CS370 Undergraduate Reading and Research

Tyler Cook
tcc@cs.utexas.edu
https://github.com/cilki/CS370

August 16, 2019

1 Introduction

Many cross-platform applications benefit from low-level information about the systems on which they run, but developers rightfully do not want to maintain the code required to get it. Although many programming languages have standard libraries that provide a decent interface to some limited information about the running system, they usually don't support a system's more intimate details such as network interface speeds, firmware versions, CPU statistics, USB device information, and so on. Manually obtaining this data is usually off-limits for cross-platform applications because doing so tremendously increases an application's complexity and maintenance workload.

Additionally, applications may wish to know about platform-specific information in the case that it is available. Accessing platform-specific information from a cross-platform context is nontrivial because different platforms often have inconsistencies which makes providing a good API difficult.

1.1 Operating System & Hardware Information (OSHI)

OSHI is a pure Java library intended to provide users with all of the information a system has to offer without requiring them to worry about how that information is obtained on their platform. It does this by exposing a set of plain Java objects that correspond to components of a generic system. Depending on the platform, OSHI uses a different routine under-the-hood to fetch the component's data. Users can then use these objects in their applications to access system information regardless of platform.

To fetch data from the system, OSHI primarily uses JNA (Java Native Access) to call native APIs without having to write native C/C++ code via JNI (Java Native Interface). OSHI also reads from files in well-known locations (like from the `/sys/class` hierarchy on Linux) and parses output from the command-line as a last resort.

Although OSHI is one of the best options for cross-platform system information access, there are architectural several issues preventing it from reaching its full potential. For example:

- The library is tightly coupled (the API directly calls utility methods to fetch data). Along with making unit testing difficult[1], this makes it impossible to change how system data are fetched at runtime.
- The API is not general enough to handle every kind of component a system might have **uniformly**. This means that adding new features (system information) in the future might require building new infrastructure within the API.

- The API provides no facility for retrieving platform-specific attributes. Since OSHI already goes through the trouble of calling native libraries, fetching platform-specific information would not require much more effort. This information would be highly valuable to certain types of applications.
- Polling certain attributes produces more garbage objects than necessary.

1.2 Redesigning OSHI

Over the course of the semester I substantially refined my preconceived ideas on the optimal architecture of a cross-platform system information library. I also came up with a few new ideas and solved many new problems in both design and implementation domains. The architecture that I developed for OSHI is radically different from the existing project and attempts to solve many of its problems without being too complicated. The most novel aspects of the design are:

- The division of OSHI into loosely coupled API and DRIVER (fetching) layers
- The use of code generation to automatically generate the API layer from a declarative specification
- The use of Java annotations in the DRIVER layer for runtime configuration of the fetching routines

Although I think my design fully addresses many of the existing issues in OSHI, there are still several issues remaining:

- The API should eventually return values wrapped in Java's `Optional` class.
- There are still some issues surrounding attributes that are not supported on all variants of a platform (for example, earlier Windows releases have slightly different attributes than modern releases).
- Compositions of components are not implemented (and therefore untested). For example, `Partition`s should be contained within a `Disk`.

I built a small reference implementation of my design in the cilki/CS370 repository that currently works for some of the attributes of network interfaces, disks, and firmware on Linux. The oshi/oshi5 repository contains my reference implementation along with all of the existing project files and boilerplate required by Maven. In addition to my primary pull request[2], I also authored a minor pull request that introduces a configuration utility for the current project[3].

2 Architecture Overview

2.1 The API Layer

One of the most obvious improvements to OSHI's design is to decouple the API from the underlying fetching logic. This makes unit testing more effective as each component can be tested independently. I experimented with several different approaches and weighed the advantages and disadvantages of each before I reached a conclusion on the API.

My primary goal in designing the API was to make it absolutely uniform (consistent) throughout. Currently, many components in OSHI are implemented differently which both increases the maintenance burden and makes actually using the API more difficult. For example, if a certain idiom is adopted for updating attributes, or a certain structure for the fetching code is employed, it should be used throughout the entire project. Doing so makes it easier to spot errors, simplifies the act of adding new components/attributes in the future, and makes it easier for the user to integrate OSHI with their applications. The way my API achieves absolute uniformity is by automatically generating it from a high-level description of the attributes.

Using automatic code generation for OSHI's API layer turned out to be an excellent idea. This approach worked well because OSHI is the kind of project that meets two important criteria:

- The system information that OSHI provides can be accurately described in a declarative file that represents a single source of truth[4].
- There are naturally many repetitive structures in OSHI.

Once I populated the definitions file with all of OSHI's attributes and components, I was able to periodically regenerate the API as I worked on it which allowed me to rapidly prototype new ideas and instantly see the results across the entire project. This workflow was highly effective.

In addition to producing an API that is uniform by default, using a code generator also ensures the project is always synchronized as changes occur in the future.

2.1.1 The Generic Container API

My first idea was to store system information in a *generic container*, similar to the way tuples are used in Python's psutil library[5]. When requesting information on a system component, the user simply receives a mutable map-like object holding the data called a container. The API generator would output a set of static objects called attribute-keys that the user can use to retrieve values from the container. Each attribute-key corresponds to exactly one attribute in the definitions file and generic containers can contain any number of attributes.

```java
// Sample usage of the Generic Container API
for (GenericContainer disk : OSHI.getSystem().getDisks()) {
    disk.get(DiskAttribute.NAME);    // Return the NAME attribute
    disk.query(DiskAttribute.NAME);  // Update and return the NAME attribute
    disk.get(CpuAttribute.MODEL);    // Throws an error since Disks are not CPUs
    disk.get(DiskAttribute.INODES);  // Throws an error if we're not on a platform
                                     // that supports INODES
}
```

To make this usage type-safe, attribute-keys are generic and reflect the type of the corresponding attribute. Therefore, when the user attempts to retrieve an attribute, the attribute-key provides the necessary type information so that the user isn't forced to cast the result.

```java
// Sample attribute keys for a disk component
class DiskAttribute {
    public static final AttributeKey<String> NAME;
    public static final AttributeKey<Long> SIZE;
    ...
}
```

I first discovered the attribute-key design pattern in a library called Netty where it was used to provide type-safe access to user-defined attributes of a network socket. I've also used it in conjunction with code generation in my own project to provide access to a hierarchy of attributes (some of which were coincidentally OSHI attributes).

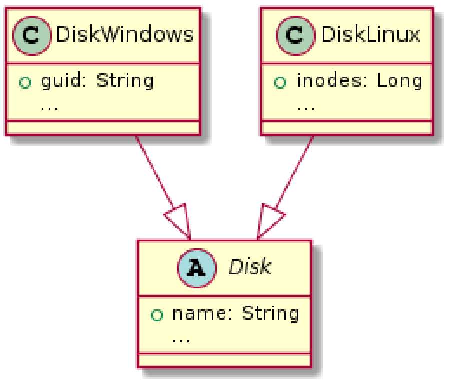
While this API is type-safe and highly flexible, it's not platform-safe which means the API allows actions that have no chance of succeeding (such as attempting to retrieve an attribute or component that doesn't exist on a particular platform). While this may be a relatively small disadvantage, I didn't think the OSHI denizens would like the map-style container objects and it certainly didn't meet my design objective of keeping usage similar to the existing project in order to reduce the migration burden for users.

2.1.2 The Concrete Container API

The idea behind this API is to take the most obvious approach to the problem and rely fully on code generation to make it feasible to write. In this API, each system component becomes a simple generated container class with a field and getter for each attribute of the component.

```java
// Sample container class for a disk component
class Disk {
    public String name; // The "name" attribute

    public String getName() {
        return name;
    }
}
```

To support platform-specific features, we can setup a simple inheritance hierarchy among the container classes:



The `DiskLinux` container provides access to both cross-platform attributes in `Disk` and attributes that are supported only on all Linux systems. The container classes can therefore be used as follows:

```java
// Cross-platform context
for (Disk disk : OSHI.getSystem().getDisks()) {
    disk.getName();         // A constant attribute (therefore queryName() isn't valid)
    disk.getReadBytes();    // Return the last value of READ_BYTES
    disk.queryReadBytes();  // Update READ_BYTES
}
```

```java

// Platform-specific context
for (DiskLinux disk : OSHI.getLinuxSystem().getDisks()) {
    disk.getName();    // A cross-platform attribute
    disk.getInodes();  // A platform-specific attribute
}
```

This example usage is nice for the user and there's no map lookup overhead to retrieve saved values, but there are two main problems:

- Not all attributes map onto an inheritance hierarchy well. For example, the inode count attribute, `INODES`, is not compatible with Windows, but is compatible with every other platform. To solve this, the `INODES` attribute must be pushed down in the hierarchy (from `Disk` into `DiskLinux`, `DiskMac`, etc). This keeps the API platform-safe, but now makes it slightly more inconvenient to get the `INODES` attribute on the majority of platforms. In other words, this API is nice for attributes that are *compatible* with only one platform, but isn't as nice for attributes that are *incompatible* with only one platform.

- The query methods can be problematic for performance. Most of the system information in OSHI is updated in batches because the operating system usually maintains system data in structs which we are able to access via JNA. Therefore, calling `query` for an attribute like `READ_BYTES` is almost guaranteed to also fetch `WRITE_BYTES`. To avoid wasteful duplicate queries, the user shouldn't call `query` on multiple attributes that are fetched in the same batch (in a short amount of time). The user is unlikely to follow this advice and, more importantly, the knowledge of which attributes are fetched together in the same batch is an implementation detail which should never be given to the user.

2.1.3 The Concrete Container + Attribute-Key Hybrid API

The API that I ended up with turned out to be a fusion of the Generic Container/Attribute-Key and Concrete Container approaches because there were several desirable qualities in each. The attribute-key pattern offered flexibility and simplicity, and the container architecture offers more attractive and safer usage (in most cases). My API is effectively the concrete container approach with attribute-keys as an optional feature.

Attribute Keys The API generator outputs a `public static final` field called an attribute-key for each attribute in the definitions file. Attribute-keys are primarily used to retrieve or query attributes, but are extensible to other tasks in the future.

Container Classes Containers are what the user receives when they ask for information on a system component. A container stores the most recent values of each attribute and can be easily serialized into JSON format. Each attribute in the definitions file produces three things in the container class:

- A public field that stores attribute's latest value
- A public getter method to return the attribute's latest value
- A public `query*` method to update and return the attribute (if the attribute is non-constant)

Container classes still have a one-to-one correspondence with components of the system and supports platform-specific features in the same way as the concrete container API. In addition to using the methods in the container classes, users may also retrieve/query attributes generically with an attribute-key. To prevent users from changing values in the container's public fields, the concrete container classes are hidden in an internal package and users interact with a public interface instead.

2.2 The Driver Layer

The DRIVER layer is where most of OSHI's business logic resides. It contains only driver classes and the query methods within those classes that perform the actual fetching of attribute data.

Originally I tried to have the driver classes implement the same interface as the corresponding container class (which would make it a compile error to forget to define a query method), but that didn't work for two reasons:

- It introduced a one-to-one correspondence between attributes and query methods which is fundamentally incompatible with batch queries. (Batch queries require a many-to-one relation between attributes and query methods).
- Fallback queries could not be implemented elegantly.

Instead of marrying attributes with query methods via an interface, I decided to use runtime annotations on the query methods. Although this moves some work from compile-time to runtime, it's acceptable for two reasons:

- I was able to use Java's "MethodHandle" API to reduce the overhead of runtime reflection in the common situation where an attribute is being polled.
- It's **impossible** to get the desired flexibility otherwise.

Here is the general structure of a query method for a disk driver:

```java
@DiskQuery(DiskAttribute.MODEL)  // Declares which attribute(s) this query method fetches
private void queryModel() {
    container.model = ...        // Fetch model string from system and update container
}
```

Attribute Enums Every attribute in the definitions file has a corresponding enum constant that is used to associate query methods (in the DRIVER layer) to attributes in container objects (in the API layer). These enums are not exposed to the user because attempting to query an attribute by its enum is not type-safe. Attribute-keys would solve this problem (since their type parameter reflects the type of the corresponding attribute), but unfortunately Java doesn't allow arbitrary objects in annotations which breaks the critical role of query annotations. Therefore, attribute-keys and attribute-enums serve the same purpose, but attribute-keys exist in the API layer and attribute-enums exist in the DRIVER layer.

Query Annotations Every query method in a driver class must be annotated with a query annotation containing one or more attribute-enums. This allows the runtime to choose a fetching routine (query method) for each attribute. Since the annotations can declare more than one attribute, this implicitly creates a many-to-one relationship between attributes and query methods which solves the problem of batch queries.

Java doesn't allow annotations to be defined with the enum supertype (`Enum<?>`), so query annotations must be automatically generated for each component (`@DiskQuery`, `@CpuQuery`, etc). This is an inconvenience, but isn't big a problem in practice.

Query Methods Query methods are **exclusively** responsible for fetching one or more attributes from a particular source. The must at least have a query annotation, but can optionally be configured with annotations like `@RequiresRoot`, `@Fallback`, and `@Timeout`.

For example, the location of the distribution name on Linux is not standardized and therefore OSHI must check around five different locations before giving up. In my design, each of these checks becomes a separate query method and the `@Fallback` annotation configures how to proceed if a method fails. This way, the runtime fully handles query ordering, permission issues, and timeouts rather than the query methods themselves.

Query Stacks Each attribute is associated to a sequence of query methods called a stack. When it's time to query an attribute, the runtime always starts at the top of the stack until it invokes a method that succeeds in querying the attribute. If the runtime reaches the bottom of the stack, then the query attempt fails. This structure has two attractive properties:

- Once initialized, repeated queries (polling) is low overhead.
- Failed query methods can be popped from the stack and therefore will never be retried.

2.3 Optimizing query requests

With attribute-keys in the API, the driver layer can elegantly handle querying multiple attributes without repeating query methods which was a problem with the Concrete Container API:

```java
// This should update the three attributes without repeating any query method
disk.query(MODEL, SIZE, NAME);

// This will update the three attributes, but will repeat query methods
disk.queryModel();
disk.querySize();
disk.queryName();
```

However, I unfortunately encountered an NP-Complete problem when trying to implement this method. Since there's no restriction on the combinations of attributes that may be used in query annotations, overlapping query methods can cause major problems. Consider the following imaginary driver class:

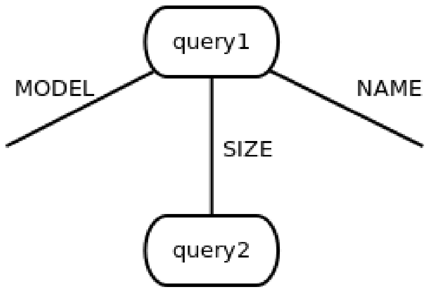
```java
@DiskQuery(MODEL, SIZE, NAME) // Three attributes are fetched by this method
private void query1() {
  ...
}

@DiskQuery(SIZE)              // The SIZE attribute overlaps with query1()
private void query2() {
  ...
}
```

After loading this driver, the runtime builds a map consisting of query stacks that looks like this:

```
MODEL: [query1]
 SIZE: [query2, query1]
 NAME: [query1]
```

If a request for the attributes `MODEL` and `SIZE` comes in, the runtime can either execute `query1` only (since it provides every attribute in the request) or both `query1` and `query2` (which leads to `SIZE` getting updated twice). We want to avoid wasteful queries, so it would be nice to build an algorithm that chooses the fewest number of query methods to satisfy each request.

This problem can be reduced into the well-known vertex cover problem by mapping each query method into a node in a graph and mapping each attribute into an edge. Each edge simply connects all of the nodes (query methods) that fetch the corresponding attribute. Therefore, a single edge can connect any number of nodes from zero to infinity (which makes this a hypergraph). Here is the reduction of the above example into the standard vertex cover problem:



Finding the smallest number of query methods that would satisfy the request is equivalent to finding the smallest number of nodes that collectively touch every edge (the vertex cover problem). To complicate things even further, query methods can also fail which means that every node has a chance of getting ”rejected” from the cover during runtime. A topic for future inquiry would be to use a known approximation algorithm for vertex cover to determine the optimal sequence in which query methods should be called. This would need to be done eagerly at initialization time to keep the query latency as low as possible.

In short, there’s no way I could guarantee that an arbitrary request would not execute the same query method more than once under the given constraints. Fortunately the query stacks in OSHI are almost always fairly simple. The majority of attributes will have only one applicable query method which eliminates all of the above complexities and uncertainties. (Because the vertex cover for a hypergraph containing exclusively edges that are connected to one node is trivial). Therefore my reference implementation usually does not wastefully invoke query methods except in rare cases of highly complex attribute overlapping.

3 Deviations from proposal

3.1 Maven Integration

I didn’t integrate the API generator with Maven which would have the benefit of automatic code generation on build. The main reason being that regenerating the API from an IDE can be done with a single click and probably in less time than it takes Maven to warm-up. Since generated code for the API will be committed to the repository, the need for Maven integration is reduced, but it will certainly need to be done before the 5.X.X branch is properly tested in a CI environment.

Since code generation is a common task for Maven, integrating OSHI's API generator into the build will consist of writing a simple Maven plugin.

3.2 Existing Fetching Code

I didn't import all of OSHI's fetching code into the new design. There were two reasons for this:

- There was a lot more fetching code than I expected, despite being moderately familiar with the codebase at the beginning.
- The fetching code is OSHI's primary business logic and as a result is the most heavily refined. Extra care should be taken to not reintroduce previously fixed bugs.

I focused my efforts on fetching code for Disks, Firmware, and Network Interfaces which should demonstrate the design's feasibility.

3.3 Attribute Definitions Format

I decided to format the attribute definitions file as JSON rather than YAML. The two primary reasons being that the Jackson JSON parser is already used elsewhere in the project and that JSON's syntax seemed to be more understandable than YAML for this problem despite being less terse.

The content of the definitions file also changed from the specification I wrote earlier this year as I discovered and solved new issues[8]. Specifically, I removed information from the file that I now consider to be implementation details like the "permissions" field which defines whether superuser permissions are required to fetch a given attribute. This information was moved into driver annotations which is much more appropriate.

3.4 Jigsaw Modularity

Java's Jigsaw feature for modularity did not make it into the new design. Although it would be nice to allow users to pull only the pieces of the project that they need, it doesn't make much sense given that almost every component depends on JNA. Altogether, OSHI is an order of magnitude smaller than JNA (around 400 KiB vs 5 MiB), so the savings of Jigsaw modularity would be relatively tiny.

3.5 Generic Drivers

Rather than implementing new generic drivers (general-purpose objects for fetching a particular resource that are used by higher-level drivers), I temporarily made use of the existing OSHI utilities for fetching. Although not required, generic drivers strongly complement the design and should be implemented in the future.

A Example Output

Output of the examples can be found below:

A.1 DiskExample.java

```
Cross-platform access:
Disk name: /sys/devices/pci0000:00/0000:00:1c.0/0000:03:00.0/nvme/nvme0/nvme0n1
    Size: 120034123776
    Read bytes: 658617856
    Write bytes: 54596608
Disk name: /sys/devices/pci0000:00/0000:00:1f.2/ata1/host0/target0:0:0/0:0:0:0/block/sda
    Size: 1000204886016
    Read bytes: 0
    Write bytes: 126464

Cross-platform access for SMART attributes:
Disk name: /sys/devices/pci0000:00/0000:00:1f.2/ata1/host0/target0:0:0/0:0:0:0/block/sda
    Power cycles: 687
    Power on hours: 16547
    Temperature: 36
```

A.2 NicExample.java

```
Cross-platform usage:
NIC: enp2s0
    ReadBytes: 1183412
    WriteBytes: 640968
    MTU: 1500
    IPv4: [10.0.4.2]
    Link Speed: 10000
NIC: enp4s0
    ReadBytes: 2043622324
    WriteBytes: 182954245
    MTU: 1500
    IPv4: [10.0.1.128]
    Link Speed: 1000
NIC: lo
    ReadBytes: 34714719
    WriteBytes: 34714719
    MTU: 65536
    IPv4: [127.0.0.1]
    Link Speed: 0
```

References

[1] https://github.com/oshi/oshi5/issues/5

[2] https://github.com/oshi/oshi5/pull/14

[3] https://github.com/oshi/oshi/pull/896

[4] https://en.wikipedia.org/wiki/Single_source_of_truth

[5] https://psutil.readthedocs.io/en/latest

[6] https://github.com/oshi/oshi5

[7] https://en.wikipedia.org/wiki/Vertex_cover

[8] https://github.com/oshi/oshi5/pull/12