How does this help? By splitting the Driver or Autonomous Op Mode in this way, we have a single abstract Op Mode that defines what the robot does in terms of these abstract operations, like `move`, that are defined by the subsystem abstract parent class, like `SkyStoneMotors`. One year, for the Rover Ruckus competition, we had two different robots, each with their own arm and claw assembly to move the minerals around. We implemented a single abstract Driver Op Mode and a single abstract arm class, but we had two concrete implementations of the arms, one for each design, and two concrete Driver Op Modes. The abstract Driver Op Mode, where most of the real work is done, was the same for both robots. The concrete arm implementation was different, of course, and there were two short concrete Driver Op Modes, whose only task was to inherit the abstract Op Mode and to declare which concrete components were being used. Changes to the *behavior* of the Op Mode took place in the abstract Driver Op. Changes to the *implementation* of the robot subsystems -- arm or claw or travel base -- took place in the concrete child classes.

Using this structure makes the robot a little more complex in some ways, but notice all the benefits that come from it. By creating an abstract layer that describes the operations the robot is able to do, even beginning programmers can become effective with just a small amount of training. Both Driver and Autonomous Op Modes are more intuitive and much simpler to program. Abstract and concrete components become reusable from one year to the next. We this year we reused the abstract and concrete layers from our travel base and arm subsystems with only a few modifications. And last but not least, we can support multiple *different* robots in the same year with the same code base.