Team 302
Software Design Principles

WET vs. DRY Principle

DRY == Don't Repeat Yourself
WET == Write Everything Twice

- DRY is good because that means there is a single place responsible for a task as opposed to being spread out. Be lazy by putting things in one place and write it once.
- WET is bad because that means if something changes, now you have to change multiple places.

SOLID Principle

Stands for:

- Single Responsibility Principle
- Open/Closed Principle
- Liskov Substitution Principle
- Interface Segregation Principle
- Dependency Inversion Principle

Single Responsibility Principle

A class should do only one thing (AND DO IT WELL!!)

- We all like the swiss army knife or multi-purpose tool, but do they really have the best screwdriver? We'd rather have a toolbox full of the best tools instead of one tool that does everything (think of each tool as a class).
- KISS (Keep It Super Simple) Principle
- Should be easy to document the class without using conditional terms

Single Responsibility

Separate classes to:

- Read Joystick inputs
- Set Chassis Motors

As opposed to one class that reads the Joystick inputs and sets the Chassis Motor Speeds

Open/Closed Principle

Software should be open for extensions but closed for modifications

- Don’t Ask/Don’t Tell
- Attributes are private
- Accessors/mutators (getters/setters) are only written when needed (try to avoid as much as possible)
- Design code such that as more functionality is added existing functionality doesn’t have to change
- Use Interfaces instead of concrete implementation

Open/Closed Principle

Shooter Aiming Example

- Creating an interface for setting the angle that doesn't assume any particular sensor type (potentiometer, encoder, limit switch, etc.)

Chassis Drive Class: Control Drive Motors Example

- Don't have Getters and Setters for each motor
- Instead have functional interfaces that don't have to change if the number of drive motors changes

Liskov Substitution Principle

Objects should be replaceable with instances of their subtypes without altering the correctness of the program (Design By Contract)

Example:

If you had a program that dealt with shapes and one of the shapes was a rectangle. Assume there were methods to set its length and width. If you added a square and decided it was a subclass of a rectangle because the area, perimeter, etc. were calculated the same way, you would violate this. Why?

Liskov Substitution Principle

```
CalcArea()
{
   Rectangle* square = new Square();
   square->SetLength( 2.0) ;
   square->SetWidth( 5.0 );
   printf( “Area = %d \n”, square->CalculateArea() );
}
```

Area = 25.0, so setting the length was effectively ignored.

You could leverage the area/perimeter similarities and subclass if you didn’t have SetLength and SetWidth methods, but rather set these values as part of the creator.

Interface Segregation Principle

Many client specific interfaces are better than one general purpose interface

- Smaller is better than bigger
- Only give access to what is needed
- Classes don’t need to know about what they don’t use

Interface Segregation Principle

Example:

Supposed you wanted only one class to depend on RobotMap.h, so you created one class (MotorsAndSensors.cpp) to manage all of the motors and sensors.

Now several problem arise:

1. The drive subsystem doesn't need/nor want to know about the shooter motors/sensors nor the intake motors/sensors
2. Method names become longer in order to distinguish between the different subsystems (e.g. SetAngle, for instance, wouldn't be clear if it was the shooter or the intake)
3. If climbing arms are added, a lot more needs to be recompiled/linked because the MotorsAndSensors class needs to add new methods to deal with the arms

Interface Segregation Principle

A couple of Solutions:

1. Create multiple classes, so the interface is smaller
2. Create interfaces and have the MotorsAndSensors implement each of these interfaces (IDrive, IIntake, IShooter, etc.). The other classes refer to the specific interface it needs.

Best Solution:

Create the specific interfaces and implement a concrete class that implements each.

Then, if for instance, the shooter changes or a new prototype is evaluated, you could just create new concrete class that implements the IShooter interface.

Dependency Inversion Principle

Depend on Abstractions, not specific concrete implementations

- Decouple classes ⇒ Use interfaces
- Logic interacts with the interface not the concrete implementations
- This allows a concrete implementation to be added that uses the same interface to work without changing everything the deals with it.

What does this mean??

Dependency Inversion Principle

1. The shooter angle is determined using a potentiometer, since this is just an analog sensor in the WPILIB, you decide to embed this logic into your class as well as the PID logic to get to specific angles.
2. If the potentiometer gets swapped out for an encoder many things change. What if the encoder change was short-term (e.g. at a competition we only had an encoder, but we want to swap back to the potentiometer between the competitions).
3. If the Angle sensor is an interface (IAngle) and the shooter aiming code (including the PID) only deals with the IAngle interface, then there could be a potentiometer class and an encoder class that both implement the IAngle interface and swapping the actual sensor would have minimal impact.