Programming Standards for Senior Design Project
Regarding Groovy and Grails

Reference:
https://tedvinke.wordpress.com/2015/03/15/basic-groovy-and-grails-code-review-guidelines/

General

- Does the code **fulfil the specifications or acceptance criteria** written in the user story, or is there a gray area?
- Do you **understand** what's been programmed? Would you be able to make modifications to it later on?
- Have the **proper Grails and Groovy constructions** been used to realize the proper things?
- Is the **code readable** and does it adhere to our guidelines of code style and naming of things? See following sections.

Naming

Use the proper structures and names. Use the correct names of Grails artifacts, such as Domain Classes, Controllers, Services, TagLibs and methods and variables.

- Packages should start with `nl.<company>` followed by application name, such as `nl.first8.myapp`

- Classes start with CAPITALS, such as `Animal`, methods and variables are written in *camelCase*, such as `displaySummary()` or `animalInstanceList`
- Try to avoid abbreviations. Prefer `domainCode` over `dmnCd`
- Does the name actually cover the subject?

Readability

General readability- and formatting guidelines to read code *easy* and have it written in a consistent manner. A part can be done automatically with the built-in *Formatter* in the IDE, e.g. `Ctrl-Shift-F` in Eclipse or GGTS, but not everything.

Not all one-liners

In example below, split up e.g. the initialization of variables, in order for its usage to become more clear.

```
out << body() << ( (attrs.int('offset')) + "-" + ( Math.min(attrs.int('count'), attrs.int('offset') + attrs.int('max') ) ) + " " + title + " " + attrs.int('count') )
```

could become

```
int start = attrs.int('offset')
int total = attrs.int('count') )
int end = Math.min(total, start + attrs.int('max') )

out << body() << start + "-" + end + " " + title + "
" + total
```

Spock tests and labels

There's plenty to say about "proper" writing Spock tests, but that's beyond the scope of this post.

Use the proper labels in Spock tests for readability. Use `and:`to separate multiple labels – with additional "textual description" after them – to make them stand out. Write

```
when: "searching for existing animal"
...
then: "we have results"
model.animalSearchResults.animalsCount == 1
flash.successMessage == "animal.all.found.message"
session.selected.size() == 1
```

as

```
when: "searching for existing animal"
...

then: "one found animal is returned"
model.animalSearchResults.animalsCount == 1

and: "all found message is displayed"
flash.successMessage == "animal.all.found.message"

and: "animal has been auto-selected"
session.selected.size() == 1
```

White-space

Use white-spaces to separate combined statements. Write

```
Integer.valueOf(params.offset?:attrs.offset)
```

as

```
Integer.valueOf(params.offset ?: attrs.offset)
```

White-space and brackets

Before and after brackets in general you do NOT have to use white spaces.

Write

```
if ( params )
if ( total > 0 )
if ( end < begin )
```

as

```
if (params)
if (total > 0)
if (end < begin)
```

Curly brackets

Use curly brackets for one-liners. Write

```
if ( end < begin )
      end = begin
```

as

```
if (end < begin) {
  end = begin
}
```

Although you can write an if-statement on one line or just the line below without the curlies, especially when there are multiple statements after each other, it will improve readability and shows intent more clearly.

Compare

```
if (end < begin)
end = begin
end = end + 1
```

with

```
if (end < begin) {
  end = begin
}
end = end + 1
```

Comments

Use comments to state the purpose of classes and methods, clarify difficult pieces of code and document the decisions (“why?”) made. For Java en Groovy code we can use Javadoc to generate API documentation in HTML format in the source code.

Javadoc on classes

Use Javadoc comments at the top of a class to describe the general purpose, such as

```
/**
 * General convenience tags for layout - header,
body and footer - purposes.
 */
class LayoutTagLib {
```

Javadoc on methods

Use Javadoc on public methods to describe the purpose and its parameters. You can use `@param` for parameters, `@return` for what it returns, when or if exceptions are thrown with `@thrown` etc.

```
/**
 * Gets the user for specified code and role.
 *
 * @param code The code, either username or email
address
 * @param role The role identification e.g. A, B or
C. Default is A.
 * @return the user or null if not found
 */
User findUser(String code, String role = "A")
```

Clean up

Remove obsolete comments, e.g. which got left behind, or which are plain wrong, or correct them!

Simplicitly

Is the code simple and does it do its work in the most simple way….but not simpler

Intuitive API

Use an intuitive API. Use similar structures or naming. Is only the actual required input needed and are for others *sensible* defaults being used? If you’re creating a `betterPaginate` tag, don’t burden the user too much with all kinds of input. Have him instead of

```
<g:betterPaginate offset="${params.offset?:0}“
count="${results.animalsCount?:0}"
max="${params.max?:0}"/>
```

allow to use your tag as

```
<g:betterPaginate count="${results.animalsCount}" />
```

where *you* take care of the defaults, and the reading from `params`etc.

Simple writing

The more complicated lines used, the more difficult it is to understand what happens and where changes should go. It turns out that the following 10 lines created by a co-worker

```
int offset = 0
int total  = 0
int max  = 0

if (params) {
    offset = Integer.valueOf(params.offset ?: attrs.offset ?: 0)
    max  = Integer.valueOf(params.max ?: attrs.max ?: 0)
} else if (attrs) {
    offset = Integer.valueOf(attrs.offset ?: 0)
    max  = Integer.valueOf(attrs.max ?: 0)
}
```

```
     total = Integer.valueOf(attrs.total ?: 0)
```

can be rewritten in just 3 lines with the same behavior:

```
int offset = Integer.valueOf(params.offset ?: attrs.offset ?: 0)
int max = Integer.valueOf(params.max ?: attrs.max ?: 0)
int total  = Integer.valueOf(attrs.total ?: 0)
```

If you have a good test, you can refactor somewhat more freely these kinds of constructions and still be fairly confident you didn't change anything unexpectedly.