Rigorous Software Engineering- Week 1 (Lectures)

- Author: Ruben Schenk
- Date: 08.03.2021
- Contact: ruben.schenk@inf.ethz.ch

1. Introduction

1.2 Challenges

The four key dimensions to the question "Why is Software so Difficult to Get Right?" are:

- *Complexity*: Modern software systems are huge, they are created by many developers over several years. They also have a high number of discrete states and execution paths.
- *Change*: Software systems often deviate from their initial design. Typical changes include: New features, new interfaces, bug fixing, performance tuning.
- *Competing Objectives*: Functionality vs usability, cost vs robustness, performance vs portability, etc.
- *Constraints*: Such as budget, time, staff and their available skills.

1.4 Solution Approaches

This course is split up into four parts:

- Part I, Writing clean code: Modularity, Coupling, Design Patterns
- Part II, Software Testing: Metrics, exhaustive, random, functional, structural testing.
- Part III, Software Analysis at Scale: Math, heap, numerical, symbolic execution, concolic execution, fuzzing
- Part IV, Modeling: Model finding, Alloy, applications to memory models

2. Documentation

2.1 Why should we document?

2.1.1 Essential vs Incidental Properties

Source code does not exspress *which properties are stable* during software evolution, moreso, which properties are `essential` and which are `incidental`.

Example: In the following code, is it *essential* that it finds some index i such that $\text{array}[i] = v$ or the *first* index i ?

```java
int find(int[] array, int v) {
    for(int i = 0; i < array.length; i++) {
        if(array[i] == v) {
            return i;
        } else {
            return -1;
        }
    }
}
```

If a developer at a later point decides to parallelize this function, it is essential to know whether or not it is important to return the first matching index or just some matching index.

2.1.2 Invariants

We look at a simple example code:

```java
HashMap<String, String> m;
m = SomeLibrary.foo();
String s = m.get("key");
```

Now, one might ask whether or not s can be `NULL`.

- What happens if the key is not found in the map? -> The function `.get()` returns `NULL`.
- What if the key is matched? Can the value still be `NULL`?

2.1.3 Overriding methods

We first introduce `lazy initialization` shown by the following code snippet:

```java
class ImageFile {
    String file;        // Path to image
    Image image;        // Loaded file

    ImageFile(String f) {
        file = f;
    }

    Image getImage() {
        if(image == null) {
            // load image
        }
```

```
            return image;
        }
    }
```

We see that we do *not* load the image on creating the `ImageFile`, i.e. in the constructor, but only when it's actually needed, i.e. in the `.getImage()` function. This style of coding is callec *lazy initialization*.

We now extend the above code snippet with the following `hashcode` function:

```
    int hashcode() {
        if(image == null) {
            return file.hashcode();
        } else {
            return image.hashcode() + file.hashcode();
        }
    }
```

And run the following demo:

```
    void demo(HashMap<ImageFile, String> m, ImageFile f) {
        m.put(f, "Hello");
        Image i = f.getImage();
        int l = m.get(f).length();
    }
```

The problem here is that we first store `"Hello"` with the hashcode of only *file* (since the image is `NULL` at this point) and then try to access the string again, after we loaded the image. But now the hashcode changed, since it now includes the hashcode of the image too (and not only of the file as before).

2.2 What to document?

One should document all *essential properties* explicitly:

- For clients: How to use the code? -> Document the interface.
- For implementors: How does the code work? Document the implementation.

2.2.1 Interface Documentation

The client interface of a class consists of:

- Constructors
- Methods
- Public fields
- Supertypes

-> We focus on methods here

Method Call

Clients need to know `how to call` a method correctly. Example:

```
class InputStreamReader {
    int reader(char cbuf[], int offset, int len) throws IOException {
        ...
    }
    ...
}
```

For the parameters we have some "obvious" restrictions:

- `cbuf` is non-null
- `offset` is non-negative
- `len` is non-negative
- `offset + len` is at most `cbuf.length`

We furthermore have an *input state restriction*, namely that the receiver must be open.

Method Result

Clients need to know `what a method returns`. As any example to the code above:

- The method returns -1 if the end of the stream has been reached without reading any characters.
- Otherwise, the result indicates how many characters have been read.

Method Effects

Clients need to know how a `method affects the state`. Again, to the example above:

- Heap effects: The result characters have been stored in `cbuf`.
- Other effects: The methods throws an `IOException` if the stream is closed or an I/O error occurs.

Global Properties
Some implementations have properties that affect all methods.

- *Consistency*: properties of states, e.g. that a list is sorted
- *Evolution*: properties of sequences of states, e.g. that a list is immutable
- *Abbreviations*: requirements or guarantees for all methods, e.g. that a list is not thread-safe

2.2.2 Implementation Documentation

Method documentation is mostly similar to interfaces. Data structure documentation is more prominent, i.e. properties of fields, internal sharing, etc. We might also document the algorithms inside the code, e.g. justification of assumptions etc.

Example: We look at the following two classes:

```
class ListRep<E> {
    E[] elems;
    boolean shared;
    ...
}

class List<E> {
    ListRep<E> rep;
    int len;
}
```

We might add the following information which could be helpful in the documentation:

- `elemes` is non-null
- `elems` is pointed to by only one object
- When the shared-field is true then shared, elems, and all elements are immutable
- ...

2.3 How to document?

There are multiple different ways to document code:

2.3.1 Comments

Comments are a simple, flexible and effective way of documenting interfaces and implementations. One might comment the `V get(Object key)` as follows:

```
/**
* Returns the value to which the specified key is mapped, or
* {@code null} if this map contains no mapping for the key.
*
* @param key the key whose associated value is to be returned
*
* @return the value to which the specified key is mapped,
* or {@code null} if this map contains no mapping for the key
*
* @throws NullPointerException if the specified key is null
* and this map does not permit null keys
*/
V get(Object key);
```

2.3.2 Types and Modifiers

Types are a powerful documentation tool and modifiers can express some specific semantic properties.

Example:

```
/* Types as documentation tool. */
HashMap<String, String> m;
m = SomeLibrary.foo();
String s = m.get("key");
```

```
/* Modifiers as documentation tool. */
class HashMap<K, V> ... {
    final float loadFactor;
}
```

2.3.3 Effect Systems

Effect systems are extensions of type systems that describe computational effects:

- Read and write effects
- Allocation and de-allocation
- Locking
- Exceptions

Example:

```
try {
    int i = isr.read();
} catch(IOException e) {
    ...
}
```

2.3.4 Metadata

Annotations allow one to attach additional syntactic and semantic information to declarations.

Example:

```
@NonNull Image getImage() {
    if(image == null) {
        // load image
    }
    return image;
}
```

2.3.5 Assertions

Assertions specify semantic properties of implementations, i.e. boolean conditions that need to hold.

Example:

```
void shrink() {
    ...
    if(rep.shared) {
        rep = new ListRep<E>();
    }
    assert !rep.shared;
    ...
}
```

2.3.6 Contracts

Contracts are stylized assertions for the documentation of interfaces and implementations. This includes:

- Method pre- and postconditions
- Invariants

Example:

```
class ImageFile {
    String file;
    invariant file != null;

    Image image;
    invariant old(image) != null ==> old(image) == image;

    ImageFile(String f) requires f != null; {
        file = f;
    }

    Image getImage() ensures result != null; {
        if(image == null) {
            // load image
        }
        return image;
    }
}
```