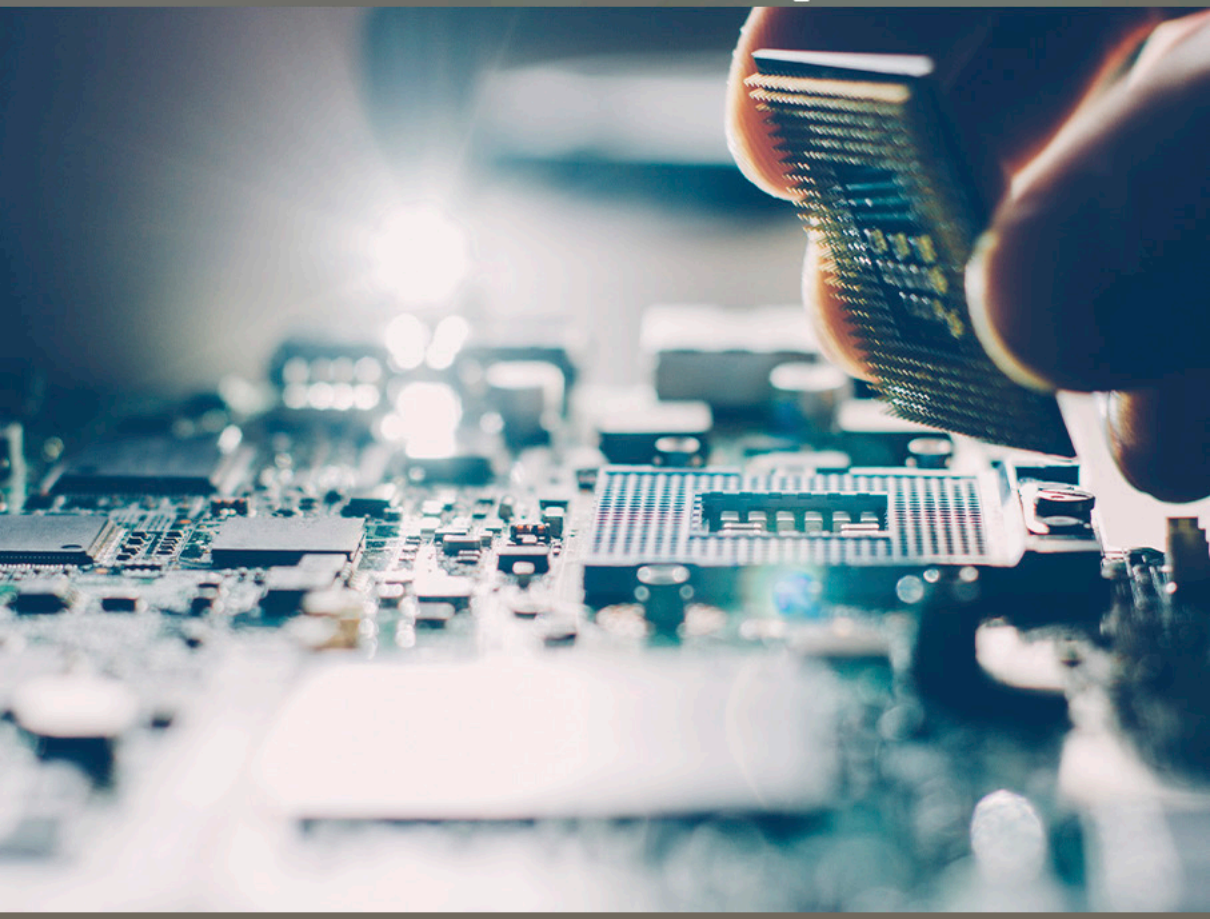
COLEÇÃO

DESAFIOS DAS ENGENHARIAS:

ENGENHARIA DE COMPUTAÇÃO



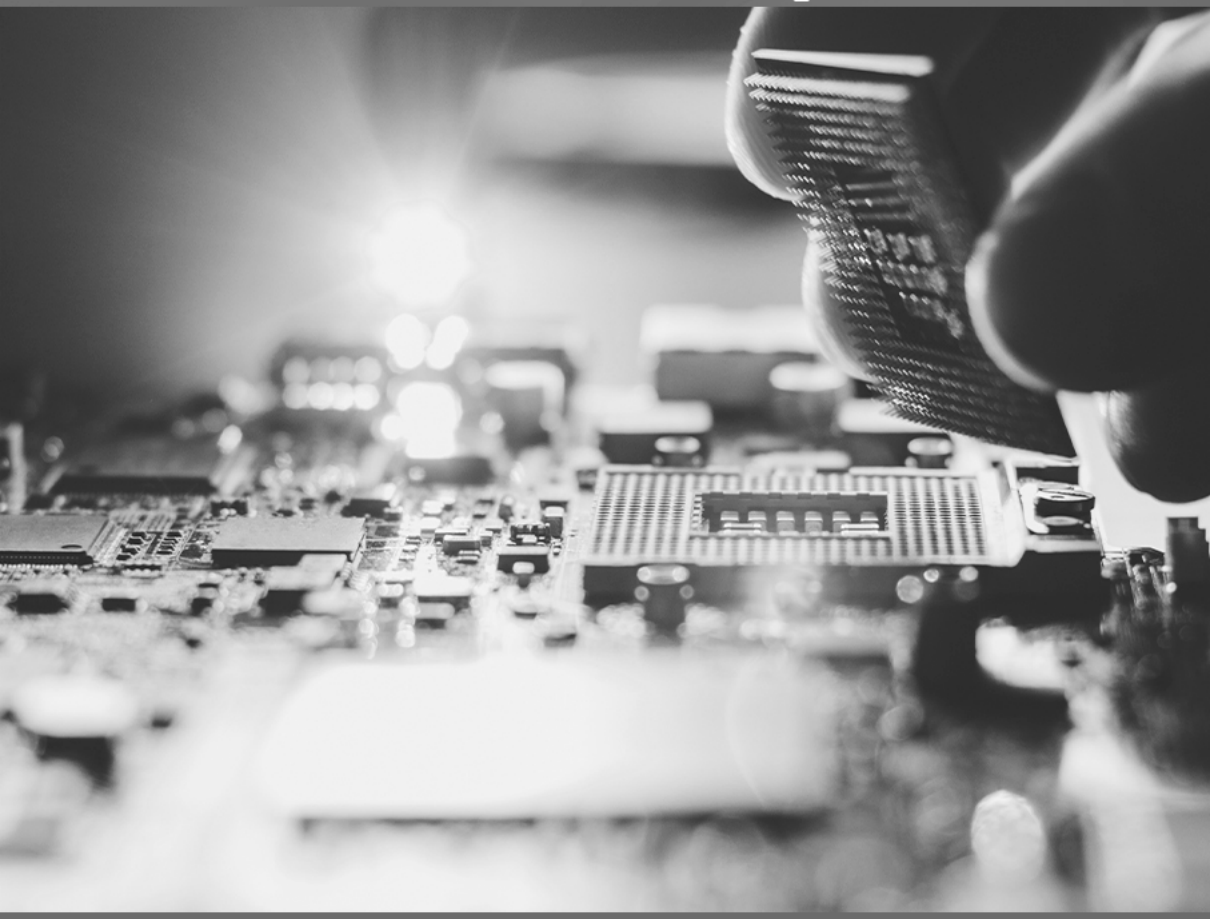
ERNANE ROSA MARTINS
(ORGANIZADOR)

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 Agrárias e Multidisciplinar

Atena Editora
Ano 2021

Atena Editora
Ano 2021

Atena Editora
Ano 2021

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

Ciências Exatas e da Terra e Engenharias

DECLARAÇÃO DOS AUTORES

Os autores desta obra: 1. Atestam não possuir qualquer interesse comercial que constitua um conflito de interesses em relação ao artigo científico publicado; 2. Declaram que participaram ativamente da construção dos respectivos manuscritos, preferencialmente na: a) Concepção do estudo, e/ou aquisição de dados, e/ou análise e interpretação de dados; b) Elaboração do artigo ou revisão com vistas a tornar o material intelectualmente relevante; c) Aprovação final do manuscrito para submissão.; 3. Certificam que os artigos científicos publicados estão completamente isentos de dados e/ou resultados fraudulentos; 4. Confirmam a citação e a referência correta de todos os dados e de interpretações de dados de outras pesquisas; 5. Reconhecem terem informado todas as fontes de financiamento recebidas para a consecução da pesquisa; 6. Autorizam a edição da obra, que incluem os registros de ficha catalográfica, ISBN, DOI e demais indexadores, projeto visual e criação de capa, diagramação de miolo, assim como lançamento e divulgação da mesma conforme critérios da Atena Editora.

Atena Editora
Ano 2021

DECLARAÇÃO DA EDITORA

A Atena Editora declara, para os devidos fins de direito, que: 1. A presente publicação constitui apenas transferência temporária dos direitos autorais, direito sobre a publicação, inclusive não constitui responsabilidade solidária na criação dos manuscritos publicados, nos termos previstos na Lei sobre direitos autorais (Lei 9610/98), no art. 184 do Código penal e no art. 927 do Código Civil; 2. Autoriza e incentiva os autores a assinarem contratos com repositórios institucionais, com fins exclusivos de divulgação da obra, desde que com o devido reconhecimento de autoria e edição e sem qualquer finalidade comercial; 3. Todos os e-book são *open access, desta forma* não os comercializa em seu site, sites parceiros, plataformas de *e-commerce,* ou qualquer outro meio virtual ou físico, portanto, está isenta de repasses de direitos autorais aos autores; 4. Todos os membros do conselho editorial são doutores e vinculados a instituições de ensino superior públicas, conforme recomendação da CAPES para obtenção do Qualis livro; 5. Não cede, comercializa ou autoriza a utilização dos nomes e e-mails dos autores, bem como nenhum outro dado dos mesmos, para qualquer finalidade que não o escopo da divulgação desta obra.

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 12

CALIDAD ÁGIL: PATRONES DE DISEÑO EN UN CONTEXTO DE DESARROLLO DIRIGIDO POR PRUEBAS

Data de aceite: 02/08/2021
Data de submissão: 18/07/2021

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

RESUMEN: Una de las prácticas más populares en desarrollo ágil y, más específicamente, Pruebas Unitarias, es el Desarrollo Dirigido por Pruebas (TDD, por sus siglas en inglés). En este dominio, el marco de trabajo JUnit se usa ampliamente en la actualidad. Aunque los patrones de diseño han ganado cada vez más popularidad, no todos los estilos y patrones arquitectónicos disponibles se pueden utilizar en un contexto TDD. Para poder utilizarlos, deben cumplir ciertas condiciones. Surge entonces la necesidad de determinar la factibilidad de que dichos mecanismos arquitectónicos puedan se probados. Este artículo presenta una prueba de concepto que forma parte de un proyecto de investigación en curso. Aquí, se lleva a cabo un análisis crítico de un pequeño conjunto de patrones de diseño y estilos arquitectónicos para determinar su capacidad de prueba en un entorno TDD. Mediante el uso de un estudio de caso, fue posible determinar la testabilidad de los patrones seleccionados, los beneficios en términos de calidad obtenidos por el uso de estos mecanismos, así como algunas restricciones para las pruebas. Para hacer esto, fue necesario implementar el patrón de prueba de *escuchador de eventos*. Como resultado adicional, se propuso un enfoque general para las pruebas de patrones de diseño.

PALABRAS CLAVE: Patrones de diseño, Desarrollo dirigido por pruebas, Calidad de Software, TDD.

AGILE QUALITY: DESIGN PATTERNS IN A TDD CONTEXT

ABSTRACT: One of the most popular practices in agile development and, more specifically, Unit Testing, is Test Driven Development (TDD). In this domain, the JUnit framework is widely used today. Although design patterns have gained more and more popularity, not all available architectural styles and patterns can be used in a TDD context. In order to use them, they must meet certain conditions. Then, the need arises to determine the feasibility that such architectural mechanisms can be tested. This paper presents a proof of concept that is part of an ongoing

research project. In here, a critical analysis of a small set of design patterns and architectural styles is carried out to determine their testability in a TDD environment. By using a case study, it was possible to determine the testability for the selected patterns, the benefits in terms of quality obtained by the usage of these mechanisms, as well as some restrictions to tests. To do this, it was necessary to implement the *event listener* test pattern. As an additional result, a general approach for design patterns tests was proposed.
KEYWORDS: Design Patterns, Test-Driven Development, Software Quality, TDD.

1 | INTRODUCCIÓN

Algunos de los mitos que aún persisten en el paradigma del DESARROLLO DE SOFTWARE ÁGIL incluyen afirmaciones como las siguientes: "la calidad del sistema se logra fácilmente con una arquitectura evolutiva", "la adaptación a los cambios en los requerimientos es fácil", "no hay que preocuparse por el *rendimiento*, la *escalabilidad*, la *seguridad*, la *usabilidad*, etc. del software hasta que su funcionalidad esté operativa". En la práctica, sin embargo, tales mitos quedan relegados y nos encontramos con la necesidad de obtener soluciones capaces de afrontar a nivel arquitectónico los compromisos típicos entre las características de calidad, tales como *usabilidad* frente a *seguridad*, *desempeño* frente a *mantenibilidad*, y *testeabilidad* frente a *rendimiento*, entre otras.

El Desarrollo Dirigido por las Pruebas (TDD, según su acrónimo inglés) surge inicialmente con la metodología XP (Beck, 1999) y ha cobrado cada vez mayor popularidad en el contexto de diferentes procesos o metodologías ágiles. TDD produce una arquitectura que surge de la no-ambigüedad de las pruebas automatizadas.

Las arquitecturas ágiles son evolutivas, no se diseñan completamente antes de escribir el código. En general, se considera que con esta práctica se aumenta la calidad del software y se consigue un código altamente reutilizable, entre otros beneficios.

En la comunidad TDD existe una cierta desconfianza acerca de la utilización de patrones de diseño en el desarrollo de pruebas unitarias (Wikipedia, 2021), que está retrasando su aceptación; sin embargo, los patrones de diseño sí son aceptados en el desarrollo de software en general.

Por otra parte, diferentes trabajos (Gomathy, 2014; Khaer et al, 2008; Khan et al, 2012; Medvidovic & Taylor, 1997; Harrison & Avgeriou, 2007; Bode et al, 2010; Ozkaya et al, 2007; Lung, 1997) han abordado los beneficios del uso de patrones de diseño en cualquier actividad relacionada con el desarrollo de software. En (Griman et al, 2006) se propone una ontología para la mantenibilidad del software y, a partir de ella, se estudian los beneficios del uso de determinados mecanismos arquitectónicos en la mantenibilidad adaptados a los tipos de mantenimiento clásicos del software. En un trabajo posterior (Griman et al, 2006b), se consideran 49 características, con sus correspondientes métricas, como base de la metodología DESMET, donde se usa el análisis de características para valorar métodos o herramientas de evaluación arquitectónica con el fin de asegurar la mantenibilidad de las

arquitecturas software.

En esta investigación presentamos una prueba de concepto realizada en el marco de un proyecto de investigación en curso. Nuestra propuesta puede aplicarse a cualquier marco de trabajo para pruebas unitarias de software. Se llevó a cabo un análisis con JUnit, un entorno de pruebas automáticas y TDD muy utilizado actualmente, de un conjunto reducido de patrones de diseño y estilos arquitectónicos para determinar su testeabilidad.

El objetivo de esta investigación a medio plazo es el análisis y caracterización de un conjunto de patrones de diseño útiles para TDD (Guerra, 2012), que pueden ser empleados con efectividad en este contexto y, por consiguiente, dentro del marco de metodologías *ágiles* (Meszaros, 2007; Beck, 1999) de desarrollo de software.

El resto del artículo se ha organizado como sigue, la sección 2 introduce la motivación que ha dado lugar a la investigación y se plantean los objetivos para la incorporación de patrones de diseño en las actividades relacionadas con TDD actualmente; se establecen las condiciones para la testeabilidad de patrones y se analiza la adecuación del conjunto seleccionado a ésta y otras características de calidad del software. En la sección 3 se comprueba, a través del caso de estudio SCACV (Sistema Automático de Control de Velocidad de un Vehículo), la testeabilidad de los patrones seleccionados. Finalmente, en la sección 4 se discuten los principales resultados obtenidos y se exponen las conclusiones del trabajo de investigación realizado.

2 | ANÁLISIS DE PATRONES PARA TDD

Esta investigación supone el comienzo de un proyecto planificado para tres años cuyo objetivo fundamental es la realización de un análisis crítico, sistemático, repetible y apoyado en casos de estudio prácticos sobre las características de calidad según (ISO/IEC, 2011) de un conjunto de patrones y estilos de diseño arquitectónico.

En este trabajo sólo se ha analizado la testeabilidad, según un método de *pruebas unitarias*, de 3 patrones que son muy utilizados actualmente en diseño y desarrollo de software, con el objetivo de promover su utilización en un entorno TDD para software complejo.

Una unidad de software ha de ser probada en su totalidad a través de la interfaz original, pero hay que tener en cuenta que cualquier estructura que se agregue por las clases que componen un patrón de diseño se convierte en una parte de la unidad de software que está siendo testeada. Por consiguiente, si como consecuencia de la utilización de un patrón, la unidad de software se convierte en más compleja, entonces será necesario testear más estados de dicha unidad. Se podría llegar incluso a la necesidad de testear un número infinito de estados si se producen dependencias de creación cíclicas. Con base en esta premisa y derivado de lo descrito anteriormente, surgen las siguientes preguntas:

1. ¿Es posible desarrollar código de tests que incorpore el uso de los patrones del

catálogo GoF (Gamma et al, 1994)?

2. ¿Qué beneficios se obtienen de la incorporación de los patrones al diseño evolutivo de la arquitectura obtenida a través de la práctica de TDD?

3. ¿Qué impacto negativo conlleva el uso de un patrón en el diseño obtenido a través de la práctica de TDD?

Condiciones de testeabilidad de un conjunto de patrones de diseño

Para responder a las preguntas planteadas anteriormente (2) y como prueba de concepto de nuestro proyecto, en primer lugar se realizó una selección de un pequeño conjunto inicial de patrones candidatos para TDD, basados en un conjunto de condiciones o criterios derivados de los conceptos y premisas de TDD, como son:

- El patrón debe poseer un enfoque funcional, por ejemplo: recibe o retoma uno o más valores.
- No deben existir dependencias de código ocultas que puedan producir efectos laterales en otros módulos. Si causa tales efectos, debe ser posible capturarlos y probarlos.
- No debe poseer un estado interno que haya de persistir tras la invocación de las operaciones o métodos de las clases que los componen.
- No se pueden crear objetos dentro del código de métodos o constructores de las clases que pertenezcan al patrón.
- Debe ser posible verificar si durante la ejecución ocurrieron todos los eventos asociados al comportamiento y prueba unitaria de un patrón.

Evaluación de las condiciones

Básicamente, las condiciones expresadas previamente convierten al patrón en un *componente software*, asegurando que cumplirá con la propiedad de *testeabilidad independiente*, es decir, un patrón encapsula todas las características que lo definen y no mantiene o crea por sí mismo dependencias con otros componentes de software que pudieran alterar su comportamiento, dependiendo del contexto en que fuese utilizado. Afirmamos que un patrón ha de poder probarse independientemente y se ha de desplegar completamente para ser utilizado correctamente. Por tanto, el patrón visto como componente posee un comportamiento similar al de una función matemática, pudiendo aceptar valores, como datos de entrada, durante su utilización o parametrizarse, pero nunca va a enviar valores que puedan cambiar el estado de su entorno o crear una dependencia con el contexto, que haga visible su estado interno. Obviamente, lo anterior necesariamente implica que los métodos de las clases del patrón no pueden crear objetos de clases externas porque se estaría permitiendo la visibilidad de parte del estado del patrón, creando referencias accesibles fuera del contexto del patrón.

En la Tabla 1 se presenta un resumen de las características de calidad relacionadas

con la evolución del software, que son propiciadas (+) e inhibidas (-) por cada patrón seleccionado para este estudio. En la tabla también pueden observarse resaltadas en *itálica* aquellas cualidades que obligatoriamente ha de cumplir un patrón para poder ser probado (testeabilidad), según las condiciones establecidas anteriormente.

Característica	Sub-característica	Cualidad	OBS	FA	MVC
Mantenibilidad	Facilidad de cambio	Cohesión *Encapsulación* *Dinamicidad*	+ + +	- + +	+ - +
	Facilidad de Análisis	Reusabilidad Complejidad	+ +	+ -	- +
	Testeabilidad	Complejidad *Independencia*	- +	- +	+ -
Eficiencia	Tiempo respuesta	Rendimiento	+	-	+
	Uso de recursos	Rendimiento	+	-	+
Funcionalidad	No acoplamiento	Complejidad *Extensibilidad*	- ++	- +	? ?
Fiabilidad	Tolerancia a fallos	Recuperabilidad Exclusión mutua Monitorización Control	- + - +	+ + - +	+ + + +
	Madurez	Previsibilidad *Sincronización*	+ +	- +	+ +

Tabla 1. Características propiciadas/inhibidas y condiciones de testeabilidad para cada patrón.

Resumen de patrones y sus beneficios

A partir del análisis de la Tabla 1, podemos afirmar que los patrones que se han utilizado para realizar este estudio: *Observador (OBS)*, *Factoría Abstracta (FA)* y *Modelo-Vista-Controlador (MVC)* cumplen todas las condiciones de testeabilidad anteriormente señaladas. Por razones de espacio, sólo presentamos a continuación un breve resumen de este análisis para el patrón Observador.

En el Observador se transmite automáticamente el cambio de estado desde los OBSERVABLES a los OBSERVADORES, provocándose la ejecución del método ACTUALIZAR() de éstos, incluidos en la lista de cada OBSERVABLE. Tal método podría tener parámetros pero sólo es ejecutado por el OBSERVABLE, el cuál en todo momento mantiene su estado oculto al resto. Tampoco es necesario crear referencias a ningún objeto OBSERVABLE porque no es necesario acceder a su estado; el propio observable se encarga de llamar a los métodos necesarios en los OBSERVADORES sin que se rompa jamás su encapsulación.

Como se muestra en la Tabla 1, la característica de *Fiabilidad* puede verse inhibida con el patrón *Observador*, ya que este patrón no propicia la *recuperabilidad* ni la facilidad de *monitorización* del software. Un cambio menor sobre un objeto OBSERVABLE puede causar

actualizaciones en cascada en los objetos OBSERVADORES y sus objetos dependientes. Puede provocar que el rastreo de los errores sea mucho más difícil. Debido a que no se conoce su origen, las operaciones de recuperación pueden resultar un trabajo duro y difícil, perjudicando la cualidad de *recuperabilidad*.

3 | CASO DE ESTUDIO SCACV

Se ha desarrollado un *applet* en Java que simula un vehículo y su control automático de velocidad (se puede acceder al ejecutable accediendo en: http://lsi.ugr.es/~ist/Documentos/practica1). Para iniciar su funcionamiento hay que pulsar el botón ARRANCAR, entonces se activa la palanca, que controla los distintos estados de control (*acelerando*, *frenando*, *activado*, *desactivado*, *reiniciando*).

Se ha programado con Swing/Java y los movimientos de la palanca se simulan accionando los distintos botones (Figura 1-a). La palanca se moverá siempre que sea una transición permitida (Figura 1-b). El botón de freno puede pulsarse en cualquier momento simulando el pedal del vehículo, mientras se mantenga pulsado el coche frenará y dejará de hacerlo en cuanto se suelte. Para apagar el motor y para repostar es necesario que el control automático se encuentre apagado y la velocidad del vehículo sea nula.

Para desarrollar el SCACV se precisaron 3 paquetes diferenciados que contenían las clases (código producido): CONTROLVELOCIDAD, MONITORIZACION e INTERFAZ y los Test Suites que activan la ejecución de todos los casos de prueba (*Test Cases*). En la Figura 2 se observa el diseño arquitectónico del SCACV propuesto, que muestra la utilización de 2 patrones: *Observador* y *MVC*. El patrón de diseño *Factoría Abstracta* es parte del patrón *MVC* (Gamma et al, 1994) y se discutirá en la prueba unitaria de los paquetes *Interfaz* (Vista) y *Simulacion* (Controlador).

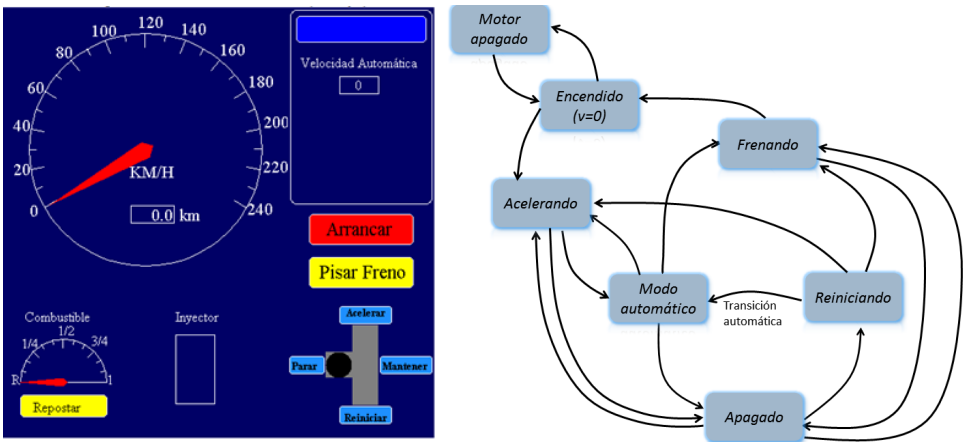


Figura 1. (a) Simulación del SCACV. (b) Estados de control del vehículo y de la palanca.

Pruebas unitarias de los patrones de diseño

Para desarrollar un proyecto de pruebas con JUnit para determinados patrones de diseño cuya prueba consiste en verificar si durante la ejecución ocurrieron todos los eventos de una lista, podemos basarnos en el *patrón de pruebas* denominado *Escuchador de Eventos*. Por ejemplo, el test del patrón Observador se realiza comprobando que cada vez que se dan las condiciones durante la ejecución, el OBSERVABLE notificó a todos los OBSERVADORES de su lista y cada uno de éstos, a su vez, ha debido ejecutar su método ACTUALIZAR() como consecuencia.

Se ha de crear una clase ESCUCHADOR que implemente la interfaz TESTLISTENER de Java y también la interfaz OBSERVADOR del patrón de diseño del mismo nombre. La primera interfaz se utiliza para recibir *eventos textuales* que ocurren en las aplicaciones. Una clase Java que esté interesada en conocer la ocurrencia de un determinado evento textual implementará esta interfaz.

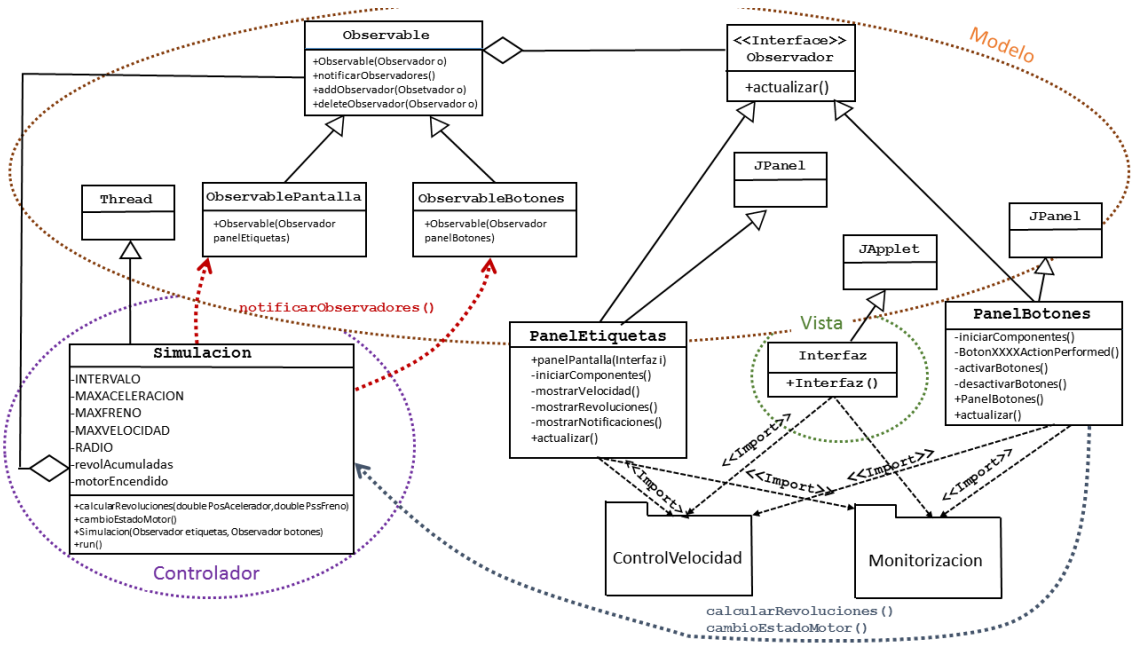


Figura 2. Diseño arquitectónico del SCACV.

Un objeto de esta clase ESCUCHADOR se registrará en una lista interna del componente de la aplicación que puede modificar su texto llamando al método ADDTEXTLISTENER(), de tal forma que cuando cambie el texto del componente, entonces se llama automáticamente al método TEXTVALUECHANGED() del objeto escuchador.

Para realizar las pruebas unitarias del SCACV, hemos incluido en 3 paquetes

diferenciados las clases de prueba para los paquetes correspondientes del código producido: ControlVelocidad, Monitorizacion e Interfaz, según el diseño arquitectónico propuesto en la Figura 2. Cada uno de los paquetes de pruebas contiene un *Test Suite* que inicia la ejecución de todos los casos de prueba (*Test Case*) de ese paquete. En el paquete Interfaz se incluyeron los casos de prueba para las clases: Interfaz, ListaObservadoresObservables, Observable, PanelBotones y Simulacion. No es necesario crear pruebas unitarias para la clase PanelEtiquetas ni para las clases que muestran indicadores en la interfaz de usuario, ya que su funcionamiento se basa en una modificación de la clase test para la propia Interfaz.

Lo anterior permite reportar los resultados obtenidos para los casos de prueba de las clases de código producido para el SCACV utilizando varios patrones de diseño de la siguiente manera:

1. Se realizó la implementación de la prueba de un *escuchador de eventos* que implementa TestListener de junit.framework.* y sirve para desarrollar las pruebas unitarias del patrón Observador utilizado en el paquete ControlVelocidad.

2. En el paquete ControlVelocidad se incluyó la descripción de los casos de prueba de las clases más relevantes, incluyendo clases de prueba abstracta, que contienen los métodos de la interfaz de los dispositivo; por ejemplo, Pedal:

```
public interface Pedal{
  public double leerEstado();
  public void soltar();
  public void actualizar();
}
```

3. No se implementaron casos de prueba para clases del tipo Almacen, ya que sus métodos son de tipo getter() y setter() y sus pruebas son triviales.

4. Por último, en el paquete de Monitorización se incluye una descripción de los casos de prueba para todas las clases que manejan los controles.

```
public class ObservadorTestListener
          implements TestListener, \emph{\emph{Observador}} {
 private List<List<Object>> eventos;
 public ObservadorTestListener() {   //Constructor
     eventos = new ArrayList<List<Object>>();
 }
 @Override
 public boolean actualizar() {
     eventos.add(nuevoObjetoEventoObservable());
     return true;
 }
 @Override
 public void startTest(Test test) {
     eventos.add(nuevoObjetoEventoInicioTest(test));
 } ...
 public List<Object> nuevoObjetoEventoObservable() {
     return Arrays.asList(new Object[ ]{"actualizar"});
 }
 public List<Object> nuevoObjetoEventoInicioTest(Test test){
     return Arrays.asList(new Object[]{"startTest",test});
 }...
 //Devuelve la lista de eventos acumulados
 public List<List<Object>> listaEventos() {
    return eventos;}
 }
}
```

Figura 3. Implementación de la clase ObservadorTestListener.

Prueba del patrón Observador. De acuerdo con el patrón de pruebas *Escuchador de Eventos*, se ha creado la clase ObservadorTestListener (Figura 3), dentro del paquete Simulacion, como escuchador de eventos que se produzcan en la aplicación, que utiliza el patrón de diseño Observador. Esta clase cuenta con una lista de eventos que se va completando con objetos que se generan cada vez que se recibe un evento y que identifican al evento en cuestión utilizando, entre otros elementos, una cadena de caracteres. Implementa cada uno de los métodos de la interfaz TestListener, que se invocan cuando se produce un evento de *inicio de test*, *fin de test*, *error* o *fallo*, en la aplicación. Para cada uno de estos eventos se inserta un objeto diferente en la lista. El método actualizar() de la Figura 3 de la interfaz Observador también está implementado, de forma que se inserta un objeto identificativo en la lista de eventos cada vez que es invocado durante la ejecución del test en notificarObservadores() del Observable, Figura 4.

```
@Test
public void testNotificarObservadores() {
 testAniadirObservador();
 observable.notificarObservadores();
 List<List<Object>>esperados= new ArrayList<List<Object>>();
 esperados.add(observador.nuevoObjetoEventoInicioTest(this));
 esperados.add(observador.nuevoObjetoEventoObservable());
 assertEquals(esperados, observador.listaEventos() );
}
```

Figura 4. Implementación de testNotificarObservadores().

Por otra parte, se crea la clase ListaObservadores/Observables que contiene los casos de prueba de un Observable: testNotificarObservadores(), testInsertaObservador(), testEliminaObservador() y el creador de la *fixture*: setUp(). En esta clase se crea un objeto ObservadorTestListener y un objeto TestResult. Al objeto TestResult se le añade el objeto ObservadorTestListener como su *escuchador*. Posteriormente, al escribir el método de prueba testNotificarObservadores() (figura 4), se crea una lista con los objetos que se espera tendrá la lista de eventos, es decir, un objeto correspondiente al inicio del test, seguido de otro correspondiente a la llamada al método actualizar() del observador.

Por último, se compara la lista de eventos de testNotificarObservadores() con la del objeto escuchador ObservadorTestListener. Si coinciden, entonces el patrón Observador de la aplicación ha pasado la prueba con éxito.

Prueba del patrón MVC: se trata ahora de conseguir realizar pruebas unitarias a una Interfaz que, si no se toman las acciones necesarias, presentará una mezcla difícil de separar entre la lógica de interacción del usuario (IU) y la denominada lógica de presentación (LP) de la aplicación. La lógica IU se refiere a la gestión de las distintas interacciones del usuario con la interfaz de la aplicación, tales como *pulsar un botón*. El problema en este caso consiste en que la lógica de IU está fuertemente acoplada al código de manejo; de hecho, la lógica IU, la LP y la de transformación de datos (TD) están empotradas juntas en el código del manejador y es difícil saber dónde comienza una y termina la otra. Para poder realizar adecuadamente las pruebas unitarias de la lógica de IU de una aplicación, tendremos previamente que probar lo siguiente:

- Toda petición desde la parte del usuario se tratará a través del Controlador.
- El Controlador contiene toda la lógica de IU.
- El Controlador no ha de mantener ningún acoplamiento con la Vista.

Si se respetan las reglas anteriores, entonces es posible realizar la prueba unitaria de la lógica IU (Controlador) de una aplicación como el SCACV. Además, la *Vista* se puede también probar si se define un modelo intermedio entre la *Vista* y el *Modelo*. De esta forma, el *Modelo* solo contendrá datos y lógica que tengan que ver exclusivamente con el negocio,

mientras que el denominado *VistaModelo* contendrá ahora los datos (anteriormente en *Modelo*) que tengan relación con la *Vista*. Se elimina cualquier posible conexión directa entre los componentes *Vista* y *Modelo* del patrón. En consecuencia, la *Vista* se conecta a *VistaModelo*, que contendrá los siguientes elementos: referencia al Modelo; lógica de TD para cada uno de los requisitos definidos en la *Vista*; LP para cada uno de los requisitos en la *Vista*.

En el caso del diseño del SCACV, el patrón MVC se aplica de acuerdo con la siguiente estructura:

- Modelo: clases de la figura 2, excepto Simulacion, Interfaz y Paneles.
- Vista: Interfaz y PanelEtiquetas.
- VistaModelo: PanelBotones.
- Controlador: clase Simulacion.

El código de los escuchadores de eventos de la interfaz de la aplicación se encuentran encapsulados en la clase PanelBotones y separados respecto de cada uno de los controles gráficos: botones, texto para completar, etc. que se le presentan al usuario a través de la Interfaz. PanelBotones contiene toda la lógica de TD que tiene un reflejo en la presentación de la Vista, ya que:

- Captura todos los eventos que se originan en la Interfaz (pulsar botones movimientos de la palanca, acelerar, etc.)
- Convierte determinados cambios en los datos del modelo (revoluciones del eje, estado del motor) a datos específicos de la Vista (representación gráfica de la velocidad, motor apagado/encendido).

De esta manera, podemos dar por terminadas las pruebas unitarias de la aplicación.

4 | CONCLUSIÓN

De forma general, para poder llevar a cabo la prueba unitaria de los patrones en el marco de trabajo JUnit es necesario implementar el patrón de pruebas *Escuchador de Eventos*. Un objeto escuchador hace posible registrar en una lista de eventos la ocurrencia de eventos fundamentales para relacionarlos con el desarrollo de la prueba, así como las llamadas a métodos de la fachada de cada patrón durante la ejecución de la aplicación. La implementación de esta técnica de prueba unitaria se resume como sigue:

1. Crear un objeto de la clase escuchador que registra en una lista interna los eventos textuales que ocurren en los componentes de una aplicación.
2. Un componente de la aplicación crea una instancia de TestResult que recoge los resultados de ejecutar un test y crea su objeto escuchador.
3. El test pasará sin producir errores o fallos si al analizar coincide el contenido de la lista interna de eventos esperados del componente con la lista de eventos

registrados en la lista de su ESCUCHADOR.

Por otra parte, se identificaron las siguientes limitaciones en cuanto a la testeabilidad de un patrón:

- Realizar pruebas unitarias de lógica IU, lógica de TD y LP, a la vez, no es posible.
- Difícil separación entre el código del Modelo, Vista y su Controlador.
- En general, la comprobación de código que contenga gestores de eventos (*event handlers*) está limitada por el hecho de que los eventos no pueden ser generados en el test de pruebas de forma real y hay que recurrir a mecanismos que activan el manejador ficticiamente (*mock handlers*) o que registran textualmente la ocurrencia de eventos o llamadas a métodos para poder programar los *test cases*.
- Para poder realizar adecuadamente las pruebas unitarias de la lógica de IU de una aplicación ha de poder garantizarse la separación total entre el código del modelo y el de la interfaz de usuario de la aplicación.

Como trabajos futuros se contempla trabajar en la identificación y análisis de características de calidad de un conjunto más amplio de patrones de diseño susceptibles de ser utilizados con las técnicas TDD.

REFERÊNCIAS

BECK, K. **XP Explained**. Addison-Wesley Professional, 1999.

BLÉ-JURADO, C. *et al*. **Diseño Ágil con TDD**. www.iexpertos.com, 2010.

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.

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

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

GUERRA, E. Fundamental test-driven development step patterns. *In:* Proceedings of the 19th Conference on Pattern Languages of Programs - PLoP, 2012.

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.

KHAER A. *et al*. On Use of Design Patterns in Empirical Assessment of Software Design Quality. *In:* International Conference on Computer and Communication Engineering, 2008.

KHAN, K.Y. *et al*. Improvement in quality of software architecture via enhanced pattern-driven architecture (EPDA). **International Journal of Information Technology and Computer Science**, v.12, 2012, pp. 31-39.

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.

MEDVIDOVIC, N. & TAYLOR, R. Framework for Clasifying and Comparing Architecture Description Languages. *In:* Software Engineering (ESEC/FSE97), 1997.

MESZAROS, G. **XUnit test patterns: refactoring test code**. Person Education, 2007.

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.

TEST-DRIVEN Design and Patterns. *In:* WIKIPÉDIA: a enciclopédia livre. [São Francisco, CA: Fundação Wikimedia], 2017. Disponível em: http://c2.com/cgi/wiki?TestDrivenDesignAndPatterns. Acesso em: 15 jan. 2021.

ÍNDICE REMISSIVO

W

COLEÇÃO

DESAFIOS DAS ENGENHARIAS:

ENGENHARIA DE COMPUTAÇÃO

www.atenaeditora.com.br
contato@atenaeditora.com.br
@atenaeditora
www.facebook.com/atenaeditora.com.br

Atena Editora
Ano 2021

COLEÇÃO

DESAFIOS DAS ENGENHARIAS:

ENGENHARIA DE COMPUTAÇÃO

www.atenaeditora.com.br
contato@atenaeditora.com.br
@atenaeditora
www.facebook.com/atenaeditora.com.br

Atena Editora
Ano 2021