

# Converting a Simple Autonomous program to a Command based autonomous program

## Converting a Simple Autonomous program to a Command based autonomous program

### The initial autonomous code with loops

```
C++

// Aim shooter
SetTargetAngle(); // Initialization: prepares for the action to be performed
while (!AtRightAngle()) { // Condition: keeps the loop going while it is satisfied
    CorrectAngle(); // Execution: repeatedly updates the code to try to make the
condition false
    delay(); // Delay to prevent maxing CPU
}
HoldAngle(); // End: performs any cleanup and final task before moving on to the next
action

// Spin up to Speed
SetTargetSpeed(); // Initialization: prepares for the action to be performed
while (!FastEnough()) { // Condition: keeps the loop going while it is satisfied
    SpeedUp(); // Execution: repeatedly updates the code to try to make the condition
false
    delay(); // Delay to prevent maxing CPU
}
HoldSpeed();

// Shoot Frisbee
Shoot(); // End: performs any cleanup and final task before moving on to the next action
}
```

#### Java

```
// Aim shooter
```

# Converting a Simple Autonomous program to a Command based autonomous program

```
SetTargetAngle(); // Initialization: prepares for the action to be performed
while (!AtRightAngle()) { // Condition: keeps the loop going while it is satisfied
    CorrectAngle(); // Execution: repeatedly updates the code to try to make the
condition false
    delay(); // Delay to prevent maxing CPU
}
HoldAngle(); // End: performs any cleanup and final task before moving on to the next
action

// Spin up to Speed
SetTargetSpeed(); // Initialization: prepares for the action to be performed
while (!FastEnough()) { // Condition: keeps the loop going while it is satisfied
    SpeedUp(); // Execution: repeatedly updates the code to try to make the condition
false
    delay(); // Delay to prevent maxing CPU
}
HoldSpeed();

// Shoot Frisbee
Shoot(); // End: performs any cleanup and final task before moving on to the next action
}
```

The code above aims a shooter, then it spins up a wheel and, finally, once the wheel is running at the desired speed, it shoots the frisbee. The code consists of three distinct actions: aim, spin up to speed and shoot the Frisbee. The first two actions follow a command pattern that consists of four parts:

1. Initialization: prepares for the action to be performed.
2. Condition: keeps the loop going while it is satisfied.
3. Execution: repeatedly updates the code to try to make the condition false.
4. End: performs any cleanup and final task before moving on to the next action.

# Converting a Simple Autonomous program to a Command based autonomous program

The last action only has an explicit initialize, though depending on how you read it, it can implicitly end under a number of conditions. The most obvious one two in this case are when it's done shooting or when autonomous has ended.

## Rewriting it as Commands

**C++**

```
#include "AutonomousCommand.h"

AutonomousCommand::AutonomousCommand()
{
    AddSequential(new Aim());
    AddSequential(new SpinUpShooter());
    AddSequential(new Shoot());
}
```

**Java**

```
public class AutonomousCommand extends CommandGroup {

    public AutonomousCommand() {
        addSequential(new Aim());
        addSequential(new SpinUpShooter());
        addSequential(new Shoot());
    }
}
```

The same code can be rewritten as a CommandGroup that groups the three actions, where each action is written as it's own command. First, the command group will be written, then the commands will be written to accomplish the three actions. This code is pretty straightforward. It does the three actions sequentially, that is one after the other. Line 3 aims the robot, then line 4

# Converting a Simple Autonomous program to a Command based autonomous program

spins the shooter up and, finally, line 5 actually shoots the frisbee. The `addSequential()` method sets it so that these commands run one after the other.

## The Aim command

C++

```
#include "Aim.h"

Aim::Aim()
{
    Requires(Robot::turret);
}

// Called just before this Command runs the first time
void Aim::Initialize()
{
    SetTargetAngle();
}

// Called repeatedly when this Command is scheduled to run
void Aim::Execute()
{
    \    CorrectAngle();
}

// Make this return true when this Command no longer needs to run execute()
bool Aim::IsFinished()
{
    return AtRightAngle();
}

// Called once after isFinished returns true
void Aim::End()
{
    HoldAngle();
}

// Called when another command which requires one or more of the same
// subsystems is scheduled to run
```

# Converting a Simple Autonomous program to a Command based autonomous program

```
void Aim:Interrupted()  
{  
    End();  
}
```

## Java

```
public class Aim extends Command {  
  
    public Aim() {  
        requires(Robot.turret);  
    }  
  
    // Called just before this Command runs the first time  
    protected void initialize() {  
        SetTargetAngle();  
    }  
  
    // Called repeatedly when this Command is scheduled to run  
    protected void execute() {  
        CorrectAngle();  
    }  
    \ }  
  
    // Make this return true when this Command no longer needs to run execute()  
    protected boolean isFinished() {  
        return AtRightAngle();  
    }  
    \ }  
  
    // Called once after isFinished returns true  
    protected void end() {  
        HoldAngle();  
    }  
  
    // Called when another command which requires one or more of the same  
    // subsystems is scheduled to run  
    protected void interrupted() {  
        end();  
    }  
}
```

# Converting a Simple Autonomous program to a Command based autonomous program

```
}  
}
```

As you can see, the command reflects the four parts of the action we discussed earlier. It also has the interrupted() method which will be discussed below. The other significant difference is that the condition in the isFinished() is the opposite of what you would put as the condition of the while loop, it returns true when you want to stop running the execute method as opposed to false. Initializing, executing and ending are exactly the same, they just go within their respective method to indicate what they do.

## SpinUpShooter command

**C++**

```
#include "SpinUpShooter.h"  
  
SpinUpShooter::SpinUpShooter()  
{  
    Requires(Robot::shooter)  
}  
  
// Called just before this Command runs the first time  
void SpinUpShooter::Initialize()  
{  
    \    SetTargetSpeed();  
}  
  
// Called repeatedly when this Command is scheduled to run  
void SpinUpShooter::Execute()  
{  
    SpeedUp();  
}  
  
// Make this return true when this Command no longer needs to run execute()  
bool SpinUpShooter::IsFinished()  
{
```

# Converting a Simple Autonomous program to a Command based autonomous program

```
\    return FastEnough();
}

// Called once after isFinished returns true
void SpinUpShooter::End()
{
    HoldSpeed();
}

// Called when another command which requires one or more of the same
// subsystems is scheduled to run
void SpinUpShooter::Interrupted()
{
    End();
}
```

## Java

```
public class SpinUpShooter extends Command {

    public SpinUpShooter() {
        requires(Robot.shooter);
    }

    // Called just before this Command runs the first time
    protected void initialize() {
        SetTargetSpeed();
    }

    // Called repeatedly when this Command is scheduled to run
    protected void execute() {
        SpeedUp();
    }

    // Make this return true when this Command no longer needs to run execute()
    protected boolean isFinished() {
        return FastEnough();
    }
}
```

# Converting a Simple Autonomous program to a Command based autonomous program

```
\    }

    // Called once after isFinished returns true
    protected void end() {
        HoldSpeed();
    }

    // Called when another command which requires one or more of the same
    // subsystems is scheduled to run
    protected void interrupted() {
        end();
    }
}
```

The spin up shooter command is very similar to the Aim command, it's the same basic idea.

## Shoot command

```
C++

#include "Shoot.h"

Shoot::Shoot()
{
    Requires(Robot.shooter);
}

// Called just before this Command runs the first time
void Shoot::Initialize()
{
    \    Shoot();
}

// Called repeatedly when this Command is scheduled to run
void Shoot::Execute()
```

# Converting a Simple Autonomous program to a Command based autonomous program

```
{  
}  
  
// Make this return true when this Command no longer needs to run execute()  
bool Shoot::IsFinished()  
{  
    return true;  
}  
  
// Called once after isFinished returns true  
void Shoot::End()  
{  
}  
  
// Called when another command which requires one or more of the same  
// subsystems is scheduled to run  
void Shoot::Interrupted()  
{  
    End();  
}
```

## Java

```
public class Shoot extends Command {  
  
    public Shoot() {  
        requires(shooter);  
    }  
  
    // Called just before this Command runs the first time  
    protected void initialize() {  
        Shoot();  
    }  
  
    // Called repeatedly when this Command is scheduled to run  
    protected void execute() {  
    }  
}
```

# Converting a Simple Autonomous program to a Command based autonomous program

```
// Make this return true when this Command no longer needs to run execute()  
protected boolean isFinished() {  
    return true;  
}  
  
// Called once after isFinished returns true  
protected void end() {  
}  
  
// Called when another command which requires one or more of the same  
// subsystems is scheduled to run  
protected void interrupted() {  
    end();  
}  
}
```

The shoot command is the same basic transformation yet again, however it is set to end immediately. In CommandBased programming, it is better to have its isFinished method return true when the act of shooting is finished, but this is a more direct translation of the original code.

## Benefits of the command based approach

Why bother re-writing the code as CommandBased? Writing the code in the CommandBased style offers a number of benefits:

- **Re-Usability** You can reuse the same command in teleop and multiple autonomous modes. They all reference the same code, so if you need to tweak it to tune it or fix it, you can do it in one place without having to make the same edits in multiple places.
- **Testability** You can test each part using tools such as the SmartDashboard to test parts of the autonomous. Once you put them together, you'll have more confidence that each piece works as desired.
- **Parallelization** If you wanted this code to aim and spin up the shooter at the same time, it's trivial with CommandBased programming. Just use AddParallel() instead of AddSequential() when adding the Aim command and now aiming and spinning up will happen simultaneously.

# Converting a Simple Autonomous program to a Command based autonomous program

- **Interruptibility** Commands are interruptible, this provides the ability to exit a command early, a task that is much harder in the equivalent while loop based code.