Design Patterns for Software Sustainability

KEVIN LANO
King's College London
London, UK
kevin.lano@kcl.ac.uk

Zishan Rahman
King's College London
London, UK
zishan.rahman@kcl.ac.uk

Lyan Alwakeel
King's College London
London, UK
lyan.alwakeel@kcl.ac.uk

Abstract

The term 'sustainable software' refers to software which has low negative impacts on the environment and/or high positive impacts, in particular, software which has low negative impact in terms of greenhouse gas emissions due to energy consumed by the software execution. In this paper we examine the relevance of a range of software design patterns and architectural styles for improving software sustainability by reducing the energy use of software execution, and we define a pattern language incorporating optimised versions of selected patterns/styles. We also provide a detailed evaluation of the impact of these patterns and styles on software energy use in particular software environments.

CCS Concepts

• **Software and its engineering → Software performance**.

Keywords

Software sustainability, design pattern, architectural style, software energy use

ACM Reference Format:
KEVIN LANO, Zishan Rahman, and Lyan Alwakeel. 2024. Design Patterns for Software Sustainability. In *31st Conference on Pattern Languages of Programs, People and Practices, October 13–16, Skamania Lodge, Columbia River Gorge, Washington, USA, 2024.* ACM, New York, NY, USA, 11 pages.

1 Introduction

According to some estimates, information technology is one of the largest single global producers of greenhouse gas (GHG) emissions, responsible for 3.9% of global emissions, and is projected to reach 14% by 2040[1].

While it is hardware which consumes energy, the software running on that hardware has major impact on the energy consumption, and hence there has been increased interest in *software sustainability* as a means to reduce the environmental impact of the digital sector.

Sustainable software (also referred to as *Green software*) is defined in [19] as

> "Software whose direct and indirect negative impacts on economy, society, human beings and environment that result from development, deployment and usage of the software are minimal and/or which has a positive effect on sustainable development."

A principal focus of software sustainability has been to consider the energy usage or consumption of the software when executed. This energy usage can then be converted into a greenhouse gas (GHG) emissions figure based on the execution platform characteristics and source of electricity supply used (e.g., renewables versus fossil fuels) [13].

In this paper we investigate the energy use of software design patterns and architectural styles. We propose some general patterns for achieving sustainable software by means of reducing the energy use of software execution, and we identify restrictions or modifications of classic design patterns and architectural styles which could help to achieve reduced software energy use.

Section 2 surveys previous related work in sustainable software. Sections 3 and 4 investigate the selection of design patterns and architectural styles for sustainability improvement, and propose new and modified patterns for sustainable software. Section 5 evaluates the reductions in energy use which can result from design pattern and architectural style choices. Finally, Section 6 gives conclusions.

2 Related Work

The field of software sustainability is over 15 years old [24], and the research in this field has been wide-ranging, covering applications of software and other information technologies in different domains to reduce the environmental harm of human activities ('IT for sustainability') as well as the reduction of the environmental impact of IT ('Sustainable IT').

2.1 Design patterns and software energy use

There has been considerable software sustainability research in the fields of programming languages and program design, whereby the energy use of different software design and implementation options are considered, including design patterns [4, 17, 27], refactorings [28], programming language choices [25] and data structure choices [18, 29]. Of particular relevance to this paper are the exploratory investigation [27] of 15 classic GoF design patterns, the specific investigations of [4, 17], and the proposal for an improved mobile app architectural style in [30]. The investigation of [27] showed that for several popular design patterns, introducing the pattern may either reduce or increase program energy use. Energy use tended to be increased by introducing patterns which involve additional object creation actions and/or additional method calls compared to the original version of a system without the pattern. Patterns which substantially reduced the number of objects (without adding extra calls), such as the Flyweight pattern, tended to reduce energy

[1] www.bjss.com/articles/the-current-state-of-the-industry-on-green-software-development

PLoP 2024, October 13–16, 2024, Skamania Lodge, Columbia River Gorge, Washington, USA

Hillside ISBN 978-1-941652-20-6

use. The investigation of [17] also found significant energy use reductions through the use of Flyweight, and significant increases for Decorator, with marginal impact from the use of Facade, Prototype and Template Method. Specific investigation of the Visitor pattern found that it increased energy use across different JVM configurations [4]. The situation for patterns is similar to that for refactorings, where certain classic refactorings (such as Extract Method) were found to increase energy use in some situations [28].

The paper [30] proposed a version, RMVRVM, of the classic MVVM mobile app architectural style, with reduction in energy usage achieved by moving the model and view model parts of the pattern off-device.

Classic design patterns, such as those of [7, 10] are primarily concerned with the improvement of software flexibility, extensibility and modularity, and such quality factors may conflict with the reduction of software energy use. For example, the use of indirection in communications between objects facilitates increased flexibility in client-server relationships and configurations, but also introduces an overhead of additional communication steps in each client-server interaction.

2.2 Software energy use measurement

Measurement or estimation of the energy use of a software system is a key initial step to enable developers to detect energy use flaws in their code and to improve the sustainability of their software [22]. Energy measurement techniques include external power meters, internal (on-chip) power sensors, or energy predictive models based on performance monitoring counters [6, 21]. Tools such as Green Algorithms [13], mlco2 [12], Codecarbon[2] and Carbontracker [3] provide estimates of software carbon footprint based on the use of computational resources and the location of such resources.

Because we are interested in evaluating the relative energy use characteristics of different software designs, independent of specific programming languages, a generalised energy use estimation approach will be used. This involves estimating the *computational cost* or effort involved in performing an operation or statement at the design level, by breaking down any computational task into a series of basic actions [11, 26]. The computational cost of an execution of an operation or statement is then the sum of the costs of the basic actions which may be executed as part of the operation/statement execution.

The basic computation actions or steps include (i) creation of an object of a class; (ii) assignment of a value to an object attribute; (iii) reading the value of an object attribute; (iv) modifying the value of an attribute; (v) declaration of a local variable; (vi) assignment of a value to a local variable; (vii) reading the value of a local variable or operation parameter; (viii) modifying the value of a local variable; (ix) performing an equality or ordering test on two values; (x) deleting an object; (xi) invoking an object operation, and other communication actions (local or remote) between software elements.

Not all of these are applicable to all programming languages, in particular, (i) and (x) are specific to object-oriented languages, however they have analogues in procedural languages, such as the dynamic allocation and deallocation of memory in C.

[2]https://codecarbon.io

Using the above approach, and making some assumptions about the relative costs of different basic steps, alternative designs can be compared in terms of their relative computational costs.

3 Design Patterns

The analysis of [27] showed that energy use may be increased by introducing design patterns which either (i) increase the number of method calls and/or (ii) increase the number of object instantiations, relative to a design without the pattern. However, (i) and (ii) are common properties of design patterns, because division of software state and computation into separate objects is a key means by which patterns increase software flexibility. Table 1 shows that all of the GoF patterns have one or both property (i) or (ii), with the sole exception of Flyweight. The properties are also true for the fundamental patterns *Immutable*, *Delegation* and *Proxy* of [10], although not for the *Interface* and *Marker Interface* fundamental patterns. Patterns based on Delegation or Proxy, such as the Adapter, Bridge, Chain of Responsibility, Command, Composite, Facade, Interpreter, Mediator or Template Method patterns, are intrinsically difficult to optimise. The same is true for patterns which essentially use the dynamic creation of objects (Decorator) or substantial processing using method calls (Visitor). Optimisation of Visitor using reflection is proposed in [4]. However, reflection is itself an energy-expensive computational technique.

To address these issues, we propose a pattern language for sustainable software, consisting of five fundamental patterns (Section 3.1), together with more energy-efficient versions of classic design patterns based upon these fundamental patterns (Section 3.2). The relations between the patterns in this pattern language are shown in Figure 1.

3.1 Fundamental patterns for energy-use reduction

We define five fundamental patterns which provide alternative strategies for software energy-use reduction:

- *Object Reuse* – reduce unnecessary object creation by memoising already-created objects (left hand side of Figure 2).
- *Reconfigure Computation* – reorganise the internal properties of a computational process, or its assignment to computational devices, to reduce overall energy use.
- *On-Demand Computation* – postpone a computation until the result is actually required, to avoid redundant computations whose result is never used. We specifically consider *On-demand Synchronisation* (right hand side of Figure 2).
- *Reschedule Computation* – reschedule computations or reorganise the order of execution of computation parts to reduce energy-use or GHG emissions.
- *Relocate Computation* – migrate processing from one computational unit to another, to improve energy efficiency of the overall computation.

3.1.1 Object Reuse. The Object Reuse pattern concerns the provision of target objects to clients. By means of a cache map (represented as a qualified association in Figure 2), the target object manager can determine if a required object already exists and is

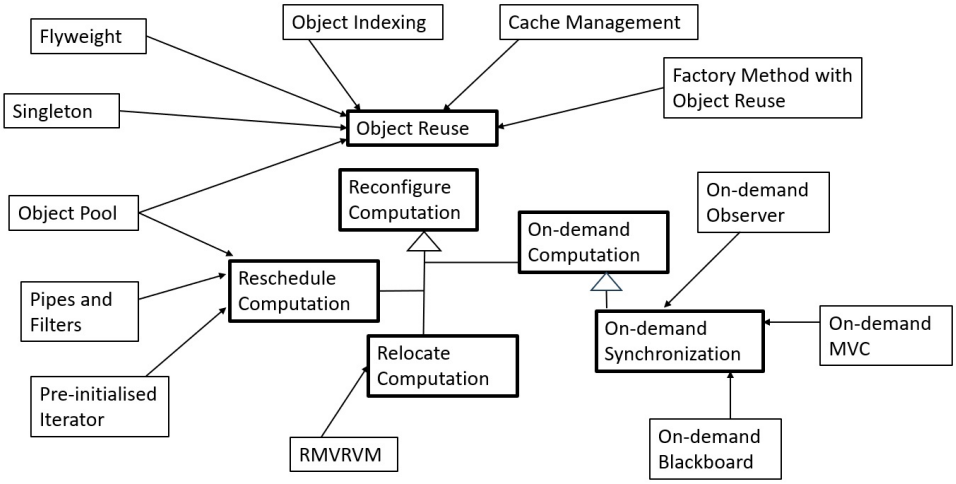

Figure 1: Pattern language for sustainable software

Table 1: Classic patterns of [7] and possible sustainability improvements

Pattern	*Adds objects*	*Adds calls*	*Sustainability improvements*
Abstract Factory	√	√	Abstract Factory with Object Reuse
Adapter	√	√	–
Bridge	√	√	–
Builder	√	√	Builder with Object Reuse
Chain of Responsibility	√	√	Re-order chains (Reschedule Computation)
Command	√	×	–
Composite	√	√	–
Decorator	√	√	–
Facade	√	√	–
Factory Method	×	√	Factory Method with Object Reuse
Flyweight	×	×	A special case of Object Reuse
Interpreter	√	×	–
Iterator	√	√	Pre-initialised Iterator (Reschedule Computation)
Mediator	√	√	–
Memento	√	√	Memento with Object Reuse
Observer	√	√	On-demand Observer
Prototype	×	√	Prototype with Object Reuse
Proxy	√	√	–
Singleton	×	√	Optimised Singleton
State	√	×	State with Object Reuse
Strategy	√	√	Strategy with Object Reuse
Template Method	×	√	–
Visitor	√	√	–

available for reuse by clients, so that (re-)creation of the object can be avoided.

Special cases of the Object Reuse fundamental pattern are:

- *Flyweight* – objects are identified and looked-up by their intrinsic attributes. These objects are typically shareable.
- *Object Indexing* [15] – objects of a class are uniquely identified and obtained by the value of an *identity* attribute of *String* type. These objects are expected to be shareable.
- *Object Pool* [10] – the pool of unused interchangeable server objects is non-empty. These objects may not be shareable.
- *Cache Management* [10] – a required data object is already in the cache. These objects may or may not be shareable.

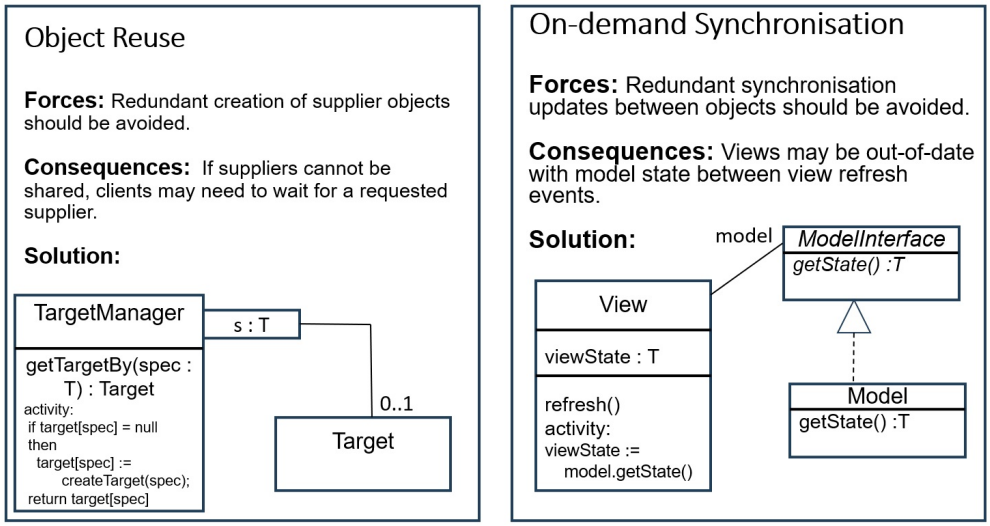

Figure 2: Object Reuse and On-demand Synchronisation fundamental patterns

Singleton could also be regarded as a special case of the pattern (the shareable single instance of the singleton class is created once and then reused when required). Another special case is the *Demand Cache* pattern [20], which memoises function results or query request results as objects, to avoid repeated redundant computation of these results.

A potential consequence of the Object Reuse pattern is that memory requirements are increased. Empirical studies have suggested that the energy costs of memory use are significantly lower than the costs of CPU and GPU computation in general [13], therefore this could be a beneficial tradeoff. Detailed computational cost analysis of patterns based on Object Reuse illustrates why they should reduce computational energy use. In the case of Object Indexing, for a class C with a single *identity* attribute *att*, the primitive object creation operation $createC(val)$ always creates a new C object with *att* value equal to *val*. The Object Indexing pattern instead uses a cache map $Cmap : SortedMap(String, C)$ and defines C creation by an operation *getCBy* defined by the following pseudocode:

```
operation getCBy(val : String) : C
activity:
  var res : C := Cmap[val];
  if res = null
  then
    res := createC(val) ;
    Cmap[val] := res
  else skip ;
  return res;
```

This coding avoids the creation of more than one instance of C with the same *att* value.

Without the pattern, P creation requests for R objects with distinct *att* values ($P \geq R$) would result in a computational cost of $P * X$ where X is the cost of $createC(val)$. With the pattern, the cost becomes

$$P * L + R * X$$

where L is the lookup cost. Therefore the cost of the P creations is reduced if

$$L < X * \frac{(P-R)}{P}$$

The Object Reuse approach could also be used to optimise the Factory Method design pattern for generating families of objects [7]. Instead of always generating new objects that satisfy required characteristics (supplied in the parameters of the factory method), the factory method could use the combination of characteristic values as a key to lookup and reuse existing objects, in preference to creating new objects. An example could be a factory method that returns random number generators which meet different required criteria for speed, statistical quality, etc. A similar approach could be used for *Abstract Factory with Object Reuse*, *Memento with Object Reuse* and *Prototype with Object Reuse*. In the case of the State and Strategy patterns, objects for different states and strategies can be created once and reused as needed. Builder can be optimised by reuse of concrete builder objects, and possibly of produced objects if these are reusable.

3.1.2 Reconfigure Computation. A general pattern for reducing computational effort is *Reconfigure Computation*. This generalises *Relocate Computation*, *Reschedule Computation* and *On-demand Computation*, and also includes cases such as dynamically changing processor clock speed and reducing numeric precision in order to reduce energy use [5].

3.1.3 On-demand Computation. *On-Demand Computation* is a special case of *Reconfigure Computation*, in which a computation is postponed until the result is actually required, thus avoiding redundant computations. In the cases considered here, the deferred computation is the synchronisation of views of a model/subject with the current model information (right hand side of Figure 2).

To avoid too-frequent *getState* calls, a time-based staleness/validity indicator of the view state can be used, i.e., *refresh*() only calls *getState* if *getState* has not been called for more than a period $\epsilon > 0$.

Special cases of this pattern are:

- On-demand Observer (Section 3.2.2).
- On-demand Blackboard (Section 4.4).
- On-demand MVC (Section 4.4).

3.1.4 Reschedule Computation. Another computation reconfiguration strategy relevant for software sustainability is *Reschedule Computation*, which involves rescheduling computations or reorganising the order of execution of computation parts [8], possibly to spread computations over longer time periods, or to use electricity at times of lower demand, when renewable energy sources are more likely to be available, hence reducing the GHG emissions due to the computations [32]. This especially applies to large-scale computational tasks such as ML model training. Pre-initialisation of objects, batching of updates to a data structure, or buffering of file operations, are also examples of this technique.

Examples of computation rescheduling patterns are:

- *Map Objects before Links* [15] – this organises model and object copying processes so that instances are copied in a first phase, without references between them, then linking references are instantiated in a second phase. This avoids recursion in object creation/copying.
- *Phased Construction* [15] and *Layered Initialisation* [10] – these reschedule construction/initialisation actions according to different constraints or policies.
- *Primed Cache* [20] – this pre-computes a set of function or request result values which are anticipated to be requested by clients.

Object pooling can also be regarded as a case of computation rescheduling: the creation of pooled suppliers is done at pool initialisation instead of during the main computation process.

Optimising the Iterator pattern by pre-initialising an array with the iteration order is another example of this fundamental pattern (Section 3.2.3).

Optimising a Pipes-and-Filters chain by changing the order of filter processes is also a computation rescheduling (Section 4.1).

3.1.5 Relocate Computation. Relocate Computation is a special case of *Reconfigure Computation* that involves moving processing from one computational unit to another. It is applicable to large-scale computational tasks such as machine learning training, in order to move these to execute upon computational resources (e.g., data centers in particular physical locations) which have lower GHG emissions [12]. It can be used to move tasks from software to special-purpose hardware, as in [9], or from local devices to the cloud [23]. It can also be used to move computations from relatively energy-inefficient devices such as mobile phones to computationally-optimised servers, as in the RMVRVM pattern [30]. However, any additional costs of data transfer and remote computation management associated with computation relocation also need to be taken into account when evaluating this pattern.

3.2 Optimised versions of specific patterns

The Singleton, Observer and Iterator classic patterns are particularly important to examine from the viewpoint of software sustainability because of their extensive use in software applications and programming languages. Each of these patterns can involve significant numbers of additional operation calls:

- A call $C.getInstance()$ is made each time the unique instance of a *Singleton* class C is used.
- Changes to a *Subject* in the Observer pattern are notified to each observing client *Observer*/*View* [7].
- Navigation actions, updates and queries on an iterator for an underlying collection may also involve delegated calls to that collection.

3.2.1 Singleton. Singleton can be optimised by replacing the static *getInstance*() operation of a singleton class C by a public static instance variable $inst : C$, which is initialised to a new C instance at application initialisation. Accesses to the unique instance are then simplified to $C.inst$ or $C::inst$ (in C++) instead of $C.getInstance()$. This enforces the optimisation of the query call by a variable access; such an optimisation may be performed by optimising compilers. However, security issues may result from direct access to a global variable, so that this optimisation cannot be used. Inlining of *getInstance* and other getters may alternatively be used, subject to the restrictions of [28] for the use of inlining of operations *op* to reduce energy use.

3.2.2 Observer. Observer was found to be one of the most energy-expensive patterns in [27]. In order to reduce the communication and processing costs of this pattern, a potential optimisation is to use an on-demand computation approach to replace notifications from the subject to the views by explicit requests from each view to the subject when the view is initialised or refreshed (Figure 3). This is particularly applicable in situations such as mobile or tablet devices where only one view will be visible at a time. Such views would be refreshed when they become visible. In the original Observer version, if there are V views, each event that changes the subject data leads to a computational cost of $V * (C + U)$ where C is the cost of an update call from the subject to a view and U the local view update cost. The optimised version instead has a cost $C + U$ for each view initialisation/refresh. Thus there is a potential for reducing computational costs if there are several $V > 1$ views and view refreshes occur less frequently than subject update events. If there are N update events and M refreshes over a given time period, then the optimised version reduces computational cost if $M < V * N$. However this approach means that views may be out-of-date wrt subject data for various periods of time. If an explicit staleness bound $\epsilon > 0$ is used, then there are a maximum of T/ϵ calls of *getState* from a given view over time period T, instead of M. The bound expresses that information staleness of time ϵ is acceptable to clients. In this version, the On-demand Observer reduces the computational cost of communication between an individual view and the subject over period T if $min(M_V, T/\epsilon) < N$ where N is the number of subject state updates over period T, and M_V the number of refresh requests for the view over this time.

The optimised pattern no longer has the Observer property that views are always synchronised with the subject, however it retains

the property that views are independent, and are responsible for updating their own state from subject data.

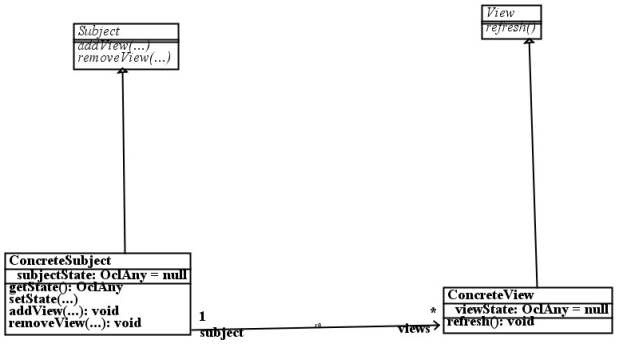

Figure 3: On-demand Observer design pattern structure

3.2.3 Iterator. Iterator may be optimised by using an array-based memoised copy of the iterated collection. The array is pre-initialised at iterator creation to the required iteration order for the collection, and is subsequently processed independently of the collection. This technique however prevents replacement of collection elements via the iterator (although the internal data of elements may be modified via the iterator, if these are objects). There is also a cost of additional memory use.

4 Architectural Styles

There has been much work in the software architecture field on architectural sustainability in the sense of ensuring that an architectural design choice can support a system over the long term, including supporting system evolution. However there have been few works specifically addressing the energy-use or (environmental) sustainability implications of architecture choices [31]. This is a significant area of research because architectural choices may have substantial effects on the energy usage of software applications [30]. As with the detailed analysis of code and evaluation of coding alternatives, the scenarios of use of a system are significant when evaluating the sustainability implications of particular architecture choices, and there may be tradeoffs between other software quality goals and sustainability. Here we consider some widely-used architectural styles, and describe approaches to improve their energy-efficiency. Certain styles are also regarded as patterns, such as Object Pool and Value Object [1].

4.1 Pipes and Filters

For the Pipes and Filters architectural style [16] (Figure 4, using UML component diagram notation), it may be possible to modify the order of execution of filters within a chain to reduce the overall computational cost and energy use.

In the case of two filter components F_1 and F_2, with F_1 preceding F_2 in the chain, with computational costs $c_i(N)$ for each F_i in terms of input dataset size N, and F_i producing output dataset size $f_i(N)$, it is beneficial to swap the order of the filters if

$$c_2(N) + c_1(f_2(N)) \ < \ c_1(N) + c_2(f_1(N))$$

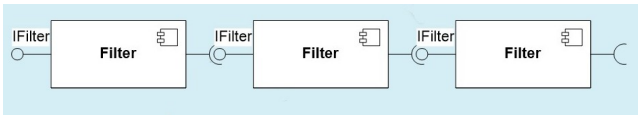

Figure 4: Pipes and Filters architectural style

In other words, if

$$c_2(N) - c_2(f_1(N)) \ < \ c_1(N) - c_1(f_2(N))$$

This can be expressed as stating that the growth rate of F_2's computational cost (between an input that is the output of F_1, and unfiltered input) is lower than the growth rate of F_1's computational cost (comparing an input that is the output of F_2, and unfiltered input). If c_1 and c_2 are the same function c, then the condition reduces to $f_2(N) \ < \ f_1(N)$ assuming that c is monotonically increasing. This means that the module that filters its input by a greater amount should go first in the chain.

Alternatively, for distinct c_1, c_2, if F_1 makes no reduction in output, then the condition becomes

$$0 < c_1(N) - c_1(f_2(N))$$

This means that if F_2 makes any reduction in output it should go first. Such re-arrangements are only possible if the filter computations themselves are order-independent. Similar computation rescheduling techniques could also be used to reduce the energy use of implementations of the Chain of Responsibility design pattern.

4.2 Object Pool

In the case of Object Pool [10], a pool of size $M > 1$ instances of a supplier component S is pre-initialised and maintained to serve client requests. If the average cost of serving a request is R, handling N requests without the pattern costs $N * (I + R)$ where I is the cost of creating and initialising an S instance. With an object pool of size M, the overall cost is $M * I + N * R$. This means that overall computational cost is reduced if $M < N$. However, the limited size of the pool may result in delays in servicing requests if more than M requests are waiting to be serviced at any time point, and the pool objects are unshareable.

4.3 Value Object

In the case of Value Object [1] being introduced to reduce the number of transfers of data across a network, if the cost of establishing a network connection is C, and the cost of data transmission is B per byte, then calls with P bytes grouped into $N > 1$ attributes cost

$$N * C + N * P * B$$

without using Value Object, because the attributes are sent in N individual remote calls $robj.setatt(attv)$ for each separate att. With Value Object, the N individual $setatt$ calls are replaced by a single remote call $robj.setatts(vo)$, and the cost of a call is

$$C + V_{N,P} + A + N * P * B$$

where $V_{N,P}$ is the cost of creating and initialising the value object vo, and A the additional costs resulting from indirect access

vo.getatt() to value object features in the called operation *setatts*. Thus introducing Value Object should be beneficial in general if

$V_{N,P} + A < (N-1) * C$

4.4 Blackboard and MVC

The Blackboard (Figure 5) and MVC architectural styles can be optimised using the On-demand Computation approach adopted for Observer in Section 3.2.

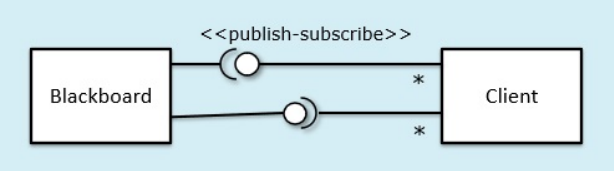

Figure 5: Blackboard architecture

In the case of Blackboard, publish/subscribe notifications from the blackboard to all clients are replaced by client query requests for the blackboard state, whenever the client view is refreshed/initialised (Figure 6). There could be V views in total, and N update events, plus M explicit view refresh events. With this optimised architecture the costs would be

$(N + M) * (D + U)$

in contrast to $N * V * (D + U)$ for the original blackboard version which broadcasts event notifications to each view, where D is cost of a call and U the cost of a client update. The optimisation therefore reduces costs if $M < N$ and $V \geq 2$.

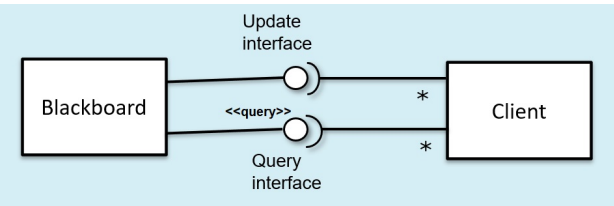

Figure 6: On-demand Blackboard architecture

Similarly, the MVC architectural style can be optimised so that views update the model via the controller, and query the model when they require an up-to-date model state (Figure 7). This optimised architecture is used by the mobile app generation strategy of AgileUML [14] and AppCraft [2].

In the cases of Blackboard and MVC, the optimised versions retain the original style property that views are independent and are responsible for presenting their representation of the data source data, but the synchronisation property of the styles is weakened.

5 Evaluation

In this section we evaluate the effect of design choices upon the energy use of implementations. We consider some example design patterns and architectural styles from Sections 3 and 4.

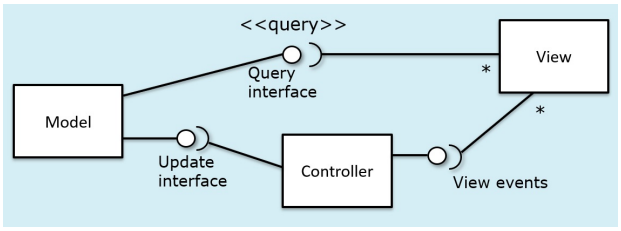

Figure 7: On-demand MVC architecture

For these experiments we evaluate energy use in milli-Watt hours (mWh) using the calculator at `https://calculator.green-algorithms.org`. We also use JoularJX [21] to cross-check the results using a different energy-use estimation approach. We consider three implementation platforms and configurations:

(1) Windows 10 OS, with JDK 8, Python 3.10 and C++11, on a laptop with a 4 core i5-9400 processor, UK location, 8GB available memory.
(2) Linux Mint 21.3, with JDK 21 and Python 3.10, on a i7-1365U processor laptop with 10 cores, UK location, 16GB available memory.
(3) MacOS 10.15 with Python 2.7 on an iMac Pro, Intel Xeon processor, 8 cores, 32GB available memory, UK location.

We consider three programming languages, representing the three main categories of 3GLs: Java (virtual machine language), C++ (compiled language), Python (interpreted language).

The varying inputs to the Green Algorithms energy estimate calculation are the processing time in ms, processor utilisation percentage and memory use (in GB). These are manually measured using performance monitoring tools on the respective platforms. The average result of three separate executions for each specific energy evaluation case is taken.

The following formula is used by [13] for the energy use of a computation in kWh:

$$t * (nc * PowerDrawCPU * cpu + MemoryUse * PowerDrawMemory) * PUE * 0.001$$

Computation time t is in hours, nc is the number of processor cores, *PowerDrawCPU* is the power in W of each core, *cpu* is the processor utilisation (ranging from 0 to 1), *MemoryUse* is the amount of memory used by the computation in GB, and *PowerDrawMemory* is the power drawn by memory use per GB. *PUE* is the fraction of energy used for computing equipment, in our evaluations this is always 1.

Thus in the case of the Windows platform, where nc = 4, PowerDrawCPU = 10.8W per core, PowerDrawMemory = 0.3725 W/GB, the energy use in Wh is then

$(durationMS/3600.0) * (cpu * 43.2 + MemoryUse * 0.3725)$

where *durationMS* is the computation duration in milliseconds.

JoularJX computes energy use of specific Java processes using on-chip performance counters, in this respect it can provide a more precise estimation than Green Algorithms tool, which does not distinguish different processes running on a device.

All data and examples can be found on Zenodo [3].

5.1 Design patterns: On-demand Observer

Here we evaluate the on-demand version of the Observer pattern described in Section 3.2.2. The standard structure of Observer is used for the unoptimised version, with a single subject and N attached observers. This leads to the sending of the order of N*N messages in response to N subject update events. In contrast, in the optimised version where observers only request subject state in response to refresh events, N subject updates and N refresh events only lead to 2*N messages. Figures 8 and 9 show a general reduction in energy use for the optimised version.

5.2 Design patterns: Object Indexing

We evaluate the impact of this pattern on energy use by measuring the energy use of the creation of N instances of a target class, with 50% of the instances having new key values. In the version without the pattern all N instances are created, whilst in the version with the pattern only $\frac{N}{2}$ instances are created. Figure 10 shows that this tends to reduce energy use.

5.3 Design patterns: Pre-initialised Iterator

For this evaluation, we compare Java implementations of an array-based pre-initialised iterator *OclIteratorOptimised* and the standard sequence-based iterator *OclIterator*. The energy use of complete iterations over collections of sizes 10,000, 100,000 and 1,000,000 elements are evaluated on the Windows platform. The results indicate that the optimised version is more energy-efficient (Figure 11).

5.4 Architectural styles: Blackboard

As described in Section 4, the Blackboard architectural style can be optimised by removing the two-way notification of view changes to the blackboard and notification of blackboard changes to views, and replacing this by unidirectional updates and data requests from the views to the blackboard.

Figures 12 (a) and (b) show that this optimisation has the expected benefits in terms of energy use. The unoptimised version has a quadratic growth in energy use with an increasing number of views, whilst the optimised version has linear growth. Similar results are obtained for the other platforms and languages.

Similar results are obtained for MVC.

5.5 Evaluation with JoularJX

We repeated the energy use measurements for Java on the Linux platform using an alternative energy-use measurement tool, JoularJX [21]. The results (Figure 13) are consistent with those obtained using the [13] calculator. Note that 1 mWh = 3.6 Joules.

6 Conclusions

This paper has examined design patterns and architectural styles from the viewpoint of sustainable software engineering. We used a general energy-use estimation approach to provide guidance on ways to select and improve pattern and style applications with respect to their energy use. Absolute assurance about the relative energy use of two or more alternative designs cannot be given, because different implementation platforms may have different characteristics which affect the relative ranking of alternative algorithms and designs. However, by using the generalised concept of *computational cost*, we have compared the typical computational loads of design alternatives. We defined general strategies for energy-use reduction, and instantiated these to provide more energy-efficient versions of several class design patterns and architectural styles. We also verified that the expected improvements in energy efficiency were actually observed for the modified patterns and styles across a range of computational platforms and programming languages.

[3] zenodo.org/records/14900982

References

[1] D. Alur, J. Crupi, and D. Malks. 2003. *Core J2EE Patterns 2nd edition.* Prentice Hall.

[2] L. Alwakeel, K. Lano, and H. Alfraihi. 2023. Towards integrating machine learning models into mobile apps using AppCraft. In *AgileMDE workshop, STAF 2023.*

[3] L. Anthony, B. Kanding, and R. Selvan. 2020. Carbontracker: Tracking and Predicting the Carbon Footprint of Training Deep Learning Models. In *ICML workshop "Challenges in Deploying and Monitoring ML Systems".*

[4] D. C. Bree and M. O'Cinneide. 2022. The energy cost of the Visitor pattern. In *ICSME.*

[5] M. Cohen, H. Zhu, S. Emgin, and Y. Liu. 2012. Energy Types. In *OOPSLA '12.* ACM, 831–849.

[6] M. Fahad, A. Shahid, R. Manumachu, and A. Lastovestsky. 2019. A comparative study of methods for measurement of energy of computing. *Energies* 12 (2019).

[7] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. 1995. *Design Patterns: Elements of reusable object-oriented software.* Addison-Wesley.

[8] R. Goncalves, R. Silva de Oliveira, and C. Montez. 2005. Design Pattern for the Adaptive Scheduling of Real-time Tasks with Multiple Versions in RTSJ. In *Proceedings of SCCC.*

[9] M. Gotz, F. Dittmann, and T. Xie. 2009. Dynamic relocation of hybrid tasks: strategies and methodologies. *Microprocessors and Microsystems* 33 (2009), 81–90.

[10] M. Grand. 1998. *Patterns in Java, Volume 1.* Wiley.

[11] D. Gries. 1971. *Compiler construction for digital computers.* Wiley.

[12] A. Lacoste et al. 2019. Quantifying the Carbon Emissions of Machine Learning. *arXiv* 1910.09700v2 (2019).

[13] L. Lannelongue, J. Grealey, and M. Inouye. 2021. Green Algorithms: Quantifying the carbon footprint of computation. *Advanced Science* 8 (2021).

[14] K. Lano, S. Kolahdouz-Rahimi, L. Alwakeel, and H. Haughton. 2021. Synthesis of mobile applications using AgileUML. In *ISEC 2021.* 1–10.

[15] K. Lano, S. Kolahdouz-Rahimi, S. Yassipour-Tehrani, and M. Sharbaf. 2018. A survey of model transformation design patterns in practice. *JSS* 140 (June 2018), 48–73.

[16] K. Lano and S. Yassipour Tehrani. 2023. *Introduction to Software Architecture.* Springer UTiCS.

[17] S. Maleki et al. 2017. Understanding the impact of object-oriented programming and design patterns on energy efficiency. In *IGSC Workshop on Sustainability in Multi/Many-core systems.*

[18] J. Michanan, R. Dewri, and M. Rutherford. 2017. GreenC5: An adaptive, energy-aware collection for green software development. *Sustainable Computing: Informatics and Systems* 13 (2017), 42–60.

[19] S. Naumann, M. Dick, E. Kern, and T. Johann. 2011. The GREENSOFT Model: a reference model for green and sustainable software and its engineering. *Sustainable Computing: Informatics and Systems* 1 (2011), 294–304.

[20] C. Nock. 2004. *Data Access Patterns – Database interactions in Object-oriented Applications.* Pearson.

[21] A. Noureddine. 2022. PowerJoular and JoularJX: Multi-platform software power monitoring tools. In *18th International Conference on Intelligent Environments.* IEEE.

[22] P. Pathania et al. 2023. Towards a Knowledge Base of Common Sustainability Weaknesses in Green Software Development. In *ASE '23.* IEEE.

[23] Priyavanshi Pathania and Rajan Dilavar Mithani. 2021. Sustainability in migrating workloads to public clouds. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering* (Virtual Event, Australia) *(ASE '20).* Association for Computing Machinery, New York, NY, USA, 166–169. doi:10.1145/3417113.3423001

[24] B. Penzenstadler et al. 2014. Systematic Mapping Study on Software Engineering for Sustainability (SE4S). In *EASE '14.* ACM.

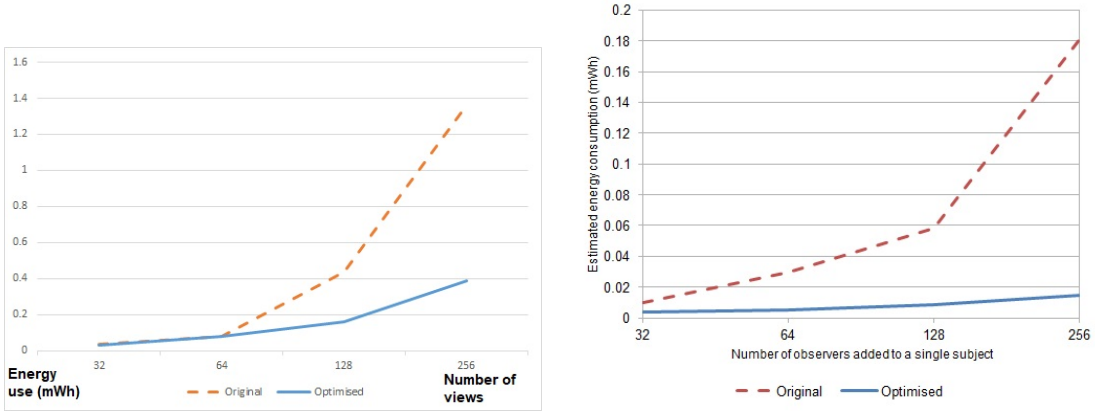

Figure 8: Observer pattern: unoptimised versus optimised versions: (a) Windows 10, Java; (b) Linux, Java

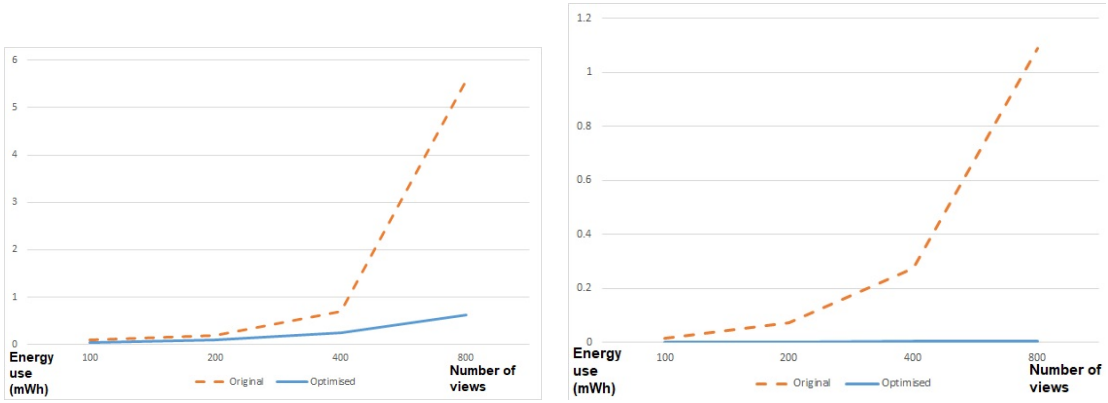

Figure 9: Observer pattern: unoptimised versus optimised versions: (a) Windows 10, C++; (b) MacOS, Python

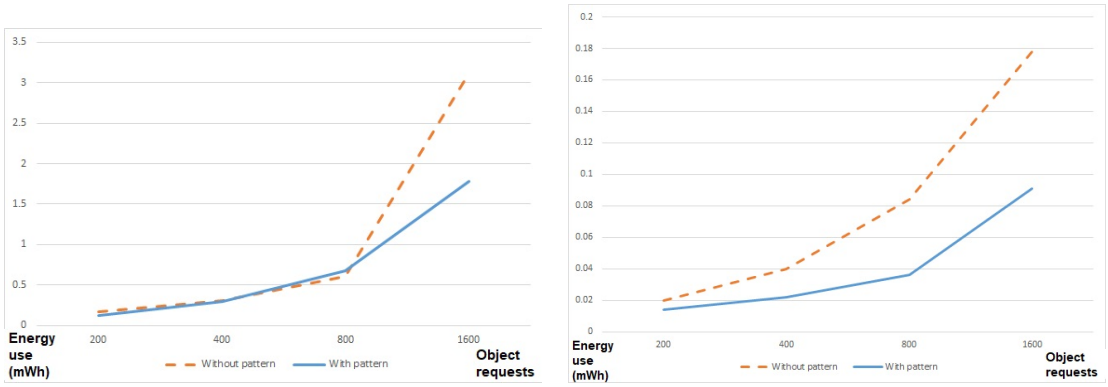

Figure 10: Object Indexing: (a) Windows 10, Java; (b) Windows 10, Python

[25] R. Pereira, M. Couto, F. Ribeiro, R. Rua, J. Cunha, J. P. Fernandes, and J. Saraiva. 2017. Energy efficiency across programming languages: How do energy, time and memory relate?. In *SLE '17*. ACM, 256–267.

[26] T. Pratt and M. Zelkowitz. 2001. *Programming Languages: Design and implementation*. Prentice Hall.

[27] C. Sahin, F. Cayel, I. Gutierrez, J. Clause, F. Kiamilev, L. Pollock, and K. Winbladh. 2012. Initial explorations on design pattern energy usage. In *International Workshop on Green and Sustainable Software*. 55–61.

[28] C. Sahin, L. Pollock, and J. Clause. 2014. How do code refactorings affect energy usage?. In *ESEM '14*. ACM.

[29] J. Singh, K. Naik, and V. Mahinthan. 2015. The Impact of Developer Choices on Energy Consumption of Software on Servers. *Procedia Computer Science* 62 (2015), 385–394.

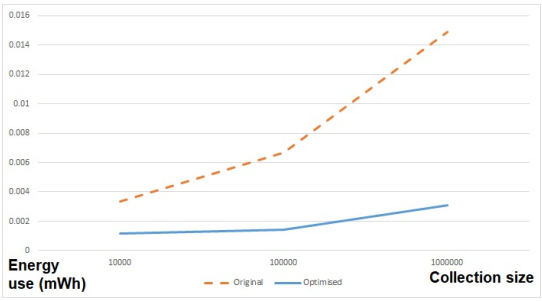

Figure 11: Iterator pattern optimisation: Windows 10, Java

[30] Lavneet Singh. 2022. RMVRVM – A Paradigm for Creating Energy Efficient User Applications Connected to Cloud through REST API. In *Proceedings of the 15th Innovations in Software Engineering Conference* (Gandhinagar, India) *(ISEC '22)*. ACM, New York, NY, USA, Article 6, 11 pages.

[31] Tiago Volpato, Ana Allian, and Elisa Yumi Nakagawa. 2019. Has social sustainability been addressed in software architectures?. In *13th European Conference on Software Architecture* (Paris, France) *(ECSA '19)*. ACM, New York, NY, USA, 245–249.

[32] C-S. Yang, C-C. Huang-Fu, and I-K. Fu. 2022. Carbon-Neutralized Task Scheduling for Green Computing Networks. *arXiv* 2209.02198v1 (2022).

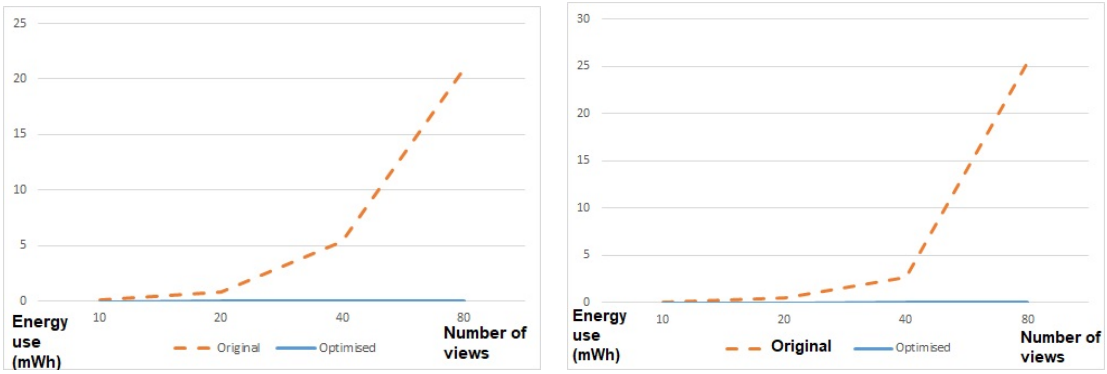

Figure 12: Blackboard style: unoptimised versus optimised versions: (a) Windows 10, Java; (b) Windows 10, C++

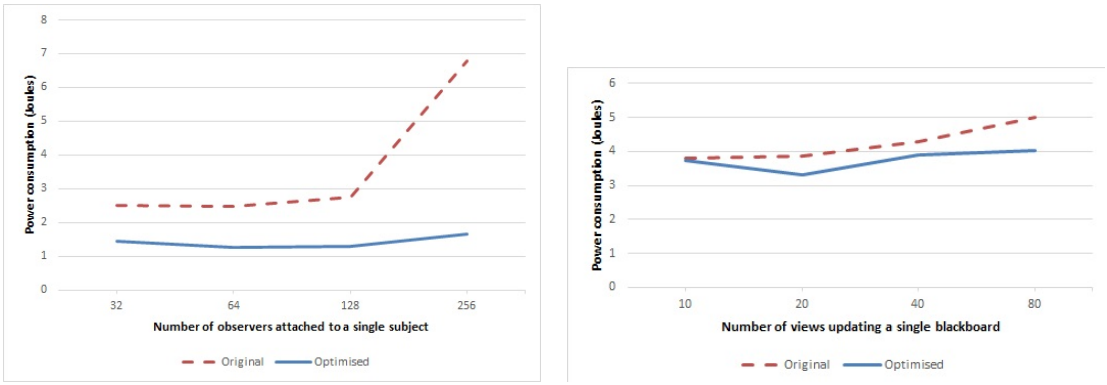

Figure 13: JoularJX Java Linux measurements for (a) Observer and (b) Blackboard