**Final Exam Review**
CPSC 2150

**Midterm 1**

**Encapsulation**—Grouping related data and operations (classes and objects). It helps us keep this data in sync

**Information Hiding**—How we control access to encapsulated data (visibility settings)

- This lets us ensure people set it correctly!
- Removes implementation details (separation of concerns)

**Separation of Concerns**—Every module has its own concerns and isolates that from the rest of the system

**Contract**—Outlines the responsibilities of the implementer (the function itself) and the client (who calls the function)

> *Precondition: The client's responsibility to the implementor.*
>
> *Postcondition:* The guarantees the implementer makes (if preconditions are met)
>
> *Invariants*: Something that is "always" true (usually about a class attribute). They are guaranteed to be true after the constructor is called, and before or after public methods, but private methods are not guaranteed!

**Entity Objects**—Represent some real world entity or abstract concept; usually a noun. *DrivetrainVelocityController*

**Boundary Objects**—Encapsulate the interactions with other systems (or the user). *Display*

**Control Objects**—Responsible for the actual flow of the program, sticks all the pieces together. *main(), opcontrol(), Routine*

**Interface**—Represents interactions between classes in a system. Describes the what classes do, but not the how. Expresses some sort of coherent concept. Interfaces allow for dynamic dispatch

**Declared Type**—The type of a variable at compile time, can be an interface, a class, or a primitive.

**Dynamic Type**—The type of a variable at runtime, must be a class or a primitive. Can use instanceof operator to determine runtime type

**Coding to Interface**—Put all your declared types in an interface. Benefits: Easier implementation swapping, and allows implementation to change without users needing to worry about it.

**Interface Specification**—Because the implementation is now swappable, we need to define all responsibilities in the interface itself. *Initialization Ensures* specifies what will be true after calling the constructor, *Defines* allow us to define a concept for the specification (usually maps to private variables), *Constraints* restrict how the defines can act (think invariants)

**Correspondences**—Allow us to connect *Defines* to private variables in a specific implementation.

**Midterm 2**

**<u>Testing</u>**

The process of verifying programs to be correct. The compiler provides a form of testing, to ensure that types match, and operations are possible, but correct programs always have stricter requirements than just compilation.

*Failure* — Any deviation of the observed behavior from the specified behavior

*Erroneous State* — the system is in a state such that further processing of the system will result in a failure.

*Fault* — the mechanical or algorithmic cause of the erroneous state (i.e. the bug)

**A fault leads to an erroneous state leads to a failure!**

*Testing* — The systematic attempt to find bugs in a planned way in the system. Cannot prove that faults do not exist, but can only show that faults do exist! You can only do this with formal verification (see below)

*Test Case* — A set of inputs and the associated expected output. Test cases show us the failure but don't show us where the bug in our code is.

> *Routine Test Case* — Tests the standard operation. Should be a simple input and output
>
> *Boundary Test Case* — Based on the contracts, test for some edge case or special number. This number should still follow the associated contract!
>
> *Challenging Test Case* — The types of test cases that, if you were writing, may find difficult or error-prone.

*Unit Testing* — Test all of the individual components of the software separately. For example, testing a single method in a class to ensure it has the correct output

- Focus on just one small part of the program

*Integration Testing* — Testing to ensure that multiple components of the system work together correctly. For example, ensuring that the entire class works as expected.

*System Testing* — Testing the entire end-user system

*Test Stub* — Partial implementation of lower-level components to aid in the testing of higher-level components. Can be avoided if you implement and test the lower-level

components first. *Example: a stub implementation of the database that gives the same response to each query to make sure your business logic works.*

*Test Driver* — Partial implementation of component that depends on the test component. Calls the test for that component to ensure it functions correctly.

**Final Review**

## String Theory

A string is a series of zero or more entries of any other mathematical type. It is not a type in Java, instead it's a mathematical structure, like a set.

An empty string is denoted by `<>`, or `empty_string`

***Concatenation***

Combines two strings together. Notated like `A o B`.

For example, if

`A = <1, 2, 3, 4>` and

`B = <4, 3, 2, 1>`, then

`A o B = <1, 2, 3, 4, 4, 3, 2, 1>`

***Length***

Gets the number of elements in the string. Notated like `|s|`

For example,

`|<1, 2, 3, 4, 5>| = 5`

`|empty_string| = 0`

***Reverse***

Reverses the order of the number of elements in the string. Notated like `Reverse(s)`

For example,

`Reverse(<1, 2, 3, 4, 5>) = <5, 4, 3, 2, 1>`

`Reverse(<>) = <>`

***Prt_Btwn***

Finds a substring of the given string, between two positions m and n. Includes m, does not include n. Notated like `Prt_Btwn(m, n, s)`

For example,

`Prt_Btwn(2, 4, <1, 2, 3, 4, 5>) = <3, 4>`

***Occurs_Ct***

Determines the number of times an item x occurs in the string. Notated like `Occurs_Ct(x, s)`

For example,

```
Occurs_Ct(2, <1, 2, 3, 2, 3, 4, 3, 2, 1, 2, 3, 4>) = 4

Occurs_Ct(7, <1, 2, 3, 2, 3, 4, 3, 2, 1, 2, 3, 4>) = 0
```

**Designing Large Systems**

*Requirements Drift — When programming complex systems, developers have a tendency to write code which slowly drifts away from its requirements.*

*Intentional Design — When working professionally, you will need to be careful, as projects can be enormous, spanning multiple teams over multiple years.* ***Old code doesn't go away!***

*Encapsulation — Group all related functionality together, so each module is responsible for different portions of the software*

*Information Hiding — How we mediate access to encapsulated information, with the purpose being to establish safeguards*

*Separation of Concerns — Make sure each component does one thing, and does it well! You should isolate side effects of your program as much as possible.*