NEW! STRONGER THAN DIRT

ENTER.
tech with a heartbeat

{clean Code}

CAUTION: EVEN BAD CODE CAN FUNCTION
40 FL OZ (1.25 QT) 1.18L

THE LAW OF DEMETER

- A module should not know about the internals of the objects it manipulates;
- It means that an object should not expose its internal structure through accessor because to do so it exposes its internal structure
- A method should not invoke methods on objects that are returned by any allowed functions.
- *Talk to friends not to strangers*.

TRAIN WRECKS

```
Final String outputDir = ctxt.getOptions().getScratchDir().getAbsolutePath();
```

- This kind of code is known as train wrecks because it looks like a bunch of couple train cars.

- It should be avoided.

```
Options opts = ctxt.getOptions();
File scratchDir = opts.getScratchDir();
Final String outputDir = scratchDir.getAbsolutePath();
```

HYBRIDS

- Are objects that are half object and half data struct.
- They have functions that do significant things, and they also have either public variables or public acessors and mutators.
- This kind of classes makes it hard to add new functions, but also make it hard to add new data structures.
- Worst of both worlds. Avoid creating them.

DATA TRANSFER OBJECTS

- These are classes with public variables and no functions.
- Especially when communicating with databases or parsing messages from sockets.
- **Active record**
  - Special forms of DTOs. They are data structures with public variables; but they have navigational methods like save and find. Tipically, they are direct translations from database tables.
  - They should not have business rules.

ENTER.
tech with a heartbeat

USE EXCEPTIONS

- … Rather than return codes
- Returning error codes, makes the caller immediately check the return to see if it is an error or a normal return.
- **Using exceptions separates the main code from the exception handling.**

WRITE YOUR TRY-CATCH STATMENTS FIRST

- The code executed in a try statement can be aborted at any time.
  - In a try-catch-finally statement, we are forcing our code to leave this block in a consistent state, no matter what happens inside the try block.
- Writing try-catch-finally statement first helps you to define what the user of that code should expect, no matter what goes wrong with the code executed in the try block.

USE UNCHECKED EXCEPTIONS

- Checked exceptions are are the exceptions that are checked at compile time. If some code within a method throws a checked exception, then the method must either handle the exception or it must specify the exception using throws keyword.

- In theory, it is a very good idea, but in practice, if there are multiple call levels, all functions should have in its signature the kinds of exceptions that the lower levels can throw, and it **violates the Open/Close principle**.

PROVIDE CONTEXT WITH YOUR EXCEPTIONS

- Each exception you throw should provide enough context to determine the source and location of the error.

- Create an informative text message and pass them along with your exceptions.
    - Mention the operation that failed and the type of failure.

DEFINE EXCEPTION CLASSES IN TERMS OF A CALLER'S NEEDS

- When defining exception classes in an application, the most important concern should be **how they are caught**.

- If different exceptions are handled in a similar way, it is a good practice to wrap them in an exception class.
  - In this way you throw your own exceptions and reduces the dependencies of external APIs.

DEFINE EXCEPTION CLASSES IN TERMS OF A CALLER'S NEEDS

```
ACMEPort port = new ACMEPort(12);
try {
 port.open();
} catch (DeviceResponseException e) {
 reportPortError(e);
 logger.log("Device response exception", e);
} catch (ATM1212UnlockedException e) {
 reportPortError(e);
 logger.log("Unlock exception", e);
} finally {
 ...
}
```

DON'T RETURN NULL

- If we return null we are essentially creating work for ourselves and forcing problems upon our callers.
  - One missing null check can make your application go out of control.
- In this cases, try to throw an exception or a Special Case object.
  - Like an empty list, for example (instead of a null).

DON'T PASS NULL

- Passing null to objects is even worse.
- In most languages, there's no real good way to deal null with passing.

INTRODUCTION

- We seldom control all the software in out systems.
- Sometimes we have to integrate third party code into our own and it should happen cleanly.

USING THIRD-PARTY CODE

- There’s a natural tension between the provider of an interface and the user of an interface.
- Providers: *Strive for broad applicability so they can work in many environments and appeal to a wide audience.*
- Users: *want the interface to work on their particular needs.*
- **And this tension can cause problems at the boundaries of our systems**.

USING THIRD-PARTY CODE

- Consider Java Map. If the application needs a Map of sensors:

```java
Map sensor = new HashMap();
```

- If the application needs to access a sensor we need to:

```java
Sensor s = (Sensor)sensors.get(sensorId);
```

- Which is not clean code!

USING THIRD-PARTY CODE

- A better way would be by using java generics that transform it to:

```
Map<Sensor> sensor = new HashMap<Sensors>();
```

- If the application needs to access a sensor we need to:

```
Sensor s = sensors.get(sensorId);
```

- However, we still depend on the Map interface which can be a problem if the map interface ever changes.

USING THIRD-PARTY CODE

- The solution to this problem is to wrap it in its own class:

```
Public class Sensors {
        private Map sensors = new HashMap();

        public Sensor getById(String id) {
                return (Sensor) sesnors.get(id);
        }

        . . .
}
```

- It hides the interface at the boundary Map.

USING THIRD-PARTY CODE

- This new interface is also tailored to meet the needs of the application and it results in code that is easier to understand and harder to misuse.
- The sensors class can enforce the design and business rules.
- **Avoid returning boundary interfaces like a map, or sending it as return of public APIs**.

EXPLORING AND LEANING BOUNDARIES

- It is a good practice to write tests for the third-party code we use.
- Integrating third party code can be difficult: A better approach of using it is to write some tests to explore our understanding of the third-party code.
- In this way we are essentially doing controlled experiments to check our understanding of that API.

LEARNING-TESTS ARE BETTER THAN FREE

- The learning-tests, cost nothing. It is a way to get the knowledge we need to use the API.

- It also helps to validate new releases of the API by running the tests we wrote to validate its usage.