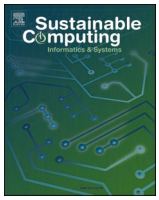
Sustainable Computing: Informatics and Systems 32 (2021) 100603

Contents lists available at ScienceDirect

Sustainable Computing: Informatics and Systems

journal homepage: www.elsevier.com/locate/suscom



Investigating the impact on execution time and energy consumption of developing with Spring

Coral Calero *, Macario Polo, Mª Ángeles Moraga

Alarcos Research Group, University of Castilla-La Mancha, Paseo de la Universidad, 4 13071 Ciudad, Real, Spain

ARTICLE INFO

Keywords:
Energy consumption
Spring
Green software
FEETINGS
Software sustainability

ABSTRACT

Respect for the environment has become a feature of life that is essential to take into account in our present-day society. Nevertheless, although software consumes large amounts of energy, in the software development sector it seems that awareness of this issue is still lacking. As software engineers, we must contribute to bringing about a change of vision in the sector. A good way of doing so is by giving companies clear guidelines on how to act. In this paper we present an example of this, focused on Spring. Spring is a server-side Java development framework that reduces the time to market of new applications, helps developers to save a great deal of development time, and hence improves their productivity. Our objective is to discover whether all these advantages are also accompanied by good energy consumption behaviour. To that end, we have compared the execution time and the energy consumption required by three releases of the same application, developed with and without Spring. This paper presents all the details of the study carried out, and sets forth our conclusions on the suitability of using Spring in the development of software applications.

1. Introduction

Our dependence on software as human beings is proved by the fact that it is nowadays difficult to find any activity which does not rely on software; people have become used to placing on software applications the responsibility for managing a significant part of their daily lives. This is true with regards to their work, but even more so in relation to their personal activities. Most of us, however, are not aware that using software implies energy consumption.

Andrae and Edler [1] project and analyse several scenarios of energy consumption in information technologies as we look towards 2030: by then, in the worst case scenario, up to 51 % of global electricity could be required for ICT (Information and Communication Technology), or 8 % in the best of possible cases, and 13 % in the expected scenario.

With software being a key part of ICT, and taking into account that society is increasingly concerned about environmental issues in almost all aspects of life, the lack of awareness regarding software consumption is surprising. This could be explained by the fact that there has not been a specific culture built up around the issue; to date, software sustainability has been the subject of only a few studies [2]. In fact, this lack of culture is patently obvious in software companies, which should take sustainability aspects into account as part of their business objectives, and incorporate them in the actions defined in their Corporative Social Responsibility (CSR) documentation. According to Calero et al. [3], the majority of the CSR actions undertaken by the top ten software companies are not related to software sustainability aspects.

Fortunately, as the use of mobile applications, embedded systems, and data center-based services expands [4], software sustainability in general, and software energy consumption in particular, are increasingly becoming a concern. It would seem obvious that, since end-users can very easily see in their energy bills the consumption of their desktop and laptop computers, they should prefer software that consumes the least possible amount of energy [5].

Little by little, then, sustainability is becoming an issue that should encourage software practitioners to develop, manufacture and use computers, servers and peripherals efficiently and effectively so as to reduce damage to the environment [6]. However, research contributions about current practices and the perspectives of software engineers in the software sustainability field are still scarce [4].

In an attempt to facilitate good practices among software engineers in developing sustainable software, we analyse in this paper whether the use of Spring affects the energy efficiency of a software application or not.

Spring [7] is a very widely used framework for developing the server

* Corresponding autor.
E-mail address: coral.calero@uclm.es (C. Calero).

https://doi.org/10.1016/j.suscom.2021.100603
Received 24 January 2021; Received in revised form 13 July 2021; Accepted 18 August 2021
Available online 28 August 2021

side of Java applications in businesses and, although there exist other frameworks with similar objectives (Spark or VRaptor, for example), Spring is the most widely extended. It is based on the design patterns *Dependency Injection* and *Inversion of Control*. Spring defines many Java annotations which are processed in runtime using the Java Reflection API. These annotations allow declarative-programming constructions to be inserted in the application. Spring helps developers save a lot of development time, since it avoids the writing of boilerplate code, so improving their productivity during development. In fact, as remarked in [7] "*Spring makes programming Java quicker, easier, and safer for everybody*." However, as will be shown in Section 2, processing the annotations to generate code at runtime implies a massive use of reflection that in turn leads to additional CPU cycles for the computer running the system.

According to Belani [8], "nearly 90 percent of Fortune 500 firms rely on Java for their desktop applications and backend development projects"and, according to a survey carried out by the Java-specialist web portal Baeldung,[1] 83.7 % of 5160 companies surveyed use some version of Spring in their software development. It is therefore of interest to investigate what the long-term impact of using this technology may be, especially considering that the programs and services offered by many 24×7 servers are implemented in Java and use Spring as supporting technology.

Our goal for this paper is to analyse the impact of the use of Spring compared to the use of a traditional approach, from the perspective not only of execution time but energy consumption as well.

From a financial perspective, and depending on the results obtained, companies might prefer to acquire systems which employ technologies that consume less energy, even though this could require a higher initial investment in their software development. Also, from the point of view of social responsibility, it may now be seen as the time to move towards more energy-efficient technologies.

This paper is structured as follows: Section 2 presents a brief overview of reflective programming and Spring, along with the main objectives of this work. Section 3 shows some related work, and in Section 4 the experiment is explained in detail. Once the foundations of the experiment have been presented, the procedure to carry it out is analysed in Section 5. The threats to validity of the experiment are explained in Section 6 and, lastly, conclusions are summarised and future work outlined in Section 7.

2. A brief overview of spring, and the experimental objective in a nutshell

Spring [7] is a technology widely-used for developing Java applications. By means of a broad set of predefined Java annotations, programmers can avoid writing a lot of boilerplate and repetitive code, thus reducing the time to market of new applications.

Java annotations consist of a word that qualifies any element (class, field, constructor or method) in a Java program. Annotated elements are processed with the Java Reflection API. The annotation processor may give the annotated element a specific behaviour. In Java, any object is an instance of a class, but the self class is also an instance of the *Class* reflective class: *john* being an instance of *Person. Person* is an instance of the *Class* reflective class: as is seen in Fig. 1, a *Class* knows its whole set of members via the reflective *Constructor, Method* and *Field* classes; the first two are *Executable,* and all of them are *Member*s.

A programmer can recover the class of an object and reflectively inspect and manipulate its members. The code in the top part of Fig. 2 is the declaration of a simple *Person* class, with two fields and two setter methods. In the bottom part, the *main* function creates an instance of that class and assigns values to its fields.

The same effect as in the bottom part of Fig. 2 can be obtained reflectively, as seen in Fig. 3: the reflective *Class* corresponding to *Person* is recovered; we then invoke its non-parameter constructor with the call to the reflective *newInstance* method (a member of *Class)*. The next two statements reflectively search the *setName* and *setLastName* methods in the class, which are later invoked with the desired arguments. Finally, *print* prints the name and last name of the object.

Elements in the *Person* class could be annotated to receive specialised processing. Suppose we want the value of the *name* field to always be written in uppercase. We could create an *@Uppercase* annotation to annotate this field, as seen in Fig. 4. The *print* method should be slightly modified so as to pass those fields with that annotation into uppercase.

Spring includes many annotations that facilitate the development of the server side of applications. Annotated elements are processed by annotation processors which give a specific treatment to each annotated element.

Spring applications are launched from a boot: before launching the application logic, the run method of *SpringApplication* reviews the correctness of all the classes in the package and subpackages. This revision inspects the annotated classes and, within each class, its annotated elements. If Spring finds any error, the run method fails and the application is not started.

From the perspective of the developer, the benefit of using Spring is clear, but our objective here is to analyse the impact that the use or non-use of Spring has on the time and energy required to perform given operations. This paper compares the execution time and energy required by two different types of Java backends that offer exactly the same set of services. Both backends implement a server of board games. The players in the games consume the services using their user agents. One of the backends is implemented using Spring, while the other is not - and so does not take advantage of the well-known declarative constructions of this framework.

The code in Fig. 5, for example, shows some lines of the *User* class of an actual project: since it is annotated as an *@Entity,* Spring knows that there must be a correspondence of this class with a table in the database called *User*. Thanks to the *@Id* annotation in the *userName* field, Spring will use the *userName* column in the table as the primary key. The *projects* field is annotated with *@OneToMany,* which tells Spring that there exists a *1..n* foreign key relationship from the *Project* table to *User*. As seen in this last example, annotations may contain additional information between brackets, which Spring can also use in order to modify the default behaviour assigned to the annotated element.

In a traditional approach, the programmer would write a specific DAO (Data Access Object) class to deal with the persistence of *User* instances. The *UserDAO* class, for example, should contain the classic CRUD operations to manipulate the instances of this persistent class in the database. With Spring, since the class is annotated as an *@Entity,* the programmer needs only to write an interface (such as that in the top side of Fig. 6). Even though the interface is empty, it offers the programmer many more operations than the basic *select, insert, delete* and *update*. The bottom part of the figure shows the operations that *CrudRepository* offers by default: *save* for inserting and updating; *findById* to build an instance from a record in the database; *deleteById* for deleting a record, etcetera.

As may be seen, Spring allows the development of applications to be accelerated, thanks to the time savings in writing of the code. However, our hypothesis is that the reflective processing of the annotated elements requires more CPU cycles, more execution time and, most likely, more energy than would be the case if performing the same tasks with no reflection, in what we have called the "traditional approach".

The experimental work presented in this paper thus attempts to determine whether the use of Spring is better than a "classical" approach in terms of both execution time and energy consumption. In an effort to obtain a more general view of the software engineering practice, we have considered the different steps relative to green software engineering, as identified in [9] and shown in Fig. 7. We thus execute our experimental work during development, operation and maintenance of the software under study (i.e. developed using Spring, and developed

[1] https://www.baeldung.com.

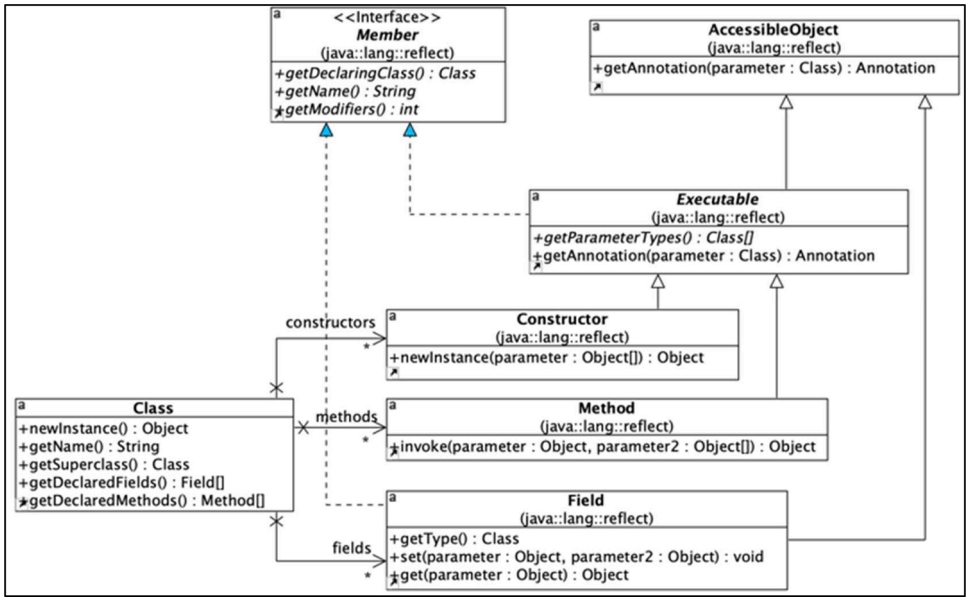


Fig. 1. Some classes of the Java Reflection API.

```
package example;

public class Person {
    private String name;
    private String lastName ;

    public void setName(String name) {
        this.name = name;
    }

    public void setLastName(String lastName ) {
        this.lastName = lastName ;
    }
}
```

```
package example;

public class Main {

    public static void main(String[] args) {
        Person john = new Person();
        john.setName ( "John");
        john.setLastName( "Smith");
    }
}
```

Fig. 2. The Person class and the code for creating an instance.

```
public static void main(String[] args) throws Exception {
    Class<Person> clazz = Person. class;

    Person john = clazz .newInstance();

    Method setName = clazz .getDeclaredMethod( "setName" ,
        String. class);
    Method setLastName = clazz .getDeclaredMethod( "setLastName" ,
        String. class);

    setName .invoke( john, "John");
    setLastName .invoke( john, "Smith");

    print(john);
}
```

Fig. 3. Reflective instantiation of *"John Smith"*.

not using Spring).

3. Software engineering measurement background

It is important that developers be aware of the energy consumption of the software they develop. However, as indicated in [10], programmers do not have much experience with regard to software energy consumption. Pinto and Castor [11] conducted a survey with software

```
package example;

public class Person {
    @Uppercase
    private String name;
    private String lastName ;

    ...
}
```

```
...

if (field.isAnnotationPresent( Uppercase .class))
    value = value.toUpperCase();
System. out.println(fieldName + ": " + value);
...
```

Fig. 4. Annotation of a field (top) and processing of *@Uppercase*.

```
@Entity
public class User {
    @Id
    private String userName ;
    @Column @JsonIgnore
    private String email;
    @Column @JsonIgnore
    private String pwd;
    @OneToMany (cascade = CascadeType. REMOVE, mappedBy = "creator")
    private List<Project> projects;
    ...
}
```

Fig. 5. A class with some Spring annotations.

developers, seeking to understand their perceptions regarding software energy consumption issues, and concluded that they do not fully understand how to write, maintain and evolve energy-efficient software systems. Software developers currently have to rely on Q&A websites, blog posts, or YouTube videos when trying to optimise energy consumption; these resources are anecdotal, not supported by empirical evidence, and may even be incorrect [4,12].

Fortunately, some efforts have been undertaken in an endeavour to obtain more reliable results. For example, Pereira et al. analysed 27 software languages [13], aiming to provide software engineers with support in deciding which language to use from the perspective of

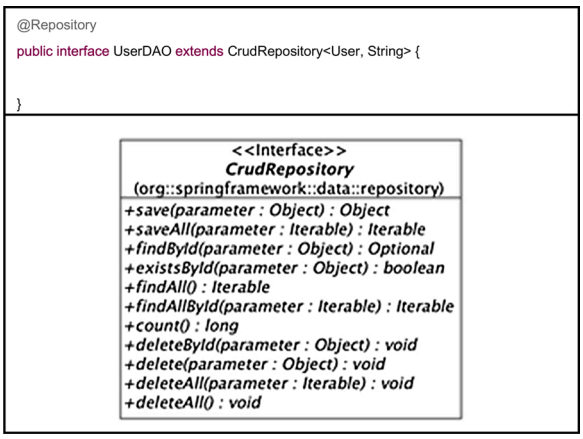


Fig. 6. The *UserDAO* interface deals with the persistence of *Users*.

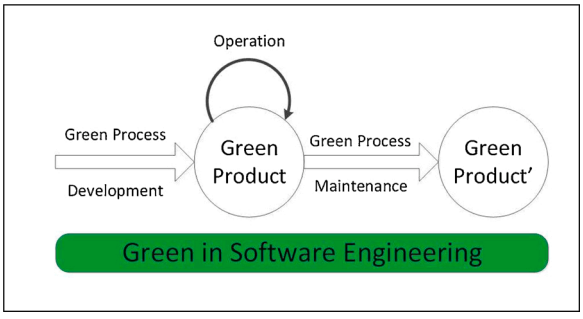


Fig. 7. Green In Software Engineering [9].

energy efficiency. To measure the energy consumption, they used Intel's Running Average Power Limit (RAPL) tool, which employs a software power model to estimate the energy at a very detailed level. This same tool was used to study, for instance, the energy behaviour of programs written in Haskell [14], the energy consumption of short code paths [15], or the energy consumption of the different Java Collection Framework (JFC) implementations [16]. In [17] the authors propose a spectrum-based energy leak localisation technique, namely SPELL, to identify inefficient energy consumption in the source code of software systems. As in the research previously mentioned, they used the RAPL tool to estimate the energy consumption. In [18] the authors propose the GreenOracle model, which is based on the dynamic traces of system calls and CPU utilisation, to estimate software energy consumption. The model was later improved, as a result of which the GreenScaler model came into being [19].

In [20] the author proposes a model for analysing the asynchronous power consumption of Android applications. Li et al. [21] calculate the source line level energy consumption of mobile apps by combining hardware-based power measurements with program analysis and statistical modeling. Zang et al. propose software application ratings based on their energy consumption [22]. Eco Droid, proposed in [23], is an approach that automatically ranks applications. A tool for monitoring power consumption of an executing application, developed in [24], is used to study how applying design patterns can impact on energy usage.

Maintenance is another stage where software energy consumption must be taken into consideration. However, as indicated in [4], energy concerns have largely been ignored during maintenance. In this respect, Cruz et al. explore whether improving energy efficiency by applying energy efficiency patterns has a negative impact on maintainability [25].

In [26] the authors try to demonstrate that, on the domain of an Android app, choosing a bundled MVP architecture can improve the sustainability and energy consumption of a system without negatively impacting system maintainability.

The general conclusions arising from the overview of energy measurement presented above include: (1) there do exist some papers related to energy consumption during software operation; (2) there are hardly any works that measure the real consumption of software during its development, and many of them are based on estimations; and (3) not enough work has so far been undertaken to study the relationship between energy consumption and maintenance.

The study we have carried out considers and seeks to minimise these weaknesses by: (1) measuring the energy consumption during the execution of a software; (2) capturing measurements of the real consumption and execution time needed; and (3) measuring these aspects during maintenance.

4. Experiment description

Our main goal is to check whether the use or non-use of Spring in the developing of the backend of applications influences execution time and energy consumption. Given that the development of an application is done once but it is executed several times, we focused our study in the long term. Therefore, measuring the energy required during the development itself is beyond the scope of this article.

To the best of our knowledge, no repositories can be found in which the same application has been developed alternatively using and not using Spring. For that reason, we decided to develop our own applications. We are well aware that, for a single example, the differences could very well be small. Nevertheless, although a single application may not have any great influence itself, the long-term impact of using one or another technology in lots of applications running throughout the whole world could mean a significant variation in cost in terms of CO_2 emissions.

To accomplish our goal, we commissioned a senior programmer (with more than 15 years of experience in Java programming) to develop two versions of the backend of the same system using Java as program languague - one with Spring and the other without Spring. Due to the fact of these two different versions being developed by the same programmer, any bias produced by having several programmers using different programming styles was eliminated. Moreover, each version has by now had three releases, which allowed us to incorporate maintenance activities into the study. We measured execution time and energy consumption for all releases of each version, using functional and load tests.

4.1. Research questions

The research questions we seek to answer are the following:

RQ1. Is there a relationship between the execution time required by an application at run time and the use of Spring during its development?

RQ2. Is there a relationship between the energy consumption needed by an application at run time and the use of Spring during its development?

4.2. Description of the system versions

The system we developed consists of a server used for the playing of several board games. One version was developed with Spring, and the other without it, but both versions (Fig. 8) offer exactly the same functionalities.

In order to play a match in one of the board games on offer, the user must first be logged into the server, which requires prior registration. If necessary, the user can recover their password at any moment. Once the user is logged in, they can select one of the games offered and play a match. If there is another player waiting to play a match in that same game, the match is started automatically; otherwise, the server creates a "pending match" and the player must wait until another user requests to join a match in the same game. When the match has two players, play

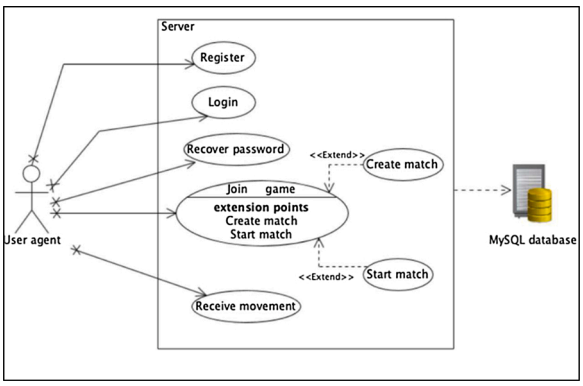


Fig. 8. Functional view of the two system versions.

begins and each user makes moves, which are collected by the server and communicated to their respective opponent via a Websocket.

All the information (users, matches and moves performed in each match) is recorded in a MySQL database (Fig. 9) which is located in a different physical machine. The CPU cycles executed by the database server do not influence our measures.

4.3. Description of the three releases

To cover the different stages shown in Fig. 7, the programmer was asked to implement three releases of each system version:

The **first release**, the development stage, offered two games: the well-known *Tic-tac-toe* and *Kuar*. In Kuar two players are given a square matrix with several unordered numbers (Fig. 10). In their user-agent, the players must sort the numbers on the top board by moving them. The moves performed by your opponent are shown on the bottom board. As shown in Fig. 10, a match of Kuar is taking place with *Lucas*, the local player, playing against *Pepe*. Note that the position of the number 5 has been changed in the bottom board from the right to the central column: this is because *Pepe* has moved number 5 in his user-agent. On the right-hand side of the image, we see that, since *Lucas* sorted all the numbers before *Pepe*, he wins the match.

In both versions (Spring and Non-Spring), the business logic is accessed through a *Manager*, which keeps a collection of the *players* currently logged on, of the *games* on offer and of the *matches* currently *in play*. As seen in Fig. 11, *Game* is an abstract class that records a collection of the *pending matches* (ongoing matches not yet completed). *Match* is also an abstract class that has an abstract *Board* and knows up to three players: *playerA* and *playerB* (which correspond to the player who created the match and the player who joined the existing match, respectively) and *winner*, a reference to either *playerA* or *playerB*, which records who has won the match. *AbstractPlayer* has two specialisations (which are not of interest in this context), depending on the method used by the user for login to the server.

Fig. 12 shows the specialisations of *Game*, *Match* and *Board* for the game *Tic-tac-toe*: since both players in a *Tictactoe* match share the same board, each *TictactoeMatch* knows just one instance of *TictactoeBoard*. The Kuar game (Fig. 13) likewise requires the same specialisations (*KuarGame, KuarMatch* and *KuarBoard*), but its implementation is more complex, because each player has their own board (although the initial configuration of the board is the same it then changes according to the players' moves).

The Non-Spring version has the three-layer design shown in Fig. 14. The persistence responsibilities of the business classes are delegated to several *DAO* (Data Access Objects) classes, which directly include the embedded SQL code necessary to execute the desired operations by means of *PreparedStatement* objects

The Spring version follows a *Model-View-Controller* pattern (Fig. 15). The *Manager* acts as the controller and is in charge of requesting Spring repositories to deal with the persistence of persistent classes.

As stated earlier, each move made by a player is recorded in the database:

- In the *Non-Spring* version, the method in charge of inserting the move is the one shown in Fig. 16: it builds a *PreparedStatement* object and sets the values to the corresponding parameters.
- In the *Spring* version, *Movement* is an *@Entity-annotated* class whose persistence is managed with an empty *CrudRepository* (see Fig. 6), whose code appears in Fig. 17.

The **second release** proceeds from a preventive maintenance intervention carried out after having analysed both versions with SonarCloud. Preventive maintenance improves the internal quality of a software product (maintainability, for example) without modifying its functionalities. Accordingly, this second release offers the same two games (*Tic-tac-toe* and *Kuar)* in exactly the same manner as in the previous release.

The reason for performing this maintenance intervention was to remove any possible bad practices of the programmer, "inherited" after many years of coding in Java, as well to gain two balanced, quantitatively-equivalent versions.

Table 1 summarises the results of the SonarCloud static analysis. Note that the Spring version shows better values in all the numeric variables, but also that both versions pass the "Quality gate", which is a set of boolean values that indicate whether the project is ready to pass to production.

The Non-Spring system has 417 lines of code more than the Spring system. Most of this difference is attributable to the DAO classes, and consists in very similar methods that perform persistence operations with the database. Most of the problems detected in the Non-Spring system (bugs, vulnerabilities, etc.) are also to be found concentrated in these classes.

When we showed the Sonar report to the developer he acknowledged that many of the highlighted issues correspond to not-so-good practices that he has been carrying out in his work over many years. For example, 36 of the 119 code smells in the Non-Spring system are attributable to methods that trigger the generic Java *Exception*, instead of a specific exception. It should be noted that after reading the Sonar report the developer fixed all the issues that were flagged in it.

Subsequently, a different senior programmer reviewed the same code and a new Sonar report, certifying that the highlighted issues did not have any influence on the different behaviour or performance of any of the releases in either of the two versions.

The **third release** corresponds to a perfective maintenance intervention. Perfective maintenance consists in the addition of new functionalities to a previous release. Hence, the programmer was asked to add a third game, *Ladders,* to both versions. Architecturally speaking, it requires the implementation of specialisations of *Game, Match* and *Board,* as well as the logic required for processing matches of this well-known game.

Fig. 18 summarises the complete development process:

- **Release 1** is a "crude version" of a server that offers *Tic-tac-toe* and *Kuar*.
- **Release 2** is a "clean version" of Release 1, after having solved all the problems detected by the static analysis. To be precise, it is a version of Release 1 after a *preventive maintenance* intervention has been carried out.
- **Release 3** corresponds to a *perfective maintenance intervention* on Release 2, since it consists of the addition of new functionalities to the system.

5. Experimental procedure

In the execution of our experiment we followed the process defined

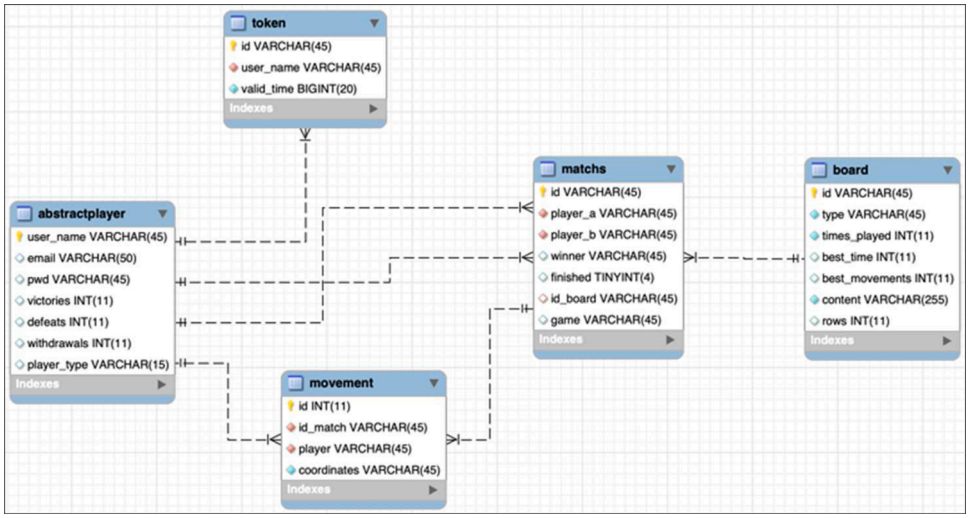


Fig. 9. Structure of the database.

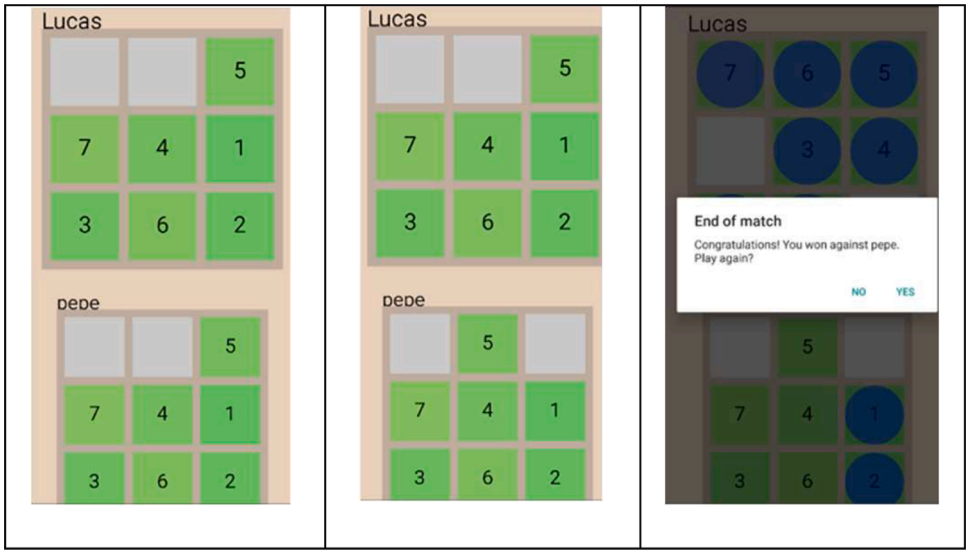


Fig. 10. The *Kuar* game.

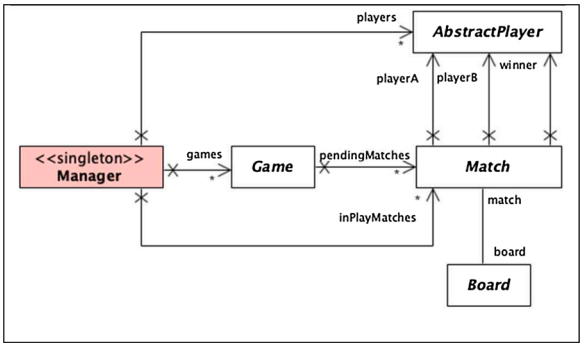


Fig. 11. General structure of the business layer in the Tic-tac-toe game.

by [27], which focuses on the analysis of software energy consumption. The process is composed of a set of phases which cover all the steps needed to carry out a correct analysis of the energy consumption of a software while it is being executed (Fig. 19).

5.1. Phase 1 (Scope Definition)

According to [27], in this phase it is necessary to: (1) define the objective, (2) choose the Software Entity Class under study and (3) choose the Software Entity, which is the software that is to be characterised by measuring its attributes. Finally, the fourth activity (4) is the development of test cases to execute and measure energy consumption.

For the first step (**definition of the objective**), the authors propose to apply Wohlin et al.'s GQM template [26]. The objective of our study can therefore be formally defined as:

- *To analyse* the execution time and the energy consumption of executing the same scenarios with both versions and in each of the three releases.
- for the *purpose* of understanding its behaviour
- *in terms* of software development, preventive maintenance and perfective maintenance, using Spring or not using Spring
- *from the point of view* of software developers, software acquirers, and software service consumers
- *in the context* of software service development, acquisition and consumption.

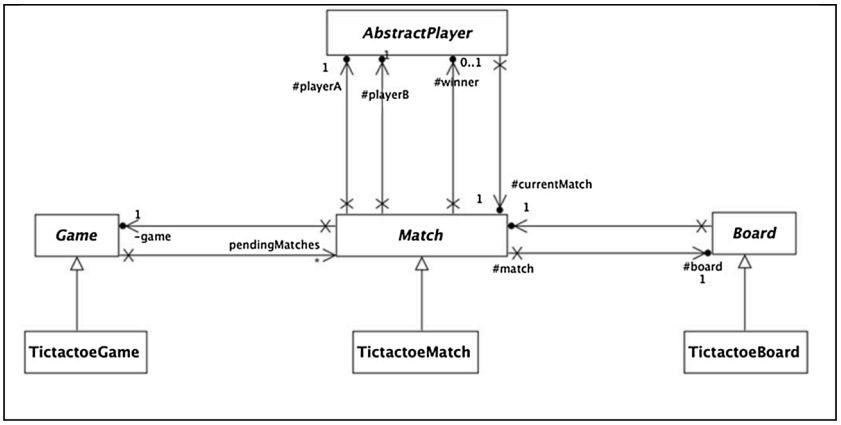


Fig. 12. Specializations for *Tic-tac-toe*.

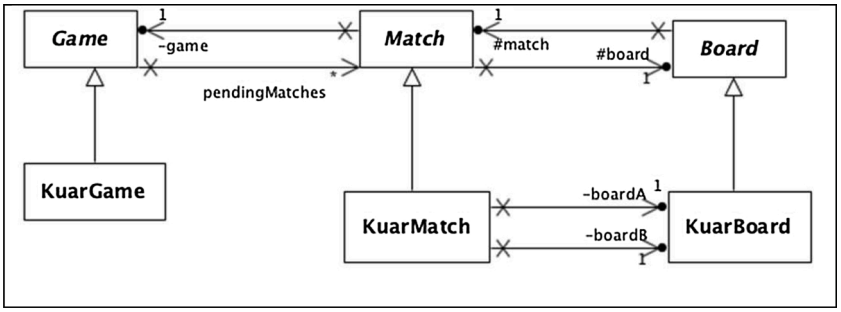


Fig. 13. Specialisations for *Kuar*.

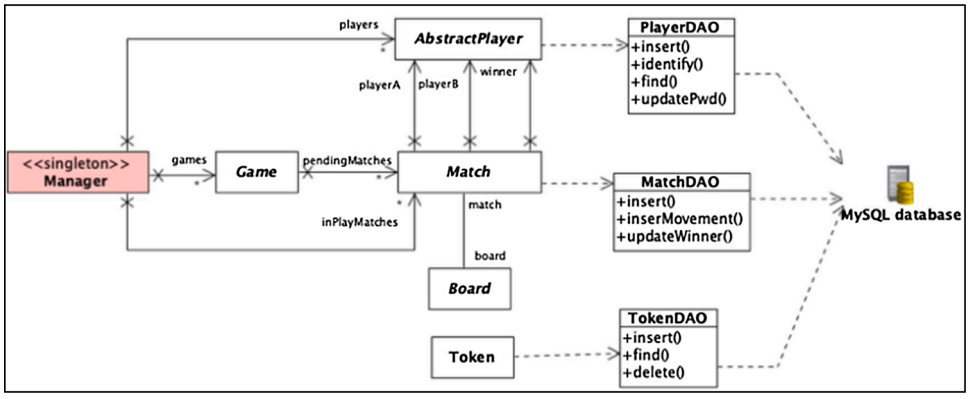


Fig. 14. Structure of the *Non-Spring* version.

It should be noted that we have focussed special attention on all the stakeholders involved, from those in software development to those in software consumption, since we believe it is important to start inculcating a social conscience about the impact that software has on global warming.

The ***Entity Classes under study*** are both versions (Spring and Non-Spring) of the same application, and their three releases together make up the ***Software Entities***.

The fourth and last step is the definition of the **test cases** that will be executed to measure energy consumption and execution time.

In this experiment, test cases are implemented as JUnit test cases that exercise exactly the same functionality in both versions. Even the code of test cases is exactly the same in both versions, with the only exceptions being: (1) the auxiliary methods that access the database to perform some assertion (number of moves inserted, for example) and (2) the method used to delete all the records before executing the tests, thus leaving the database empty, in an initial state that allows the reproducibility of test cases.

Fig. 20 shows the *setUp* method of both versions: several statements in the *Non-Spring* version, and just a single call to the *deleteAll* method of a repository in the Spring version. Since all the foreign key relationships in the database are *on delete cascade*, deleting all the players also deletes all the records in all the tables (as per the relational schema in Fig. 9).

The test cases exercise all the functionalities shown in the use case diagram of Fig. 8. For **Releases 1 and 2**, they test:

1) User register, with both valid and invalid credentials.
2) User login, with both valid and invalid credentials.
3) Password recovery, both in the normal scenario (successful recovery) and in an invalid one (invalid token).

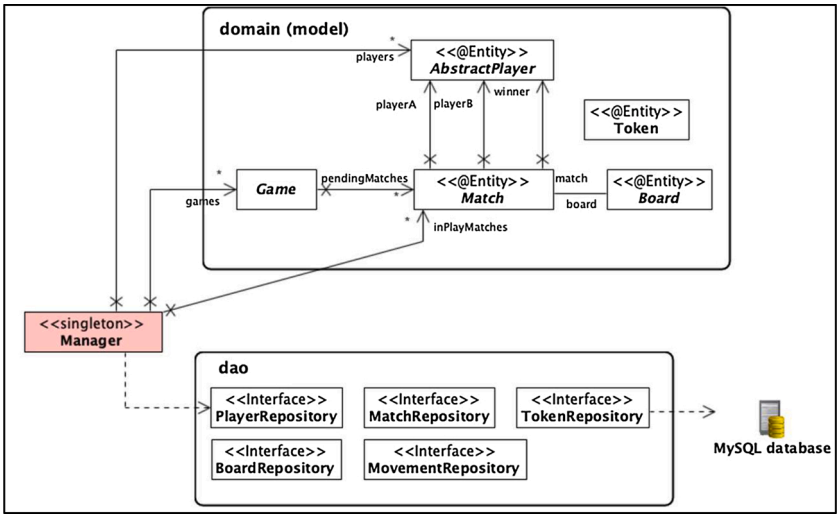


Fig. 15. Structure of the *Spring* version.

```
public static void insertMovement(Match match,
 AbstractPlayer player, Integer[] coordinates ) throws Exception {
      String sCoordinates ="";
      for (int i=0; i<coordinates .length-1; i++)
            sCoordinates +=coordinates [i]+",";
      sCoordinates +=coordinates [coordinates .length-1];
      GinsengConnection bd=null;
      try {
            bd=Broker.get().getBd();
            String sql="insert into movement (id_match, player, coordinates) values (?, ?,
?)" ;
            PreparedStatement ps=bd.prepareStatement( sql);
            ps.setString(1, match.getIdMatch());
            ps.setString(2, player.getUserName());
            ps.setString(3, sCoordinates );
            ps.executeUpdate();
      }
      catch (Exception e) {
            throw e;
      }
      finally {
            if (bd!=null)
               bd.close();
      }
}
```

Fig. 16. Insertion of a movement in the Non-Spring version.

```
@ Entity(name= "movement")
public class Movement {
      @ Id @ GeneratedValue(strategy= GenerationType.AUTO,
                                                generator= "native")
      @ Column(name= "id")
      private Long id;
      @ ManyToOne @ JoinColumn(name= "id_match")
      private Match match;
      @ ManyToOne @ JoinColumn(name= "player")
      private AbstractPlayer player;
      private String coordinates ;

      ...
}
```

Fig. 17. *Movement* is an *@Entity* in the Spring version.

4) Two players play a single match of Tic-tac-toe.
5) Two players play a single match of Kuar.
6) Parallel execution of 100 user registers, 100 logins and 50 simultaneous matches of Tic-tac-toe and Kuar with the 100 logged-in users.

For **Release 3**, which includes the Ladder game after the perfective maintenance intervention, we extended the 6th test to include 50 simultaneous matches of Ladder.

As may be seen, the 6th test simulates an important load of players connected simultaneously to the system and executing operations concurrently, all of them requiring access to the database.

It is important to highlight the coverage reached by the tests: it must be high enough to ensure that all the scenarios are being executed in both system versions.

Table 2 shows the statement coverage of the tests, measured by the Eclemma[2] code coverage plugin for Eclipse. The shaded-in cells correspond to non-present classes in some of the products. In the Spring version, the repositories are not included, since they are mere interfaces with non-implemented methods.

There are some noteworthy differences in the results for some classes, although for our context they are not meaningful. For example, the coverage on *Board* reaches 49 % in the Non-Spring system and 28 % in the Spring system. Table 3 shows the coverage details of *Board* in both versions:

- In the Non-Spring system, and setting aside the non-visited (and non-used) methods, the constructor *Board()* and the methods *set_id*, *setBestTime* and *setTimesPlayed* are explicitly called in the *KuarBoardDAO* class when a board is loaded from the database.
- In the Spring system, and since *Board* is an *@Entity* class, the self-Spring **framework** reflectively calls the constructor and its setter methods. However, Eclemma is not able to detect these calls.

Another apparently significant difference appears in the coverage of *KuarGame*, with 26 % in the Non-Spring system and 100 % in the Spring one. This difference proceeds from a method called *getRandomBoard (boolean testingMode)*, which is thought to return a random board from the database when the *testingMode* parameter is *false*, and always the

Table 1
Sonar summary of the first release of both versions.

	Non-Spring	Spring
LOC	1,456	1,039
Bugs	11	1
Vulnerabilities	18	1
Code smells	119	82
Security hotspots	4	2
Duplications	1.2 %	0.0 %
Quality gate	Passed	Passed
Technical debt	3 days	2 days

[2] https://www.eclemma.org/.

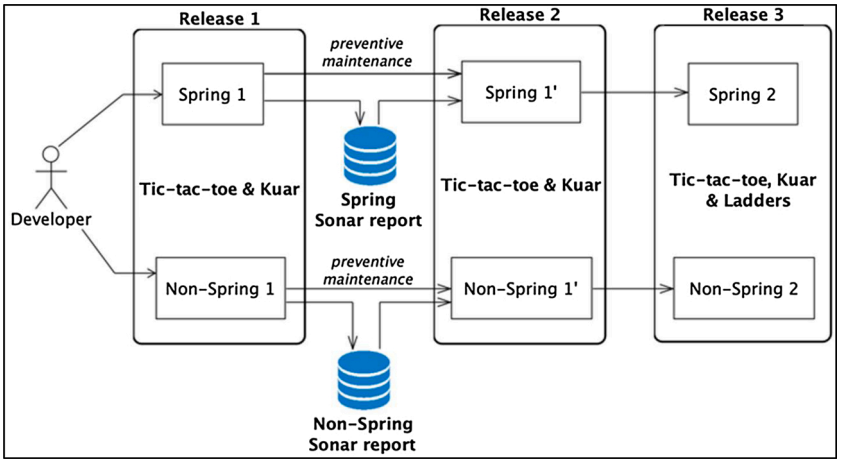


Fig. 18. Summary of the releases.

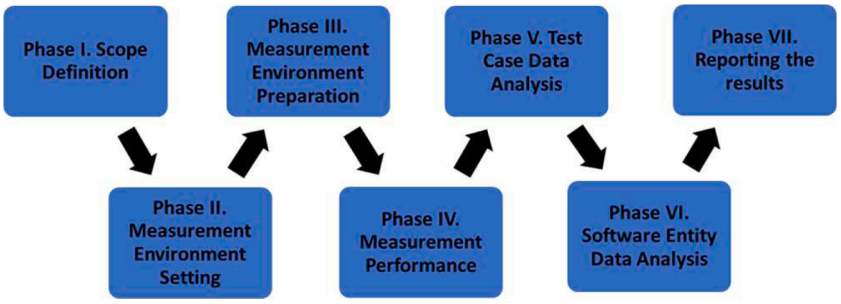


Fig. 19. Process for evaluating the energy efficiency of the software [27].

Non-Spring	public void setUp() {
	try {
	DAOConfiguration. *pool*=Pool. *POOL*;
	GinsengConnection bd =
	Broker. *get*().getBd();
	String sql="delete from abstractplayer" ;
	PreparedStatement ps =
	bd.prepareStatement(sql);
	ps.executeUpdate();
	bd.close();
	} catch (SQLException e) {
	fail();
	}
	}

Spring	public void setUp() {
	playersRepo .deleteAll();
	}

Fig. 20. An example of the differences in the auxiliary methods.

same board otherwise: this method fulfils this requirement in the Non-Spring version (i.e. it actually returns a random board), but has a "fake" implementation in the Spring version (i.e. it does not check the value of the parameter and always returns the same board). The coverage reached in the *KuarGame* class of both products is therefore, for our goal, exactly equivalent.

It seems clear that the test suite designed covers the functionalities of the products widely enough that we can proceed with the analysis of the measurements obtained.

5.2. Phase 2 (Measurement Environment Setting)

This phase is intended to satisfy the objective defined in the first phase. It consists of five activities:

Table 2
Statement coverage reached by the test suite.

		Statement coverage (%)	
Package	Class	Non-Spring	Spring
dao	Broker	84	
dao	Broker.BrokerHolder	63	
dao	GinsengConnection	100	
dao	GinsengPooledConnection	100	
dao	KuarBoardDAO	81	
dao	Pool	58	
dao.noSpring.noPool	MatchDAO	82	
dao.noSpring.noPool	PlayerDAO	53	
dao.noSpring.noPool	SimplePlayerDAO	73	
dao.noSpring.noPool	TokenDAO	62	
domain	AbstractPlayer	100	80
domain	Board	49	28
domain	Game	82	86
domain	Manager	95	97
domain	Manager.ManagerHolder	67	63
domain	Match	90	87
domain	Movement		100
domain	Player	92	82
domain	SimplePlayer	77	100
domain	Token	92	100
domain.kuar	KuarBoard	94	90
domain.kuar	KuarGame	26	100
domain.kuar	KuarMatch	100	100
domain.tictactoe	TictactoeBoard	67	67
domain.tictactoe	TictactoeGame	100	100
domain.tictactoe	TictactoeMatch	100	100
	Mean	**79**	**86**

Table 3
Differences in the coverage of *domain.Board*.

Method	Non-Spring	Spring
Board()	100 %	0%
Board(Match)	100 %	100 %
get_id()	100 %	100 %
getBestMovements()	0 %	0 %
getBestTime()	0 %	0 %
increaseTimesPlayed()	0%	0 %
load(String)	0 %	0 %
set_id(String)	100 %	0 %
setBestMovements(int)	0 %	0 %
setBestTime(int)	100 %	0 %
setMatch(Match)	100 %	100 %
setTimesPlayed(int)	100 %	0 %
toJSON()	0 %	0 %

1) Selection of the measuring instrument used to perform the time and power consumption measurements of the software analysed.
   The measurement environment to be used to evaluate the energy efficiency of the software is FEETINGS (Framework for Energy Efficiency Testing to Improve eNviromental Goals of the Software [28], a framework for measuring and analysing the energy consumption of a software application (see Fig. 21). FEETINGS consists of two main elements: (i) an EET (Efficient Energy Tester), which is a device that measures the energy consumption of a set of hardware components when the *Software Entity* is executed in the DUT (Device Under Test); and (ii) an ELLIOT, which is the software application that processes and analyses the data collected by the EET.
2) Specifications of the Device Under Test (DUT). The DUT is a desktop computer that allows the execution of the test cases in order to carry out the time and energy consumption measurements. In our experiment the DUT had the following specifications:
   1. Asus M2N-SLI Deluxe motherboard.
   2. AMD Athlon tm 64 × 2 Dual Core 5600 + 2,81 GHz processor.
   3. 4 modules of 1GB DDR2 MHz RAM memory.
   4. Seagate Barracuda 7200 500Gb hard disk drive.
   5. Nvidia XfX 8600 GTS graphics card.
   6. Power supply 350 W AopenZ350-08Fc.
   7. Windows as operating system.
3) Selection of the measures to be used for the analysis, which will be applied to the three releases of both versions. In this study, the measures are:
   - Energy consumption of the processor, the HDD and the DUT (the overall energy consumption). Since we focus on the long-term impact of offering software services in a system running 24 × 7 with no GUI, we will not collect the energy consumed either by the graphics card or the monitor.
   - Execution time obtained from the computer clock.
   - As we have explained, some other measures were taken with SonarCloud to improve the internal quality between the first and the second release of both versions. The measurements considered were LOC, Bugs, Vulnerabilities, Code Smells, Security Hotspots, Duplications, Quality Gate and Technical Debt.
4) Checking that no other software is running in the background and interrupting all services and processes that may affect the baseline measurement of consumption. This will be executed when the experiment is run.
5) Finally, the fifth activity is to obtain a baseline of the energy consumption. In our case, as we are performing the measurements on the same environment for all the releases and versions, the possible extent of any effect on the baseline energy consumption is the same for all of them, and so does not add noise to the data for comparison purposes. It is worth emphasising that our objective is to find out if there is any significative difference in execution time and energy consumption between using Spring and not using Spring; the goal is not to discover what this difference is exactly, because this will always depend on the environment in which the applications are run.

Fig. 21. The EET device measures the energy consumed by the DUT.

5.3. Phase 3 (Measurement Environment Preparation)

This phase is composed of three activities: checking that no other software is running in the background; determining the number of times each measurement should be repeated; and configuring the testbed, by installing the Software Entity and the services required in the DUT. In this last activity, the chosen Software Entity must also be prepared in such a way as to enable the execution of the defined test cases.

To enable the EET to take the measurements, the DUT (a desktop computer) is not directly connected to the power supply, but rather to the EET; it records the power consumed, stores these data on an SSD memory card, and supplies them to the DUT. Once these physical connections have been made, the protocols for executing the experiment can be defined. Following these protocols will assure that the three steps of this phase are accomplished.

The protocol for the measurement of the Non-Spring version of each release will thus be:

1) The Device Under Test is cleaned.
2) The Non-Spring version is stored on the DUT.
3) The test suite is launched against the version.
4) The execution time and energy consumption measures, which have been collected by the EET, are saved onto a file.

We then performed the same above steps with the Spring version.

In order to avoid undesirable effects, and to guarantee the reliability of the analysis and statistical results obtained, steps 3 and 4 of each protocol were recorded 101 times, and each set of test cases was launched 101 times against each version. By repeating the measurement 101 times it is possible to detect if there is any other software running in the background, as the results would then present differences between them.

5.4. Phase 4 (Measurement Performance)

This phase consists of the execution of the experiment using the defined tests on the Software Entities described, according to the protocol defined, on the DUT specified, and using the EET to take the measurements (Fig. 21).

During the 101 instances of each execution of each release of both product versions, the EET saves a huge amount of data on the memory card; this must later be processed in order to obtain the final results. In Fig. 22, the data collected by the EET are shown.

5.5. Phase 5 (Test Case Data Analysis)

The main goal of the fifth phase is the processing and analysis of the energy consumption data of each of the test cases defined in the first

```
Sample Nr., time, HDD1, HDD2,  Graph1, Graph2, Proc1, Proc2, tempEET, tempPROC
1 ,38 ,1.23710942 ,14.33710956 ,1.17187500 ,0.04882812 ,1.85546875 ,1.80664062
2 ,47 ,1.33476567 ,14.48359394 ,1.22070312 ,0.09765625 ,1.85546875 ,1.85546875
3 ,56 ,1.13945317 ,14.63007831 ,1.12304687 ,0.09765625 ,1.90429687 ,1.85546875
4 ,66 ,1.28593754 ,14.23945331 ,1.12304687 ,0.04882812 ,1.85546875 ,1.90429687
5 ,76 ,0.94414062 ,14.48359394 ,1.22070312 ,0.04882812 ,1.80664062 ,1.90429687
6 ,89 ,1.23710942 ,14.23945331 ,1.22070312 ,0.14648437 ,1.90429687 ,1.85546875
7 ,99 ,1.28593754 ,14.23945331 ,1.17187500 ,0.04882812 ,1.95312500 ,1.85546875
8 ,108 ,1.38359379 ,14.19062519 ,1.12304687 ,0.04882812 ,1.80664062 ,1.95312500
9 ,118 ,0.84648437 ,14.53242206 ,1.22070312 ,0.14648437 ,1.70898437 ,1.85546875
10 ,128 ,1.04179692 ,14.23945331 ,1.17187500 ,0.09765625 ,3.41796875 ,3.07617187
```

Fig. 22. An excerpt of the raw data saved by the EET.

phase. This phase is composed of two different activities: preparation of the raw data, and the statistical analysis of the values obtained from the measurements of the defined test cases. Both activities are carried out by means of ELLIOT (part of FEETINGS). For the first activity, the average values of each of the measurements are calculated. This necessary step enables us to work with a single value that derives from a large number of values captured by the EET. During this activity we have to check for possible outliers, removing them from the set of data. As we mentioned in the previous section, we repeated the measurement 101 times. Consequently, during this activity we have to clean the data and remove invalid executions. An execution can be considered invalid for one of two different reasons: a wrong execution, or an outlier. In the former, all of the values obtained are not consistent with the rest of the results for the other executions. In such case, it is possible that another program may have been running in the background. In the latter, the execution is eliminated because it shows an outlier when one or more values are either too high or too low with respect to the rest of the results for the other executions.

In our case, we have removed the following executions (see Table 4):

Once the data have been processed and prepared, the descriptive statistics of the values obtained are calculated (See Table 5- Release1, Table 6– Release 2 and Table 7 – Release 3).

Boxplots enable us to study the distributional characteristics of a group of scores, as well as the level of the scores. The scores are sorted, and four equal-sized groups are made from the ordered scores. The lines dividing the groups are called quartiles, and the groups are referred to as quartile groups. The boxplots of the consumption data for our study are shown in Figs. 23–25.

If the four sections of a boxplot are uneven in size this means that, while many measurements have similar values in certain parts of the scale, in other parts the measurements are more variable. In our case, all the boxplots, except the ones for Release 2 with Spring and for the Total consumption (see Fig. 24), are even in size.

The median is represented by the line in the box. The interquartile range box represents the middle 50 % of the data (values from the 2nd to the 3rd quartile).

The whiskers extend from either side of the box. The whiskers represent the ranges for the bottom 25 % and the top 25 % of the data values, excluding outliers. As we can see in Figs. 23–25, there are no outliers because no data point is located outside the whiskers of the boxplot.

The plots which show a similar distribution of the data points for the first and second quartiles to the data of the third and fourth quartiles, represent a normal distribution. The boxplots are skewed left for Release 1 HDD with non-Spring, Release 2 HDD, and Release 3 HDD and Processor with Non-Spring, whereas they are skewed right for Release 1 HDD with Spring, and Release 2 Total with Non-Spring.

If we compare the boxplot between releases and the same component, that is the boxplot of HDD for Release 1 with Spring and with Non-Spring, or the boxplot of Processor for Release 1 with Spring and with Non-Spring, we can observe that the median line lies outside the box of the comparison boxplot; thus, there is a difference between the two groups. This means that there are differences between the consumption of the releases, depending on the use or not of Spring.

Finally, we can analyse the interquartile range to examine how the data is dispersed. As is shown in Figs. 23–25, in general the box length is small, therefore the data are not greatly dispersed.

Table 4
Cleaning of executions.

	Release 1		Release 2		Release 3	
	Spring	Non-Spring	Spring	Non-Spring	Spring	Non-Spring
Wrong execution		2		3		
Outlier		1	2	6	2	9
Total number of valid executions	101	98	99	92	99	92

5.6. Phase 6 (Software Entity Data Analysis)

In Phase 6, the results of the execution time and the energy consumption for both versions of each release can be compared, in an effort to arrive at a conclusion related to the RQ posed at the beginning of this study.

In the following tables we will show the mean values obtained for the time required for running the test suite (Time column), and the mean figures for the energy consumed by the hard disk (HDD column), the processor (Processor column) and the DUT (Total column).

Table 8 shows the mean results for Release 1 (Tic-tac-toe and Kuar with no refactoring).

As can be observed, in Release 1, the Non-Spring version requires almost 16 % more time to perform the same operations. In relation to the energy required, the Spring version requires more for the HDD, the processor and, in general, for the DUT.

The hard disk demands more energy than the processor, although the difference in the amount of energy consumed by the processor is especially significant; this is likely to be due to the extra effort required by the reflective engine that processes Spring annotations.

Our initial conclusion regarding this first release is that using Spring is not positive, either from the point of view of energy consumed or the execution time.

The **second release** offers the same functionalities as the previous one, but it includes some changes in the code, in line with the Sonar Cloud report (see Table 1).

The summary of the results for Release 2 appears in Table 9. Execution time is again greater in the Spring version, taking some 13 % more time than is required by the Non-Spring version.

It is worth noting that the corrections introduced in the code reduce the differences between the Non-Spring and the Spring versions. It should be highlighted that once more, in this latter version, the hard disk energy consumption is the highest for the components, but the biggest difference in consumption between using or not using Spring is in the processor.

As a general conclusion, after the preventive maintenance it seems that although the differences between Spring and Non-Spring versions are minor, the consumption pattern remains the same as in Release 1; for instance, the use of Spring gives a worse result from the point of view of energy consumption than it does when not using it.

However, looking at the figures obtained, the values for Release 2 are, in general, worse than those for Release 1, which in theory should not make sense, bearing in mind that in Release 2 we are supposedly improving upon Release 1.

If we look at the changes made, we observe that before doing the changes derived from the analysis with SonarCloud, when a search is

Table 5
Values for Release 1.

	Execution Time (s)		Energy Consumption (in watts*second)					
			HDD		Processor		Total	
	Spring	Non-Spring	Spring	Non-Spring	Spring	Non-Spring	Spring	Non-Spring
Mean	4,32	3,73	67,61	58,25	32,94	18,90	801,86	529,28
Standard Deviation	0,49	0,53	0,14	0,18	0,26	0,33	6,03	7,12
Median			67,97	58,43	34,10	14,38	803,77	518,77
Max	5,09	4,58	79,46	71,94	20,55	24,25	967,69	673,20
Min	3,02	2,51	46,11	38,42	39,16	10,87	521,81	340,44

Table 6
Values for Release 2.

	Execution Time (s)		Energy Consumption (in watts*second)					
			HDD		Processor		Total	
	Spring	Non-Spring	Spring	Non-Spring	Spring	Non-Spring	Spring	Non-Spring
Mean	4,53	4,02	71,16	61,78	34,66	19,98	825,59	590,64
Standard Deviation	0,47	0,43	0,26	0,59	0,39	0,76	6,55	6,28
Median			71,51	62,92	36,53	15,56	828,22	573,69
Max	5,53	6,03	87,86	92,10	42,17	27,23	1014,65	803,81
Min	3,02	3,02	47,79	44,72	24,11	7,12	566,52	462,19

Table 7
Values for Release 3.

	Execution Time (s)		Energy Consumption (in watts*second)					
			HDD		Processor		Total	
	Spring	Non-Spring	Spring	Non-Spring	Spring	Non-Spring	Spring	Non-Spring
Mean	34,65	24,07	532,82	355,59	151,13	87,05	5149,62	3190,41
Standard Deviation	2,35	0,71	0,17	0,84	0,24	0,53	2,65	1,00
Median			532,95	369,01	115,06	84,62	4767,03	3097,31
Max	39,19	25,63	599,92	394,37	178,34	107,57	5798,14	3424,26
Min	30,64	22,60	467,25	310,91	126,15	42,32	4607,24	2929,44

carried out on the database (for example, when a user logs in), the search goes to the repository and returns the object that is being searched for (Fig. 26, top row). However, Sonar recommends to check whether the object exists before it is recovered (Fig. 26, bottom row). Some times (when it is certain that the instance exists in the databse), the implementation of this recommended improvement may suppose a slight "deoptimisation", which has a certain negative impact on the time, since this implementation is executed many times; hence the results obtained for Release 2 are higher. Nevertheless, we consider that this increase is not very significant, and that the pattern of comparison between Spring and Non-Spring in this second release subsequent to the preventive maintenance is maintained.

The **third and last release** consists of the addition of a new game to the system and in measuring the execution time and energy consumption of the tests.

As may be seen (Table 10), in this case also the differences between both versions are very significant: executing the same scenarios requires almost 44 % more time with Spring, almost 50 % more energy to read from and write on the hard disk, in excess of 70 % more energy is required by the processor, and there is an increase of around 60 %-plus in the energy required by the computer. Again, the biggest difference in the energy required between the Spring and the non-Spring versions is in the processor.

The perfective maintenance carried out therefore can be seen to have a negative impact on the execution time and energy consumption required by the Spring version. In fact, it is the worst scenario of the three studied, as the percentage of the differences for all the measures (with the exception of the processor percentage of Release 1), is much higher than for the two previous releases.

Finally, considering the three releases together, we can conclude that, quantitatively speaking, the greatest amount of energy is used by the hard disk, because it requires the continuous spin and the physical shift of the R/W head. Percentage-wise, the biggest difference is in the consumption of the processor.

If we look at the progression of the different measures among releases, we obtain the following results (Figs. 27–30).

The preventive maintenance, being just the correction of bugs, presents only a small increase in the execution time and in the energy required by the hardware components and the whole computer that is under analysis, mainly due to the actions performed in compliance with the Sonar indications. However, the consumption pattern between Spring and Non-Spring remains the same as in the other releases. On the other hand, the perfective maintenance represents an extension of the system and as such there is a big difference among its values and those of the previous releases.

In any case, the values required by the Spring version in all the releases are greater than those required by the Non-Spring version.

As a general conclusion, it seems that products developed without using Spring are better in all the conditions and for all the measures. This could indicate that, although Spring has some advantages for programmers, once the product starts to run this advantage disappears in benefit of the Non-Spring development.

These conclusions allow us to answer our research questions accordingly:

RQ1. Is there a relationship between the execution time required by an application when it is run and the use of Spring during its development?

It seems that, from the study undertaken and the data acquired, there

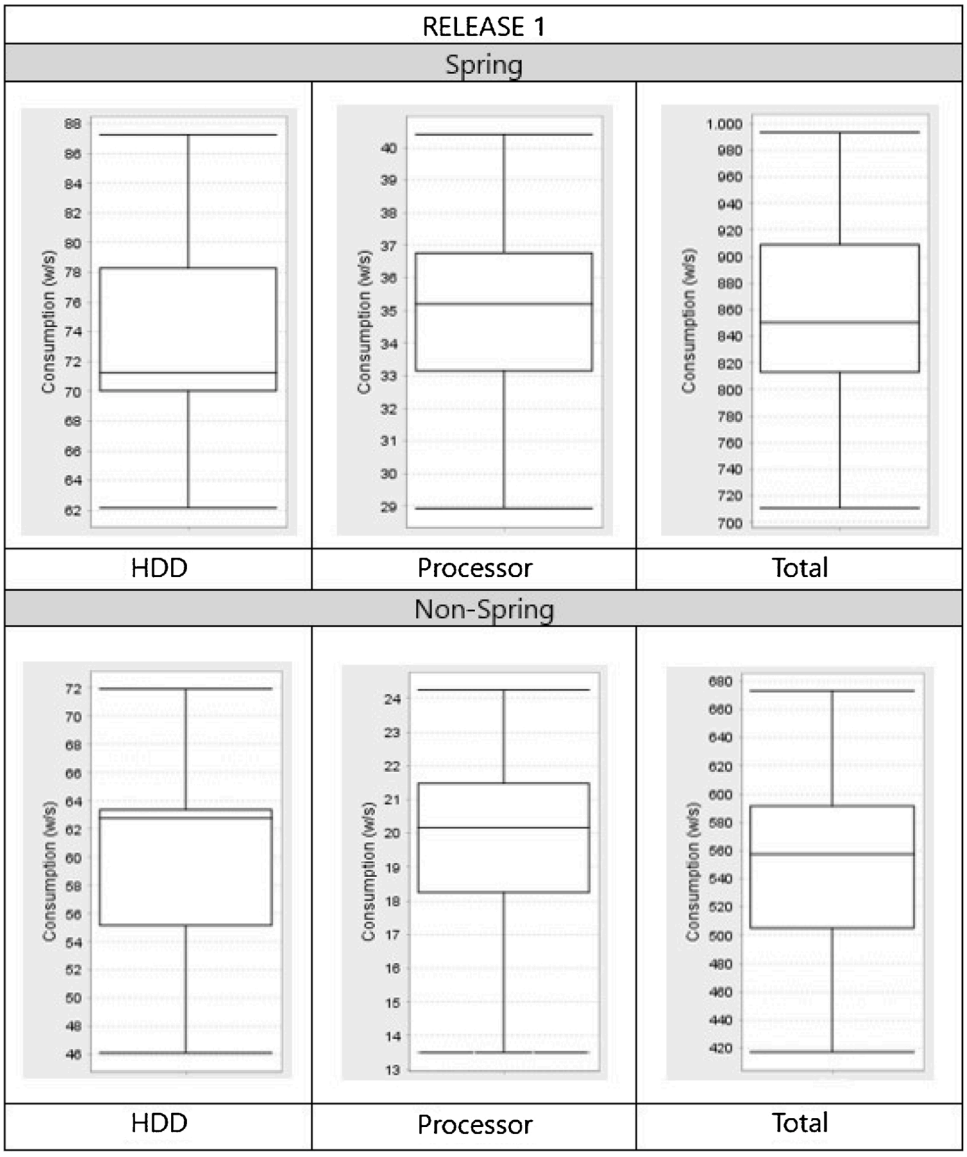


Fig. 23. Boxplots for Release 1.

is a negative relationship between the use of Spring and the time needed by the application for it to run. This implies that from the point of view of execution time, it seem to be better to avoid the use of Spring.

RQ2. Is there a relationship between the energy needed to run an application and the use of Spring during its development?

Based on the data from our study, the use of Spring is not positive - neither for the consumption of the hard disk, nor the computer; it is especially not positive from the point of view of the processor, where the consumption of the Spring version is greater than that of the Non-Spring version in all three releases.

5.6.1. Contextualizing the results

In spite of our previous comment that we wish to compare the results without paying precise attention to the consumption data of the different scenarios (this is why we have not calculated and eliminated the baseline), in an endeavour to give clarity as to what the differences obtained imply, we nevertheless present an estimation of the difference in the CO2 footprint in the long-term, using the consumption information obtained for the two versions of Release 3. The calculus will be based on the continuous execution of the system for six months (15,552000 s). As every country and territory produces electricity employing different technologies, we will use the CO2 footprint calculators of Rensmart.com[3] (which has data from the European Union), along with data of other important energy consumer countries:

- In the EU, Sweden is the country that requires the least emissions to produce electricity (0.013 Kg of CO_2 per Kw h, while Estonia needs 0.819 Kg/Kw h The mean of the European Union is 0.30 Kg per Kw h
- In United States[4], the mean is 0.45 Kg/Kw h
- China[5] requires 0.56 Kg/Kw h
- India[6] produces 0.60 Kg per Kw h

Table 11 summarises the data for the EU and the USA, showing the difference in CO_2 footprint between six months' usage of Spring and Non-Spring, respectively. The difference is significant, and it relates only

[3] https://www.rensmart.com/Calculators/KWH-to-CO2.
[4] U.S. Energy Information Administration: https://www.eia.gov/.
[5] https://www.climate-transparency.org/wp-content/uploads/2019/11/B2G_2019_China.pdf.
[6] http://erpc.gov.in/wp-content/uploads/2018/06/carbon-emissions-from-power-sector-7062018.pdf.

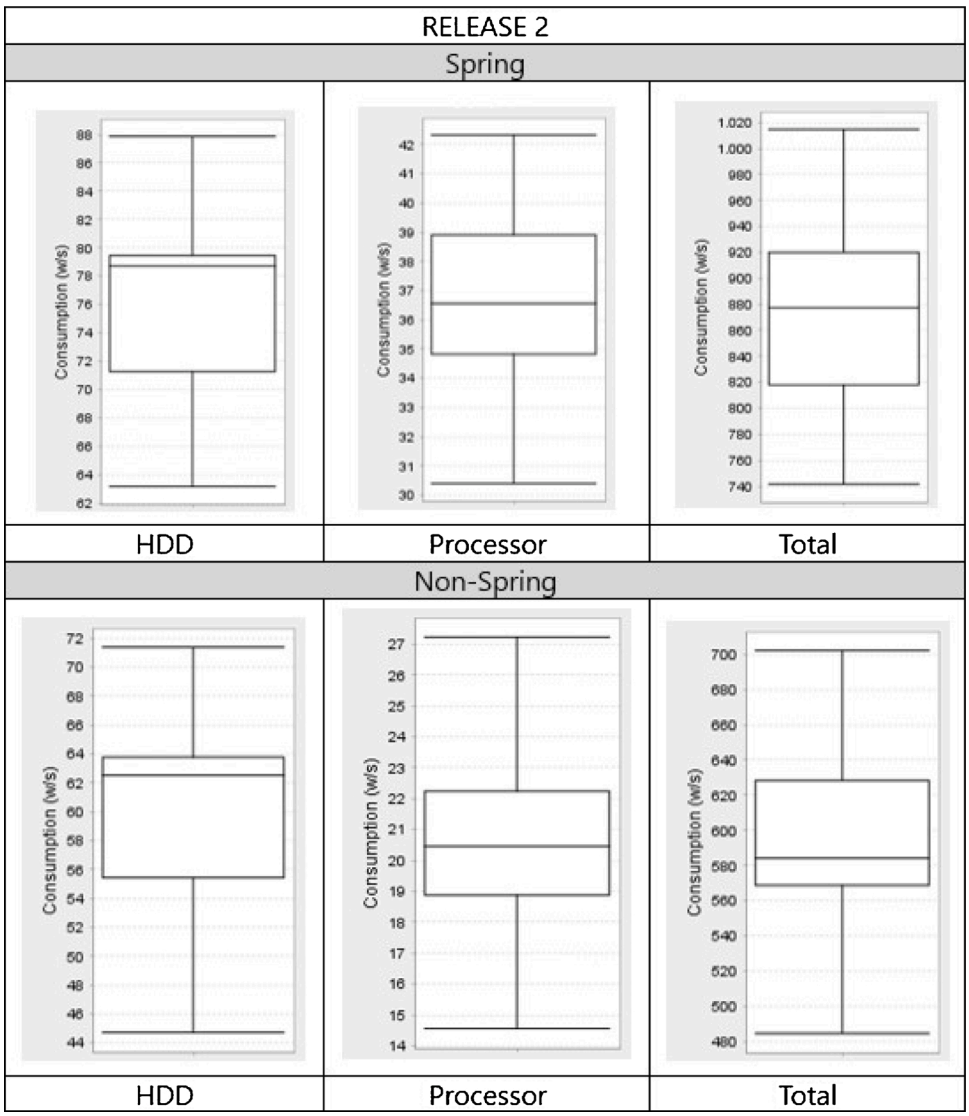


Fig. 24. Boxplots for Release 2.

to Release 3 of our example. This comparison gives us an idea of the great impact that not using Spring might have on the environment.

5.7. Phase 7 (Reporting the results)

The last phase (Phase 7) is about documenting the study performed, describing the entire process followed, and presenting the results regarding the energy consumption of the software.

This activity includes the writing-up of detailed documentation to explain the whole process and the results obtained in this paper.

6. Threats to validity

Despite being very methodical when preparing and performing our experiment, we are nonetheless aware of some possible threats to its validity. In the paragraphs below we analyse the construct, internal and external validity in accordance with the definitions of Wohlin et al. [29].

Construct validity is the "degree to which the independent and the dependent variables are accurately measured by the measurement instruments used in the experiment".

In this experiment, the **dependent variables** are the execution time and the consumption of energy required for executing the test scenarios; these have been objectively measured by the Efficient Energy Tester developed in our laboratory, which has been validated as a reliable device for measuring the energy efficiency of running software. Moreover, all the executions have been repeated 101 times, and both products have run on the same physical machine, operating system and configuration. So as to avoid any possible bias from the execution of the database server, this was placed on a different physical machine.

The **independent variable** is the use or not of Spring in the system version.

With regard to the **test cases** used, we have followed the recommendations given in [27]: the test cases execute the necessary functionality of the software product that is to be measured (they reach high coverage in both versions); they are independent and do not affect the subsequent test case (the database is emptied after each test execution); they simulate user inputs and focus on specific software tasks; they can be executed against both system versions (with the minor exception of the small amount of details explained in Section 5.1 above).

Internal validity is the degree of confidence in a cause-effect relationship between factors of interest and the results observed.

Due to the nature of both experiments, all variables have been controlled, so threats to internal validity are minimised. In particular, the test cases are the same, and run the same scenarios and situations in both products (developed with and without Spring). It is worth noting again that the repetition of the experiments 101 times also helps to

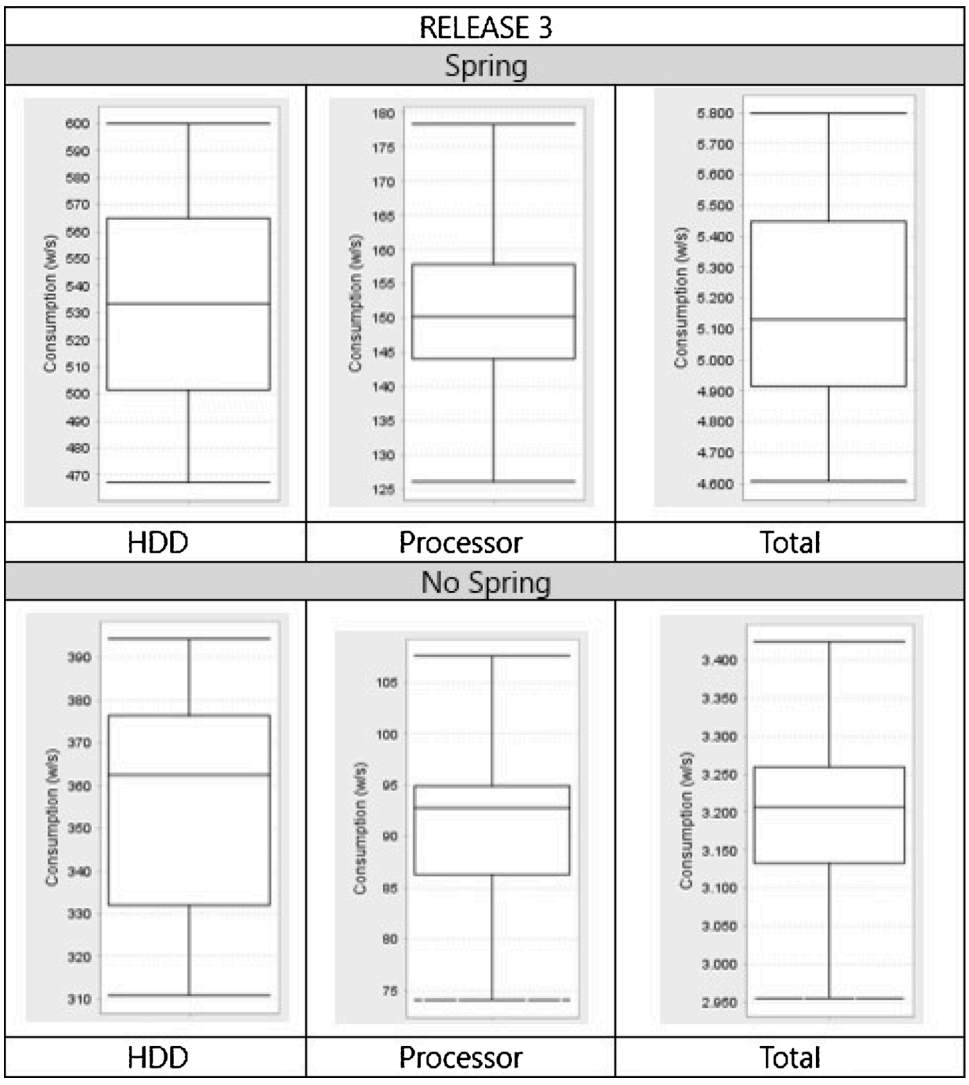


Fig. 25. Boxplots for Release 3.

Table 8
Mean results for Release 1.

		Average energy consumption (in watts*second)		
	Execution time (seconds)	HDD	Processor	Total
Non-Spring	3,73	58,25	18,9	529,28
Spring	4,32	67,61	32,94	801,86
Difference	15,82 %	16,07 %	74,29 %	51,50 %

```
User user = userDAO.findById(userName).get();
```

```
Optional<User> optUser = userDAO.findById(userName);
if (optUser.isPresent()){
 User user = optUser.get();
      …
}
```

Fig. 26. Changes when searching in the database.

Table 9
Mean results for Release 2.

		Average energy consumption (in watts*second)		
	Execution time (seconds)	HDD	Processor	Total
Non-Spring	4,02	61,78	19,98	590,64
Spring	4,53	71,16	34,66	825,59
Difference	12,69 %	15,18 %	73,47 %	39,78 %

Table 10
Mean results for the third release.

		Average energy consumption (in watts*second)		
	Execution time (seconds)	HDD	Processor	Total
Non-Spring	24,07	355,59	87,05	3190,41
Spring	34,65	532,82	151,13	5149,62
Difference	43,96 %	49,84 %	73,61 %	61,41 %

minimise any possible bias.

External validity is the "degree to which the research results can be generalized to the population under study and other research settings". The greater the external validity, the more the results of an empirical study can be generalised to actual software engineering practice.

As in most software engineering experiments, the extenal validity is

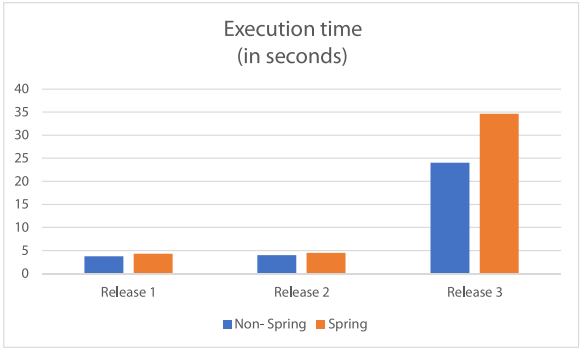


Fig. 27. Comparing execution time among the three releases.

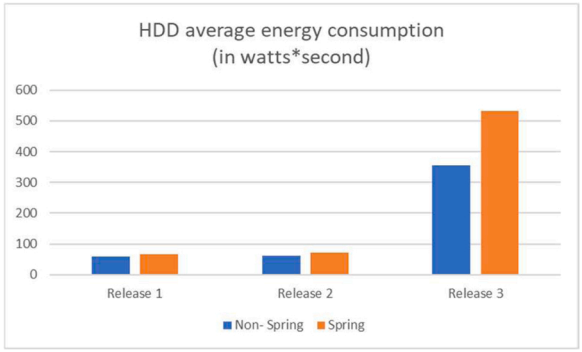


Fig. 28. Comparing HDD average energy consumption among the three releases.

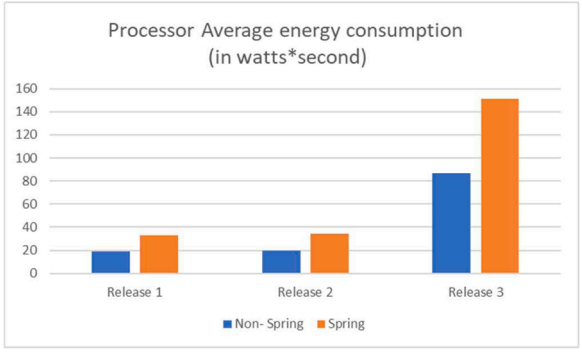


Fig. 29. Comparing average processor energy consumption among the three releases.

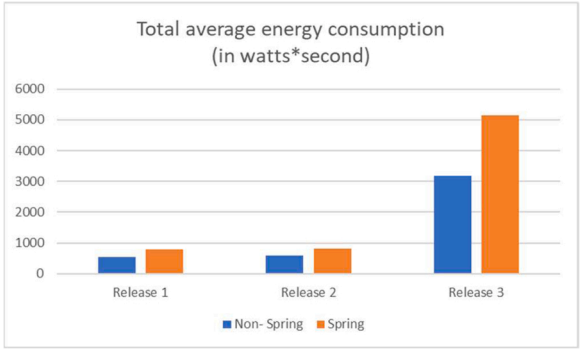


Fig. 30. Comparing Total average energy consumption among the three releases.

Table 11
Total CO_2 footprint (in tons of CO_2) of Release 3 during six months.

	Country footprint (Kg/ Kw h	Non-Spring (13,910.8 Kw h	Spring (22,302.4 Kw h
EU	0.30	4.17	6.69
USA	0.45	6.26	10.04
China	0.56	7.79	12.49
India	0.60	8.35	13.38

the weakest aspect of our experiment. The system is in fact small, and cannot be absolutely generalised to any other system.

However, it should be noted that:

- Both versions have been developed by a senior programmer with many years of experience in Java development.
- Both versions follow classical architectural patterns, which have been widely tested and applied in an infinite number of projects.
- The second release of both versions eliminated the possible bad smells that the programmer had committed in the first one.
- It is also clear that the reflective processing of annotations requires greater computational effort. This experiment has not done anything other than measure its possible impact on a sample system.

All of these facts lead us to affirm that our results are generalisable to any given Spring application.

Furthermore, in order to allow the scientific community to replicate our experiments, the source code of the three versions of both systems is available for their use in the laboratory package generated for the experiment, as follows:

- The Non-Spring system is available for download at: https://bitbucket.org/macariopolo/ginsengnoorm
- The Spring system can be found at: https://bitbucket.org/macariopolo/ginsengconjpa

7. Conclusions and future work

Respect for the environment is something that has become a mandatory element of life in present-day society. Many sectors have already incorporated this concept as a key element of their business, treating it as an important business objective.

It is apparent, however, that in the software development sector such sensitivity towards this issue has not yet taken root, and that there are currently very few initiatives to design software in a way that respects the environment. The fact is that, although software is certainly a great facilitator in the world in which we live today, it is also a great energy consumer. Although there exists a multitude of applications designed to help the world and its inhabitants to function better, and millions of individuals use this software on a massive scale, awareness as to the energy that this software demands is still lacking.

Software development companies need to be aware of the impact that their products have on the environment, and begin to take on the task of improving its energy efficiency as an integral part of their business objectives. Unfortunately, as shown in the study carried out by [3] regarding the CSR policies of the top ten software companies, the efforts devoted to improving the environmental impact of software represent a truly minimal percentage of all the activities undertaken by these companies.

As software engineers, we ourselves must contribute to a change of vision in this sector. One good way to set about doing so is to give companies clear guidelines on how to act.

Accordingly, in this paper we set out to discover if the use of Spring in developing software is better, from the point of view of energy consumption, than not using Spring. We chose Spring since it is a widely-used technology for developing Java applications that reduces the

time to market of new applications. This technology can help developers save a lot of development time, by eliminating the need for writing boilerplate code and thereby improving the productivity of those developers. Of course, these advantages may seem very attractive from the business point of view. However, the massive use of reflection implies additional effort for the computer running the system. We therefore decided to study whether this increment in the computer's needs affects its energy consumption in a significant way and compared the energy needed by the same application when developed with and without using Spring.

The paper shows the three different releases developed for the same application, using Spring on the one hand, and not using Spring on the other. Thus, six different applications have been developed. First of all, in order to study the maintainability, a preventive maintenance was carried out to the first release of the application. As a result, a second release was obtained. Next, a perfective maintenance was performed and the outcome of this was a third release. The energy required for each release was measured, recovered and analysed by means of FEETINGS, a framework for measuring and analysing the energy consumption of a software application.

By way of conclusion, we have been able to ascertain that the releases developed using Spring need much more energy than those developed without using it. Although it would be necessary to conduct more testing and use different applications, developed by different programmers, the outcomes shown in this paper can be considered as a first aproximation. Moreover, taking into account the philosophy and the characteristics of Spring we consider that the results should be confirmed.

Based on the results previously explained, our recommendation to the software industry would be take into account not only the benefit of using Spring from the point of view of development time or time to market but also the negative aspect of the increased energy consumption that using Spring entails. In such way, the industry should seek to balance the two perspectives and so try to minimise the use of Spring in developing software wherever possible. Evidently, such change may have an economic impact, because the time to market could be greater and the productivity of workers less. Nevertheless, a positive impact upon indirect earnings related to enhanced brand reputation could compensate for these less desirable effects, and turn out to have an overall positive impact on the economic situation of the industry. In fact, in 2019 the high-risk investment fund TCI Fund Management decided to invest only in companies who have a convincing strategic plan to combat the catastrophic climate situation, a decision which gave them substantial profits - such that they became the most profitable fund in 2019. Also, BlackRock, the world's leading investment fund, has joined the climate crusade, with its CEO having defined climate change as the most single-most decisive and determinant factor in shaping the long-term perspectives on capital in companies around the world [29].

We need software industries that show they are beginning to be aware of the importance of reducing the impact of software on the environment; we need them to step forward and put the environment at the forefront of their business objectives.

As a future work and in order to obtain more robust results we plan to replicate this study in other computers. It is worth to emphasize that in this case it will be neccesary to eliminate the baseline consumption of the computer.

Moreover, we will continue to study different software engineering techniques from the perspective of energy efficiency (for example, and related to the study presented in this paper, we will look at the consumption of the Spring-based web server). As part of this, we are studying the energy consumption of different databases of different complexity, as well as other types of non-gaming applications, to corroborate whether the results are confirmed. We aim to produce a set of good practices to be used in the IT industry and which would be connected to the CSR of software companies. Our final goal is to design a label classification system that enables the classification at different levels of software companies and of the software they develop, in function of their respective practices from an energy efficiency point of view.

Declaration of Competing Interest

The authors do not have any conflict of interest

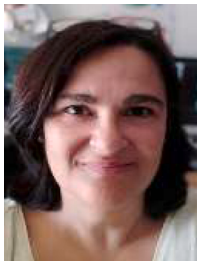
Acknowledgements

This work was part of the BIZDEVOPS-Global (RTI2018-098309-B-C31), supported by the Spanish Ministry of Economy, Industry and Competitiveness and by European FEDER funds, and was also part of the SOS and TESTIMO projects (No. SBPLY/17/180501/000364 and SBPLY/17/180501/000503, respectively), funded by the Department of Education, Culture and Sport of the Directorate General of Universities, Research and Innovation of the JCCM (Regional Government of the Autonomous Region of Castilla-La Mancha).

References

[1] A.S.G. Andrae, T. Edler, On global electricity usage of communication technology: trends to 2030, Challenges 6 (1) (2015) 117–157, https://doi.org/10.3390/challe6010117.
[2] N. Wolfram, P. Lago, F. Osborne, Sustainability in software engineering, Sustainable Internet and ICT for Sustainability (SustainIT) (2017), https://doi.org/10.23919/SustainIT.2017.8379798.
[3] C. Calero, I. García-Rodríguez De Guzmán, M. Moraga, F. García, Is software sustainability considered in the CSR of soft-ware industry? Int. J. Sustain. Dev. World Ecol. 26 (5) (2019) 439–459, https://doi.org/10.1080/13504509.2019.1590746.
[4] I. Manotas, C. Bird, R. Zhang, D. Shepherd, C. Jaspan, C. Sadowski, L. Pollock, J. Clause, An Empirical Study of Practitioners' Perspectives on Green Software Engineering, Austin, TX, 2016, pp. 237–248, https://doi.org/10.1145/2884781.2884810.
[5] A. Hindle, Green Software Engineering: The Curse of Methodology, Suita, 2016, pp. 46–55, https://doi.org/10.1109/SANER.2016.60.
[6] A.J. Bokolo, A.M. Mazlina, Development of a green ICT model for sustainable enterprise strategy, J. Soft Comput. Decis. Support Syst. 3 (2016) 1–12.
[7] Spring, «Spring makes Java simple». https://spring.io/.
[8] G. Belani, Programming Languages You Should Learn in 2020, IEEE Computer Society https://www.computer.org/publications/tech-news/trends/programming-languages-you-should-learn-in-2020.
[9] C. Calero, M. Piattini, Puzzling out Software Sustainability, Sustain Comput Inform. Syst 16 (2017) 117–124, https://doi.org/10.1016/j.suscom.2017.10.011.
[10] C. Pang, A. Hindle, B. Adams, E. Hassan, What do programmers know about the energy consumption of software? IEEE Softw. 33 (2016) 83–89, https://doi.org/10.1109/MS.2015.83.
[11] G. Pinto, F. Castor, Energy efficiency: a new concern for application software developers, Commun. ACM 60 (12) (2017) 68–75, https://doi.org/10.1145/3154384.
[12] D. Li, W.G.J. Halfond, An Investigation Into Energy-saving Programming Practices for Android Smartphone App Development, 2014, pp. 46–53.
[13] R. Pereira, M. Couto, F. Ribeiro, R. Rua, J. Cunha, J.P. Fernandes, J. Saraiva, energy efficiency across programming languages: how Do energy, time, and memory relate?, in: Proceedings of the 10th ACM SIGPLAN International Conference on Software Language Engineering, New York, NY, USA, 2017, pp. 256–267, https://doi.org/10.1145/3136014.3136031.
[14] L.G. Lima, F. Soares-Neto, P. Lieuthier, F. Castor, G. Melfe, J.P. Fernandes, On Haskell and energy efficiency, J. Syst. Softw. 149 (2019) 554–580, https://doi.org/10.1016/j.jss.2018.12.014.
[15] M. Hähnel, B. Döbel, M. Völp, H. Härtig, Measuring energy consumption for short code paths using RAPL, SIGMETRICS Perform Eval. Rev. 40 (3) (2012) 13–17, https://doi.org/10.1145/2425248.2425252, ene.
[16] R. Pereira, M. Couto, J. Saraiva, J. Cunha, J.P. Fernandes, The influence of the java collection framework on overall energy consumption, in: Proceedings of the 5th International Workshop on Green and Sustainable Software, New York, NY, USA, 2016, pp. 15–21, https://doi.org/10.1145/2896967.2896968.
[17] R. Pereira, T. Carção, M. Couto, J. Cunha, J.P. Fernandes, J. Saraiva, SPELLing out energy leaks: aiding developers locate energy inefficient code, J. Syst. Softw. 161 (2020) 110463, https://doi.org/10.1016/j.jss.2019.110463.
[18] S.A. Chowdhury, A. Hindle, GreenOracle: estimating software energy consumption with energy measurement corpora, 2016 IEEE/ACM 13th Working Conference on Mining Software Repositories (MSR) (2016) 49–60.
[19] S. Chowdhury, S. Borle, S. Romansky, A. Hindle, GreenScaler: training software energy models with automatic test generation, Empir. Softw. Eng. 24 (4) (2019) 1649–1692, https://doi.org/10.1007/s10664-018-9640-7, ago.
[20] S. Nakajima, Model-based power consumption analysis of smartphone applications, ACESMB@MoDELS (2013).

[21] D. Li, S. Hao, W.G.J. Halfond, R. Govindan, Calculating source line level energy information for android applications, in: Proceedings of the 2013 International Symposium on Software Testing and Analysis, New York, NY, USA, 2013, pp. 78–89, https://doi.org/10.1145/2483760.2483780.
[22] C. Zhang, A. Hindle, D.M. German, The impact of user choice on energy consumption, IEEE Softw. 31 (3) (2014) 69–75.
[23] R. Jabbarvand, A. Sadeghi, J. Garcia, S. Malek, P. Ammann, Ecodroid: an Approach for Energy-based Ranking of Android Apps., 2015, pp. 8–14.
[24] C. Sahin, F. Cayci, I. Manotas, J. Clause, F. Kiamilev, L. Pollock, K. Winbladh, Initial explorations on design pattern energy usage, 2012 First International Workshop on Green and Sustainable Software (GREENS) (2012) 55–61.
[25] L. Cruz, R. Abreu, J. Grundy, L. Li, X. Xia, Do energy-oriented changes hinder maintainability? 2019 IEEE International Conference on Software Maintenance and Evolution (ICSME) (2019) 29–40.
[26] S.A. Chowdhury, A. Hindle, R. Kazman, T. Shuto, K. Matsui, Y. Kamei. GreenBundle: An Empirical Study on the Energy Impact of Bundled Processing, 2019, pp. 1107–1118, https://doi.org/10.1109/ICSE.2019.00114.
[27] J. Mancebo, F. García, C. Calero, A process for analyzing the energy efficiency of soft-ware, Sent Inf. Softw. Technol. (2020).
[28] J. Mancebo, C. Calero, F. Garcia, M. Moraga, I. Garcia-Rodriguez de Guzman, FEETINGS: Framework for Energy Efficiency Testing to Improve Environmental Goal of the Software, Sustainable Computing: Informatics and Systems, 30, 2021, https://doi.org/10.1016/j.suscom.2021.100558. ISSN 2210-5379.
[29] C. Wohlin, P. Runeson, M. Höst, M.C. Ohlsson, B. Regnell, A. Wesslén, Experimentation in Software Engineering, Springer-Verlag, Berlin Heidelberg, 2012.



Calero, C. is a Professor at the University of Castilla-La Mancha in Spain and has a PhD in Computer Science. Her research interests include: software quality, software quality models, software measurement and software sustainability definition, evaluation, measurement and assessment.

Polo, M. is a Professor of Computer Science at the University of Castilla-La Mancha. He has Msc and Phd degrees in Computer Science. His main research areas are related to the automation of software processes, especially testing.

Moraga, M. has a PhD in Computer Science. She received her MSc in Computer Science and her Technical Degree in Computer Science from the University of Castilla-La Mancha (UCLM). She is currently associate professor at the Escuela Superior de Informática of the University of Castilla-La Mancha in Ciudad Real (Spain). She is a member of the Alarcos Research Group which specializes in Information Systems, Databases and Software Engineering in that University. Her research interests are: software quality and software sustainability.