Un estudio de evaluación de la eficiencia energética de patrones de diseño software

David Carretero[0009-0005-6675-4184], Olivia Poy[0000-0003-1504-2361], Ignacio García[0000-0002-0038-0942], Félix García[0000-0001-6460-0353], Mª Ángeles Moraga[0000-0001-9165-7144] y Coral Calero [0000-0003-0728-4176]

Grupo Alarcos, Universidad de Castilla-La Mancha, Paseo de la Universidad 4, Ciudad Real, España
{david.carretero, oscar.poy}@alu.uclm.es, {felix.garcia, ignacio.grodriguez, mariaangeles.moraga, coral.calero}@uclm.es

Abstract. La sostenibilidad está cobrando cada vez un valor más relevante en el ámbito del desarrollo de software, pero desarrollar software sostenible es todo un reto, habida cuenta del desconocimiento sobre la influencia que tienen en el consumo del software las diferentes estructuras de código. Con el objetivo de comenzar a crear un cuerpo de conocimiento que se traduzca en buenas prácticas a la hora de desarrollar software sostenible, este trabajo plantea la evaluación de la eficiencia energética de tres patrones de diseño: *Singleton*, *Composite y State*. Los resultados permiten observar cuándo el uso de cierto patrón de diseño disminuye el consumo del software o lo incrementa. Así mismo, estas conclusiones serán vitales para poder aplicar indicadores como la *deuda ecológica*, que ayudarán a cuantificar los recursos energéticos que comprometemos con el software cuando éste acumula decisiones de diseño que lo condicionan a un mayor consumo.

Keywords: Software sostenible, Green-In, consumo energético, patrón de diseño, deuda ecológica.

1 Introducción

La Ingeniería del Software dota al programador de multitud de técnicas y herramientas que aprovechan el conocimiento adquirido a lo largo del tiempo para mejorar tanto la calidad del software como la eficiencia en el desarrollo de este. Dicho conocimiento toma la forma de buenas prácticas tales y como pueden ser los patrones de diseño [1] (que adaptamos a nuestros proyectos confiando en el buen resultado que han mostrado anteriormente), y prácticas a evitar como son los *bad-smells* y anti-patrones [2] (que pueden afectar a diversas dimensiones de la calidad del software). Estas buenas prácticas han contribuido, en el ámbito de las dimensiones de tiempo, coste y calidad, al desarrollo exitoso de proyectos y la obtención de un producto de calidad.

Un factor que tradicionalmente no se ha tenido presente en la Ingeniería del Software es el de la sostenibilidad, aunque recientemente haya empezado a tener más relevancia [3]. Este creciente interés ha motivado que tengamos que replantear la validez

del conocimiento que hasta ahora considerábamos aplicable. Una de las dimensiones clave de la sostenibilidad es la medioambiental, y en particular en este trabajo abordamos cómo impacta en el consumo energético la forma de desarrollar un software, tratando así la perspectiva Green-IN.

Dado que actualmente existe cierto grado de desconocimiento acerca de cómo los estilos de programación pueden influir en el consumo energético (tanto en su aumento como en su reducción) [4] [5] es necesario partir de estructuras que tienen un alto impacto en el desarrollo de software, como son los mencionados patrones de diseño y *bad-smells*. Dada la frecuencia con la que estos se usan o evitan, su análisis permitiría ofrecer indicios y pistas de cómo empezar a mejorar el consumo energético en el software, creando buenas prácticas para el desarrollo de software sostenible. Algunas experiencias [6] presentan antipatrones de diseño como "*God-Class*", que aunque tiene un impacto positivo sobre la mantenibilidad, implican un mayor consumo energético con respecto a una misma versión del software menos mantenible.

Esta situación nos plantea un importante análisis "*trade-off*" sobre en qué medida algunas decisiones de diseño ya probadas, pueden tener un efecto negativo en el consumo energético del software. Sin embargo, antes de poder tomar decisiones para llevar a cabo un desarrollo del software sostenible es necesario evaluar en qué medida existe un impacto positivo o negativo con respecto a dichas decisiones. Por ello en este trabajo se presenta un estudio en el que se analiza cómo tres patrones altamente extendidos (*Singleton*, *Composite* y *State*) implican un mayor o menor consumo energético con respecto a una versión funcionalmente equivalente del software que no los implementa.

Este artículo se organiza de la siguiente manera: la sección 2 presenta una revisión de estudios sobre el impacto que pueden tener los patrones de diseño y *bad smells* en el consumo; la sección 3, muestra el marco de trabajo empleado en este artículo para la evaluación de los patrones bajo estudio mencionados; la sección 4, presentan los resultados de la medición del consumo de distintas versiones de software con y sin los patrones bajo análisis; y finalmente, la sección 5 presenta algunas conclusiones que pueden extraerse del estudio, así como algunas líneas de investigación a seguir en el futuro.

2 Trabajos relacionados

En relación con el efecto de los patrones de diseño en el consumo de energía, en [7] se evaluaron los patrones *State, Strategy* y *Template*, concluyendo que las soluciones sin aplicar el patrón eran más eficientes desde el punto de vista energético que las soluciones con patrón, entre un 17 % y un 54 % de diferencia de consumo y con un 79 % de los casos en los que no aplicar el patrón producía mejoras de consumo. Según Sahin et al [8] el impacto puede variar mucho, ya que para algunos patrones como por ejemplo *Factory method, Prototype, Bridge* y *Strategy* el impacto de su aplicación fue relativamente pequeño (menos del 1 %), mientras que para otros patrones el impacto fue moderado (por ejemplo, *Abstract factory, Flyweight, Observer*) o incluso sustancial (por ejemplo, *Decorator*). Además, se observó que, en cada categoría, hay patrones que tienen impacto positivo y otros, impacto negativo. En comparación con [8], en [7] se proporciona un análisis más detallado aplicando 2 métricas: SLOC (Líneas de Código

Fuente) y MPC (Métodos Ponderados por Clase), observando diferencias estadísticamente significativas en las métricas entre los casos en los que el patrón es eficiente energéticamente y no, observándose de media un 65,83% más de líneas de código y un 43,37% más de invocaciones de métodos en los casos en los que se aplicó el patrón que en los que no. El estudio de Bunse et al. [9] abordó software de dispositivos móviles y se analizaron los patrones *Facade, Abstract Factory, Observer, Decorator, Prototype* y *Template Method*. Se observó que el uso de patrones aumentaba el consumo aunque de forma ligera, salvo el patrón *Decorator*, con un consumo de más del doble de energía que su homólogo sin patrón. Por su parte, Litke et al [10] realizaron un análisis de 5 patrones de diseño (*Factory Method, Adapter, Observer, Bridge* y *Composite*) sobre 6 aplicaciones de ejemplo desarrolladas en C++, observando diferencias en los tres primeros, de los cuales *Factory Method* y *Adapter* no suponían una amenaza grave para el consumo de energía porque el número de instrucciones ejecutadas permanecía casi inalterado y los accesos a la memoria no aumentaban significativamente, debiéndose la diferencia principalmente al tamaño del código. Sin embargo, sí se identificó una sobrecarga significativa al emplear el patrón *Observer*. En [11] se evaluó el impacto del uso de patrones en aplicaciones móviles abordando los patrones *Observer, Singleton, Facade, Abstract factory* y *Template*, observando que los valores de consumo de energía aumentaron tras la implementación del patrón *Singleton*. Noureddine y Rajan [12] abordaron cómo mejorar la eficiencia energética de los patrones de diseño sin perder las ventajas esenciales de una mejor legibilidad, mantenimiento y reutilización del código. Su hipótesis de partida era la influencia de las optimizaciones del compilador, que permiten ahorrar energía sin necesidad de modificar el software o el hardware existentes. Por ello realizaron una comparación de la sobrecarga de consumo de energía causada por 21 patrones de diseño y exploraron en detalle los efectos de 2 patrones de diseño (*Observer* y *Decorator*). Como objetos de estudio para la comparativa se analizaron las aplicaciones que utilizan la solución con patrones, la solución sin patrones y una alternativa optimizada para los patrones de diseño que se integraban en las aplicaciones realizando cambios en los compiladores y sugiriendo como resultado que las transformaciones de los patrones *Observer* y *Decorator* son capaces de proporcionar reducciones en el consumo de energía del orden del 4,32 % al 25,47 %.

También existen diversos trabajos de refactorización de código fuente para eliminar *bad smells* así como sus efectos en la eficiencia energética como en [6], que observaron que el código refactorizado tenía un mayor tráfico de mensajes, con un tiempo de ejecución también mayor y con un mayor consumo de energía. En Silva [13] se abordó la aplicación de la técnica de refactorización "*inline method*", observando una mejora de rendimiento y reducción en el consumo de energía. Además, en Park [14] se realiza un análisis del efecto en el consumo de energía de todas las técnicas de refactorización de código sugeridas por Fowler [2]. Estos estudios evidencian el interés en estudiar el efecto de la refactorización en el consumo, porque por el simple hecho de refactorizar no queda demostrada una mejora en el mismo y la refactorización debe orientarse también a mejoras en consumo, como se realiza en [15], definiendo los "*energy bad smells*", y proponiendo una técnica de refactorización para la reducción de consumo de energía, demostrando así su utilidad. Otros estudios relevantes que abordan los efectos

de la refactorización para eliminar *bad smells* son los que podemos encontrar en [16] [17] [18] [19] [20].

En resumen, se observa un creciente interés en analizar el impacto del uso de patrones sobre el consumo de energía, así como el uso de la refactorización, habiéndose usado principalmente estimadores para obtener las mediciones de consumo. Cabe destacar que se observa una gran heterogeneidad en las conclusiones, lo que remarca la necesidad de crear una base empírica más sólida. En este trabajo se pretende aportar evidencia adicional sobre el uso de tres patrones de diseño, donde además se ha usado un dispositivo hardware para obtener mediciones de consumo del software evaluado más preciso y realista.

3 Marco de Trabajo y planificación del estudio

Para llevar a cabo el estudio empírico se utilizó el marco de trabajo FEETINGS (*Framework for Energy Efficiency Testing to Improve eNviromental Goals of the Software*) [21], que ayuda a la medición, el análisis y la visualización del consumo de energía de una aplicación software, y cuenta con: (i) un entorno metodológico denominado GSMP, que incluye las actividades necesarias para llevar a cabo los estudios de eficiencia energética de software, desde el diseño del estudio hasta el análisis y la comunicación de los resultados; y (ii) un entorno tecnológico que incluye:

1. EET: un dispositivo hardware capaz de medir el consumo energético de un conjunto de componentes hardware (procesador, disco duro, tarjeta gráfica, monitor) cuando se ejecuta un software en el PC, que denominaremos DUT (Device Under Test).
2. ELLIOT: aplicación software para el procesamiento y análisis de los datos recopilados por EET.

A continuación, se describe la planificación del estudio siguiendo el proceso GSMP. En particular nos centraremos en describir las variables, entidades software a evaluar y el entorno de medición y su preparación.

3.1 Variables y entidades software a evaluar

En el contexto de este estudio, podemos definir la variable independiente como el "uso del patrón de diseño software", cuyos valores serían: (1) Uso del patrón; (2) No uso del patrón. Por otro lado, la variable dependiente es la "Energía consumida (w/s)", calculada mediante el consumo de energía (w), obtenido con EET y el tiempo de ejecución (s). El consumo de energía se refina en tres medidas más concretas: (una para cada componente hardware sensorizado por EET): Procesador (consumo de energía del procesador), HDD (consumo de energía del disco duro) y DUT (consumo de energía total).

En cuanto a las entidades a analizar, se seleccionaron tres patrones de diseño software representativos, concretamente los patrones *Singleton*, *Composite* y *State*. Para cada patrón se realizaron mediciones sobre dos versiones de un mismo sistema: una que

implementa el patrón, y otra que no, cuyos códigos fueron extraídos de [22–24]. Para tener una muestra suficientemente representativa de, por un lado, los patrones y, por otro, la influencia del uso de patrones en el consumo energético, se hicieron seis casos de prueba diferentes por cada patrón, de acuerdo con una serie de criterios: (i) determinar el consumo energético del propio uso del patrón y (ii) determinar si el número de repeticiones está directamente relacionado con el consumo energético del patrón.

Se planificaron tres casos de prueba de un software con el patrón implementado y tres sin patrón. Asimismo, los diferentes casos de prueba se ejecutaron con un número diferente de repeticiones, dependiendo del patrón de diseño a medir, pero asegurando que las repeticiones eran suficientemente diferentes en cada caso de prueba para poder observar mayores diferencias en el consumo energético de cada patrón.

En resumen, para este estudio se han definido seis tipos diferentes de casos de prueba por cada patrón. Estos casos de prueba se agrupan en pares y cada par incluye un caso de prueba con patrón y otro sin patrón, pero con la misma funcionalidad y repeticiones, además de complejidad similar. La tabla 1 muestra un resumen de estos casos de estudio junto con el número de ejecuciones para cada caso.

Tabla 1. Resumen de los casos de prueba

Caso de prueba (Con y sin patrón)	Patrón estudiado	Número de ejecuciones
Casos de prueba 1 y 2	Singleton	750000
Caso de prueba 3 y 4	Singleton	500000
Caso de prueba 5 y 6	Singleton	250000
Caso de prueba 7 y 8	Composite	500000
Caso de prueba 9 y 10	Composite	250000
Caso de prueba 11 y 12	Composite	100000
Caso de prueba 13 y 14	State	4000000
Caso de prueba 15 y 16	State	2000000
Caso de prueba 17 y 18	State	1000000

3.2 Preparación del entorno de medición

La medición de consumo de energía se llevó a cabo mediante EET, mientras que el software ELLIOT se usó para el análisis de los datos obtenidos por EET. También se utilizó la plataforma SonarCloud para verificar que la complejidad del código era similar en los distintos casos de prueba. La configuración del DUT utilizada para ejecutar el estudio empírico se muestra en la tabla 2, donde se puede observar que se trata de un PC con una configuración convencional y sin altas capacidades de procesamiento, con el fin de que los resultados obtenidos puedas ser más generalizables.

Tabla 2. Especificaciones DUT

Elemento DUT	Características
Monitor	LCD Monitor Philips 170S6FS
Placa base	Asus M2N-SLI Delux
Procesador	AMD athlon tm 64 X2 Dual Core 5600+ 2,81 GHz
RAM	4 Modules of 1GB DD42 MHz
Tarjeta gráfica	Nvidia XfX 8600 GTS
Disco duro	Seagate Barracuda 7200 500Gb
Fuente de alimentación	350 W Aopenz350-08Fc

En el DUT se instaló Eclipse IDE 4.13.0. y el jre de Java. Para asegurar la consistencia de las medidas tomadas, la ejecución de cada uno de los casos de prueba se repitió 20 veces y los valores de las variables dependientes fueron por lo tanto el promedio de estas ejecuciones. De este modo se disponía de un tamaño de muestra suficientemente grande para mitigar el impacto de los valores atípicos debidos, por ejemplo, a fluctuaciones puntuales de energía que puedan surgir.

La forma en la que se han tratado las posibles amenazas a la validez de este estudio de acuerdo con las recomendaciones de [25] se describen a continuación. En relación con la fiabilidad de las mediciones (validez de constructo), hemos utilizado EET para medir el consumo real, habiendo sido previamente validado por comparación con un *gold estándar* [26].

En cuanto a los factores de validez interna que pueden afectar, los más destacables están relacionados con las condiciones en las que se realizaron las mediciones, destacando que se usó el mismo DUT y se tomaron medidas para que estuviese en las mismas condiciones para la realización de cada una de las diferentes ejecuciones (ej. sin aplicaciones en segundo plano, etc.). Además, se realizaron varias repeticiones de cada caso de prueba (concretamente 20) para mitigar los posibles valores atípicos relacionados con el consumo. Por último, relacionado con la validez externa, se han usado casos de ejemplo representativos adaptados de [22–24] para evaluar específicamente el uso o no de los patrones, siendo conveniente en el futuro aplicarlo a casos de empresa.

4 Resultados del estudio

En este apartado se describen los resultados del estudio analizando cada uno de los tres patrones evaluados.

4.1 Patrón *Singleton*

Este patrón [22] permite asegurar, por construcción, que sólo existirá una única instancia de un objeto de una clase, proporcionando así un único punto de acceso global a

dicha instancia. Este patrón por otro lado vulnera el principio de responsabilidad única. Para este patrón, las repeticiones que se establecieron fueron:

- Dos casos de prueba que se ejecutan 750.000 veces.
- Dos casos de prueba que se ejecutan 500.000 veces.
- Dos casos de prueba que se ejecutan 250.000 veces.

En la Figura 1 se muestran los resultados obtenidos y las gráficas de consumo para este patrón:

Media	ConSingleton 750k	SinSingleton 750k	ConSingleton 500k	SinSingleton 500k	ConSingleton 250k	SinSingleton 250k
Tiempo (s)	12,80595	375,44705	9,02535	238,2746	5,5257	117,1136842
HDD (w)	13,70087243	14,81841654	14,42096807	14,88359701	14,3642812	15,17126511
Procesador (w)	5,692872996	6,543519927	6,31907969	6,86002842	5,583314921	7,576378089
DUT (w)	193,9840138	184,3240894	199,9632448	206,0247518	201,5128611	211,8838855

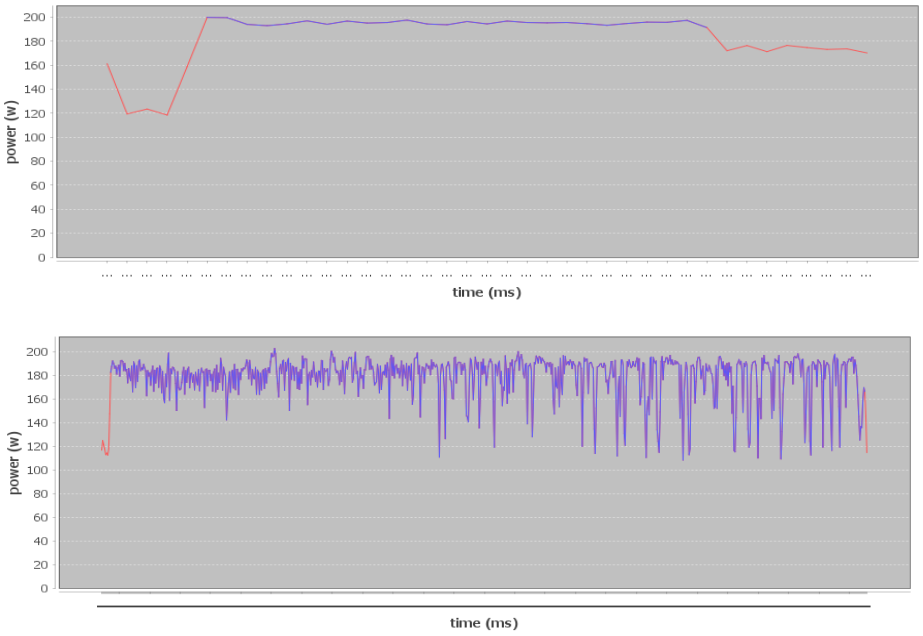

Fig. 1. Patrón Singleton: Resultados y gráficos de consumo de los casos de prueba con patrón (parte superior) y sin patrón (parte inferior)

A partir de los resultados del consumo de energía de cada uno de los seis casos de prueba definidos, se pueden extraer las siguientes conclusiones:

- Se aprecia cómo entre las versiones con y sin patrón existen claras diferencias en los tiempos de ejecución. En el caso de 250k repeticiones, el código sin patrón es un 2119 % más lento, en 500k es 2640 % y en 750k llega a ser 2931 %. Esto quiere decir que el código con patrón puede hacer muchas más iteraciones que el código sin patrón por unidad de tiempo. Considerando los demás parámetros excepto en la

variable DUT se puede observar como siempre se sigue la misma tendencia: el código sin patrón consume algo más de energía. Desde un 3 % de variación en el HDD de las 500k repeticiones hasta un 26 % más de consumo en el procesador en el caso de prueba de 250k repeticiones.

- Hay otra peculiaridad, en este caso en el DUT, ya que con 250K repeticiones el consumo del código sin patrón es un 4,8 % mayor, en 500k es un 2,9 % mayor y en 750k consigue consumir 5 % menos que el código con patrón, esto puede deberse a que con bajas repeticiones merece más la pena crear la instancia de la clase en cada repetición, pero según estas aumentan merece más la pena aplicar el patrón y que sólo haya una instancia de la clase.
- La gráfica de consumo de energía del código con patrón *Singleton* (ya sea de HDD, de DUT, o procesador) es lineal mientras que la del código sin patrón oscila bastante (ver Fig. 1). Un posible origen para este efecto puede ser que el código sin patrón tiene que instanciar el objeto de la clase en cada iteración, por lo que a eso se deben esos picos de consumo, mientras que en el código con patrón, la instancia del objeto se crea una única vez en toda la ejecución del programa, por eso el consumo es más estable.

Para obtener el consumo total de energía de cada caso de prueba (en w/s) se multiplicó el tiempo promedio (s) de ejecución de las 20 repeticiones de cada caso de prueba por el promedio del consumo (w). Estos resultados se muestran en la Fig. 2, donde se incluyen también las gráficas de consumo de procesador y total (DUT).

	ConSingleton	SinSingleton	ConSingleton	SinSingleton	ConSingleton	SinSingleton
mean (w/s)	750k con	750k sin	500K con	500K sin	250K con	250K sin
hdd	175,4526873	5563,530775	130,1542842	3546,383124	79,37270863	1776,762752
procesador	72,90264694	2456,745253	57,03190588	1634,570528	30,85172326	887,297551
dut	2484,149582	69203,9356	1804,738272	49090,46532	1113,499617	24814,50245

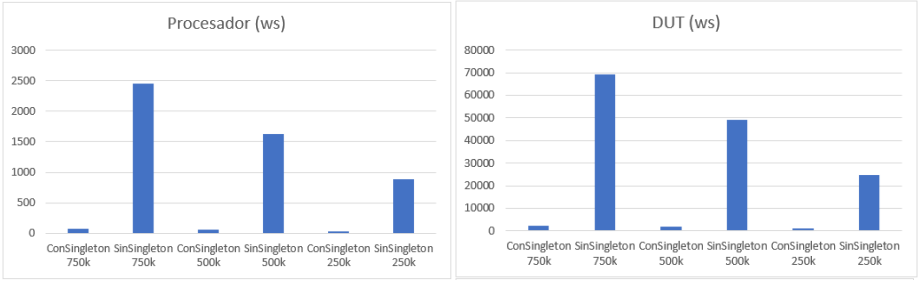

Fig. 2. Patrón Singleton: Resultados de consumo total (ws) (parte superior) y gráficos de consumo de procesador y total (DUT) de los casos de prueba con patrón y sin patrón (parte inferior)

Respecto al código sin patrón, se observa en la Fig. 2 que a mayor número de repeticiones hay un mayor consumo tanto en procesador, como en HDD y DUT, siempre consumiendo más que el código con patrón.

En conclusión, teniendo en cuenta los resultados obtenidos es aconsejable usar el patrón *Singleton* en lugar de una solución sin patrón para reducir así el consumo energético del sistema.

4.2 Patrón *Composite*

Composite [23] es un patrón de diseño estructural que permite componer objetos en estructuras de árbol, de modo que un objeto es siempre, o una parte de un todo, o un todo compuesto por varias partes. Las repeticiones para los casos de prueba de este patrón fueron:

- Dos casos de prueba que se ejecutan 500.000 veces.
- Dos casos de prueba que se ejecutan 250.000 veces.
- Dos casos de prueba que se ejecutan 100.000 veces

La tabla 3 muestra el impacto en el consumo de energía registrado por EET para cada uno de los casos de prueba.

Tabla 3. Energía consumida (w) para los casos de prueba del patrón Composite

	ConComposite	SinComposite	ConComposite	SinComposite	ConComposite	SinComposite
Media	500k	500k	250k	250k	100k	100k
Tiempo (s)	42,9356	40,46225	19,97610526	17,63275	6,797722222	7,796473684
HDD (w)	16,07684381	16,12222186	15,36373414	15,03623049	15,06301411	15,02148243
Procesador (w)	8,72777907	8,769642889	7,588385769	7,59670793	7,46063594	7,618906917
DUT (w)	219,7054917	218,8263472	211,836718	207,5639205	200,5015072	206,5974636

De acuerdo con los resultados mostrados en la tabla 3:

- Entre las versiones con y sin patrón existen pocas diferencias en los tiempos de ejecución. En el caso de 500k y 250k repeticiones, el código con patrón es un 6 % y 13 % más lento respectivamente. En el caso de 100k repeticiones ocurre el caso contrario, el código sin patrón es un 14 % más lento. En este sentido se observa que a bajas repeticiones es más recomendable aplicar el patrón, pero según se van aumentando las repeticiones, no merece la pena aplicarlo, aunque también hay que tener en cuenta que las diferencias de tiempo son mínimas por lo que el consumo total no sería muy diferente.
- Considerando las mediciones obtenidas sobre el componente HDD, se puede observar que el código sin patrón en los casos de 100k y 250k consume un 2 % menos que el código con patrón, pero en el caso de 500k repeticiones se da el efecto contrario. No sigue la misma tendencia que el parámetro del tiempo, pero de nuevo los resultados son muy similares.
- Respecto al consumo registrado del procesador se observa la misma tendencia: el código sin patrón consume algo más de energía. Desde un 0,1 % de variación en el caso de las 250k repeticiones hasta un 2 % más de consumo en el caso de prueba de 100k repeticiones. De nuevo los tiempos son muy similares por lo que el consumo total no tendría mucha variación tal como se analiza a continuación.

Se puede concluir a partir de estos resultados que, a altas repeticiones aunque el caso de prueba tarde un poco más, consume menos en los parámetros HDD y Procesador, esto puede ser debido a la escalabilidad que proporciona la estructura en árbol del patrón Composite, la cual es recorrida de forma recursiva simplificando el tratamiento de los objetos creados [23].

La Fig. 3 muestra el consumo total de energía y las gráficas de consumo del procesador y del DUT.

	ConComposite	SinComposite	ConComposite	SinComposite	ConComposite	SinComposite
Media(w/s)	500k con	500k sin	250k con	250k sin	100k con	100k sin
HDD	690,268935	652,3413716	306,9075703	265,1300932	102,3941857	117,1145924
Procesador	374,732431	354,839483	151,5863929	133,9508518	50,71533072	59,40060728
DUT	9433,187109	8854,206366	4231,672577	3659,922719	1362,953551	1610,731688

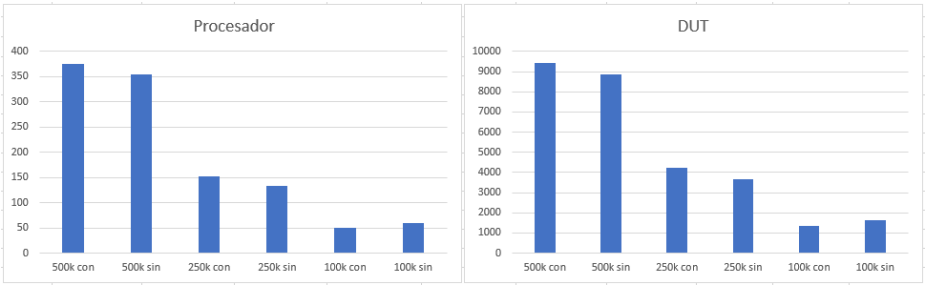

Fig. 3. Patrón Composite: Resultados de consumo total (ws) (parte superior) y gráficos de consumo de procesador y total (DUT) de los casos de prueba con patrón y sin patrón (parte inferior)

Tal como se observa en la Fig. 3, hay una clara diferencia de consumo de energía total (w/s): en el caso de 100k repeticiones, el caso de prueba sin patrón consume un 18% más, pero en los casos de 250k y 500k repeticiones se invierte la situación, consumiendo el caso de prueba con patrón un 15% y un 6% más respectivamente. Como conclusión se puede recomendar a bajas repeticiones aplicar el patrón, pero si estas aumentan lo mejor es no aplicarlo desde el punto de vista de la eficiencia energética.

4.3 Patrón *State*

State [24] es un patrón de diseño de comportamiento que permite a un objeto alterar su comportamiento cuando su estado interno cambia, además está estrechamente relacionado con el concepto de la máquina de estados finitos. Este patrón sugiere que se creen nuevas clases para todos los estados posibles de un objeto y se extraigan todos los comportamientos específicos del estado para colocarlos dentro de estas clases. Tal y como se realizó con los otros patrones, cada par de casos de prueba supuso la ejecución con un número diferente de repeticiones, concretamente:

- Dos casos de prueba que se ejecutan 4.000.000 veces.
- Dos casos de prueba que se ejecutan 2.000.000 veces.

- Dos casos de prueba que se ejecutan 1.000.000 veces.

La tabla 4 muestra los resultados obtenidos (tiempo y energía) de las ejecuciones de los distintos casos de prueba.

Tabla 4. Tiempo (s) y energía consumida (w) para los casos de prueba del patrón State

Media	4M conState	4M sinState	2M conState	2M sinState	1M conState	1M sinState
Tiempo (s)	29,6625	29,28942105	14,51855	15,98589474	8,48845	7,77505
HDD (w)	16,02544882	16,0216316	16,01183501	16,02443374	16,0288873	16,0103773
Processor (w)	8,174823079	8,187296319	8,210755074	8,160091308	8,33980652	8,22150806
DUT (w)	213,4677484	215,2316352	217,6845565	213,8710156	221,1714069	215,314104

A partir de los resultados mostrados en la tabla 4 se pueden extraer las siguientes conclusiones:

- Entre las versiones con y sin patrón existen muy pocas diferencias en los tiempos de ejecución. Es decir, no se observa ninguna tendencia en este sentido debida al número de repeticiones.
- Respecto al consumo del procesador, el código sin patrón en los casos de 1M y 2M consume menos que el código con patrón, menos de un 2 %, pero en el caso de 4M de repeticiones es lo contrario, el código sin patrón consume un 0,15 % más, esto podría deberse a que con el patrón *State* los diferentes estados están representados por un único atributo y no en uno diferente por cada estado, además, con muchas repeticiones el caso de prueba sin patrón estaría comprobando demasiadas condiciones al tener varios atributos, lo que hace que el consumo sea más elevado [24].
- Considerando el consumo del HDD se puede observar que no sigue la misma tendencia: el código sin patrón solo consume algo más de energía en el caso de 2M de repeticiones, concretamente un 0,07 % más, mientras que para los otros dos casos, es el código con patrón el que consume un 0,05 % más. Se puede apreciar que no sigue ninguna tendencia clara pudiendo deducir que, desde el punto de vista de la eficiencia energética, ambas implementaciones son equivalentes.
- Respecto al consumo de todo el PC (DUT), se sigue la misma tendencia que con el procesador. A bajas repeticiones no merece la pena aplicar el patrón, mientras que aumentando las repeticiones se invierte la situación consumiendo más el caso de prueba en el que no se aplica el patrón

El consumo total de energía y las gráficas de consumo del procesador y del DUT para el patrón *State* se muestran en la Fig. 4.

Media (w/s)	4M conState	4M sinState	2M conState	2M sinState	1M conState	1M sinState
HDD	475,3548757	469,264314	232,4686272	256,1649109	136,0604084	124,481484
Procesador	242,4856896	239,8011692	119,2082581	130,4463607	70,79203065	63,9226363
DUT	6331,987087	6304,009988	3160,464117	3418,919543	1877,402429	1674,07792

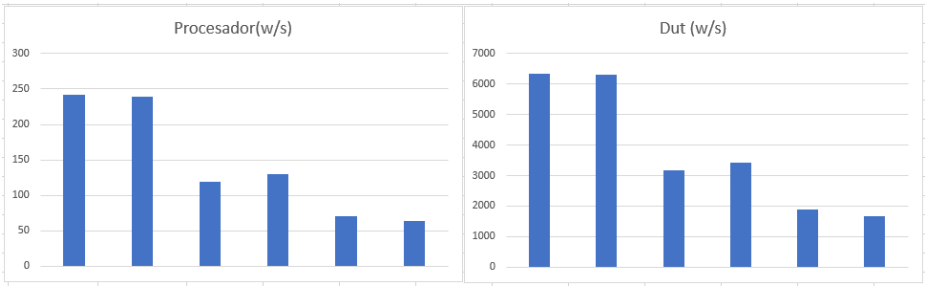

Fig. 4. Patrón State: Resultados de consumo total (ws) (parte superior) y gráficos de consumo de procesador y total (DUT) de los casos de prueba con patrón y sin patrón (parte inferior)

Tal como se puede observar en la Fig. 4, hay una pequeña diferencia de consumo de energía total (w/s) pero sin seguir ninguna pauta clara: en los casos de 1M y 4M de repeticiones puede observarse que la aplicación del patrón no mejora el consumo, mientras que en el otro caso sí, aunque el porcentaje de variación es mínimo.

En definitiva, en lo que respecta al patrón *State* no se observa una pauta clara de su efecto sobre el consumo de energía por lo que no se puede dar una recomendación al respecto.

5 Conclusiones

En este trabajo se ha presentado un estudio de evaluación de eficiencia energética del uso de los patrones *Singleton*, *Composite* y *State*. Para poder observar su efecto en distintos escenarios de aplicación, los casos de prueba se han diseñado considerando diferente número de repeticiones en el uso del patrón (o código sin patrón). Por otra parte, para proporcionar mediciones más fiables de consumo de energía se ha usado el entorno FEETINGS, dado que incluye en su entorno tecnológico un dispositivo de medición hardware, además de un entorno metodológico para llevar a cabo los estudios de medición de una manera sistemática. Como resultado, para el caso del patrón *Singleton* se pueden dar unas recomendaciones más claras desde el punto de vista de consumo de energía, dado que el estudio evidencia que su uso ayuda a reducir el consumo energético del sistema. Respecto al patrón *Composite* podemos afirmar de acuerdo con los resultados obtenidos, que a bajo número de repeticiones (de uso) es mejor aplicarlo respecto a una solución sin patrón ya que, si no, se estaría aumentando el consumo energético. En el caso del patrón *State* no se pueden dar recomendaciones, ya que no se observan tendencias claras al respecto teniendo en cuenta el número de repeticiones del patrón, por lo que no podemos confirmar los resultados de otros estudios como [7] donde su uso parece perjudicar la eficiencia energética del sistema. En relación con otros estudios del patrón *Composite* [13] se confirma con nuestro estudio que a altas repeticiones no se favorece la eficiencia energética, pero a bajas repeticiones podría favorecerla.

Las conclusiones que se obtienen de este estudio, así como los siguientes experimentos que se lleven a cabo analizando otros patrones de diseño o *bad-smells* que, aun siendo negativos desde un punto de la mantenibilidad podrían aportar valor sobre la reducción del consumo energético del software, nos permitirán obtener indicadores especializados que nos indicaran en qué medida un software compromete el consumo energético. Uno de estos indicadores es la "*Deuda Ecológica*", que se define como "*el coste (en términos de uso de recursos) de entregar un sistema de software con un grado de greenability inferior al nivel de los requisitos no funcionales establecidos por las partes interesadas, más el coste necesario para refactorizar el sistema en el futuro*" [27], inspirada en el conocido concepto de deuda técnica [28].

El trabajo futuro se centrará en: seguir realizando nuevos estudios que confirmen o amplíen la evidencia empírica de los estudios realizados; extender los casos de estudio para evaluar también otros aspectos claves en el diseño como el uso de antipatrones y *bad-smells*; generar recomendaciones y buenas prácticas a partir de los resultados obtenidos. Aunque se espera validar los resultados en equipos de nueva generación, consideramos que los efectos de la ejecución del software seguirán siendo similares, aunque puedan variar el tiempo y el consumo de la medición.

Agradecimientos

Este trabajo ha sido financiado por los siguientes proyectos: OASSIS (PID2021-122554OB-C31/ AEI/10.13039/ 501100011033/FEDER, UE); EMMA (Project SBPLY/21/180501/000115, funded by CECD (JCCM) and FEDER funds); SEEAT (PDC2022-133249-C31 funded by MCIN/AEI/ 10.13039/501100011033 and European Union NextGenerationEU/PRTR); PLAGEMIS (TED2021-129245B-C22 funded by MCIN/AEI/ 10.13039/501100011033 and European Union NextGenerationEU/PRTR).

References

1. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Reusable Object-Oriented Software. , Reading, Mass (1994).
2. Fowler, M., Beck, K., Brant, J., Opdyke, W., Roberts, D., Gamma, E.: Refactoring: Improving the Design of Existing Code. Addison-Wesley Professional, Reading, MA (1999).
3. Calero, C., Piattini, M.: Puzzling out Software Sustainability. Sustainable Computing: Informatics and Systems. 16, (2017). https://doi.org/10.1016/j.suscom.2017.10.011.
4. Pinto, G., Castor, F.: Energy efficiency: a new concern for application software developers. Commun. ACM. 60, 68–75 (2017). https://doi.org/10.1145/3154384.
5. Kern, E., Hilty, L.M., Guldner, A., Maksimov, Y.V., Filler, A., Gröger, J., Naumann, S.: Sustainable software products—Towards assessment criteria for resource and energy efficiency. Future Generation Computer Systems. 86, 199–210 (2018). https://doi.org/10.1016/j.future.2018.02.044.

6. Perez-Castillo, R., Piattini, M.: Analyzing the Harmful Effect of God Class Refactoring on Power Consumption : GREEN SOFTWARE. IEEE softw. 31, 48–54 (2014).
7. Feitosa, D., Alders, R., Ampatzoglou, A., Avgeriou, P., Nakagawa, E.Y.: Investigating the effect of design patterns on energy consumption. Journal of Software: Evolution and Process. 29, e1851 (2017). https://doi.org/10.1002/smr.1851.
8. Sahin, C., Cayci, F., Manotas, I., Clause, J., Kiamilev, F., Pollock, L., Winbladh, K.: Initial explorations on design pattern energy usage. 2012 1st International Workshop on Green and Sustainable Software, GREENS 2012 - Proceedings. (2012). https://doi.org/10.1109/GREENS.2012.6224257.
9. Bunse, C., Stiemer, S.: On the Energy Consumption of Design Patterns. Softwaretechnik-Trends. 33, 7–8 (2013). https://doi.org/10.1007/s40568-013-0020-6.
10. Litke, A., Zotos, K., Chatzigeorgiou, A., Stephanides, G.: Energy Consumption Analysis of Design Patterns. World Academy of Science, Engineering and Technology, International Journal of Electrical, Computer, Energetic, Electronic and Communication Engineering. (2007).
11. Qasim, A., Munawar, A., Hassan, J., Khalid, A.: Evaluating the Impact of Design Pattern Usage on Energy Consumption of Applications for Mobile Platform. Applied Computer Systems. 26, 1–11 (2021). https://doi.org/10.2478/acss-2021-0001.
12. Noureddine, A., Rajan, A.: Optimising Energy Consumption of Design Patterns. Presented at the May 1 (2015). https://doi.org/10.1109/ICSE.2015.208.
13. Silva, W., Brisolara, L., Corrêa, U., Carro, L.: Evaluation of the impact of code refactoring on embedded software efficiency. Presented at the January 1 (2010). https://doi.org/10.13140/2.1.1481.8249.
14. Park, J., Hong, J.-E., Lee, S.-H.: Investigation for Software Power Consumption of Code Refactoring Techniques. Presented at the International Conference on Software Engineering and Knowledge Engineering (2014).
15. Lee, J.-W., Kim, D., Hoing, J.-E.: Code Refactoring Techniques Based on Energy Bad Smells for Reducing Energy Consumption. KIPS Transactions on Software and Data Engineering. 5, 209–220 (2016). https://doi.org/10.3745/KTSDE.2016.5.5.209.
16. Rodriguez, A., Longo, M., Zunino, A.: Using bad smell-driven code refactorings in mobile applications to reduce battery usage. Presented at the Simposio Argentino de Ingeniería de Software (ASSE 2015) - JAIIO 44 (2015).
17. Verdecchia, R., Saez, R., Procaccianti, G., Lago, P.: Empirical Evaluation of the Energy Impact of Refactoring Code Smells. Presented at the 5th International Conference on ICT4S , Toronto, Canada February 2 (2018). https://doi.org/10.29007/dz83.
18. Palomba, F., Di Nucci, D., Panichella, A., Zaidman, A., De Lucia, A.: On the impact of code smells on the energy consumption of mobile applications. Information and Software Technology. 105, 43–55 (2019). https://doi.org/10.1016/j.infsof.2018.08.004.
19. Sehgal, R., Mehrotra, D., Nagpal, R., Sharma, R.: Green software: Refactoring approach. Journal of King Saud University - Computer and Information Sciences. 34, 4635–4643 (2022). https://doi.org/10.1016/j.jksuci.2020.10.022.
20. Guamán, D., Pérez, J., Valdiviezo-Diaz, P.: Estimating the energy consumption of model-view-controller applications. J Supercomput. (2023). https://doi.org/10.1007/s11227-023-05202-6.

21. Mancebo, J., Calero, C., Garcia, F., Moraga, M., Guzmán, I.: FEETINGS: Framework for Energy Efficiency Testing to Improve eNvironmental Goal of the Software. Sustainable Computing: Informatics and Systems. 30, 100558 (2021). https://doi.org/10.1016/j.suscom.2021.100558.
22. Singleton, https://refactoring.guru/es/design-patterns/singleton, last accessed 2023/04/26.
23. Composite, https://refactoring.guru/es/design-patterns/composite, last accessed 2023/04/26.
24. State, https://refactoring.guru/es/design-patterns/state, last accessed 2023/04/26.
25. Wohlin, C., Runeson, P., Höst, M., Ohlsson, M.C., Regnell, B., Wesslén, A.: Experimentation in Software Engineering. Springer, Berlin, Heidelberg (2012). https://doi.org/10.1007/978-3-642-29044-2.
26. Mancebo, J., Guldner, A., Kern, E., Kesseler, P., Kreten, S., Garcia, F., Calero, C., Naumann, S.: Assessing the Sustainability of Software Products—A Method Comparison. In: Schaldach, R., Simon, K.-H., Weismüller, J., and Wohlgemuth, V. (eds.) Advances and New Trends in Environmental Informatics. pp. 1–15. Springer International Publishing, Cham (2020). https://doi.org/10.1007/978-3-030-30862-9_1.
27. Guzmán, I.G.-R., Piattini, M., Pérez-Castillo, R.: Green Software Maintenance. In: Calero, C. and Piattini, M. (eds.) Green in Software Engineering. pp. 205–229. Springer International Publishing, Cham (2015). https://doi.org/10.1007/978-3-319-08581-4_9.
28. Kruchten, P., Nord, R.L., Ozkaya, I., Falessi, D.: Technical debt: towards a crisper definition report on the 4th international workshop on managing technical debt. SIGSOFT Softw. Eng. Notes. 38, 51–54 (2013). https://doi.org/10.1145/2507288.2507326.