An Approach to Robot Control

SoftwareBug2.0, FRC Team 1425

September 11, 2015

Contents

List of Figures

1 Introduction

This whitepaper describes a method of structuring robot code used in the FIRST Robotics Competition. Many sources give examples of how to get basic functionality working or how to use specific sensors but fewer have approached how to structure the system beyond splitting code by the physical components controlled. This paper is intended to be language-neutral but is aimed at teams who already have some basic level of proficiency in their chosen language.

2 Overview

The control of a robot can be broken into several categories of tasks: 1) defining goals, 2) writing outputs, 3) reading sensors, 4) state estimation, and 5) control algorithms. Each of these tasks can be simple or complex.

2.1 Example: Pneumatic Arm

Consider the following system: A robot arm is moved up when a pneumatic piston is retracted and down when it is extended. The pneumatic piston is extended when a button on a gamepad is held down. Breaking this down into parts one might get:

1. Goals: The user can indicate that the arm should be either up or down by pressing a button therefore the goals are “up” and “down”.
2. Writing outputs: The system must know which physical connectors are wired to the solenoid that controls the piston and what values they must be given to cause the piston to extend or retract.
3. Reading sensors: There aren’t any sensors in this example so this step isn’t needed.
4. State estimation: This part would estimate the position of the arm. To do this it might use readings from sensors or look at the outputs that are being sent to the arm. The resulting estimate would could be used either to feed to control algorithm or as feedback to the drivers. However, in this it turns out that the control algorithm doesn’t need an estimate of what the arm is doing, so this step can be skipped.
5. Control algorithm: Turns goals and state estimates into outputs. In this case, when the goal is “up”, it should choose the output that makes the arm go up, and if the goal is “down” it should choose the output that makes the arm go down.

2.2 Example: Wheeled Shooter

Now consider a more complicated example: A wheeled shooter is controlled by a knob which chooses a speed between 0 and 3000 RPM. A sensor creates a high pulse for each revolution of the wheel.

1. Goals: A wheel speed between 0 and 3000 RPM.
2. Writing outputs: The system needs to know which motor and information about how to drive the motor. For example, a Victor SP would require a fraction of maximum power while a Talon SRX might instead use PID constants and a set point.
3. Reading sensors: How to access the results from sensor that’s used to track the speed. This should probably be in some type of processed form such as RPM instead of the raw digital input reading.
4. State estimation: The state would be the speed of the wheel which can be estimated to be the speed that the sensor reads.
5. Control algorithm: If a Talon SRX was used then might be the control algorithm choose be to choose the PID constants and vary the set point based on the indicated goal. If this is a Victor SP then there might need to be an actual implementation of PID or anither algorithm such as bang-bang control.

3 A Partial Implementation

3.1 Example: Pneumatic Arm

The first step is choosing a data type to represent the goals. With the arm example, one could use the following (C++11 code):

```
enum class Goal{UP,DOWN};
```

Which creates a data type called "Goal" that stores only "UP" or "DOWN". Alternatively, since there are only two options:

```
using Goal=bool;
```

or equivalently

```
typedef bool Goal;
```

could be used. These allow 'Goal' to be used as a boolean type.

Next choose a type that represents an output of the system. For example, any of the following could work (again in C++11):

```
enum class Output{UP,DOWN};

//because Goal and Output have the same options in this case
using Output=Goal;

using Output=bool;//possible since only 2 options
```

Once these data types are chosen pieces of code to do the following three actions are needed:

- Reading the button on the gamepad and generating a *Goal*
- Generating *Output* given the *Goal*
- Writing the *Output* to the hardware

If this was interfacing with team 1425's 2015 robot code then these might look something like:

```
auto arm_goal=driver_gamepad.button[2]?Goal::UP:Goal::DOWN;
```

(Note that the meaning of the *auto* keyword changed in C++11.)

```
Output control(Goal g){
   return (g==Goal::UP)?Output::UP:Output::DOWN;
}
```

```
void write_output(Robot_outputs &robot_outputs,Output out){
   robot_outputs.relay[3]=(out==Output::UP);
}
```

3.2 Example: Wheeled Shooter

Create a *Goal* type:

```
using Goal=float;
```

Or alternately make a type that can hold only 0-3000. Language-level support for refinement types isn't necessary. The following will suffice:

```
class F{
   float data;

   public:
   F(float f):data(f){
      assert(f>=0 && f<=3000);
   }

   float get()const{ return data; }
};
```

After the *Goal* type has been chosen goals can be read in:

```
//Assume the driver station is gives 0-5 for 0-5 volts
goal=inputs.driver_station.analog[2]/5*3000;
```

Next create an *Output* type:

```
//If directly controlling motor power
using Output=float;

//If an external device (e.g. Talon SRX) is running PID loop
struct Output{
   float p,i,d,set_point;
};
```

For the case with the Talon SRX this could result in:

```
void write_outputs(Robot_outputs &robot_outputs,Output out){
   auto &x=robot_outputs.talox_srx[0];
   x.p=out.p;
   x.i=out.i;
   x.d=out.d;
   x.set_point=out.set_point;
}
```

Since this part of the robot has a sensor define a type to hold results from it:

```
using Input=float;//RPMs
```

and a function to read it:

```
Input read_input(Robot_inputs in){
  return in.talon_srx[0].velocity;
}
```

The estimation code is trivial:

```
using Status=Input; /*Since they both hold a single wheel speed*/
Status estimate(Input in){ return in; }
```

Here are a couple different control algorithms:

```
/*Use a Talon SRX's internal PID controller (and assume Output was
    defined appropriately)*/
Output control(Goal goal){
  return Output{1,.2,0,goal};
}

/*Assume output has been defined as fraction of available power (-1 to
    1)*/
Output control(Input in,Goal goal){
  //bang-bang controller
  return goal>in;
}
```

4 Complete Parts List

In order to allows the code for the various parts of the robot to be used interchangably a common interface is defined. For many subsystems this is more elaborite interface than strictly necessary; many of the steps will have trivial implementations.

4.1 Type *Input*

As described earlier, this is a data type that stores what's been read from sensors that is relavant to the subsystem. A few examples:

```
//For when there are no sensors
struct Input{};
```

```
//for when there is only one sensor which yields an integer
using Input=int;

//for multiple sensors
struct Input{
   bool top_limit,bottom_limit;
   float height;
};
```

4.2 Type *Input_reader*

The *Input_reader* converts raw data from sensors to the format defined by *Input*. It also defines the inverse - it calculates sensor inputs that would create a given *Input* variable. The first operation is used in normal operation of the robot. The second operation is useful for testing and debugging code. A no-op example (C++):

```
struct Input{}; //As before

struct Input_reader{
   Input operator()(Robot_inputs)const{
      return Input{};
   }

   Robot_inputs operator()(Robot_inputs a,Input)const{
      return a;
   }
};
```

A more typical example:

```
enum class Input{OPEN,CLOSED};

struct Input_reader{
   int digital_io_port;

   Input operator()(Robot_inputs a)const{
      return a.dio[digital_io_port]?Input::OPEN:Input::CLOSED;
   }

   Robot_inputs operator()(Robot_inputs r,Input b)const{
      a.dio[digital_io_port]=(b==Input::OPEN);
      return r;
   }
};
```

4.3 Type *Output*

The *Output* type should fully describe the outputs from the subsystem to the robot.

```
//Sufficient for a device with a single solenoid
using Output=bool;

//Kitbot drivebase, controlled by power levels
struct Output{
   float motor_l,motor_r;
};

//Valid, but atypical. Could be used for diagnostic purposes.
struct Output{};
```

4.4 Type *Output_applicator*

The *Output_applicator* works similarly to the *Input_reader*. Given an *Output* it determines the physical outputs of the robot needed, and the reverse: calculating what value in an *Output* would correspond to a given set of robot outputs. Unlike the *Input_reader*, both functions of the *Output_applicator* are used in normal operation. (The function that goes from *Robot_outputs* to *Output* is used to allow transparent support for disabled mode.)

```
//For a single solenoid
struct Output_applicator{
   Robot_outputs operator()(Robot_outputs r,Output out)const{
      r.relay[3]=out;
      return r;
   }

   Output operator()(Robot_outputs a)const{
      return a.relay[3];
   }
};

//For a simple drivebase
struct Output_applicator{
   Robot_outputs operator()(Robot_outputs r,Output out)const{
      r.pwm[0]=out.motor_l;
      r.pwm[1]=out.motor_r;
      return r;
   }

   Output operator()(Robot_outputs a)const{
      return Output{a.pwm[0],a.pwm[1]};
   }
};

//The degenerate case
struct Output_applicator{
   Robot_outputs operator()(Robot_outputs a,Output)const{ return a; }
   Output operator()(Robot_outputs)const{ return Output{}; }
};
```

4.5 Type *Goal*

The *Goal* type could be very high level or very low level.

```
//If what the system attempts to do does not change
struct Goal{};

//Arm with only two interesting positions
enum class Goal{UP,DOWN};

//Wheeled shooter always given a target RPM
using Goal=float;

//Wheeled shooter that can be given either a target RPM or power level
class Goal{
   public:
   enum class Type{POWER,RPM};

   private:
   Type type_;
   float value_;

   Goal(Type t,float f):type_(t),value_(f){}

   public:
   Type type()const{ return type_; }

   float rpm()const{
      assert(type_==Type::RPM);
      return value_;
   }

   float power()const{
      assert(type_==Type::POWER);
      return value_;
   }

   static Goal power(float f){
      assert(f>=-1 && f<=1);
      return Goal{Type::POWER,f};
```

```
	}

	static Goal rpm(float f){
		assert(f>=0 && f<=3000);
		return Goal{Type::RPM,f};
	}
};
```

4.6 Type *Status*

A *Status* is meant to be a human-readable estimate of the what the state of the system.

```
//It is valid to produce an information-free status
struct Status{};

//An arm that has two stopping positions
enum class Status{UP,DOWN,MOVING};

//A wheeled shooter's speed
using Status=float;
```

4.7 Type *Status_detail*

The *Status_detail* serves a similar purpose to the *Status* class but may contain extra information which is not interesting to a user or which is not designed to be relied upon by other subsystems of the robot. The information held in *Status* should all be derivable from the information in *Status_detail* but reverse does not have to be true.

The simplest option is to make *Status_detail* the same as *Status*:

```
using Status_detail=Status;
```

Some situations may call for something somewhat more complex, however. For example, the status might look like the following to implement a PID controller:

```
struct Status_detail{
	float speed;
	float i_accum;
	float d_speed;
};
```

4.8 Type *Estimator*

The *Estimator* is the heart of the system, integrating sensor and other data and calculating values relavant to the control algorithm. An *Estimator*'s basic form:

```
struct Estimator{
   void update(Time,Input,Output);
   Status_detail get()const;
};
```

The simplest case appears when *Status_detail* is empty:

```
struct Estimator{
   void update(Time,Input,Output){}
   Status_detail get()const{ return Status_detail{}; }
};
```

For a subsystem with an arm that has two stopping positions, and it is important to know when it is at one of the ends but no sensors are present:

```
enum class Status_detail{UP,DOWN,MIDDLE};

using Time=float; //a number of seconds

struct Estimator{
   Time start_time,last_time;
   Output last_output;

   Estimator():start_time(-1),last_time(-1){}

   void update(Time t,Input,Output output){
      if(last_time==-1 || last_output!=output){
         start_time=last_time=t;
         last_output=output;
      }
   }

   /*arm is at an end if the output has been constant for 1.5 seconds,
       otherwise it's in the middle*/
   Status_detail get()const{
      Time elapsed=last_time-start_time;
      if(elapsed<1.5) return Status_detail::MIDDLE;
      return
          (last_output==Output::UP)?Status_detail::UP:Status_detail::DOWN;
   }
};
```

Note that the *Estimator* is not given a *Goal* as input.

4.9 Function *status*

The *status* function takes a *Status_detail* and returns a *Status*. This often implemented as the following:

```
//This works if Status_detail and Status are the same type
Status status(Status_detail a){ return a; }
```

It's also common for the *Status* to take the following form:

```
Status status(Status_detail a){ return a.x; }
```

4.10 Function *control*

The function *control* has this signature:

```
Output control(Status,Goal);
```

The *control* functions from section 2 might be modified as follows:

```
//Pneumatic arm
Output control(Status,Goal goal){
   return (goal::Goal::UP)?Output::UP:Output::DOWN;
}

//Wheeled shooter w/ bang-bang control
Output control(Status current_speed,Goal desired_speed){
   return desired_speed>current_speed;
}
```

4.11 Function *ready*

The *ready* function indicates when a *Goal* has been accomplished.

```
//Simplest case
bool ready(Status,Goal){ return 1; }

//Pneumatic arm where Status is one of UP,DOWN,MOVING
bool ready(Status status,Goal goal){
   if(goal==Goal::UP) return status==Status::UP;
   return status==Status::DOWN;
}

//Wheeled shooter; Status and Goal are both RPMs
bool ready(Status status,Goal goal){
   //close enough if status and goal are within 50 RPMs
   return fabs(status-goal)-50;
}
```

5 The System

The flow of information within the robot software and the system consisting of the robot software and hardware and the operator will now be examined.

5.1 Assembling the Parts

The code for the system can be put together in this manner:

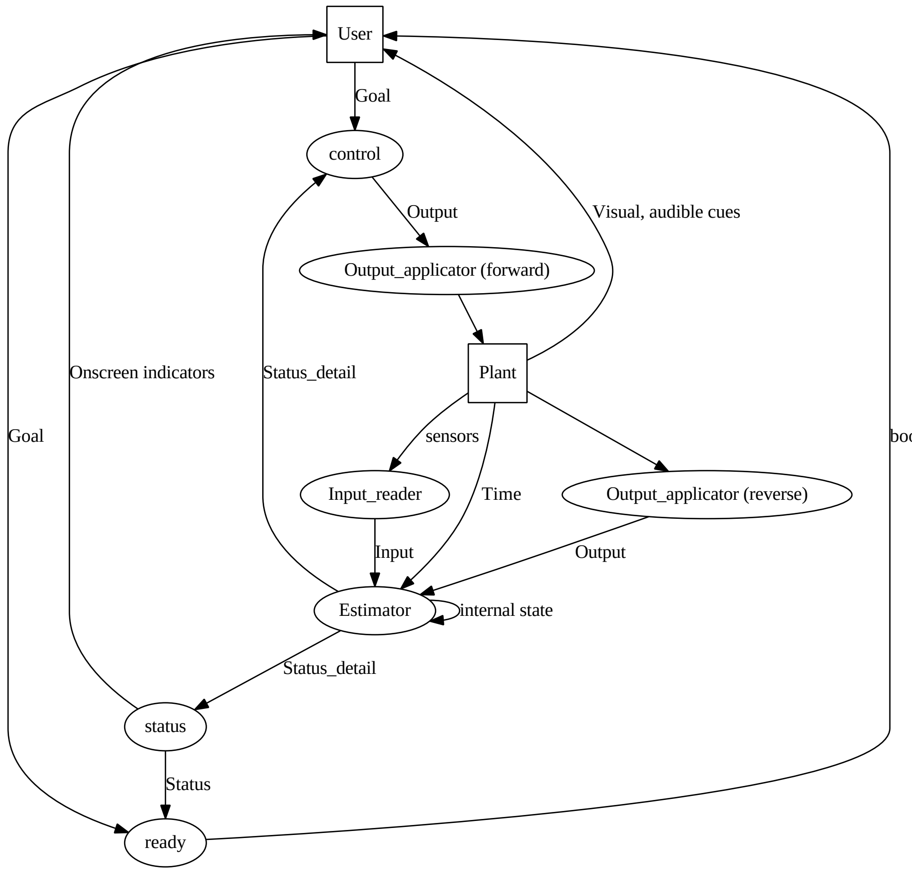
```
struct Arm{
   struct Input{};
   struct Input_reader{
      Input operator()(Robot_inputs)const{ return Input{}; }
      Robot_inputs operator()(Robot_inputs a,Input){ return a; }
   };
   enum class Output{UP,DOWN};
   struct Output_applicator{
      Robot_outputs operator()(Robot_outputs,Output)const;
      Output operator()(Robot_outputs)const;
   };
   using Goal=Output;
   struct Status{};
   using Status_detail=Status;
   struct Estimator{
      void update(Time,Input,Output){}
      Status_detail get()const{ return Status_detail{}; }
   };

   Input_reader input_reader;
   Output_aplicator output_applicator;
   Estimator estimator;
};
Arm::status status(Arm::Status_detail a){ return a; }
Arm::Output control(Arm::Status,Arm::Goal goal){ return goal; }
bool ready(Arm::Status,Arm::Goal){ return 1; }
```

5.2 Driving the system

And the code to call this drive this might look like:

```
Arm arm;
Robot_outputs r_out;
while(1){
   Robot_inputs in=read_inputs();
   auto goal=read_driver_station();
```

Figure 1: System Diagram
Ovals represent software functions. Squares represent physical entities. "User" represents the drivers and/or a higher level of control software. "Plant" in this diagram represents the physical robot and its environment as well as any extra layers of software between the program and the physical system. Arrows represent the flow of information. Labels next to arrows denote the type of information that flows.



```
   arm.estimator.update(
      get_time(),
      arm.input_reader(in),
      arm.output_applicator(r_out)
   );
   auto status_detail=arm.estimator.get();
   auto current_status=status(status_detail);
   cout<<current_status<<"\n";
   cout<<"ready:"<<ready(current_status,goal);
   auto output=control(status_detail,goal);
   r_out=arm.output_applicator(r_out,output);
   write_outputs(r_out);
}
```

The dependency graph in this code should mirror the figure on page 14.

6 Testing

Since the different parts do not directly call each other they can be tested independently. Once that works some basic testing can be performed on the whole.

6.1 *Input_reader*

The *Input_reader* functions are tested by creating a list of values for *Input* (possibly exhaustive) and verifying that the two functions are inverses at least for the given list.

6.2 *Output_applicator*

The *Output_applicator* is tested in the same way as *Input_reader*.

6.3 *Estimator*

Typically this is tested by seeing creating some sequence of events that can produce each of a chosen set of *Status_detail* values. Also, lists of *Input* and *Output* values are cycled through to see that no unexpected results occur.

6.4 *status*

The *status* function is fed all out of outputs that testing *Estimator* produces. The results are compared to a set of known *Status* values.

6.5 *control*

The *control* function is tested by running with all pairs of *Status* and *Goal*.

6.6 *ready*

The testing is conducted in the same manner as *control*.

6.7 The Whole Subsystem

To simulate the whole subsystem a simulator is created by instantiating an extra *Estimator*. For each value of *Goal* the simulator is used to see if the output of *ready* goes high within some time limit. Typically the time limit is an amount of simulated time.

7 Error Handling

This system makes no special accommodation for error conditions, and it is not expected that components will throw exceptions which they do not themselves catch. Some suggestions for handling error in the various conditions are handled.

Input_reader If errors are possible make *Input* able to represent *INVALID*, *NaN*, or some other appropriate value.

Output_applicator Typically errors are not possible and no mechanism to handle them is provided.

Estimator If problems are encountered a ‘safe’ *Status_detail* value should be returned. ‘Safe’ means that it would not cause *control* to choose outputs that would damage the robot.

controller, status, and ready No error handling mechanism is provided. These functions should be total.

8 Extending Subsystems

8.1 *Input_reader*

The *Input_reader* can be adjusted without modifying the rest of the code:

- Changing IO pins. This is useful when there is more than one of the same device on the robot.
- Talking to a device on different types of interfaces. For example, a sensor might have an analog or a digital mode.
- The type of sensor could be switched as long as it gives the same information out. For example a touch sensor could be replaced by a proximity sensor with a threshold function.

8.2 *Output_applicator*

The *Output_applicator* can be replaced with alternate versions in a similar manner to the *Input_reader*.

8.3 *Estimator*

Different algorithms can be switched into the *Estimator* freely as long as the *Status_detail* type doesn't need to be changed. For example, a state machine could be switched to fuzzy logic or a Kalman filter.

8.4 *status*

Typically the *status* function isn't extended unless *Status_detail* changes.

8.5 *control*

For many purposes *control* as a pure function is somewhat limiting. For example, it doesn't allow implementing all of a PID loop inside of it. To implement a PID the options are either to either make an offboard controller run the loop (recommended) or add extra state to *Estimator* and *Status_detail*. If one found that this was commonly needed in practice *control* could be made a closure.

8.6 *ready*

Like *status*, *ready* is not typically modified alone. The implementation of *ready* should be obvious given the definitions of *Status* and *Goal*. When the output of ready is not as desired typically *Goal* is redefined (also necessitating modification to *control* and *ready*).

For example consider controlling an arm where the options for *Goal* are *UP* or *DOWN*, but sometimes the robot as a whole should be considered ready even if the arm hasn't reached either *UP* or *DOWN*. In this case a third value could be added to *Goal* called X, where X would denote that the readiness of the system doesn't depend on the state of the arm. The value of *ready* when given the goal X would be true regardless of the state estimate. The output from *control* when given a goal of X could be chosen arbitrarily.

With the same physical setup as the previous example it's also possible that although the robot's readiness should not depend on what a subsystem is doing the subsystem could still be given a task. New possiblities could be added to *Goal* called *UP_NONBLOCK* and *DOWN_NONBLOCK*, which would denote that the arm should go up or down but that *ready* should return *true* regardless. A more general solution would be to add a boolean variable called "nonblocking" to a structure that defined the *Goal*.

In a similar manner, error bounds for a continuous-valued *Status* could be made adjustable by additions to *Goal*.

9 Combining Subsystems

One of the most powerful properties of this style of code organization is its composablity. Subsystems can be combined into larger subsystems, and typically this is continued until there is one subsystem named *Top_level* which encapsulates all controllable parts of the robot. Subsystems might be orthogonal, meaning that the operations are not interdependant, or the subsystems might have some interaction. For example, a non-orthogonal subsystem might have two different arms that can reach the same location, but if they were both told to reach that location they would collide. A subsystem that contained the two arms might contain logic to avoid collisions.

9.1 Example: Orthogonal Subsystems

Each of the data types is combined by concatenation:

```
struct Input{
   Arm::Input arm;
   Drivebase::Input drivebase;
};

struct Status{
   Arm::Status arm;
   Drivebase::Status drivebase;
};

//...
```

The functions are defined as follows:

Input_reader, Output_applicator, Estimator, control, status The functions are run sequentially and the results are concatenated. Example:

```
Input Input_applicator::operator()(Robot_inputs in){
   return Input{arm(in),drivebase(in)};
}
```

ready The functions are run sequentially and returns true iff all component *ready* functions return true.

9.2 Example: Non-orthogonal Subsystems

Consider two single-jointed arms which are independently driven but which share a pivot point and will interfere if their output angles are within 20 degrees of each other. This could be defined the same as for the orthogonal case except that *control* might be defined like so:

```
Output control(Status_detail st,Goal g){
    return Output{
        control(st.arm1,g.arm1),

        /*force the target for arm2 to always be at least 20 degrees above
            the goal for arm1*/
        control(st.arm2,max(g.arm1+20,g.arm2))
    };
}
```

10 Conclusion

A method for controlling robots in the FIRST Robotics Competition has been presented. It seperates the input, output, and other sections or robot code and encourages thinking in terms of higher-level abstrations of the robot's state. It increases modularity and testability of a codebase compared to the standard organizations. This approach has been used by team 1425 (Error Code Xero) for its 2014 and 2015 entries and has been found minimize conflicts between changesets. To report any errors please contact SoftwareBug2.0 on ChiefDelphi.

Appendix A Java Implementation

The follows is a listing of a possible way to implement this in Java. It compiles with no errors but does generate warnings about type erasure. These should not be a problem, however the below is certainly not the only way this could be implemented. For example, it might be simpler using reflection rather than generic interfaces.

```
class Robot_inputs{}
class Robot_outputs{}

class Time{
    float seconds;
}

interface Input_reader<Input>{
    Input read(Robot_inputs a);
    Robot_inputs write(Robot_inputs a,Input b);
}

interface Output_applicator<Output>{
    Robot_outputs write(Robot_outputs a,Output b);
    Output read(Robot_outputs a);
}

interface Estimator<Input,Output,Status_detail>{
    void update(Time a,Input b,Output c);
```

```
  Status_detail get();
}

interface Ready<Status,Goal>{
  boolean run(Status a,Goal b);
}

interface Control_interface<Output,Status_detail,Goal>{
  Output run(Status_detail a,Goal b);
}

interface Status_convert<Status_detail,Status>{
  Status run(Status_detail a);
}

interface Subsystem<Input,Output,Status_detail,Status,Goal>{
  Input_reader<Input> input_reader();
  Output_applicator<Output> output_applicator();
  Estimator<Input,Output,Status_detail> estimator();
  Control_interface<Output,Status_detail,Goal> control_interface();
  Status_convert<Status_detail,Status> status_convert();
  Ready<Status,Goal> ready();
  Goal goal();
}

class Arm implements Subsystem{
  class Input{}

  class Input_reader1 implements Input_reader<Input>{
    public Input read(Robot_inputs a){
      return new Input();
    }

    public Robot_inputs write(Robot_inputs a,Input b){
      return a;
    }
  }

  class Output{}

  class Output_applicator1 implements Output_applicator<Output>{
    public Output read(Robot_outputs a){
      return new Output();
    }

    public Robot_outputs write(Robot_outputs a,Output b){
      return a;
    }
  }
  class Status_detail1 extends Status_detail{}

  class Estimator1 implements Estimator<Input,Output,Status_detail1>{
    public void update(Time a,Input b,Output c){}

    public Status_detail1 get(){
      return new Status_detail1();
    }
  }

  class Status{}
  class Goal{}

  public Input_reader1 input_reader(){ return new Input_reader1(); }
  public Output_applicator1 output_applicator(){ return new
      Output_applicator1(); }
  public Estimator1 estimator(){ return new Estimator1(); }

  class Ready1 implements Ready<Status,Goal>{
    public boolean run(Status a,Goal b){ return true; }
  }
  public Ready1 ready(){ return new Ready1(); }

  class Status_convert1 implements Status_convert<Status_detail,Status>{
    public Status run(Status_detail a){
      return new Status();
    }
  }
  public Status_convert status_convert(){ return new Status_convert1();
      }

  public Control_interface<Output,Status_detail,Goal>
      control_interface(){
    class A implements Control_interface<Output,Status_detail,Goal>{
      public Output run(Status_detail a,Goal b){
        return new Output();
      }
    }
    return new A();
  }

  public Goal goal(){ return new Goal(); }
}

class Test{
  static Robot_inputs read_inputs(){
    return new Robot_inputs();
  }

  static void write_outputs(Robot_outputs a){}
```

```java
  static Time get_time(){
    return new Time();
  }

  static <Input,Output,Status_detail,Status,Goal> void
      run(Subsystem<Input,Output,Status_detail,Status,Goal> a){
    Input_reader<Input> ir=a.input_reader();
    Output_applicator<Output> oa=a.output_applicator();
    Estimator<Input,Output,Status_detail> est=a.estimator();
    Control_interface<Output,Status_detail,Goal>
        control=a.control_interface();
    Status_convert<Status_detail,Status> status=a.status_convert();
    Ready<Status,Goal> ready=a.ready();

    Robot_outputs robot_outputs=new Robot_outputs();
    Robot_inputs robot_inputs;
    Goal goal=a.goal();

    for(int i=0;i<10;i++){
      robot_inputs=read_inputs();
      Input input=ir.read(robot_inputs);
      est.update(get_time(),input,oa.read(robot_outputs));
      Status_detail st_detail=est.get();
      Status st=status.run(st_detail);
      Output output=control.run(st_detail,goal);
      oa.write(robot_outputs,output);
      System.out.println("Status:"+st);
      System.out.println("Ready:"+ready.run(st,goal));
      write_outputs(robot_outputs);
    }
  }

  public static void main(String[] args){
    Arm arm=new Arm();
    run(arm);
  }
}
```