Clean Code in Practice

1. Introduction

Do you use Clean Code practices on a daily basis. If yes, why? If no, why?

This post, follows the InterVenture[1] Jam Session talk, "Clean Code in Practice". The talk is targeting an interesting phenomena.

> Majority of developers are familiar and appreciative about the Clean Code practices. However, in practice they use only a handful of routines.

This is a personal observation that I keep asserting as true. Let me give you an example. During the talk, amongst the thirty something developers of varying experience, all have said that they have heard about Clean Code. When followed with a question of who's using the routines on a daily basis. There was only a single hand still standing.

What is behind this mismatch? Is it that we don't believe in the benefits of Clean Code? Or is it that the costs are not worth the benefit?

Clean Code in practice talk tries to asses the value of Clean Code when applied to real life applications.

We target a well known Spring PetClinic app as an example of a source code that is on pair to with the code we do on our everyday basis. We'll review it from the Clean Code perspective and asses the value of the changes brought upon the applied Clean Code practices.

This series of blog posts presents the findings.

1.1. Cycling Helmet Analogy

Why don't we use CC routines more often? I can express my belief with an analogy. Do you use a cycling helmet when going for a joy ride? Despite being an undisputable good thing, many don't. The belief that you won't really need

[1] https://www.interventure.info/

it, combined with a slight nuisance of having to carry it around, topped with a potentially ruined hair style, makes the decision of ignoring the helmet quite likely. Even if it can be a life saver.

It's not that we're not aware of the risk. I rarely see kids without the helmet. However, we ourselves, we feel as pro's. We ride for a long time, and believe competent enough that we will not fall. It will not happen to us, not to us or not this time.

Until we do fall. Afterwards, the helmet is a must, and we frown upon people ignoring it.

1.2. Is Clean Code Only for Beginners?

Just like with the helmet, as we become seasoned developers we start trusting ourselves with the code. First of all, we do tests. We also use architecture frameworks such as Spring, Angular or something third. So, if we are testing, if we're using MVC, the rest is cosmetic, right? In addition, we don't have all the time in the world to polish every corner of the application. Experienced developers tend to underestimate clean code.

At least I speak about my personal perspective from the early days. What changed for me is that I've joined a project that is in production for a loooong time. We deal with uber classes, measured in thousands of lines of code. Do you think that the class having several thousand lines is properly tested? Do you think it does something important? The only thing we can rely on is the messy code that we need to understand, day in, day out. We're reaching near zero productivity. We're afraid of breaking things. We're stuck.

The point we reached is perfectly described in Clean Code: A Handbook of Agile Software Craftsmanship[2]. There's something about the personal experience that makes you become a believer. My second take on clean code book, was as reading the Agatha Christie's novel.

For the two years I'm practicing Clean Code routines in a much greater extent then ever before, I see a shift for the better. This is what I'm trying to communicate.

[2] https://www.amazon.com/Clean-Code-Handbook-Software-Craftsmanship/dp/0132350882

1.3. Code Cleaning Spring PetClinic

Spring PetClinic is a good target as it is in a really good shape and familiar to millions. In my belief, its the only demo application that has its own community. In addition Spring as framework, just as clean code, is on a mission of enabling apps that are easy to change and evolve.

Following are the findings of Spring PetClinic review, sorted by the degree of drama:

2. Classes

Amongst all the weak spots that our clean code review uncovers, the one discussed in this section is both nasty and sneaky. Other weak spots are good placeholders for readability boosting, but the anomaly we're about to discuss, is a design flow waiting to blow-up in your face.

When it comes to classes, the main design guideline to be followed is to keep them small. The interesting bit is that the size of a class is measured, not in lines of code, rather in responsibilities.

2.1. Principle of Least Astonishment

In clean code, one rule applies, expect the expected, or unexpect the unexpected.

Observe the VetTests.java[3] in the `clean-code` branch:

VetTests.java.

```
@Test
public void testAddSpeciality() {
    assertThat(vet.getSpecialties().size()).isEqualTo(0);
}
```

This test passes, so no suprises there. However, lets add a speciality to a `vet`:

VetTests.java.

```
@Test
public void testAddSpeciality() {
```

[3] https://github.com/interventure-growingtogether/spring-petclinic-clean-code/blob/clean-code/src/test/java/org/springframework/samples/petclinic/vet/VetTests.java

```
    assertThat(vet.getSpecialties().size()).isEqualTo(0);

    Specialty s = new Specialty();
    s.setName("oftamology");
    vet.getSpecialties().add(s);
    assertThat(vet.getSpecialties().size()).isEqualTo(1);
}
```

To my suprise, this test is failing. This is unexpected, and unexpected behaviour is what clean code is aiming to prevent.

> You know you are working on clean code when each routine you read turns out to be pretty much what you expected.
>
> — Ward Cunningham

The failing test, forces us to see what is the actual implementation of the `getSpecialities()` method.

2.2. Single Responsibility Principle

Class should have a single, clear and well defined responsibility. Its often phrased like this:

> The Single Responsibility Principle (SRP) states that a class or module should have one, and only one, reason to change.
>
> — Robert C. Martin *Clean Code - A Handbook of Agaile Software Craftmanshift*

If you use code completion, the methods that you see should all make sense. With `Vet.java`, the case is as follows:

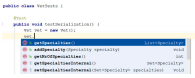

Figure 1. vet code completion

Can you tell what is the difference btw `getSpecialtiesInternal()` and `getSpecialties()`? Without looking at the code, I would wrongly assume that there exist two fields, `specialties` and `specialtiesInternal`.

There's both `specialities` and `specialitiesInternal` method. Obviously, the class is having two representations of `specialities`, one internal (whatever that

might be), one public. Just by reading the names, one can notice that the `Vet.java` is violating the Single Responsiblity Principle.

2.3. Mixing Levels of Abstraction

> Good software design requires that we separate concepts at different levels and place them in different containers. We don't want lower and higher level concepts mixed together.
>
> — Robert C. Martin *Clean Code - A Handbook of Agaile Software Craftmanshift*

Here is the complete Vet.java[4]:

Vet.java.

```
import java.util.*;

import javax.persistence.*;
import javax.xml.bind.annotation.XmlElement;

import org.springframework.beans.support.MutableSortDefinition;
import org.springframework.beans.support.PropertyComparator;
import org.springframework.samples.petclinic.model.Person;

@Entity
@Table(name = "vets")
public class Vet extends Person {

    @ManyToMany(fetch = FetchType.EAGER)
    @JoinTable(name = "vet_specialties", joinColumns = @JoinColumn(name
 = "vet_id"), inverseJoinColumns = @JoinColumn(name = "specialty_id"))
    private Set<Specialty> specialties;

    protected Set<Specialty> getSpecialtiesInternal() {
        if (this.specialties == null) {
            this.specialties = new HashSet<>();
        }
        return this.specialties;
    }
```

[4] https://github.com/interventure-growingtogether/spring-petclinic/blob/master/src/main/java/org/springframework/samples/petclinic/vet/Vet.java

```
    protected void setSpecialtiesInternal(Set<Specialty> specialties) {
        this.specialties = specialties;
    }

    @XmlElement
    public List<Specialty> getSpecialties() {
        List<Specialty> sortedSpecs = new
 ArrayList<>(getSpecialtiesInternal());
        PropertyComparator.sort(sortedSpecs,
                new MutableSortDefinition("name", true, true));
        return Collections.unmodifiableList(sortedSpecs);
    }

    public int getNrOfSpecialties() {
        return getSpecialtiesInternal().size();
    }

    public void addSpecialty(Specialty specialty) {
        getSpecialtiesInternal().add(specialty);
    }

}
```

Considering the main responsibility of the class, the only reasonable expectation is that the import section holds only `java.util.*` and `javax.persistence.*` packages. However, we see imports from `import  javax.xml.*` and `import org.springframework.*`.

If you check how and where the class is being used, you'll notice that its properties are directly accessed from the view as well. This is what's behind the method name confusion, the `getSpecialities()` is accessed from the view.

2.4. Class Clean Code Refactoring

In fact, the proper SRP refactored `Vet.java` class should look something like:

Vet.java.

```
@Entity
@Table(name = "vets")
public class Vet extends Person {

    @ManyToMany(fetch = FetchType.LAZY)
```

```
    @JoinTable(name = "vet_specialties", joinColumns = @JoinColumn(name
 = "vet_id"), inverseJoinColumns = @JoinColumn(name = "specialty_id"))
    private Set<Specialty> specialties;

    public Set<Specialty> getSpecialties() {
        if (this.specialties == null) {
            this.specialties = new HashSet<>();
        }
        return this.specialties;
    }
}
```

The nastiness of this weak spot reflects in the fact that the impact of change required for fixing it is quite fair. Spring Petclinic, obivously for brewity sake, is missing a business layer. Parts of the responsibilities of this layer are moved to the Vet.java entity, e.g. getSpecialites, addSpecialites method etc. part is moved to Vets.java class, and parts are right in the vetList.html.

Check the code in `clean-code` branch to see the fixes:

VetViewModel.java.

```
@XmlRootElement
public class VetViewModel {

    private String firstName;
    private String lastName;
    private String specialties;

    public VetViewModel(Vet vet) {
        this.firstName = vet.getFirstName();
        this.lastName = vet.getLastName();
        this.specialties = toSpecialities(vet);

    }

    private String toSpecialities(Vet vet) {
        return vet.getSpecialties()
            .stream()
            .map(NamedEntity::getName)
            .sorted()
            .collect(Collectors.joining(" "));
    }
```

```
    public String getFirstName() {
        return firstName;
    }

    public String getLastName() {
        return lastName;
    }

    public String getSpecialties() {
        return specialties;
    }
}
```

First we include a VetService.java[5]:

VetService.java.

```
@Service
public class VetService {

    private VetRepository vetRepository;

    public VetService(VetRepository vetRepository) {
        this.vetRepository = vetRepository;
    }

    @Transactional
    public List<VetViewModel> findVets() {
        return vetRepository.findAll().stream()
        .map(vet -> new VetViewModel(vet))
        .collect(Collectors.toList());
    }
}
```

In addition to small refactors to the test and `VetController` we can consider this SRP violation fixed.

2.5. Class Clean Code Refactoring Conclusions

Classes should have a single responsibility. Often times, we take shortcuts and reuse existing classes in additional context. Typically, this can be OK, if we are

[5] https://github.com/interventure-growingtogether/spring-petclinic-clean-code/blob/clean-code/src/main/java/org/springframework/samples/petclinic/vet/VetService.java

early in the process of development, however, the more the project develops, the more there will be divergence between scenarios. Eventually, difference use cases will yield different classes. This is something to keep in mind, so that the impact of change is not too big.

The nasty bit was that we have to change a lot, the sneaky bit has to do with tests, and it follows in the separate blog post

3. Testing

There's at least half a dozen reasons why Spring PetClinic is so well geared for clean code review.

First, the expectation that the code quality level is already high enough, as it is being reviewed by thousand of souls on a daily basis.

Second, Spring as a frameworks strives for the similar goals as clean code, promoting good design, that is testable and easy to change.

Third, crucial for this section, its quite well tested, JaCoCo plugin reports coverage that is above 90%.

3.1. How Many Tests Should be in a Test Suite?

Is code coverage 90% enough? It sounds like it for sure. Here's a quote for you:

> How many tests should be in a test suite? Unfortunately, the metric many programmers use is "That seems like enough." A test suite should test everything that could possibly break. The tests are insufficient so long as there are conditions that have not been explored by the tests or calculations that have not been validated.
>
> — Robert C. Martin *Clean Code - A Handbook of Agaile Software Craftmanshift*

Still, CC celebrates the usage of a coverage tool, and 90% is surely a good starting point.

3.2. Test Everything That Can Break

Remember the previous post? We we're examining the `Vet.java`. There was one technical issue that immediate caught my attention. The relation toward `specialties` was marked as EAGER. In fact, its a well known anti-pattern[6].

It's basically like `all you can eat` restaurant, where no-one is really checking the tickets. This way you can bring all of your relative, and relatives of your relatives, and eat all that you can with only a single ticket pay.

If specialities were to have simliar kind of eager relations, retrieving a name of a single Vet would result in many more records pulled from the database.

The reason why this is intriguing even from the clean code perspective is due to the fact that the `specialities` are accessed from the view. I was accepting that `EAGER` is there to hide the LazyInitializationException[7]. To my suprise, the app was working even after changing `EAGER` to `LAZY`.

The confusion lasted for a moment only. I learned that Spring Boot Application is using a property: `spring.jpa.open-in-view` set to `true` by default.

This is also a very suspicous default considering that Open Session in View is also considered an anti-pattern[8]

If anyone is interested in details about why is the default value set to true, read the debate at this thread[9]

The summary of the debate is what Phill Web from the spring team mentioned:

> After a lot (and I do mean a lot) of discussion we've decided to leave things as they are. The primary reason is that people upgrading are likely to face very subtle issues that only manifest themselves in certain circumstances.
>
> If we were starting from a clean slate we may well have picked a different default, but we think the pain of changing the default

[6] https://vladmihalcea.com/eager-fetching-is-a-code-smell
[7] https://vladmihalcea.com/the-best-way-to-handle-the-lazyinitializationexception/
[8] https://vladmihalcea.com/the-open-session-in-view-anti-pattern/
[9] https://github.com/spring-projects/spring-boot/issues/7107

> (even at a major release) is going to cause more bugs than leaving things as they are.
>
> — Phil Webb

I understand both points in debate. One was that Spring Boot attempts at giving a pleasent start experience for the developers. They should not be faced with a technical exception firing in the view at the very beginning of learning. On the other hand, the closer the code gets to production, the more problematic it gets to change it.

But, going back to clean code. From the clean code perspective, the default value should have been the one that doesn't cause any fear of change. The default should have been `false`.

3.3. Test Clean Code Conclusion

We mentioned in previous post that coupling an Entity to view is nasty due to the impact of change needed to fix it. The sneaky part is something different. If, for the reasons given above, we change EAGER to LAZY, and `spring.jpa.open-in-view` to false, and run the application, we'll get an error page as soon as we try to list all vets.

This is fine, the changes in the clean code branch show how to remedy this. Plus, we have a 90% test coverage that will surely discover that the application is not working in production.

But let's really make the two changes on the original PetClinic. Let us, also, run all the test to see which one is failing. I'll use the dots to fake a suspense of following a console log … AND….BOOM!

All test are passing?!

Application is failing in production, but all test are passing. This is the kind of green I would never like to see. False positive.

Despite the high coverage rate, tests are not testing everything that can possible break.

That is it, GAME OVER for Spring Petclinic application.

4. Functions

In our previous posts, we discovered places where Spring Petclinic is violating the Single Responsibility Principle.

We also learned, in quite a "dramatic" post :) that despite the high test coverage, tests are not testing everything that can possible break.

This post will be more fun and relaxing, we'll be discussing some of the function-related clean code violations, and we're in for a treat.

4.1. Functions Must be Simple

Clean code's view on the subject of functions is clear and precise. Functions should be simple, do one thing, and do it well. Simplicity is implied by the size, number of arguments, number or nested constructs within the function, just to name a few.

There exits a famous example of having a boolean as function argument. Effectively, with a boolean as an argument, your're having a function that is doing at least two things. One for the `true` case, one for the `false` case.

4.2. Flag Arguments in Functions

Let us examine one such method:

Owner.java.

```
public Pet getPet(String name, boolean ignoreNew) {
        name = name.toLowerCase();
        for (Pet pet : getPetsInternal()) {
            if (!ignoreNew || !pet.isNew()) {
                String compName = pet.getName();
                compName = compName.toLowerCase();
                if (compName.equals(name)) {
                    return pet;
                }
            }
        }
        return null;
    }
```

Take a moment, and try to understand what this method is doing. Its a handfull of lines, yet, its not easy to grasp it completely. Let us do something about it.

4.3. Extracting Methods

As said earlier, whenever we have a function that accepts a boolean argument, we're in fact having two functions, one for the `true`, and one for the `false` case.

The first thing to do is to extract these functions. IDEs help a lot with this kind of refactors.

The `getPet` method is called from two places, but for one of the two `getPet(String name)` the IDE is showing that is not being used at all.

Clean Code rules are simply in this regard, we should remove unused functions. This leaves us with a single call, where the `ignoreNew` is set to `true`.

If we know that `ignoreNew` is always true, we know that `!ignoreNew` is always false. This means that we can simply the if condition and remove the argument.

Our method now looks like:

Owner.java.

```
public Pet getPet(String name) {
        name = name.toLowerCase();
        for (Pet pet : getPetsInternal()) {
            if (!pet.isNew()) {
                String compName = pet.getName();
                compName = compName.toLowerCase();
                if (compName.equals(name)) {
                    return pet;
                }
            }
        }
        return null;
    }
```

4.4. Functions Should be Small

Common reasoning and knowing a language also helps. If we want to make the above functoin smaller, we can simply notice that calls to `lowerCase()` can be omitted by switching to `equalsIgnoreCase` in the condition.

The method now looks:

Owner.java.

```
public Pet getPet(String name) {
        for (Pet pet : getPetsInternal()) {
            if (!pet.isNew()) {
                if (pet.getName().equalsIgnoreCase(name)) {
                    return pet;
                }
            }
        }
        return null;
    }
```

4.5. Remove Tabs

The nested constructs such as nested if statements, nested loops or combination of if statements and loops are know to imped readability. By extracting functoins, we can always limit nesting to a single level. However, in the particular case, Java 8 stream constructs help us tackle all of the nesting with something like this:

```
public Optional<Pet> getPet(String name) {
       return getPetsInternal().stream()
            .filter(p -> !p.isNew())
            .filter(p -> p.getName().equalsIgnoreCase(name))
            .findFirst();
    }
```

Note that the we changed the return type of a function. We're not returning `null` anymore, and thus, the caller is not forced to do a null check. A simple call to `isPresent` on the optional instance is good

4.6. Extract Boolean Clauses

The final touch on the method doesn't really belong to function related clean code rules, rather to naming rules (we're yet to speak about them). Still, we'll mention it here. A nice and safe readability booster happens when we extract predicate or if conditions into appropriately named methods.

Final look on the method is:

```
public Optional<Pet> getPet(String name) {
       return getPetsInternal().stream()
            .filter(isAlreadyExisting())
            .filter(hasSameName(name))
            .findFirst();
    }
```

Let us try to answer the initial question now. What is `getPet` method doing? Retrieves already existing pet with a matching name.

5. Comments

Do good developers comment their code? Do comments make the code easier to understand?

Clean Code frowns upon comments. I find the common clean code comment phrases quite amusing.

First, comments are failures. Failures to express the intent in code.

Second, comments are lies waiting to happen.

On top of this, avoid comments that are redundant. They don't any additional value to the code they are referring to. Finally, they inevitably fall out of sync with the code.

Let us see if these statement hold true when reviewing pet clinic example

5.1. Comments are Failures

Whenever you can you should express the comment in code. Observe this example:

VetController.java.

```
@GetMapping("/owners")
    public String processFindForm(Owner owner, BindingResult result,
 Map<String, Object> model) {

        // allow parameterless GET request for /owners to return all
 records
```

```
        if (owner.getLastName() == null) {
            owner.setLastName(""); // empty string signifies broadest
 possible search
        }

        // find owners by last name
        Collection<Owner> results =
 this.owners.findByLastName(owner.getLastName());
        if (results.isEmpty()) {
            // no owners found
            result.rejectValue("lastName", "notFound", "not found");
            return "owners/findOwners";
        } else if (results.size() == 1) {
            // 1 owner found
            owner = results.iterator().next();
            return "redirect:/owners/" + owner.getId();
        } else {
            // multiple owners found
            model.put("selections", results);
            return "owners/ownersList";
        }
    }
```

Can be refactored to:

VetController.java.

```
@GetMapping("/owners")
    public String findOwners(Owner owner, BindingResult result,
 Map<String, Object> model) {
        String searchTerm = owner.getLastName() != null ?
 owner.getLastName() : FIND_ALL;
        Collection<Owner> owners =
 this.owners.findByLastName(searchTerm);
        if (isEmpty(owners)) {
            result.rejectValue("lastName", "notFound", "not found");
            return "owners/findOwners";
        } else if (hasUniqueOwner(owners)) {
            owner = owners.iterator().next();
            return "redirect:/owners/" + owner.getId();
        } else {
            model.put("selections", owners);
            return "owners/ownersList";
        }
    }
```

We did the following, we expressed the `// no owners found` comment, in the code as `isEmpty(owners)`.

We also refactored to hacky, confusing and errorProne statements:

```
// allow parameterless GET request for /owners to return all records
if (owner.getLastName() == null) {
    owner.setLastName(""); // empty string signifies broadest possible
 search
}
```

to something like

```
String searchTerm = owner.getLastName() != null ? owner.getLastName() :
 FIND_ALL;
```

There's a few more changes that follow the same line of thinking, but you've caught the gist by now.

Indeed, most of the comments can truly be expressed in code. It takes some will to do it though.

5.2. Comments Lie

While it may sound funny, comments are indeed lies to be happen'. Let's revisit again the `getPet` method:

Owner.java.

```
/**
     * Return the Pet with the given name, or null if none found for
 this Owner.
     *
     * @param name to test
     * @return true if pet name is already in use
     */
    public Pet getPet(String name, boolean ignoreNew) {
        name = name.toLowerCase();
        for (Pet pet : getPetsInternal()) {
            if (!ignoreNew || !pet.isNew()) {
                String compName = pet.getName();
                compName = compName.toLowerCase();
                if (compName.equals(name)) {
```

```
                    return pet;
                }
            }
        }
        return null;
    }
```

Check the method comment `@return true if pet name is already in use`. In fact, `getPet` is never returning `true`. Perhaps it used to, but this has changed and the comment left standing and missinforming.

5.3. Avoid Redundant Code

> Comments should say things that the code cannot say for itself.
>
> — Robert C. Martin *Clean Code - A Handbook of Agaile Software Craftmanshift*

At the very gates of Spring PetClinic Application, there stands the:

PetClinicApplication.java.

```
/**
 * PetClinic Spring Boot Application.
 *
 * @author Dave Syer
 *
 */
@SpringBootApplication
public class PetClinicApplication {

    public static void main(String[] args) {
        SpringApplication.run(PetClinicApplication.class, args);
    }

}
```

Check the class comment and see what does it say, that can't be inffered by reading the first two lines of code already.

Even the `author Dave Syer` (Dave, if you ever read this, love your work :)), is redundant. Its enough to consult commit history to learn that, yes, there's Dave, but also, there's Antoine Rey, Stephane Nicoll etc.

Redundant comments tend to clutter the code with details that are needless, therefore are better of removed.

5.4. Comments Fall Out of Sync

Observe the two methods from the `VetController.java`:

VetController.java.

```
@GetMapping("/vets.html")
  public String showVetList(Map<String, Object> model) {
      // Here we are returning an object of type 'Vets' rather than a
 collection of Vet
      // objects so it is simpler for Object-Xml mapping
      Vets vets = new Vets();
      vets.getVetList().addAll(this.vets.findAll());
      model.put("vets", vets);
      return "vets/vetList";
  }

  @GetMapping({ "/vets" })
  public @ResponseBody Vets showResourcesVetList() {
      // Here we are returning an object of type 'Vets' rather than a
 collection of Vet
      // objects so it is simpler for JSon/Object mapping
      Vets vets = new Vets();
      vets.getVetList().addAll(this.vets.findAll());
      return vets;
  }
```

You likely spot code duplication. The natural first reflex is to extract the duplicated code in a single method.

Say that you extract the method but capture the comment as well:

VetController.java.

```
@GetMapping("/vets.html")
  public String showVetList(Map<String, Object> model) {
      Vets vets = findVets();
      model.put("vets", vets);
      return "vets/vetList";
  }
```

```
 @GetMapping({ "/vets" })
 public @ResponseBody Vets showResourcesVetList() {
     // Here we are returning an object of type 'Vets' rather than a
collection of Vet
     // objects so it is simpler for JSon/Object mapping
     Vets vets = findVets();
     vets.getVetList().addAll(this.vets.findAll());
     return vets;
 }

 private Vets findVets() {
     // Here we are returning an object of type 'Vets' rather than a
collection of Vet
     // objects so it is simpler for Object-Xml mapping
     Vets vets = new Vets();
     vets.getVetList().addAll(this.vets.findAll());
}
```

However, you decide to go about it, you'll end up with one of the two comments falling out of sync, because, while the code is the same, the comments are different.

5.5. Comments Conclusion

As you can see, comments are overestimated. They rot quicker then the code.

This is not to say that you should never write a comment. If you're dealing with a non-expressive code such as regexp or properties files, comments can be of help. If you're commenting on a hack, or deal with a generated code, using comments is adding value.

Most of the times, however, comments can either be expressed in code, or be removed as redundant. Lowering the comments to the very essential has one positive consequence. Comments become more notable and better maintained. People will read the handful of comments, if they know they are there for a good reason.

6. Names

Naming Rules is my favorite set of rules. They provide great benefits at low cost. Name refactorings rarely cause things to break and with modern IDEs are quite an easy operation.

Proper naming has one major objective. You should be able to have a reasonable expectation of what a class or a method is doing, just by reading its name.

6.1. Names in Spring PetClinic

We'll be checking class, method and variable names along the following checklist:

Naming issues

- ❏ names that are not properly revealing the intent
- ❏ declarations that are far from the actual usage
- ❏ long names of short scoped variables
- ❏ short names of long scoped variables
- ❏ names clarified with comments
- ❏ names with encodings
- ❏ names with meaningless suffix
- ❏ names with meaningless prefix

Namings in Spring PetClinic are quite according to the Clean Code standards. There are no big findings.

Spring organizes its code by feature packages. The layout can be portrayed as the following:

6.2. Use Solution Domain Names

> Remember that the people who read your code will be programmers. So go ahead and use computer science (CS) terms, algorithm names, pattern names, math terms, and so forth.
>
> — Robert C. Martin *Clean Code - A Handbook of Agaile Software Craftmanshift*

The listed concepts comprise the solution domain. I would say that this idea applies for well known framework concepts as well. If you are a Spring developer,

you know what to expect when you see `Controller` `Service` or `Repository` in a class name.

6.3. Inspecting Spring PetClinic Class Names

With the solution domain rule in mind, let us inspect the class names within the `owner` package:

Anyone with more than one spring project under his belt, should have no issues with `OwnerController`, `OwnerRepository`, `PetConroller`, `PetRepository`, `VisitController`. Even `PetTypeFormatter` and `PetValiator` are common concepts, only not used as often.

This leaves us with `Owner` and `Pet`. I would bet five bucks that they are model classes, entities even. I would win the bet, they are, so all expected here.

Let us try with `vet` package:

Do you notice something peculiar here? I was inspecting class names exactly in the similar fashion, and I did get confused with `Vets` class. If we have a `Vet` model class, why would we have `Vets` class. What would be a difference between a `Vets` and a list of `Vet` instances?

By opening the `Vets.java`[10] class, we see that its a model that is passing the data to the view, but there's nothing in the name that implies this.

6.4. Inspecting the Method Names

On a method level, we can say that PetClinic is doing a pretty good job. Let us examine the `OwnerController.java` method names:

OwnerController.java.

[10] https://github.com/interventure-growingtogether/spring-petclinic-clean-code/blob/original/src/main/java/org/springframework/samples/petclinic/vet/Vets.java

```
setAllowedFields
initCreationForm
processCreationForm
initFindForm
processFindForm
initUpdateOwnerForm
processUpdateOwnerForm
showOwner
```

Each method name starts with a verb. We quickly notice a pattern btw. `init` and `process` used interchangeably.

Slightly more engaging is figuring out what the methods of the `VetController.java` are doing:

VetController.java.

```
showVetList
showResourcesVetList
```

What is the difference between the two methods? After inspecting the code, I believe that better names would be:

VetController.java.

```
showAllVetsInView
findAllVets
```

Guess what the methods are doing based on the original names, and then based on the changed names

6.5. Inspecting Variable Names

Finally, let's examine variable names. If I'm not missing anything of the obvious, the only naming issue worth commenting, we can find in the otherwise ugly `getPetName()` method:

Owner.java.

```
    public Pet getPet(String name, boolean ignoreNew) {
        name = name.toLowerCase();
        for (Pet pet : getPetsInternal()) {
```

```
            if (!ignoreNew || !pet.isNew()) {
                String compName = pet.getName();
                compName = compName.toLowerCase();
                if (compName.equals(name)) {
                    return pet;
                }
            }
        }
        return null;
    }
```

Notice the `compName` variable. The name is meaningles. Considering the short scope, having something like `pn` would be a better fit. We discussed this complete method in the Functios section, so in any case it doesn't exist in the `clean-code` version.

6.6. Naming Conclusion

Naming is power that we should learn how to use. We want to give ourselves and others, the option to have proper expectations based on the name without having to reach for the code. Naming refactors are typically safe, and its quite common that as we go, we find better names and change them. This accounts for continuous renaming.

Spring PetClinic does look as the application that followed this routine.

7. Conclusion

What we learned by code cleaning Spring Petclinic is that coverage tool and high test coverage rate is not enough. We must make sure that we test everything that can possible break, whatever that might be in practice.

Through our review of names, functions and comments we learned that its easy to gain many little wins, that are quick and safe to do, and leave the code simpler, readable and easier to maintain.

There's no proper excuse of not learning clean code routines. Being critical of the rules is understandable. Not taking the time to learn them, is simply irresponsible and lazy. When it comes to practicing and applying the rules, for most of the time, the only thing needed, is to care enough and to do the changes.

> Clean code always looks like it was written by someone who care
>
> — Michael Fathers *Author of Working Effectively with Legacy Code*