|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| UMGC Capstone | | | | |
|  |  | | |  |
| DevSecOps | | | | |
|  | | Plan |  | |

TABLE OF CONTENTS

[Introduction 3](#_Toc52111529)

[1. Getting started 4](#_Toc52111530)

[*Setting up your environment using Make* 5](#_Toc52111531)

[2. Git and GitHub 6](#_Toc52111532)

[Process Overview 6](#_Toc52111533)

[Criteria for Merging to Master 7](#_Toc52111534)

[Guide 7](#_Toc52111535)

[Best Practices 8](#_Toc52111536)

[3. Testing 9](#_Toc52111537)

[Expectations 9](#_Toc52111538)

[Test Examples 10](#_Toc52111539)

[Integration Test Example 12](#_Toc52111540)

[Unit Test 12](#_Toc52111541)

[Writing Testable Code With Dependency Injection 13](#_Toc52111542)

[How to Uncouple the World 14](#_Toc52111543)

[4. Coding Standards 15](#_Toc52111544)

[Java 15](#_Toc52111545)

[Enforcement 15](#_Toc52111546)

[5. Code Quality And Security 16](#_Toc52111547)

[SonarQube 16](#_Toc52111548)

[Discussion 16](#_Toc52111549)

# Introduction

This document outlines the standards, best practices, and processes that are to be adhered to throughout the course of the UMGC software engineering capstone project by each development team. To be successful in the course, teams will be required to utilize every tool they have acquired throughout the course of their post-graduate education at UMGC. The primary audience of this document is the development teams. This document is intended to guide the development teams on propper practices and ensure development proceeds as smoothly as possible.

This remainder of this document will operate within the context of the UMGC software engineering capstone project context, referring to the business as the class as a whole and project teams as development teams. The purpose of this document is to inform, and thus the remainder of the document will be

Revision History

|  |  |  |  |
| --- | --- | --- | --- |
| Name | Date | Reasons for change | Version |
| Dustin Emerson | 9/5/20 | Initial Draft | V 0.1 |
| Dustin Emerson | 9/12/20 | Add Section on Testing | V 0.4 |
| Taylor Goodlett | 9/13/20 | Start Fleshing Out Style Guide | V 0.5 |
| Taylor Goodlett | 9/15/20 | Fix GitHub Diagram | V 0.6 |
| Dustin Emerson | 9/16/20 | Clean Up | V 0.7 |
| Taylor Goodlett | 9/16/20 | Finalize for initial release to DevTeams | V 0.8 |
| Dustin Emerson | 9/27/20 | Create and release Version 1 | V 1.0 |

# Getting started

Our key goal is to enable the development teams to be successful by shortening the development lifecycle through automated continuous delivery and ensure a high level of quality through automated testing. DeveSecOps encourages all teams to set up their environment how they see fit. However, having the same tools the DevSecOps team is using in the Azure Pipeline will allow you to tap into the benefits of our toolchain on your local machine. The DevSecOps team is utilizing the following tools to build and deploy software:

**Required**:

* *git*– Source control (Note: every developer is required to use git)

**Optional**:

* *Linux*– Our Operating system (OSX will also suffice)
  + Ways of getting Linux on Windows:
    - Windows Subsystem for Linux (WSL) – Requires Windows 10 professional edition. We recommend Core OS if you are going this route.
    - Oracle VM Virtual Box – Easy to use and free virtual machine to run Linux on.
* *Bash*– Our shell
* *Make* – Build automation tool
* *Docker* – Containerization

**Note**:

The above tools are not required for you to develop. You are welcome to set up your development environment as you see fit. This is simply a helping hand to get set up. If you are planning on setting up your environment yourself, you can skip the next subsection. Keep in mind you will need to locate and install the dependencies of your application yourself.

## *Setting up your environment using Make*

Requirements:

* *Bash*
* *Make*
* *Docker*

Depending on the project the make commands could vary. For example, the chatbot team has a separate docker repository with a static database that only has two commands. For this reason, the only command you will ever truly need to get started is the help command:

# Display makes recipes and targets.

~$ make help

Running this command will provide you with the documentation required to set yourself up for success.

# Git and GitHub

Throughout the twelve weeks of this course, development teams will use git as their version control system. In addition to git, teams will utilize GitHub for hosting their work, tracking bugs, and facilitating code reviews.

## Process Overview

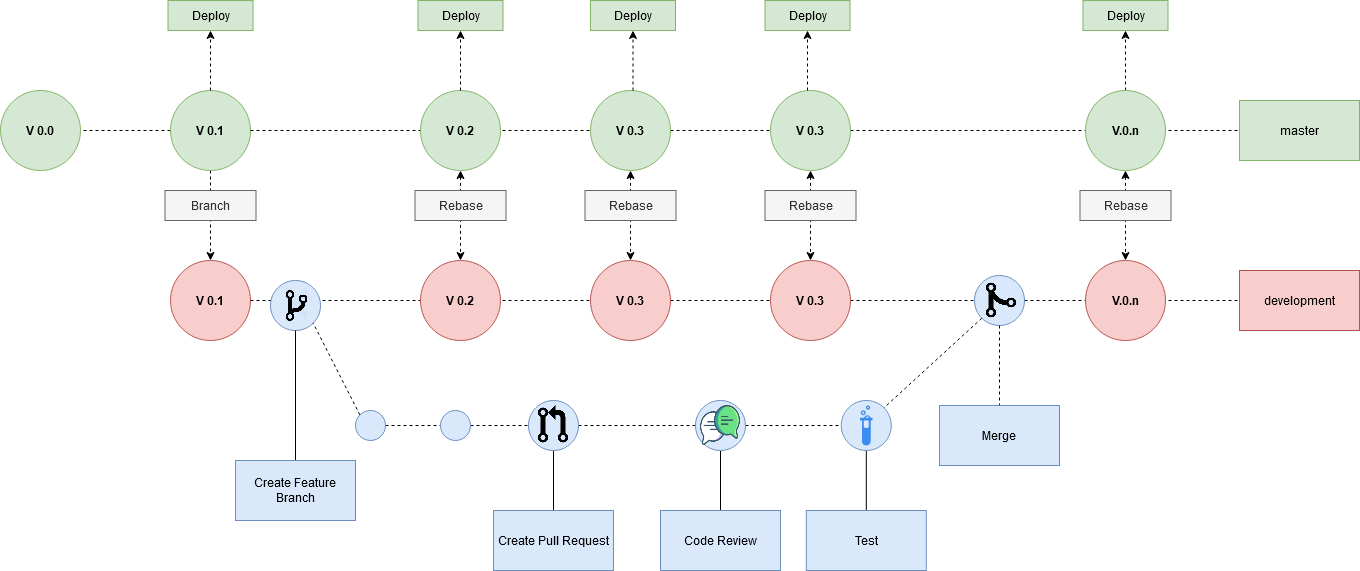


Figure n Version Control Process Overview

Projects repositories will be set up as such:

1. There is a mainline **master** branch that requires a member of DevSecOps to approve code being pushed to it. This branch will be pipelined as such where the project deploys after a build is successful.
2. There is a **development** branch forked off of master. This is the branch the dev teams will use when working. The team can decide how often they wish to move the development branch into master.

Development will be performed using a pull request model. Developers will checkout a new branch on their local machine off of the development branch when they begin developing a new feature. When the feature is ready to be presented for review, they will push their branch to GitHub and create a pull request for peer review. Once a member on their team and a member of DevSecOps has approved, the pull request will be tested. Finally, once passing tests, the branch will be merged into master.

## Criteria for Merging to Master

* The pull request must build.
* The code being added must pass a LINT evaluation.
* The code added or changed must be tested.
* The pull request cannot contain any confidential or personal information (passwords, social security, etc.)
* The pull request cannot contain binary files.
* 70%-80% Code Coverage in tests

## Guide

1. Ensure the repository is cloned on your local machine:

git clone https://github.com/umgc/umgc.project.n.git

1. Checkout the development branch:

git checkout development

git pull origin development

1. Checkout a new branch for your feature

git checkout -b my-feature

1. Develop feature
2. Push branch to GitHub

git push origin **my-feature**

^ This is your branch name from earlier

1. Create a pull request on GitHub
   1. Follow [GitHub's documentation](https://docs.github.com/en/github/collaborating-with-issues-and-pull-requests/creating-a-pull-request) for creating a pull request.
2. Incorporate Feedback from team and DevSecOps
3. Once the pull request is accepted, and tests pass, your work is ready to be merged into production.
   1. Follow [GitHub's documentation](https://docs.github.com/en/github/collaborating-with-issues-and-pull-requests/merging-a-pull-request) for merging a pull request.

## Best Practices

Outside of the process outlined above, which is enforced by rules written by DevSecOps for the repositories, teams are free to use git as they see fit. However, in order to prepare oneself for a career in software development, some best practices are outlined below:

Commit Messages:

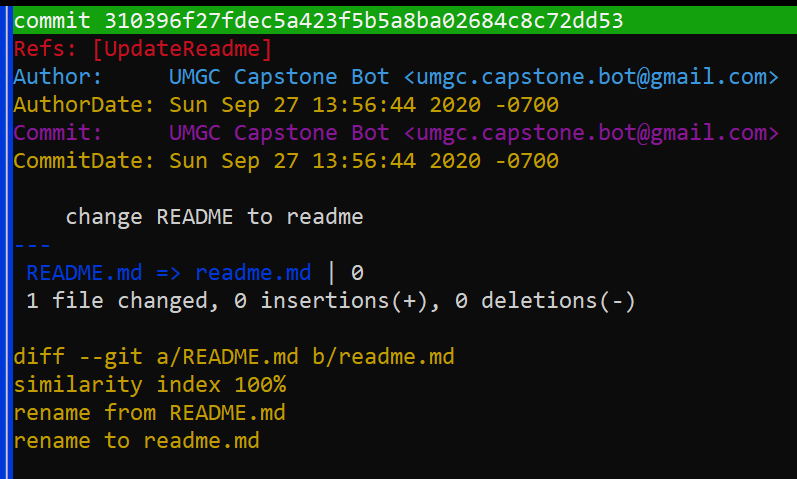
Development teams shall do their best to adhere to the git [documentation](https://git-scm.com/docs/git-commit)'s advice that commits messages shall consist of a "single short (less than 50 character) line summarizing the change, followed by a blank line and then a more thorough description" (Git, 2020).

Branch Names:

Development teams shall do their best to name their branches in such a way that it reflects the purpose of the code added in the merge (i.e., add-context-menu or fix-oom-in-filter).

Changing File Names:

Depending on the case sensitivity of the target file system you could really destroy someones local repository by making commit where tou change filename.java => FileName.java



If someone is on a Mac, which is case insensitive but case preserving, they will be unable to get their operating system to agree that it is up to date with origin. If you must perform the above either rename it to something unique that has the correct casing, or make a commit where you delete the file, and make a separate commit on top where you add it back with the correct name.

# Testing

In order to ensure the code produces is complete, we require all features to be tested. There are two types of tests we are interested in:

1. Integration Tests which tests a collection of components (Required)
2. Unit Tests which test a specific component

Sometimes it's very difficult to correctly Unit Test software due to high coupling between classes. In order to not slow down development, at minimum, Integration tests are required. However, one should aspire to have unit tests for their components and integration tests for their features.

The testing framework we will be using is [JUnit](https://junit.org/junit5/).

## Expectations

We expect the following to be accomplished with your tests.

* Every feature shall at least have an integration Test.
* At least 70% code coverage shall be required.
* Test performance is reliable and does not intermittently fail.
* Tests are atomic and do not require earlier tests to run in order to succeed.
* Tests clean up for themselves. All data produced during test execution shall be cleaned up by the time the test completes.

## Test Examples

Below is a simple application written to demonstrate testing software called `AddThings.` The application simply adds objects together. What that means is dependent on the Adder used.

AddThings.java

*/\*\*  
 \* Adds objects together if possible  
 \*/*public class AddThings {  
 private ArrayList<Adder<?>> adders = new ArrayList<>() {{  
 adders.add(new NumberAdder());  
 adders.add(new StringAdder());  
 }};  
  
 */\*\*  
 \* Will add two objects together and return the desired result as defined by  
 \* the appropriate Adder  
 \*  
 \** ***@param*** *an object to be added  
 \** ***@param*** *b object to be added  
 \** ***@return*** *a product of adding a and b  
 \*/* Optional<Object> add(Object a, Object b) {  
 if (!isSameType(a, b))  
 throw new IllegalArgumentException(  
 "Objects are not the same type { \n" +  
 "\t a is " + a.getClass() +  
 "\t b is " + b.getClass()  
 );  
  
 for (Adder adder : adders) {  
 if (adder.isType(a)) {  
 return Optional.*of*(adder.add(a, b));  
 }  
 }  
  
 return Optional.*empty*();  
 }  
  
 */\*\*  
 \* Returns if a  
 \** ***@param*** *a the object with which to compare its types  
 \** ***@param*** *b the reference object with which to compare type to a's type  
 \** ***@return*** *{****@code*** *true} if this a's class is the same as the b's class;  
 \* {****@code*** *false} otherwise.  
 \*/* boolean isSameType(Object a, Object b) {  
 return a.getClass().equals(b.getClass());  
 }  
}.

Adder.java

*/\*\*  
 \* Adds two objects of type {****@code*** *T} together  
 \** ***@param*** <*T*> *The type of object the Adder will handle  
 \*/*public interface Adder<T> {  
 T add(T a, T b);  
 boolean isType(Object o);  
}

NumberAdder.Java

*/\*\*  
 \* Adds numbers together, what will they think of next  
 \*/*public class NumberAdder implements Adder<Number> {  
 */\*\*  
 \* Adds {****@param*** *a} and {****@param*** *b} together such that  
 \*  
 \* a + b -> c or 1 + 2 -> 3  
 \*  
 \** ***@param*** *a a number  
 \** ***@param*** *b another number  
 \** ***@return*** *the sum of {****@param*** *a} and {****@param*** *b}  
 \*/* @Override  
 public Number add(Number a, Number b) {  
 return a.intValue() + b.intValue();  
 }  
  
 */\*\*  
 \* Determines if the object passed in is a Number  
 \*  
 \** ***@param*** *o the <i>alleged</i> Number  
 \** ***@return*** *true if {****@param*** *o} is a Number;  
 \* otherwise false  
 \*/* @Override  
 public boolean isType(Object o) {  
 return o instanceof Number;  
 }  
}

StringAdder.Java

*/\*\*  
 \* Adds strings together, what will they think of next!  
 \*/*public class StringAdder implements Adder<String> {  
 */\*\*  
 \* Adds, or rather concatenates, {****@param*** *a} and {****@param*** *b} together such that  
 \*  
 \* a + b -> ab or "Oh Hi" + "Mark" = "Oh HiMark"  
 \*  
 \** ***@param*** *a a String  
 \** ***@param*** *b another String  
 \** ***@return*** *concatenated value of {****@param*** *a} + {****@param*** *b}  
 \*/* @Override  
 public String add(String a, String b) {  
 return a + b;  
 }  
  
 */\*\*  
 \* Determines if the object passed in is a String  
 \** ***@param*** *o the <i>alleged</i> String  
 \** ***@return*** *true if {****@param*** *o} is a String;  
 \* \* otherwise false  
 \*/* @Override  
 public boolean isType(Object o) {  
 return o instanceof String;  
 }  
}

## Integration Test Example

Integration tests are the easiest to As described above; an integration test shall test a collection of components. For this example, we will be testing AddThings.java using, including its dependency on its Adders.

class AddThingsIntegrationTest{  
 @Test  
 public void AddNumber() {  
 *assertEquals*(3, new AddThings().add(1, 2).get());  
 }  
  
 @Test  
 public void AddString() {  
 *assertEquals*("TacoPocket",  
 new AddThings().add("Taco", "Pocket").get());  
 }  
  
 @Test  
 public void AddNotSupported() {  
 *assertEquals*(Optional.*empty*(), new AddThings().add(new Object(), new Object()));  
 }  
  
 @Test  
 public void ExceptionThrownWhenArgumentsDoNotMatch() {  
 *assertThrows*(IllegalArgumentException.class,  
 () -> new AddThings().add(1, "Hi Mom"));  
 }  
  
}

A key aspect that should be noted in the above example is that this test verifies the correctness of the entire suite of components, not just a single class.

## Unit Test

A Unit Test, by comparison, will only test a specific component. In this exercise, we will write a unit test for StringAdder.java

class StringAdderTest {  
  
 @Test  
 void add() {  
 *assertEquals*("YabbaDabba", new StringAdder().add("Yabba", "Dabba"));  
 *assertEquals*("nullDabba", new StringAdder().add(null, "Dabba"));  
 *assertEquals*("Yabbanull", new StringAdder().add("Yabba", null));  
 *assertEquals*("nullnull", new StringAdder().add(null, null));  
 }  
  
 @Test  
 void isType() {  
 *assertTrue*(new StringAdder().isType("Hi Mom"));  
 *assertFalse*(new StringAdder().isType(24));  
 }  
}

## Writing Testable Code With Dependency Injection

Often developers develop with the idea of testing as an afterthought, something to be considered when the component is finished being written. Doing so often leads to components that are difficult to debug due to high coupling. In the below guide, we will illustrate how to compose one's component in such a way that dependencies are injected at runtime. This will permit the injection of mock dependencies during test execution.

In the above example, it might occur to you that creating a pure Unit Test for AddThings would be difficult. This is due to the high coupling between AddThings.java and the Adders. This is illustrated here:

public class AddThings {  
 // We couple AddThings to the Adders here:  
 private ArrayList<Adder<?>> adders = new ArrayList<>() {{  
 add(new NumberAdder());  
 add(new StringAdder());  
 }};

How could we straighten this up so that we can have a Unit Test for AddThings.java in addition to its integration test? The answer is we can inject the dependencies during its construction.

*/\*\*  
 \* Adds objects together if possible  
 \*/*public class AddThings {  
 // We couple AddThings to the Adders here:  
 private ArrayList<Adder<?>> adders = new ArrayList<>();  
  
 public AddThings() {  
 this(new NumberAdder(), new StringAdder());  
 }  
  
 // Inject dependency here  
 AddThings(Adder<?>... adders) {  
 this.adders.addAll(Arrays.*asList*(adders));  
 }

A quick note on what's going on here: Initially, when we first wrote AddThings.java, we baked in the behavior that it had a list of Adders containing an instance of NumberAdder.Java and StringAdder.Java. No matter what we made, that list would contain those instances as soon as the constructor was called. By rewriting the class such that those *dependencies* are supplied by a default constructor, which in turn would take an array of adders, we have created a loosely coupled class that can have those dependencies supplied if we so wish.

This allows us to inject a mock object into our class during test execution shown below:

class AddThingsTest {  
 private Adder<String> mockAdder = new Adder<>() {  
 @Override  
 public String add(String a, String b) {  
 return ("taco".equals(a) && "bell".equals(b))  
 ? "taco bell" : "Doritos";  
 }  
  
 @Override  
 public boolean isType(Object o) {  
 return o instanceof String;  
 }  
 };  
  
 @Test  
 void add() {  
 *assertEquals*("taco bell",  
 new AddThings(mockAdder).add("taco", "bell").get());  
 *assertEquals*("Doritos",  
 new AddThings(mockAdder).add("McDonald's", "milk").get());  
 }  
}

## How to Uncouple the World

This practice can be taken even further if you explore functional interfaces. For example, imagine you have a method that is meant to accept a userID and return an email. In order for this to be properly unit tested, we can use the Function Interface.

Instead of thinking of our operation as a call to the database, performing a query, retrieving a result, and returning the email. We can think of it as a function that accepts a String and Returns a String.

import java.util.function.Function;  
  
public class ClassyClass {  
 private Function<String, String> userEmailProvider;  
   
 public ClassyClass() {  
 Function<String, String> myUserEmailProvider = new Function<>() {  
 @Override  
 public String apply(String uid) {  
 // Connect to database  
 // query for uid  
 // return users email  
 return "taco@timmycorp.com";  
 }  
 };  
 }  
   
 // Inject your own email provider here:  
 ClassyClass(Function<String, String> userEmailProvider) {  
 this.userEmailProvider = userEmailProvider;  
 }  
}

# Coding Standards

## Java

We will be conforming to the Google Java Style Guidelines.

<https://google.github.io/styleguide/javaguide.html>

There are instructions located on the link above for integrating the style guidline with whatever IDE you are using.

## Enforcement

We will be enforcing this guidline using [spotless](https://github.com/diffplug/spotless/tree/main/plugin-maven#google-java-format), a maven/gradle plugin that will verify your coding convesions on build. When building your project, you will see output like the following if you are not conforming to the style :

user@machine repo % mvn spotless:check

[ERROR] > The following files had format violations:

[ERROR] src\main\java\com\diffplug\gradle\spotless\FormatExtension.java

[ERROR] -\t\t····if·(targets.length·==·0)·{

[ERROR] +\t\tif·(targets.length·==·0)·{

[ERROR] Run 'mvn spotless:apply' to fix these violations.

user@machine repo % mvn spotless:apply

[INFO] BUILD SUCCESS

user@machine repo % mvn spotless:check

[INFO] BUILD SUCCESS

As displayed in the above example, spotless will even be able to fix your style violations for simple cases.

# Code Quality And Security

## SonarQube

The plan is for code quality and security will be enforeced with [SonarQube](https://www.sonarqube.org/). SonarQube makes use of static and dynamic code anysis to verify the following:

* Code Reliability
* Code Security
* Code Maintainability

## Discussion

In order to perform an accurate analysis Sonaqube needs to run it’s analysis over time. In order to do it needs to run on a server. This is a problem for DevSecOps as doing so will quickly run our Azure account out of money.   
  
Currently we are investigating a workaround for this issue. The current idea is that we can take what we learned form spinning up the static database for the chatbot team, and employ it for SonarQube. Basically, we would include SonarQube in your project. There will be a command you can run to spin up SonarQube on your local host.   
  
To get the best results we would need to see if we can commit updates to the SonarQube images locally to kind of hack in some information over time cheaply.