Chapter 5
Patterns and Energy Consumption: Design, Implementation, Studies, and Stories

Daniel Feitosa, Luís Cruz, Rui Abreu, João Paulo Fernandes, Marco Couto, and João Saraiva

Abstract Software patterns are well known to both researchers and practitioners. They emerge from the need to tackle problems that become ever more common in development activities. Thus, it is not surprising that patterns have also been explored as a means to address issues related to energy consumption. In this chapter, we discuss patterns at code and design level and address energy efficiency not only as the main concern of patterns but also as a side effect of patterns that were not originally intended to deal with this problem. We first elaborate on state-of-the-art energy-oriented and general-purpose patterns. Next, we present cases of how patterns appear naturally as part of decisions made in industrial projects. By looking at the two levels of abstraction, we identify recurrent issues and solutions. In addition, we illustrate how patterns take part in a network of interconnected components and address energetic concerns. The reporting and cases discussed in this chapter emphasize the importance of being aware of energy-efficient strategies to make informed decisions, especially when developing sustainable software systems.

D. Feitosa (✉)
University of Groningen, Groningen, The Netherlands
e-mail: d.feitosa@rug.nl

L. Cruz
Delft University of Technology, Delft, The Netherlands
e-mail: l.cruz@tudelft.nl

R. Abreu
Faculty of Engineering, University of Porto & INESC-ID, Porto, Portugal
e-mail: rui@computer.org

J. P. Fernandes
CISUC and University of Coimbra, Coimbra, Portugal
e-mail: jpf@dei.uc.pt

M. Couto · J. Saraiva
HASLab/INESC TEC and University of Minho, Braga, Portugal
e-mail: marco.l.couto@inesctec.pt; jas@di.uminho.pt

© Springer Nature Switzerland AG 2021

C. Calero et al. (eds.), *Software Sustainability*,
https://doi.org/10.1007/978-3-030-69970-3_5

5.1 Introduction

The existence of *patterns* cannot be dissociated from our daily life. We may reason about patterns as concrete observations that are grouped into coherent categories. Patterns help us understand and describe our world. As an example, the evolutionary theory proposed by Charles Darwin was synthesized based on his understanding of patterns emerging from the observations he conducted during his voyage. Patterns can also be found in music, and in this context it has been shown that only a few musical notes sustain the essential melody of landmark music pieces.

Patterns are also well known to both researchers and practitioners in the software development world. In between the various definitions and types of patterns, there is a common understanding that they encapsulate solutions to recurrent problems [1]. A collection of recurrent problems that have become ever more apparent involves energy efficiency, as the growing energy demand associated with ICT usage is already a concern [2]. Notably, energy consumption is an issue with data/computation centers and their massive energy footprint [3], and, nowadays, the ubiquitous use of battery-powered devices such as smartphones [4].

Within ICT, energy consumption is an issue that needs to be addressed not only at hardware and firmware level, but also at the software, or application, level. Indeed, energy efficiency is a multifaceted problem, which encompasses networks, hardware, drivers, operating systems, and applications. In this chapter, we focus on applications and address the problems as systems of forces that can be fully or partially addressed by patterns [1]. In this context, software optimizations have been discussed at source code, design, and architecture level, from which we focus on the first two.

At the code level, we find solutions that are platform-specific and also commonly language-specific, which benefit from being more straightforward to apply. As the scopes open up, design patterns can be language-agnostic and generalizable to a broader range of software domains.

In this chapter, we aim to demonstrate how patterns with various scopes can help build energy-efficient software. Moreover, we discuss patterns that address energy consumption as the main concern (i.e., energy patterns), and patterns that were not initially intended to serve that purpose but have an energy-related side effect. To that end, the subject matter is organized as depicted in Fig. 5.1. In particular, we present energy-oriented code patterns in Sect. 5.2, move on to energy-oriented design patterns in Sect. 5.3, and elaborate the impact of general purpose design patterns on energy efficiency in Sect. 5.4.

Finally, we also illustrate how patterns appear naturally as part of decisions made in industrial projects. Thus, in Sect. 5.5 we present cases from open source projects where energy efficiency issues were factored in and a pattern was applied as part of the solution.

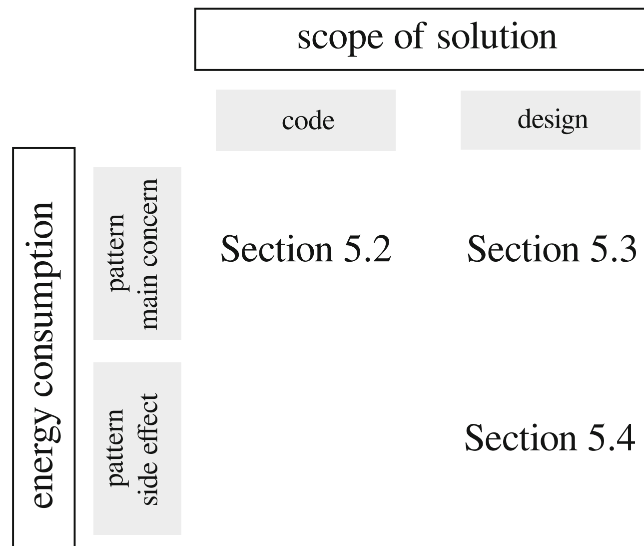

Fig. 5.1 Types of pattern solution addressed in the chapter

5.2 Code-Level Patterns

In this section, we focus on code-level patterns that have been shown to exhibit *greedy* energy consumption behaviors. Identifying patterns at the code level facilitates their transformation into more efficient alternatives, an approach that is widely known as *refactoring*. The potential of code refactorings is maximized when it is possible to automatically realize them, namely using tools that locate code fragments that can be improved and replacing them with the documented alternatives.

The patterns we consider in this section are specific to mobile application development. Mobile devices are these days an essential component of our daily lives, to support both our personal and professional activities. In this context, battery life is one of the principal factors that influence the satisfaction of mobile device users [5], and a recent survey in the US ranked battery life as the most important factor influencing purchasing decisions [6]. Battery life is such a concern that it has been suggested that nine out of ten users suffer from anxiety when their devices are low on battery [7], and this anxiety is under discussion within the Diagnostic and Statistical Manual of Mental Disorders as a potential clinical condition named *nomophobia*, which reflects the fear of not being able to use one's mobile phone [8].

The perception that an application causes excessive battery consumption is actually one of the most common causes for bad app reviews in app stores [9, 10]. This has raised the awareness of mobile application developers regarding the impact their applications have on battery life. In fact, while it has been shown that developers often seek information on how to improve the energy profile of their applications, they rarely receive proper advice [11–13].

Our focus is on code patterns reported as energy greedy within Android. We will refer to these patterns as EGAPs—Energy-Greedy Android Patterns. We synthesize contributions from several works that have documented and validated energy-oriented code refactorings. Specifically, we focus on energy-greedy patterns that

have been automatically refactored in a large-scale empirical study involving 600+ applications [14]. We describe each pattern using the template shown next.[1]

Field	Description
Problem	A recurrent energy efficiency problem where the pattern can be used.
Solution	Generic and reusable solution to the problem.
Example	An illustration of a practical usage of the energy pattern.

The problem and solution for each pattern are tentatively provided by high-level descriptions that we believe can be understood by a broad audience of users and developers. Complementarily, we provide concrete instances of each pattern as snippets whose interpretation is specially oriented toward Android application developers.

5.2.1 Patterns

Below we describe each type of pattern that we considered in [14].

Pattern: Draw Allocation

This pattern is detected by Android *lint*,[2] and is the first of five EGAPs whose energy impact analysis was included in [15, 16].

Problem **Draw Allocation** occurs when new objects are allocated along with draw operations, which are very sensitive to performance. In other words, it is a bad design practice to create objects inside the **onDraw** method of a class which extends a View Android component.

Solution The recommended alternative for this EGAP is to move the allocation of independent objects outside the method, turning it into a static variable.

Example The code snippet to the left should be transformed to the one on the right.

```
public class CMView extends View {
  @Override
  protected void onDraw(Canvas c){
    RectF rectF1 = new RectF(); ✗
    ...
    If(!clockwise) {
      rectF1.set(X2-r, Y2-r, X2+r,
        Y2+r);
      ...
} } }
```

```
public class CMView extends View {
  RectF rectF1 = new RectF(); ✓
  @Override
  protected void onDraw(Canvas c) {
    ...
    if(!clockwise) {
      rectF1.set(X2-r, Y2-r, X2+r,
        Y2+r);
      ...
} } }
```

[1] When the practical usage is obvious, we will exclude the illustrative example.

[2] *Lint* is a code analysis tool, provided by the Android SDK, which reports upon finding issues related to the code structural quality.

Pattern: Wakelock

Wakelock is the second Android *lint* performance issue [15–18].

Problem **Wakelock** occurs whenever a wakelock, a mechanism to control the power state of the device and prevent the screen from turning off, is not properly released, or is used when it is not necessary.

Solution The alternative here would be to simply add a **release** instruction.

Example The code snippet to the left should be transformed to the one on the right.

```
public class DMFSetTempo extends
... {
  PowerManager.WakeLock l;
  public void onClickBtStart(View
v) {
    wakelock.acquire(); ✓
  }
  @Override()
  public void onPause() {
      super.onPause(); ✗ }
  }
```

```
public class DMFSetTempo extends
... {
  PowerManager.WakeLock l;
  public void onClickBtStart(View
v) {
    wakelock.acquire(); ✓
  }
  @Override()
  public void onPause() {
    super.onPause();
    if (l.isHeld()) l.release(); ✓
  }
}
```

There exist other types of wakelocks for resources such as Sensor, Camera, and Media. They differ from the Screen only in the mechanism used to release the lock.

Pattern: Recycle

Recycle is another Android *lint* performance issue [15, 16].

Problem **Recycle** is detected when some collections or database-related objects, such as **TypedArrays** or **Cursors**, are not recycled or closed after being used. When this happens, other objects of the same type cannot efficiently use the same resources.

Solution The alternative in this case would be to include a **close** method call before the method's return.

Example The code snippet to the left should be refactored to the one on the right.

```
public Summoner getSummoner(int id) {
  SQLiteDatabase db =
      this.getReadableDatabase();

Cursor c = db.query( ... );
...
return summoner; ✗
}
```

```
public Summoner getSummoner(int id) {
  SQLiteDatabase db =
      this.getReadableDatabase();

Cursor c = db.query( ... );
...
c.close(); ✓
return summoner;
}
```

Pattern: Obsolete Layout Parameter

The fourth Android *lint* performance issue, **Obsolete Layout Parameter**, is the only one that is not Java-related [15, 16].

Problem The view layouts in Android are specified using **XML**, and they tend to suffer several updates. As a consequence, some parameters that have no effect in the view may still remain in the code, which causes excessive processing at runtime.

Solution The alternative is to parse the **XML** syntax tree and remove these useless parameters.

Example The next snippet shows an example of a view component with parameters that can be removed.

```
<TextView android:id="@+id/centertext"
  android:layout_width="wrap_content" android:
layout_height="wrap_content"
  android:text="remote files"
 ✕layout_centerVertical="true" ✕layout_alignParentRight="true" >
</TextView >
```

Pattern: View Holder

View Holder is the last Android ***lint*** performance issue [15, 16], whose alternative intends to make a smoother scroll in ***List Views***.

Problem The process of drawing all items in a ***List View*** is costly, since they need to be drawn separately.

Solution To reuse data from already drawn items, therefore reducing the number of calls to **findViewById()**, known to be energy greedy [19].

Example Every time **getView()** is called, the system searches on all the view components for both the **TextView** with the id "label" (❶) and the **ImageView** with the id "logo" (❷), using the energy-greedy method **findViewById()**. The alternative version is to cache the desired view components, with the following approach:

```
public View getView(int p, View v, ViewGroup par) {
  LayoutInflater inflater = ...

  v = inflater.inflate(R.layout.apps , par, false);
 TextView txt=(TextView) v.findViewById(R.id.label);  ❶
 ImageView img=(ImageView) v.findViewById(R.id.logo);  ❷
 return row;
}
```

```
static class HolderItem {
  TextView txtView; ImageView imgView;
}
public View getView(int p, View v, ViewGroup par) {
  HolderItem hld; LayoutInflater inflater = ...

 if (v == null) {              ❸
  v = inflater.inflate(...); hld = new HolderItem();
 hld.txtView = (TextView) v.findViewById (...);     ❹
 hld.imgView = (ImageView) v.findViewById (...);     ❺
 v.setTag(hld);
 } else { hld = (HolderItem) v.getTag(); } »     ❻
 TextView txt = hld.txtView; ImageView img = hld.imgView;
 ...
}
```

Condition ❸ evaluates to true only once, which means instructions ❹ and ❺ execute once, i.e., **findViewById()** executes twice, and its results are stored in the **ViewHolderItem** instance. The following calls to **getView()** will use cached values for the view components **txt** and **img** (❻).

Pattern: HashMap Usage

This EGAP is related to the usage of the **HashMap** collection [17, 20–22].

Problem The usage of **HashMap** is discouraged, since the alternative **ArrayMap** is allegedly more energy efficient, without decreasing performance.[3]

Solution To simply replace the type **HashMap**, whenever used, by **ArrayMap**.

Pattern: Excessive Method Calls

Unnecessarily calling a method can penalize performance, since a call usually involves pushing arguments to the call stack, storing the return value in the appropriate processor's register, and cleaning the stack afterwards.

Problem **Excessive Method Calls** was explored by [20, 23], showing that the energy consumption in Android applications can be decreased by removing method calls inside loops that can be extracted from them.

Solution The alternative is to replace the method call by a variable that is declared outside the loop, and is initialized with the return value of the method call extracted.

[3] As stated in the Android *ArrayMap* documentation: http://bit.ly/32hK0y9.

Example An example of an extractable method call would be one which receives no arguments, and is accessed by an object that is not transformed in any way inside the loop.

Pattern: Member Ignoring Method

This EGAP addresses the issue of having a non-static method inside a class, and which could be static instead [17, 20].

Problem Having a method not declared as static, but which does not access any class fields, does not directly invoke non-static methods, and is not an overriding method. This causes multiple instances of the method to be created and used at runtime, which can be avoided.

Solution Use static methods as these are stored in a memory block separated from where objects are stored, and no matter how many class instances are created throughout the program's execution, only an instance of such a method will be created and used. This mechanism helps in reducing energy consumption.

5.3 Energy Design Patterns

In the previous section, we learned about code patterns that are specific to a given platform (i.e., Android) or paradigm (i.e., object oriented) and how they affect energy consumption. In this section, we bring these patterns to a higher level of abstraction: we delve into design patterns that provide reusable solutions that generalize to any software of a given domain and that are not coupled with any particular development framework or paradigm.

The energy patterns in this section do not give any particular advice on coding practices. Rather, they help software engineers create energy-efficient software by design. Nevertheless, these patterns may have a direct impact on the feature set of the application and ultimately on the user experience.

In this particular case, we focus on energy patterns in the mobile domain. We present a catalog of 22 energy patterns that are commonly used for mobile applications. This catalog is the result of an empirical study with more than 1700 mobile applications [24] to document energy patterns that are commonly adopted by iOS and Android software engineers and are expected to generalize to any mobile platform.

We describe each energy pattern with the template used in the previous section, explaining the problem and the solution while providing an illustrative example.

5.3.1 Patterns

Below we pinpoint different design patterns to develop energy-efficient mobile applications.

Pattern: Dark UI Colors

Provide a dark UI color theme to save battery on devices with AMOLED[4] screens [25–28].

Problem One of the major sources of energy consumption in mobile devices comes from the screen. Thus, mobile applications that rely on the screen for all use cases, such as video apps or reading apps, can significantly drain the battery.

Solution Opt for dark colors when designing the UI. Smartphones typically feature screens that are more energy efficient with dark colors. Depending on the application, users can be given the option to choose between a light and a dark UI theme. Alternatively, a special trigger (e.g., when battery is running low) can activate the dark UI theme.

Example In a reading app, provide a theme with a dark background using a light foreground color to display text. When compared to themes using light background colors, a dark background will have a higher number of dark pixels.

Pattern: Dynamic Retry Delay

When trying to access a resource that is failing or not responding, increase the waiting time before attempting a new access.

Problem Mobile apps often need to exchange data with different resources (e.g., connect to a server in the cloud). It may happen that the communication with these resources fails and a new attempt needs to be made. However, if the resource is temporary, the app will repeatedly try to connect to the resource with no success, leading to unwanted energy consumption.

Solution After each failed connection, increase the waiting time before the next attempt. A linear or exponential growth can be used for the waiting interval. Upon a successful connection or a given change in the context (e.g., network status), the waiting time can be set back to the original value.

[4]AMOLED is a display technology used in mobile devices and stands for Active Matrix Organic Light Emitting Diodes.

Example Consider the scenario in which an app with a news feed is not able to communicate with the server to retrieve updates. The naive approach is to continuously poll the server until the connection is successful—i.e., the server is available. Instead, a dynamic retry delay can be used by, for example, adopting the Fibonacci series[5] to increase the time between sequential attempts.

Pattern: Avoid Extraneous Work

Avoid tasks in the mobile application that do not add enough value to the user experience or whose results quickly become obsolete.

Problem Typically, mobile applications execute multiple tasks at the same time. However, there are cases in which the results of these tasks are not immediately presented to the user. For example, when the application is synchronizing real-time data that does not immediately meet the information needs of the user, it may become obsolete before the user actually accesses it.

This is even more evident when apps are running in the background. The phone will be using resources unnecessarily to update data that will never be used.

Solution Define the minimal set of data that is presented to the users. In addition, disable all the tasks that are not affecting the data being displayed to the user.

Example Consider a plot with a time series of real-time data that is being continuously updated. When the user scrolls up/down, the plot might move out of the visible area of the UI. In this case, updating the plot is a waste of energy. Drawing operations related to updating the plot should be ceased and restarted when the plot is visible again.

Pattern: Race-to-idle

Resources or services ought to be closed as soon as possible (e.g., location sensors, wakelocks, screen) [15, 29–31].

Problem Mobile apps resort to different resources and components that need to be stopped after being used. After activating a given resource, it starts operating and is ready to respond to the app’s requests. Even if the app is not making any request, the resource will waste energy until it is properly closed.

Solution Make sure resources are inactive when they are not necessary by manually closing them. Static analysis tools may help identify cases of resources that are not being properly closed—e.g., *Facebook Infer*, *Leafactor* [16].

[5]Fibonacci series is a sequence of numbers in which each number is the sum of the two preceding numbers (i.e., 1, 1, 2, 3, 5, 8, etc.).

Example Wakelocks are commonly used by mobile applications to prevent phones from entering sleep mode. Different types of wakelocks can be used; for example, there are wakelocks specific for the screen, CPU, and so on. Always implement event handlers that listen to the application events of the entering or leaving background. Implement handlers for the events that are fired when the app goes to background, and release wakelocks accordingly.

Pattern: Open Only When Necessary

Open/start resources/services immediately before they are required. This is similar to the pattern ***Race-to-idle***.

Problem Resources, such as location sensors or database connections, must be activated before they are ready to use. Once a given resources is opened, it actively consumes more energy. Thus, it should only be opened immediately before its usage. In particular, resources should not be activated upon the creation of the view or activity where it operates.

Solution Activate resources and services immediately before they are needed. This will also prevent the activation of resources that are never used [29].

Example In a video call app, the camera is used to share the faces or images of the different participants in the call. The camera should only start capturing video when it is actually being displayed in the view to the user.[6]

Pattern: Push over Poll

Using push notifications is more energy efficient than actively polling for notifications.

Problem Mobile apps typically resort to notifications to get updates from resources (e.g., from a server). The naive approach to getting updates is by reaching the resource and asking it for updates. The downside is that, by continuously asking a server for updates, it might be making several requests without any update. This leads to unnecessary energy consumption.

Solution Use push notifications to get updates. Note—for Free and Open Source applications this is a big challenge because it requires having a cloud messaging

[6]A real example where the camera was being initiated too early can be found here: https://github.com/signalapp/Signal-Android/commit/cb9f225f5962d399f48b65d5f855e11f146c bbcb (visited on June 15, 2020).

server set-up. For example, in the case of Android there is no good open source alternative to Google's Firebase Cloud Messaging.

Example In a social network app, instead of actively reaching the server to provide relevant notifications to the user, the app should prescribe push notifications.

Pattern: Power Save Mode

Implement an alternative execution mode in which some features are dropped to ensure energy efficiency. In some cases, user experience is hindered.

Problem When the battery level is low, users may want to make sure they will not lose connectivity before reaching a power station and charging their phone.

Solution Implement a power save mode that only provides the minimum functionality that is essential to the user. This mode can be manually activated by the user or through power events (e.g., when battery reaches a given level) raised by the operative system. In some cases, the mobile platform already features this out of the box—e.g., this is enforced in iOS for use cases using the **BackgroundSync** APIs.

Example Reduce update intervals, disable less important features, or disable UI animations.

Pattern: Power Awareness

Features operate in a different way regarding the battery level or depending on whether the device is connected to a power station.

Problem There are some features that, despite improving user experience, are not strictly necessary for users—e.g., UI animations. Moreover, there are low-priority operations that do not need to be executed immediately (e.g., backup data in the cloud).

Solution Adjust the feature set according to its power status. Even when the device is being charged, the battery level may be low and it is better to wait for a higher battery level before executing any intensive task.

Example Postpone intensive tasks, such as cloud syncing or image processing, until the device reaches a satisfactory power level, typically above 20%.

Pattern: Reduce Size

Minimize the size of data being transferred to the server.

Problem Mobile apps typically transfer data with servers over an internet connection. Such operations are battery intensive and should be reduced to a minimum. There are cases in which the size of the data can be reduced without affecting user experience.

Solution Exclusively transmit data that is strictly necessary and compression techniques whenever possible.

Example Enable gzip content encoding when sending data over HTTP requests.

Pattern: WiFi over Cellular

Postpone features that require a heavy data connection until a WiFi network is available.

Problem Mobile apps typically need to synchronize data with a server. However, cellular data connections (e.g., 4G) tend to be energy greedy.

Solution WiFi connections are usually a more energy-efficient alternative to cellular connections [32]. These are use cases that do not require real-time sync and should be postponed until a WiFi connection is available.

Example Consider a music stream application that allows users to play their favorite songs and to organize them in playlists. In addition, the app allows users to play the playlists offline—i.e., when there is no internet connection. When a new song is added to a given offline playlist, the app waits for a WiFi connection before downloading the song.

Pattern: Suppress Logs

Avoid intensive logging as much as possible. Overusing logging leads to significant energy consumption, as found in previous work [33].

Problem Logging is commonly used to simplify debugging. However, there is a trade-off between having the necessary information and energy efficiency that needs to be considered.

Solution Manage logging rates to a maximum of one message per second.

Example In a mobile app that is processing real-time data, avoid logging this behavior. If necessary, enable logging only for debugging executions.

Pattern: Batch Operations

Bundle multiple operations instead of running them separately. This will avoid putting the device into an active state many times in the same time window.

Problem Executing operations separately leads to extraneous energy consumption related to turning a particular resource on and off—this is typically called *tail energy consumption* [23, 34, 35]. Executing a task often induces tail energy consumption related to starting and stopping resources (e.g., starting a cellular connection).

Solution Combine multiple operations in a single one to optimize tail energy consumption. Although background tasks can be expensive, very often they have flexible time constraints. For example, a given background task that needs to be executed eventually does not need to be executed at a specific time. Thus, it can wait for other operations to be scheduled before it is executed.

Example Use Operative System-wide APIs tailored for job scheduling (e.g., 'android.app.job.JobScheduler,' 'Firebase JobDispatcher'). These APIs manage multiple background tasks occurring in a device to guarantee that the device will exit sleep mode (or doze mode) only when the tasks in the waiting list really need to be executed.

Pattern: Cache

This pattern proposes the use of cache mechanisms to avoid unnecessary operations.

Problem A common functionality in mobile apps is to display data fetched from a remote server. A potential issue with that need is that an app may fetch the same data from the server multiple times during the lifetime of the mobile app.

Solution Mobile apps should put in place caching mechanisms to avoid fetching data from the server [36]. Moreover, lightweight strategies to decide whether to refresh the data in the cache need to be implemented to guarantee that the mobile app is displaying the up-to-date data.

Example Consider a social network app that displays profiles of other users. Instead of downloading basic information and profile pictures every time a given profile is opened, the app can use data that was locally stored from earlier visits.

Pattern: Decrease Rate

This pattern proposes to increase the time between syncs/sensor reads as needed.

Problem It is common for mobile apps to perform certain operations periodically. A potential issue is that, if the time between two executions is small, the app will be executing operations more often.

Solution Increase the time-between-operations to find the minimal time interval that would compromise user experience, while having a positive impact on the energy consumption. This time-between-operations can be manually tuned by developers, defined by users, or even found in an empirical way. One could also envisage more sophisticated and dynamic solutions that can also use context (e.g., time of day, history data) to infer the optimal update rate.

Example Consider a news app that gathers news from different sources, doing so by fetching the news of a given source in its own thread. Instead of triggering updates for all threads at the same rate, use data from previous updates to infer the optimal update rate of these threads. Connect to the news source only if updates are expected.

Pattern: User Knows Best

This pattern proposes to offer capabilities to allow users to enable/disable certain features to save energy.

Problem The number of features offered by a mobile app and power consumption is a trade-off generally considered when devising energy-efficient solutions. However, there is no **one–size-fits-all** user as far as this trade-off is concerned. There are users who might be satisfied with fewer features but better energy efficiency, and vice versa.

Solution The possibility for users to customize their preferences regarding energy-critical features is therefore important. This customization should be intuitive and an optimal default set of preferences.

Example Consider a mail client for POP3 accounts as an example. One can imagine that a user may want their mail client to check/poll for new messages every other minute, and others—depending on the time of day—much less often. As there is no automatic mechanism to infer the optimal update interval, the best option is to allow users to define it.

Pattern: Inform Users

This pattern proposes to inform the user when the app is performing any battery-intensive operation.

Problem It is known that there are use cases in mobile apps that require a substantial amount of energy. In turn, one can activate features to be energy efficient at the cost of user experience. We argue that if users do not know the expected behavior from the mobile app, they may flag its operation as failing.

Solution Inform users about battery-intensive operations or energy management features. This could be done by flagging (e.g., via alerts) this information in the user interface.

Example Alert users when (1) a power-saving mode is active or (2) a battery-intensive operation is being executed.

Pattern: Enough Resolution

This pattern proposes that data should be pulled or provided with high accuracy only when strictly necessary.

Problem Users tend to use precise data points when fetching and/or displaying data. An issue with such a strategy is that the collection and manipulation requires more resources, entailing, naturally, high-energy consumption. There are, however, use cases where dealing with low-resolution data suffices.

Solution Developers should find the trade-off between data resolution and app/user needs as well as user experience.

Example Take as an example a running app that is able to record running sessions. The app shows the user the current overall distance to a given location. Instead of using precise real-time processing of GPS or accelerometer sensors, which can be energy greedy, a lightweight method could be used to estimate this information with lower but reasonable accuracy. Evidently, at the end of the session, the accurate results would still be processed, but without real-time constraints.

Pattern: Sensor Fusion

This pattern proposes using data from low-power sensors to decide whether to fetch data from high-power sensors.

Problem Operations to interact with distinct sensors or components may be energy greedy, causing the app to consume a substantial amount of energy. Therefore, such operations should be executed only in case of absolute necessity.

Solution Making use of data sources that entail low power consumption (such as alternative low-power sensors) may prevent the need to execute an energy-greedy operation.

Example As an example, one can imagine using the accelerometer to infer whether the user has changed location, and only interacting with the energyintensive GPS to obtain a more precise location in case of a location change.

Pattern: Kill Abnormal Tasks

This pattern proposes to offer capabilities to interrupt energy-greedy operations (e.g., using timeouts, or users input).

Problem Mobile apps may trigger an operation that unexpectedly consumes more energy than anticipated (e.g., taking a long time to execute).

Solution Offering an intuitive way for end users to interrupt an energy-greedy operation would help to fix this issue. Alternatively, a fair timeout could be included for energy-greedy tasks or wakelocks.

Example As an example, consider a mobile app that features an alarm clock. Implementing a fair timeout for the duration of the alarm, in case the user is not able to turn it off, will prevent the battery from being drained.

Pattern: No Screen Interaction

This pattern proposes to allow interaction without using the display whenever possible.

Problem There are mobile apps that involve constant use of the screen. However, there may be cases in which the screen can be replaced by less power-intensive alternatives.

Solution Enable users to use alternate interfaces (e.g. audio) to communicate with the app.

Example As an example, consider a navigation app. There are use cases in which users may be using audio instructions only, having no need the see updates on the screen. This strategy is commonly adopted by audio players that use the earphone buttons to play/pause or skip songs.

Pattern: Avoid Extraneous Graphics and Animations

Graphics and animations are at the forefront as far as improving the user experience is concerned, but can also be battery intensive. Therefore, this pattern proposes to use them with care [37]. This is well aligned with what is recommended in the official documentation for iOS developers.[7]

Problem Mobile apps often feature impressive visual effects. However, they need to be properly tuned to prevent the battery from being drained quickly. This has been shown to be particularly critical in e-paper devices.

Solution Study the importance and impact of visual effects (such as graphics and animations) to the user experience. The improvement in user experience may not be sufficient to overcome the overhead imposed on the energy consumption. Therefore, developers should consider avoiding using visual effects or high-quality graphics, and should instead resort to low frame rates for animations when viable and/or feasible.

Example For instance, high frame rates may make sense while playing a game, but a lower frame rate may suffice while in the menu screens. In other words, use a high frame rate only when the user experience requires it.

Pattern: Manual Sync, On Demand

This pattern proposes to execute tasks if, and only if, requested by the user.

Problem Some tasks may be energy intensive, but not really needed to give the best user experience of the app. Hence, they could be avoided.

Solution Providing a mechanism in the UI (e.g., button) which allows users to trigger energy-intensive tasks would be helpful in letting the user decide which tasks he wants to trade off for energy consumption.

[7] *Energy Efficiency Guide for iOS* Apps—Avoid Extraneous Graphics and Animations available here: https://developer.apple.com/library/archive/documentation/Performance/Conceptual/EnergyGuide-iOS/AvoidExtraneousGraphicsAndAnimations.html (visited on June 15, 2020).

Example Take as an example a beacon monitoring app. There may be situations in which the user does not need to keep track of her/his beacons. This app could implement a mechanism to let the user (manually) start and stop monitoring.

5.4 Object-Oriented Patterns

In this section, we focus on patterns that are tailored to a certain programming paradigm. In particular, we discuss the Gang of Four (GoF) patterns, a popular catalog of object-oriented (OO) design patterns proposed by Gamma, Helm, Johnson, and Vlissides [38] that describe recurring solutions to common OO problems. Although these patterns do not primarily target energy efficiency, they do have an impact that ought to be considered when designing sustainable systems. To refresh the reader's mind, we present two such patterns.

Pattern: Template Method

An algorithm must accommodate custom steps while maintaining the same overall structure [38].

Problem Software systems oftentimes implement behaviors that are similar, containing only a couple of steps that differ. Maintaining the code for each behavior independently incurs greater effort. Moreover, there is a risk that patches will not be applied uniformly among similar instances, which may unnecessarily (and potentially erroneously) diverge the designs.

Solution The overarching steps among all behaviors should be implemented in a single component. The steps that are implemented differently between the behaviors are accessed via an interface. The individual behaviors must now inherit the general component and only implement the interfaced steps.

Example A library implements several supervised learning classification algorithms. The steps to create and use such an algorithm are similar, e.g., configure model, define features and response variables, train model, and predict new values. In this scenario, template methods can be used on steps such as train and predict, while centralizing the implementation of the overall classification task.

Pattern: State

A single component may alter its states with different behaviors as if the component had been replaced [38].

Problem One or more behaviors of a component depend on a state that is only identifiable at runtime. Although the state is mutable, the set of possible states and the different ways behavior is implemented are well defined.

Solution The component consists of an interface accessible to other components (i.e., clients). Each state implements the interface. The state of the component is reassessed internally upon the execution of an implemented behavior.

Example A sensor component offers the behaviors *read_data*, *turn_on*, *turn_off*, and *get_state*, which are implemented for the states **enabled**, **disabled**, and **defective**. Upon an unsuccessful read in the **enabled** state, the component changes its state to **defective**. Otherwise, the state is defined via *turn_on* and *turn_off*.

The GoF patterns can be grouped according to the purpose they serve, i.e., to create objects, to organize structure, or to orchestrate behavior. A pattern instance comprises the association of one or more classes and interfaces fulfilling the various roles described by the pattern. For example, the instance of a State pattern comprises an interface that is implemented on a set of state classes that can provide a different behavior for the predefined actions, which are in turn accessed by a context (client) class.

As the reader may already know or have noticed by now, the GoF patterns do not address energy problems by intent. However, design pattern instances (like any design) have effects on quality attributes. Moreover, the instantiations of a design pattern are not uniform, nor are their effects on quality attributes [39]. In particular, several studies suggest that the effect of a pattern on a quality attribute depends on factors such as the number of classes, invoked methods, and polymorphic methods [39–41].

Considering the systematic use of OO features (e.g., polymorphism) in pattern instances, one may expect a potential impact (positive or negative) on energy consumption. Furthermore, researchers consistently find that, at least on Java systems, approximately 30% of the classes participate in one or more instances of GoF patterns [42–44]. This picture adds up to a growing concern and interest in the research community. In this context, if a pattern instance is not the optimal design solution, an alternative (non-pattern) design solution can be applied. Several authors (including GoF design pattern advocates) have proposed such alternatives [38, 45–49].

In efforts to investigate the aforementioned effect, Litke et al. [50] studied the energy consumption of five design patterns[8] through six toy examples and were able to detect a negligible consumption overhead for the Factory Method and Adapter pattern. Sahin et al. [51] investigated 15 design patterns[9]; however, there were some inconclusive results, as they could observe both an increase and a decrease in energy consumption. To shed further light on the matter, Noureddine and Rajan [52] examined in detail two design patterns for which they identified a significant overhead, namely Observer and Decorator patterns. The comparison involved not only pattern and alternative non-pattern solutions but also a transformed pattern solution that optimizes the number of object creations and method calls. Although the pattern solution showed overheads between 15% and 30%, the optimized solution reduced these observations by up to 25%.

The preceding work shows that there is indeed a potential systematic effect of GoF patterns in energy consumption and that negative effects may be countered on certain cases. Such knowledge is relevant for both greenfield projects (i.e., fresh development), where it can support an energy-smart application of patterns, and brownfield projects (e.g., refactoring of a system to a new purpose), where it can inform decisions on what parts of the system to refactor. However, to fulfill these goals, more insights and guidelines are necessary to fully understand what influences the energy consumption of GoF patterns.

To that end, one of the authors was the lead researcher in a study to investigate the effect of Template Method and State/Strategy patterns on energy consumption [53]. In particular, an experiment was set up to compare the energy consumption of pattern and alternative (non-pattern) solutions and, more importantly, to examine factors that influenced the observed results. To improve accuracy, the energy measurements were collected at both system and method level. The energy efficiency of pattern instances was analyzed at the method level, from which both the size (measured in source lines of code—SLOC) and the number of foreign calls (measured via the message passing coupling metric—MPC[10]) were assessed.

The results of the study showed that the non-pattern solutions consume less energy than their pattern counterpart. However, as in other studies, there were cases in which the pattern solution had a similar or marginally lower energy consumption. One of the main contributions of this work is the investigation of the related factors. Upon examining the SLOC and MPC metrics, it was possible to establish that instances of GoF patterns tend to provide an equitable or more energy-efficient solution when used to implement logic with longer methods and multiple calls to external classes, i.e., complex behaviors. These findings are illustrated in Fig. 5.2, which compares the energy consumption of pattern (y-axis, left chart) and non-pattern (x-axis, left chart) solutions for all assessed methods. These data points

[8] Factory Method, Adapter, Observer, Bridge, and Composite.

[9] Abstract Factory, Bridge, Builder, Command, Composite, Decorator, Factory Method, Flyweight, Mediator Observer, Prototype, Proxy, Singleton, Strategy, and Visitor.

[10] Number of invocations to methods that are not owned or inherited by the class being measured.

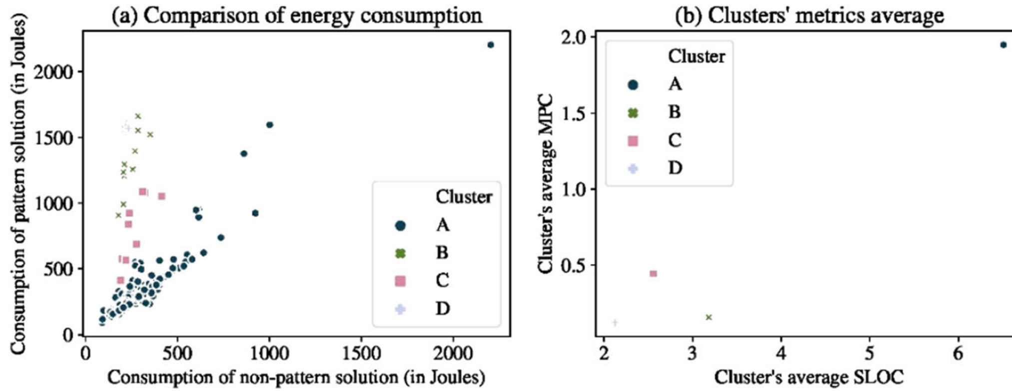

Fig. 5.2 Comparison of energy consumption and associated factors

are clustered by energy efficiency (distinguished by shape and color) and the average SLOC and MPC of each cluster are depicted in the right-hand chart.

These findings serve to reiterate and discuss a set of recurring concerns around the use of GoF patterns. First, they should only be applied if the extra (design) complexity that they introduce is lower than the one that they resolve. In other words, if the context or logical complexity is trivial, the design solution should also be trivial. Otherwise, quality attributes, including energy efficiency, are likely to deteriorate [40, 41]. For example, longer methods reduce the ratio between localization time of the overall computation (i.e., logic) and thus also the overall overhead caused by the polymorphic mechanism.

Finally, note that as patterns promote improved structuring of the source code, energy efficiency may also be achieved through more efficient bytecode. For example, we observed that the Java Virtual Machine applies internal optimizations when pattern-related methods comprise a set of external invocations (i.e., to methods that are not owned or inherited by the pattern class). Such optimizations might not be triggered in a non-pattern alternative, as the structure is altered.

5.5 Patterns in Context

In this section, we present a series of cases describing situations in which patterns can help improve the energy consumption of software-intensive systems. These cases were extracted from real projects or created based on scenarios that practitioners may regularly encounter. As the cases comprise the application of patterns, we resort to a well-known template for capturing design decisions related to patterns described by Harrison et al. [54]. Each case is described according to the fields presented in Table 5.1. We clarify that there are additional fields available in the template by Harrison et al., e.g., related patterns and related requirements. However, we restricted our analyses to the parts of the systems on which we report, and thus we do not establish links between decisions within a project.

Table 5.1 Template for documenting pattern-related decisions

Field	Description
Context	Scenario (incl. constraints) in which the pattern is (or would be) applied.
Problem	Stakeholders' concern that must be addressed.
Alternatives	Alternatives (according to forces) that have been considered to tackle the issue.
Solution	Generic solution (provided by the pattern) to the design problem.
Rationale	Rationale of applying the pattern's solution in relation to the forces.
Pattern	Pattern name.
Consequences	Context and implications of applying the pattern.
Notes	Relevant points that do not fit in another field.
Source	Origin of the case, or description of the fictional context.

Case: Android Token

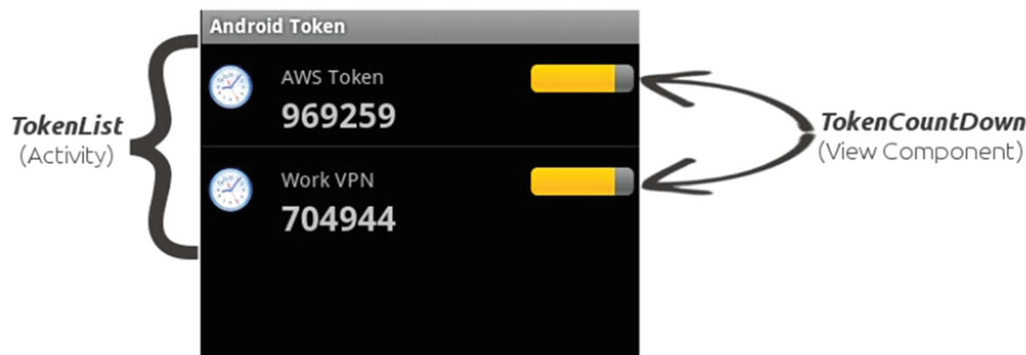

Fig. 5.3 The main activity of *Android Token*

Android Token is an application suited for generating and managing One-Time Password (OTP) tokens, to be used in software requiring Open Authentication (OATH). It is completely free and open source, and is available in the F-Droid application catalog.

Context The main purpose of this application is to provide information regarding the properties of the generated tokens, such as their value, where are they being used, and how much time is left until the token expires. As such, the application's main view (which is managed by the main *Activity*, depicted in Fig. 5.3) shows a list of all tokens, with all of the aforementioned properties displayed. Since the information per token is the same, it is expected that there will be several identical view components displayed (such as labels or progress bars).

Problem Drawing the same type of view components for each token means repeating almost the exact same task, but with different values. Once an application is created, the Android system puts all the metadata of all view components within the application inside the same wrapper class. Each *Activity* is then responsible for fetching the required components to be drawn in their associated layouts. The fetching process is available in Android only through an API call already known to be energy greedy [19]. Moreover, due to how Android internally handles the

process of swapping between activities, moving to a new activity or going back to a previously visited one means redrawing all the components. As expected, this has a huge impact on the amount of work performed by both the CPU and the GPU.

Essentially, this problem creates two optimization challenges. The first one is the excessive number of component fetching and redrawing tasks, which should be reduced. Second, since the actual component's *draw* operation itself is repeated several times, it should be focused only on the component-drawing process, and not on tasks such as setting up of any kind, or creating objects.

Solution For the first problem, the solution requires a caching strategy, to avoid unnecessary fetching, and to optimize the redrawing process. Therefore, the *Activity* responsible for fetching and drawing the view components should internally keep a copied reference of each one, collected the first time they are drawn.

The second problem can be tackled by reducing to a minimum the number of instructions not related to the drawing process. As such, creating new objects should be avoided in the *onDraw* method of a view component, as described in the Android documentation.[11]

Rationale Caching view components means reducing the effort required by the CPU to traverse through (potentially) all existing components, and avoiding unnecessary calls to an energy-greedy Android API. It also means reducing the effort required by the GPU to redraw the same components. Avoiding object allocation inside the *onDraw* method is also a CPU effort reduction optimization, since many objects require an expensive initialization procedure.

Ultimately, reducing the effort on these tasks translates to reducing the energy consumed by the application, and consequently increasing the device's battery uptime.

Pattern The patterns that provide the solution to the aforementioned problems are commonly known as *ViewHolder* and *DrawAllocation*, respectively.

Consequences Implementing both the patterns has a significant impact on code readability and maintainability, especially for *ViewHolder*. It requires including an inner class inside the *Activity* to hold the view components, and to increase the complexity of the fetching/drawing method. As for *DrawAllocation*, developers should preallocate objects (by using class variables), which, depending on the type of object, may require additional effort and reduce the code readability. When applying both patterns on an existing application, it also means restructuring code, critical to the application, with a new concept, which can be a delicate and costly task.

[11] Android View documentation: https://developer.android.com/training/custom-views/custom-drawing#createobject

Case: Nextcloud Android app

Nextcloud is a file hosting service client-server solution for file hosting services. Anyone can install it on their own private server. It is distributed under the General Public License v2.0 open-source license, which also means that anyone can contribute to the project. It provides a software suite with a cloud server and client apps for different desktop and mobile platforms. In this particular case, we are looking at their Android app.

Context As in most mobile apps for cloud services, data exchanging is a recurrent task in their feature set. In the case of Nextcloud, all the files need to be synchronized with the different user devices. Thus, whenever a new file is added or updated, it needs to be uploaded to the cloud server.

Problem Uploading files is a resource-intensive task that may take a few minutes to execute. This may considerably reduce battery level. However, there are cases in which the user is not so interested in having all the files immediately uploaded to the server. Depending on the user context, the trade-off between file consistency and battery level may be different.

Solution Allow the user to define when the app should prioritize energy efficiency above other features. Typically, mobile operating systems already provide a power save mode that can be activated manually or when the battery reaches a critical level (e.g., 20% of full capacity). All the apps have access to this setting and can change their behavior accordingly. In the example of Nextcloud, developers decided to deactivate any file upload during this mode.

Rationale The power save mode is a deliberate user action that expresses that the user is prioritizing battery life above other features. Thus, it is important that energy-intensive features, such as file transfers, are avoided.

Pattern Power Save Mode.

Alternatives The patterns Inform Users (i.e., warn users of energy-intensive actions) and Power Awareness (e.g., change behavior according to the battery level) can also be used in this context.

Consequences This strategy can have a big impact on the user experience. It is important that users understand that during this mode their files are not going to be uploaded to the server. Thus, this behavior should be properly flagged in the user interface, so that users are well informed of it. In this particular case, the Nextcloud app allows users to override the *Power Save Mode* behavior by clicking on a button that manually triggers a synchronization with the server. Finally, some studies have found evidence that, when not coded properly, this pattern may hinder the maintainability of the project.

Notes This pattern is usually supported by any modern mobile operating system. It is always a good practice to implement this pattern in a mobile app.

Source This case is reported in the Nextcloud app's GitHub project: https://github.com/nextcloud/android/commit/8bc432027e0d33e8043cf401922

Case: K-9 Mail

K9-Mail is a free and open-source e-mail client for Android. It was first written in 2008 and it is still under active development, being one of the oldest Android apps. Like any mobile application, K-9 Mail runs under limited energy resources. Battery life needs to be optimized to prevent hindering user experience. Thus, along the history of its project, we encounter a number of code changes that were made to improve energy efficiency.

Context An important activity done by an e-mail client app is synchronizing data and communicating with e-mail providers. For example, when new emails appear in the user's inbox, the app needs to communicate with the server and download this new data.

Problem Servers do not always work as intended. There are many reasons for servers being unreachable: slow or no internet connection, too many users accessing the server, server is down for maintenance, and so on. This means that the features requiring server communication will fail until the required server can be reached again. Typically, the communication can be established after a few unsuccessful attempts. Thus, it is common that for asynchronous tasks the app will try the communication again after some delay. However, in some cases the server may be unreachable for hours or days. This means that the app will silently be draining the battery while continuously attempting to establish a connection with the server. Debugging this behavior is not trivial since the app will not necessarily fail but the task keeps running in the background. In this particular case, K9-Mail is trying to communicate with the server to set up the synchronization mechanism IMAP IDLE protocol.[12]

Solution The typical fix for this situation is creating a threshold for the maximum number of times a communication can fail. After this defined threshold, the app should permanently stop trying to reach the server. In addition, it is a good practice to increase the delay between attempts. For example, while the initial attempts can be made within a few seconds, the following delays should be subsequently increased.

Rationale Often when a server is not reachable within seconds, it is due to a more severe communication problem. Thus, it is unwise to continuously attempt new

[12]IMAP IDLE is a feature defined by the standard RFC 2177 that allows a client to indicate to the server that it is ready to accept real-time notifications.

connections. It is better to kill the task and wait for the user to trigger a new attempt later. This approach gives more control to the user to define whether (1) the task is indeed critical and battery life is not so important or (2) the other way around.

Pattern This pattern is commonly known as *Dynamic Retry Delay*.

Alternatives Alternatives (according to forces) that have been considered to tackle the issue.

Consequences The main consequence of this approach is that new code needs to be added to accomplish this behavior. It is always a good practice to use existing APIs to schedule this kind of task in the background.

Notes The same problem can be found in other features of a mobile app, for example syncing with a wearable device, getting location data, and accessing.

Source This issue was found by K-9 Mail developers and their solution can be found on GitHub: https://github.com/k9mail/k-9/commit/86f3b28f79509d1a4d

Case: WebAssembly design

WebAssembly is an assembly-like language that can be executed in modern web browsers.[13] With this context in focus, the language was designed to produce a compact binary that can be executed with near-native performance, i.e., comparable to binaries compiled for native platforms (e.g., x86, ARM).[14]

The WebAssembly project has a repository dedicated to its design[15] and the bug tracking system is used to discuss issues related to it. Among the discussed issues are matters related to energy efficiency.

Context The WebAssembly group aims at providing a Just-in-Time (JIT) interface part of its specification.[16] However, the level of detail provided in the specification dictates the level of flexibility that library implementations would have. For example, depending on the level of detail in the specification, a library could allow for more undefined behaviors, e.g., at what moment a function definition is evaluated and how deep the checking goes.

Problem The specification of the moment in which a function is evaluated also requires the specification of when errors are reported. This concern was brought up and discussed in an issue opened on the aforementioned GitHub repository.[17] In

[13] https://developer.mozilla.org/en-US/docs/WebAssembly
[14] https://webassembly.org/
[15] https://github.com/WebAssembly/design
[16] https://webassembly.org/docs/jit-library/
[17] https://github.com/WebAssembly/design/pull/719

short, developers argued about the proper moment for a JIT compiler to flag a malformed or not fully implemented feature (e.g., function or module) as an error.

Alternatives The main alternatives discussed by the developers were threefold:

- *Ahead of time*. Maintain the current situation and enforce the validation of features as early as possible. This option provides a more deterministic solution but also may result in waste of resources.
- *Lazy loading*. Modify the expected behavior to validate features at call time. This option allows potential savings w.r.t. resources as modules and functions will only be validated and loaded if used, which may oftentimes not be the case.
- *Mixed approach*. Use lazy loading by default, but provide a compiler setting (*WebAssembly.validate*) that allows compilation ahead of time. This option will require library developers to maintain the two behaviors.

Solution Although the aforementioned issue is still open at the time of writing this chapter, the current solution is to partially abandon WebAssembly.

Rationale A specification that allows for a greater degree of lazy loading gives library developers the freedom to define the level of aggressiveness of the JIT compiler and balance responsiveness with other aspects, notably startup performance, battery, and memory. Furthermore, some stakeholders expected that WebAssembly code would be mainly generated by tools, which provides less room for true positives (i.e., actually malformed or defective features).

Pattern Lazy loading.

Consequences There are three main side effects raised by those involved in the discussion. First, the JIT compilation is abstracted from developers, who lose some control over optimization (e.g., for parallelizing loading tasks). However, it is expected that the benefits outweigh the optimizations that could be manually implemented. Second, although validation is performed at call time, the time at which errors are thrown is non-deterministic. This behavior may change entirely if a variable is set to enforce validation ahead of time. Finally, it is possible that non-deterministic aspects of the compiler may make testing more complicated. However, foreseeable problems can be averted by enforcing feature validation ahead of time (manually or by setting).

Source This issue was found by WebAssembly developers and their solution can be found at the aforementioned link.

5.6 Conclusions

In this chapter, we addressed energy efficiency as a pattern-related problem, where issues are not unique and reoccur systematically in a variety of software systems. In particular, we looked at two levels of abstraction, namely code and design, to

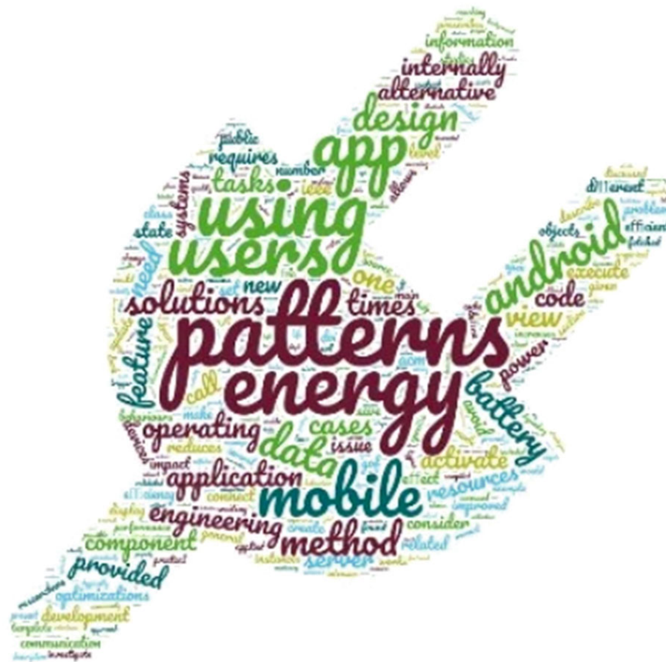

Fig. 5.4 Word cloud of chapter content

identify recurrent issues and solutions. Furthermore, we acknowledge that parts of a system are rarely islands, isolated from each other, and rather comprise a network of interconnected components, in which other patterns may be in play. Thus, we also considered and discussed energy efficiency from two perspectives: as a main concern of patterns and as a side effect of applying patterns.

To consolidate the concepts in this chapter, we showed how the different patterns were used in four real scenarios. These use cases emphasize the importance of being aware of energy-efficiency strategies to make informed decisions when developing sustainable software systems. In Fig. 5.4, we depict the most recurrent words in this chapter and, in light of the presented knowledge, we provide the following takeaway messages and advice.

There exists a consolidated list of refactorings for code-level patterns that can consistently be explored to improve the energy efficiency of Android mobile applications. Along these lines, we should, however, note that we have previously shown that combining as many individual refactorings as possible most often, but not always, increases energy savings. The interested reader may consult all the details on the magnitude and realization of the expected savings in [14].

On a different level of abstraction, design patterns have been used to improve energy efficiency. These patterns ought to be considered when designing software with critical energy requirements, such as mobile applications. By gaining knowledge about these patterns, developers can learn from the vast experiences of different developers across different platforms.

Finally, even if a pattern is not intended to address energy-related issues, it may still have a substantial effect on energy consumption. Thus, it is paramount to not

only be aware of the patterns applied in the system but also how to harvest their benefits while avoiding detriments to the overall energy consumption of the system. As a rule of thumb for OO systems, we suggest avoiding the application of patterns to encapsulate trivial functionality, e.g., small in size or that do not communicate with other classes.

References

1. Buschmann F, Meunier R, Rohnert H, Sommerlad P, Stal M (1996) Pattern-oriented software architecture: a system of patterns, vol 1. Wiley
2. Andrae A, Edler T (2015) On global electricity usage of communication technology: trends to 2030. Challenges 6(1):117–157. https://doi.org/10.3390/challe6010117
3. Power consumption in data centers is a global problem. https://www.datacenterdynamics.com/en/opinions/power-consumption-data-centers-global-problem/. Accessed 10 Jun 2020
4. Pinto G, Castor F (2017) Energy efficiency: a new concern for application software developers. Commun ACM 60(12):68–75. https://doi.org/10.1145/3154384
5. Thorwart A, O'Neill D (2017) Camera and battery features continue to drive consumer satisfaction of smartphones in US. https://www.prnewswire.com/news-releases/camera-and-battery-features-continue-to-drive-consumer-satisfaction-of-smartphones-in-us-300466220.html. Accessed 06 Feb 2019
6. The most wanted smartphone features. https://www.statista.com/chart/5995/the-most-wanted-smartphone-features. Accessed 24 Jan 2018
7. Mickle T (2018) Your phone is almost out of battery. Remain calm. Call a doctor. https://www.wsj.com/articles/your-phone-is-almost-out-of-battery-remain-calm-call-a-doctor-1525449283. Accessed 05 Feb 2019
8. Bragazzi NL, Del Puente G (2014) A proposal for including nomophobia in the new dsm-v. Psychol Res Behav Manag 7:155. https://doi.org/10.2147/PRBM.S41386
9. Fu B, Lin J, Li L, Faloutsos C, Hong J, Sadeh N (2013) Why people hate your app: making sense of user feedback in a mobile app store. In: Proc. ACM SIGKDD 19th Int. Conf. Knowledge Discovery and Data Mining (KDD '13). ACM, Chicago, IL, pp 1276–1284. https://doi.org/10.1145/2487575.2488202
10. Khalid H, Shihab E, Nagappan M, Hassan AE (2015) What do mobile app users complain about? IEEE Softw 32(3):70–77. https://doi.org/10.1109/MS.2014.50
11. Manotas I, Bird C, Zhang R, Shepherd D, Jaspan C, Sadowski C, Pollock L, Clause J (2016) An empirical study of practitioners' perspectives on green software engineering. In: Proc. IEEE/ACM 38th Int. Conf. Software Engineering (ICSE '16), pp. 237–248. IEEE, Austin, TX. https://doi.org/10.1145/2884781.2884810
12. Pang C, Hindle A, Adams B, Hassan AE (2016) What do programmers know about software energy consumption? IEEE Softw 33(3):83–89. https://doi.org/10.1109/MS.2015.83
13. Pinto G, Castor F, Liu YD (2014) Mining questions about software energy consumption. In: Proc. 11th Working Conf. Mining Software Repositories (MSR '14). ACM, Hyderabad, pp 22–31. https://doi.org/10.1145/2597073.2597110
14. Couto M, Saraiva J, Fernandes JP (2020) Energy refactorings for android in the large and in the wild. In: Proc. IEEE 27th Int. Conf. Software Analysis, Evolution and Reengineering (SANER '20). London, ON, pp 217–228. https://doi.org/10.1109/SANER48275.2020.9054858
15. Cruz L, Abreu R (2017) Performance-based guidelines for energy efficient mobile applications. In: Proc. IEEE/ACM 4th Int. Conf. Mobile Software Engineering and Systems (MobileSoft '17). IEEE, Buenos Aires, pp 46–57. https://doi.org/10.1109/MOBILESoft.2017.19