAS ENGENHARIAS:

ENGENHARIA DE COMPUTAÇÃO

Atena Editora

Ano 2021