paulmurraycbr.github.io

Arduino the Object Way

54-68 minutes

Arduino the Object Oriented way

Introduction

In the early to mid 1990's, programmers began coding in a new way - the Object Oriented way.

Ok, that's a total lie. OO programming and languages had been around for years. But it was around this time that it became big commercially, spurred by the introduction of OO features to existing languages, such as C++, and by new languages and platforms such as Java

Ok, that was a total lie, too. The truth is: I learned Java back in the late 90's and got myself a java job in 2000 or so. And that was when I personally started doing this stuff.

Object oriented programming is part of a progression in language design over the last 50 years:

0. Machine code
1. Assembler, Macro assembler
2. Compiled languages
3. Block-structured languages
4. Object-oriented languages

At each stage in this progression, the new programming languages provide constructs that "wrap" blocks of code that you had to write in the old one. Assembler macros wrap chunks of assembler. Compiled languages such as Fortran did the grunt work of converting expressions into assembler. "Fortran" is a shortening of "formula translation": the new thing was that a compiler would turn "a * (b + c)" into machine code. Block-structured languages replaced the usual idioms for flow of control with "if" and "while" statements, which Wirth proved mathematically was sufficient to do anything that you could do with GOTO. Functions and subroutines and the idea of a "stack" … well, that had been around for a long time - there's support for it on the chip. But "functions" are a formalisation of it.

At each stage of the progression it becomes easier to make more complex things. The features of the language mean that there's fewer balls in the air as you code. Once you write a function and get it right, you no longer need to think about what is in that function - you just use it for what it *does*. Another very important feature is that this isolation of chunks of code makes it much easier for teams of programmers to build something.

Arduino programming - at least, what you see on the Arduino message boards - seems to be stuck at step 3. But OO coding has a great deal to offer in terms of making Arduino code simpler and more robust. it's well worthwhile getting your head around it.

This page is about how I personally address a typical project posted on the Aruino boards, using C++ objects. It works for me.

The discussion on the discussion boards is about this page is here. You may want to consider some criticisms there of this page here, although it might be worthwhile skipping some of the quibbling about using `byte` instead of `int` .

The OO Way

The OO way of programming is like building electronics, or building something mechanical, or building anything, really. You use something simple to make something moderately complicated, and those moderately complicated things to make something very complicated. The trick is: when it comes time build the very complicated thing, you work with *what* your moderately complicated things do and you temporarily forget about *how* they do it.

In our headlamp example, we have two things. We have a button that emits a short-click and a long-click. And we have a headlamp that turns on and off and that cycles brightness. If we can make those two things work, and hook them together - job done.

Adding the tail light is straightforward. Our tail light has two independent on/off variables: day/night, and braking/not braking. We program a thing that can do that, and then we program up the interactions between the things.

What is an object?

An object is a bundle of variables (holding state) and functions that operate on those variables.

Whenever you code things for an arduino this is what you wind up doing anyway. You will always have sets of variables that work togther (in OO language: that have strong cohesion), and blocks of code that mainly deal with those variables. A function is just a block of code that you have separated out and given a name. If you have a thing that needs to blink, your will inevitably have a set of variables like `blinkTimeMs` and `blinkState` to keep track of the blink, and functions named `handle_blink` and such.

C++, however, allows you to bundle these things together.

Sure, you can do this without classes and objects. If you code Arduino, you already *are* doing these things without classes and objects. All you have to do is be careful to keep track of things and it helps to give everything sensible names.

But the same applies to all of the progression of languages over the years. Assembler doesn't allow you to do anything you can't do with raw opcodes. Compilers don't allow you to do anything that you can't do in assembler. Block-structured languages don't allow you to do anything that you can't do with gotos. All these things do is make it easier for you - the programmer - to get it right.

Bundling your variables and code for those variables together has a few advantages.

Your code becomes easier to understand and modify, because the variables that the code needs are right there, next to the code.

It becomes much easier to create multiple objects that all do the same thing. A set of blinkers, each one with its own output pin. A pair of servos that both behave in the same way. You no longer need to declare multiple variables all with slightly different names, or have them in arrays.

It becomes easier to build compound objects, such as a pair of relays which must never be both on at the same time, requiring a delay between turning one off and the other on.

Objects can conceal parts of themselves from other objects (in fact, that's the default). You can build an object and be confident that it will work correctly because other parts of your sketch *cannot* interfere with its internals. Conversely, if another part of your sketch *needs* to interfere with its internals, then this signals to you - the programmer - that there's something you haven't thought of and might need to take into account.

But the most important thing is difficult to quantify: it changes the way that you *think* about what you are doing, it changes the *way* you code.

And finally, doing objects the way I suggest here makes managing asynchonous, event-driven interfaces to sketches much easier and more natural to do. Blink-without-delay for everything.

The Example Problem

In the code that is sometimes posted to the Arduino message boards, things start to get messy when coders are having to deal with several independent devices, which is to say clusters of pins - input and output.

One example is here: #367748 - Reading length of button-press.

The OP has a bike headlamp. this headlamp has a button that you can press 5 times to cycle through 5 output settings, and a power switch. the OP would like to have one button to control it. A long click should turn the lamp on and off, and a short click should cycle the brightness. Furthermore, when the lamp is turned on, the OP would like the arduino to cycle the light up to its previous level of brightness. The outputs to cycle the lamp should be pulses with a duration of 100ms, separated by 100ms. The output to turn the lamp on and off should simply be HIGH/LOW on a digital out.

Simple stuff - you would think. But the output clicks to cycle the lamp up to a level of brightness take time. What happens if there's a short click *while* this is happening? A long click? How do the timing of the input clicks on the controller button interact with the timing of the output to the lamp? Oh, and there should be a 50ms debounce delay on the button, too.

What happens is that people try to code this up by having long chains of nested `if` statements, trying to account for all the possibilities. This never works out, because the number of possibilities explodes combinatorially. These sketches also tend to have a rat's nest of variables declared at the top.

Let's add something to the OP's sketch - a tail light and a brake sensor. If the headlamp is on, then the tail light should flash

intermittently (100ms every second). The tail light should also shine continuously if the brakes are on.

Adding this to a sketch structured in the usual way people attempt to write these things - you'd just give up. But if you code things the Object Oriented way, it becomes easy. Easier, anyway. Do-able.

Part I - the headlamp and control button

Enough! Show me the code!

Ok, let's start with the control button. I will drop it onto pin 7.

```
const byte BUTTON_PIN = 7;

class Button {
};

Button button;

void setup() {
  pinMode(BUTTON_PIN, INPUT_PULLUP);
}

void loop() {
}
```

This defines a class for the button, and a single instance of the button class named `button` .

Obviously, it does nothing yet. However, in the programming pattern I am describing here, we give every object a `setup` and a `loop` method (functions inside classes are called methods). The `setup` method of every object is invoked in the main `setup` , and the `loop` method of every object is invoked in the main `loop` .

```
const byte BUTTON_PIN = 7;

class Button {
  public:
    void setup() {
      pinMode(BUTTON_PIN, INPUT_PULLUP);
    }

    void loop() {
    }
};

Button button;

void setup() {
  button.setup();
}

void loop() {
  button.loop();
}
```

And this is the key to this pattern for building Arduino projects. Every object gets a turn, even if it does nothing. And every object guarantees that it returns *quickly* from it's `loop` method. And each object does not rely on the other objects getting their slices of time in any particular order. We rely on the fact that there is a single thread in an Ardiono sketch. Each object guarantees that at the end of each iteration of its `loop` , it is in a suitable state for other objects to interrogate it and call its public methods.

Notice that I have had to introduce a `public` declaration. As I mentioned, the default behaviour for objects is that their internals are not visible to other things or to the main sketch. Anything that needs to be visible must be in a `public` section.

At this stage, we can introduce the usual code for button behaviour which you have all seen a million times before.

```
const byte BUTTON_PIN = 7;

class Button {
    int state;
    unsigned long buttonDownMs;
```

```
  public:
    void setup() {
      pinMode(BUTTON_PIN, INPUT_PULLUP);
      state = HIGH;
    }

    void loop() {
      int prevState = state;
      state = digitalRead(BUTTON_PIN);
      if (prevState == HIGH && state == LOW) {
        buttonDownMs = millis();
      }
      else if (prevState == LOW && state == HIGH) {
        if (millis() - buttonDownMs < 50) {
          // ignore this for debounce
        }
        else if (millis() - buttonDownMs < 250) {
          // short click
        }
        else  {
          // long click
        }
      }
    }
};

Button button;

void setup() {
  button.setup();
}

void loop() {
  button.loop();
}
```

We can see here a bit of weirdness - the button state is inside the button class, but the pin that it uses is a constant defined outside. This is a problem if you want - for instance - more than one button with similar behaviour.

To fix this, we move the pin number into the button class using a constructor. Constructors get invoked *before* any `setup` , so its important that they don't attempt to talk to the outside world at all. They are purely for initializing the object into a sensible pre-setup state.

```
class Button {
    const byte pin;
    int state;
    unsigned long buttonDownMs;

  public:
    Button(byte attachTo) :
      pin(attachTo)
    {
    }


    void setup() {
      pinMode(pin, INPUT_PULLUP);
      state = HIGH;
    }

    void loop() {
      int prevState = state;
      state = digitalRead(pin);
      if (prevState == HIGH && state == LOW) {
        buttonDownMs = millis();
      }
```

```
      else if (prevState == LOW && state == HIGH) {
        if (millis() - buttonDownMs < 50) {
          // ignore this for debounce
        }
        else if (millis() - buttonDownMs < 250) {
          // short click
        }
        else  {
          // long click
        }
      }
    }
};

Button button(7);

void setup() {
  button.setup();
}

void loop() {
  button.loop();
}
```

You can do other things inside your constructor, but in this case all we are doing is assigning a simple variable.

At this point our button is complete, except for the code for what actually gets done when a long and short click is performed. So just for testing purposes, let's have our button change the state of a pair of LEDs on pin 8 and 9. The LED on pin 8 will change for a short click, and the LED on pin 9 will change in response to a long click.

```
class Button {
    const byte pin;
    int state;
    unsigned long buttonDownMs;

  public:
    Button(byte attachTo) :
      pin(attachTo)
    {
    }


    void setup() {
      pinMode(pin, INPUT_PULLUP);
      state = HIGH;
    }

    void loop() {
      int prevState = state;
      state = digitalRead(pin);
      if (prevState == HIGH && state == LOW) {
        buttonDownMs = millis();
      }
      else if (prevState == LOW && state == HIGH) {
        if (millis() - buttonDownMs < 50) {
          // ignore this for debounce
        }
        else if (millis() - buttonDownMs < 250) {
          // short click
          // TEST BY TOGGLING PIN 8
          digitalWrite(8, !digitalRead(8));
        }
        else  {
          // long click
          // TEST BY TOGGLING PIN 9
```

```
          digitalWrite(9, !digitalRead(9));
        }
      }
    }
};

Button button(7);
void setup() {
  button.setup();

  // FOR TESTING
  pinMode(8, OUTPUT);
  pinMode(9, OUTPUT);
}

void loop() {
  button.loop();
}
```

Sans issues with the damn PCB-mount buttons jumping out of the breadboard (grr!), it works exactly as it should.

Step 2 - the light.

Next step is the light controller. It has two outputs - a power on/of and a brightness cycle. The brightness cycler switch needs pauses between clicks. For purposes of my demo, I will make the pauses long: a 250ms on and a 750ms off.

The controller needs two inputs - power on/off and brightness cycle. Our problem is: what happens if things happen *while* the brightness is cycling?

I'll over-engineer this a little and make the brightness cycle a sub-object. It has two input events: a "cycle by one" and a "cancel". What state does it need? Well, it needs to know if it is currently doing a click, it needs to know when it started doing the click that it is currently doing, and it needs to know how many clicks are pending on the queue.

Without further ado, the code looks like this:

```
class ClickQueue {
    const byte pin;
    // make these slow for testing
    const unsigned CLICK_DOWN_MS = 250;
    const unsigned CLICK_TOTAL_MS = 1000;


    enum State {
      NOT_CLICKING = 0,
      CLICK_DOWN = 1,
      CLICK_PAUSE = 2
    } state;

    unsigned long clickStartMs;
    int pendingClicks;

  public:
    ClickQueue(byte attachTo) :
      pin(attachTo) {
    }

    void setup() {
      pinMode(pin, OUTPUT);
      state = NOT_CLICKING;
      pendingClicks = 0;
    }

    void loop() {
      switch (state) {
        case NOT_CLICKING:
          if (pendingClicks > 0) {
            pendingClicks --;
```

```
            digitalWrite(pin, HIGH);
            clickStartMs = millis();
            state = CLICK_DOWN;
          }
          break;
        case CLICK_DOWN:
          if (millis() - clickStartMs > CLICK_DOWN_MS) {
            digitalWrite(pin, LOW);
            state = CLICK_PAUSE;
          }
          break;
        case CLICK_PAUSE:
          if (millis() - clickStartMs > CLICK_TOTAL_MS) {
            state = NOT_CLICKING;
          }
          break;
      }
    }

    void addAClick() {
      pendingClicks++;
    }

    void cancel() {
    }
} clickQueue(11);
```

There are a few things to note about this, which I believe are good practise.

First, any variable that holds a physical quantity should be suffixed with the unit of measurement of that quantity. This is a coding standard born of bitter experience. Variables, parameters, and functions dealing with time should be suffixed `Day` `Min` `Sec` or `Ms` accordingly. This also applies to currency, although we don't often encounter that in Arduino programming.

Notice that the variable `pin` is not the same variable in the two classes. We say that they are in different "scopes". An advantage of scoping is that you don't have to keep coming up with new and creative ways of naming your variables.

Use an enum for states. Enums are better than `const int` or `#define` when a variable might hold one of a small set of related values, because the compiler will check the types for you and make sure you are using the right one. It also means that you can use namespaces, for instance if you have several things each with an INIT state.

The `loop` method reads any state that it needs to (millis(), pin reads, `getValue()` calls to other objects), and this is followed by a switch statement that operates as a state machine. We have a branch for each state that the object might be in, and each branch of the switch statement follows the form:

```
if(some condition) {
  do stuff;
  state = new state;
}
else if(some other condition) {
  do other stuff;
  state = different new state;
}
else {
  do stuff if nothing else matches;
  state = other different new state;
}
```

You can skip the final `else` . If the "do stuff" or the conditions are complex, move them into a function. However, the assignment of the state should be done in the switch statement in the loop. That way, by inspection of the loop alone you can map put the flow of state changes in the object.

You might notice that I don't handle the case where a button finished being pressed but there are more presses on the queue. It doesn't matter - *the next time slice will take care of it*. The key is that `loop()` gets called frequently, because none of the other objects hang up with `delay()s` .

But alternatively, then yes you can make the 'end of click' processing do it. The problem is that there would then be two places where whatever needs to be done to start a click gets done. A block of duplicate code. This is usually a Bad Thing.

So what you would normally do is pull out the code (the digital write and the setting of the start time) into a private function and call that from both places. If you name that function clearly (eg: `beginClickDown`), then it might be reasonable to set the value of `state` in

that function. As is often the case, it doesn't matter what you do, so long as you are consistent and name things appropriately.

So let's test it! I'll modify the button click code like so:

```
        else if (millis() - buttonDownMs < 250) {
          // short click
          // TEST BY TOGGLING PIN 8
          digitalWrite(8, !digitalRead(8));
          clickQueue.addAClick();
        }
```

And - it doesn't work. Of course it doesnt: I have to gibe this new object some time slices.

```
void setup() {
  button.setup();
  clickQueue.setup();

  // FOR TESTING
  pinMode(8, OUTPUT);
  pinMode(9, OUTPUT);
}

void loop() {
  button.loop();
  clickQueue.loop();
}
```

And now it works like a charm. When I short-click the button four times, the LED on pin 8 switches on/off (the testing code) and the LED on pin 11 flashes 4 times slowly.

Cancel

Ok, what about cancel? Specifically, what do we do if a cancel occurs while we are in the middle of a click? What if we turn off the click, and then have another click immediately after? This requires a decision. I am going to go the simple way and have cancel simply turn off the click unconditionally. This is because I know that in this case, the sketch will be hitting cancel only when the power to the light is going to be going off, so having hanging state is not really a concern.

```
    void cancel() {
      pendingClicks = 0;
      digitalWrite(pin, LOW);
      state = NOT_CLICKING;
    }
```

If we need things like cool-down periods before turning off the power - well, we would have to code for that. In this simple case, no.

So let's test it:

```
        else  {
          // long click
          // TEST BY TOGGLING PIN 9
          digitalWrite(9, !digitalRead(9));
          clickQueue.cancel();
        }
```

And again, it works perfectly. A series of short clicks will queue up a series of flashes, and a long click will cancel them all.

The rest of the light: composition

Ok, now that we have a thing that can click the clicky thing to advance the light through its brightness levels, its time to think about power on/off.

The main thing we want is that if power goes of and then on again, we want the arduino to advance the brightness to its prior knows state.

So, lets get the basics going. We write our basic headlamp class:

```
class Headlamp {
    const byte powerOutPin;
    const byte brightnessOutPin;

  public:
    Headlamp(byte powerOutAttach, byte brightnessOutAttach) :
      powerOutPin(powerOutAttach),
      brightnessOutPin(brightnessOutAttach)
```

```
    {
    }

    void setup() {
    }

    void loop() {
    }

    void powerToggle() {
    }

    void brightnessClick() {
    }

} headlamp(10, 11);
```

Alter the button code to sent messages to that instead of the click queue

```
        else if (millis() - buttonDownMs < 250) {
          // short click
          // TEST BY TOGGLING PIN 8
          digitalWrite(8, !digitalRead(8));
          headlamp.brightnessClick();
        }
        else  {
          // long click
          // TEST BY TOGGLING PIN 9
          digitalWrite(9, !digitalRead(9));
          headlamp.powerToggle();
        }
```

And alter the main setup and loop to include this new object,

```
void setup() {
  button.setup();
  clickQueue.setup();
  headlamp.setup();

  // FOR TESTING
  pinMode(8, OUTPUT);
  pinMode(9, OUTPUT);
}

void loop() {
  button.loop();
  clickQueue.loop();
  headlamp.loop();
}
```

Now, at this point I want to think about pins. I want to think of the headlamp as a single object, with a power output pin and a brightness click output pin. But the brightness pin needs to be operated by a ClickQueue. What I have will kinda work, because both the click queue and the headlamp are listening to pin 11, but that's the wrong way to go about things: a pin should be "owned" by only one object.

So what I am going to do is to build a compound object by composition. The click queue will be part of the headlamp.

This does not mean that the `Headlamp` class needs to have the the `ClickQueue` class inside it. Instead, it means that the `Headlamp` class has an instance of the `ClickQueue` class in it.

The result looks line this:

```
class Headlamp {
    const byte powerOutPin;
    ClickQueue brightnessClicker;

  public:
    Headlamp(byte powerOutAttach, byte brightnessOutAttach) :
      powerOutPin(powerOutAttach),
      brightnessClicker(brightnessOutAttach)
```

```
    {
    }

    void setup() {
      brightnessClicker.setup();
    }

    void loop() {
      brightnessClicker.loop();
    }

    void powerToggle() {
    }

    void brightnessClick() {
    }

} headlamp(10, 11);
```

Because the brightness clicker is part of the headlamp, the headlamp becomes responsible for setting it up with its constructor, and for calling its `setup` and `loop` methods.

Consequently, we get rid of the clickQueue instance altogether - it's all headlamp now.

From here, we put the code into the headlamp that keeps track of what the brightness is and turns things on and off. The result looks like this:

```
class Headlamp {
    const byte powerOutPin;
    ClickQueue brightnessClicker;
    boolean isOn;
    int brightness;

  public:
    Headlamp(byte powerOutAttach, byte brightnessOutAttach) :
      powerOutPin(powerOutAttach),
      brightnessClicker(brightnessOutAttach)
    {
    }

    void setup() {
      brightnessClicker.setup();

      pinMode(powerOutPin, OUTPUT);
      isOn = false;
      digitalWrite(powerOutPin, LOW);

      brightness = 0;
    }

    void loop() {
      brightnessClicker.loop();
    }

    void powerToggle() {
      if (isOn) {
        brightnessClicker.cancel();
        digitalWrite(powerOutPin, LOW);
        isOn = false;
      }
      else {
        digitalWrite(powerOutPin, HIGH);
        delay(50);
        brightnessClicker.addClicks(brightness);
        isOn = true;
      }
```

```
    }

    void brightnessClick() {
      if (isOn) {
        brightnessClicker.addAClick();
        brightness = (brightness + 1 ) % 5;
      }
    }

} headlamp(10, 11);
```

You may notice that I have had to add a "addClicks" method to the clicker. This is not a big deal.

You may also notice that I have added a 50ms delay to the power toggle. This is a potential source of trouble. But the only thing that is going to happen is that a short click might be ignored after a long click to turn the lamp on. This is something we can live with.

And you know what? It works exactly as it should. Short clicks increment the brightness when the lamp is on; Long clicks turn it off and on; and when the lamp is turned on there's a series of clicks to bring it up to the previous level of brightness.

Job done. Well - there's some refinements to make.

An aside

Re-reading this after having not visited for a few months, there's something worth noting in this section. The comment "now that we have a thing that can click the clicky thing". I don't say "I have a function" to do it or "come code" to do it. I have a thingy to do it.

When you code the OO way, once you have built an object you proceed to think of it as a black box. It's like plugging an IC into a circuit. Notice that none of the code in `Headlamp` deals with the pin at all - `Headlamp` doesn't even save the value of `brightnessOutAttach`. It' relies entirely on `brightnessClicker` to deal with it.

`Headlamp` doesn't "think" about `brightnessClicker` in terms of pins, timings, internal state. All it knows is that `brightnessClicker` is a `ClickQueue`.That means you can add as many clicks to it as you want, and you can at any time cancel all the clicks. That's all. Which is good. The whole point of encapsulating code and state in a class is that once you have written (**and tested!**) `ClickQueue` you can unload from your current train-of-thought all the design you did about *how* it works. All you need to care about from here on in is *what it does*.

Separating out the button behaviour: polymorphism

It really irks me that our very abstract short-click/long-click button has a bunch of code in it that's specific to headlamps in particular. I want to separate out the bit that worries about timing from the bit that thinks about headlamps.

The way I am going to do this is with inheritane. My base button class will only think about timings, and will call a `longClick` and `shortClick` method. But here's the thing: these methods will not be implemented. It looks like this:

```
class Button {
    const byte pin;
    int state;
    unsigned long buttonDownMs;

  protected:
    virtual void shortClick() = 0;
    virtual void longClick() = 0;

  public:
    Button(byte attachTo) :
      pin(attachTo)
    {
    }


    void setup() {
      pinMode(pin, INPUT_PULLUP);
      state = HIGH;
    }

    void loop() {
      int prevState = state;
      state = digitalRead(pin);
      if (prevState == HIGH && state == LOW) {
        buttonDownMs = millis();
```

```
      }
      else if (prevState == LOW && state == HIGH) {
        if (millis() - buttonDownMs < 50) {
          // ignore this for debounce
        }
        else if (millis() - buttonDownMs < 250) {
          shortClick();
        }
        else  {
          longClick();
        }
      }
    }

};
```

We then have what we call a subclass of button whose job it is to know how to work a headlight. Crucially, that subclass has everything that a button has - it uses the button's `setup` and loop . So the only thing it needs is that it needs to supply the definition of what to do if there's a short and a long click.

```
class HeadlampControlButton: public Button {
  public:
    HeadlampControlButton(byte attachTo) :
      Button(attachTo) {
    }
  protected:
    void shortClick() {
      headlamp.brightnessClick();
    }

    void longClick() {
      headlamp.powerToggle();
    }
};

HeadlampControlButton button(7);
```

With this change, the sketch - well - does exactly what it did before :). But now, if we want to make other long/short clicky buttons on difefrent pins doing different things, it's easy.

Wiring up the buttons with constructors: composition by reference

The next thing that irritates me is how the HeadlampControlButton object is tied to the headlamp object defined in the main part of the sketch. What if we want two headlamps and buttons? What if we want to call it something different?

So what I will do is have a *HeadlampControlButton* hold a reference to its headlight. It is given this reference in the constructor, which means that the thing that builds it is responsible for working out which headlight it is supposed to be tied to.

Because the object does not "own" the headlight, it will not call the `setup` and `loop` methds.

It looks a little like this:

```
class HeadlampControlButton: public Button {
    Headlamp &lamp;

  public:
    HeadlampControlButton(byte attachToPin, Headlamp &attachToHeadlamp) :
      Button(attachToPin),
      lamp(attachToHeadlamp) {
    }
  protected:
    void shortClick() {
      lamp.brightnessClick();
    }

    void longClick() {
      lamp.powerToggle();
    }
};
```

At this point, the main part of the sketch looks like this

```
Headlamp headlamp(10, 11);
HeadlampControlButton button(7, headlamp);

void setup() {
  button.setup();
  headlamp.setup();
}

void loop() {
  button.loop();
  headlamp.loop();
}
```

See how it is now the declarations in the main part of the sketch that defines who uses what pins and how they interact. Each component only "knows" about its own little world and can be swapped in and out on one place without needing other coding.

The full, working sketch

As you have been so patient looking at these code fragments and snippets, here is the full thing.

```
class ClickQueue {
    const byte pin;
    // make these slow for testing
    const unsigned CLICK_DOWN_MS = 250;
    const unsigned CLICK_TOTAL_MS = 1000;

    enum State {
      NOT_CLICKING = 0,
      CLICK_DOWN = 1,
      CLICK_PAUSE = 2
    } state;

    unsigned long clickStartMs;
    int pendingClicks;

  public:
    ClickQueue(byte attachTo) :
      pin(attachTo) {
    }

    void setup() {
      pinMode(pin, OUTPUT);
      state = NOT_CLICKING;
      pendingClicks = 0;
    }

    void loop() {
      switch (state) {
        case NOT_CLICKING:
          if (pendingClicks > 0) {
            pendingClicks --;
            digitalWrite(pin, HIGH);
            clickStartMs = millis();
            state = CLICK_DOWN;
          }
          break;
        case CLICK_DOWN:
          if (millis() - clickStartMs > CLICK_DOWN_MS) {
            digitalWrite(pin, LOW);
            state = CLICK_PAUSE;
          }
          break;
        case CLICK_PAUSE:
          if (millis() - clickStartMs > CLICK_TOTAL_MS) {
```

```
            state = NOT_CLICKING;
          }
          break;
      }
    }

    void addAClick() {
      pendingClicks++;
    }

    void addClicks(int clicks) {
      pendingClicks += clicks;
    }

    void cancel() {
      pendingClicks = 0;
      digitalWrite(pin, LOW);
      state = NOT_CLICKING;
    }
};

class Headlamp {
    const byte powerOutPin;
    ClickQueue brightnessClicker;
    boolean isOn;
    int brightness;

  public:
    Headlamp(byte powerOutAttach, byte brightnessOutAttach) :
      powerOutPin(powerOutAttach),
      brightnessClicker(brightnessOutAttach)
    {
    }

    void setup() {
      brightnessClicker.setup();

      pinMode(powerOutPin, OUTPUT);
      isOn = false;
      digitalWrite(powerOutPin, LOW);

      brightness = 0;
    }

    void loop() {
      brightnessClicker.loop();
    }

    void powerToggle() {
      if (isOn) {
        brightnessClicker.cancel();
        digitalWrite(powerOutPin, LOW);
        isOn = false;
      }
      else {
        digitalWrite(powerOutPin, HIGH);
        delay(50);
        brightnessClicker.addClicks(brightness);
        isOn = true;
      }
    }

    void brightnessClick() {
      if (isOn) {
```

```
        brightnessClicker.addAClick();
        brightness = (brightness + 1 ) % 5;
      }
    }

} ;

class Button {
    const byte pin;
    int state;
    unsigned long buttonDownMs;

  protected:
    virtual void shortClick() = 0;
    virtual void longClick() = 0;

  public:
    Button(byte attachTo) :
      pin(attachTo)
    {
    }


    void setup() {
      pinMode(pin, INPUT_PULLUP);
      state = HIGH;
    }

    void loop() {
      int prevState = state;
      state = digitalRead(pin);
      if (prevState == HIGH && state == LOW) {
        buttonDownMs = millis();
      }
      else if (prevState == LOW && state == HIGH) {
        if (millis() - buttonDownMs < 50) {
          // ignore this for debounce
        }
        else if (millis() - buttonDownMs < 250) {
          shortClick();
        }
        else  {
          longClick();
        }
      }
    }

};

class HeadlampControlButton: public Button {
    Headlamp &lamp;

  public:
    HeadlampControlButton(byte attachToPin, Headlamp &attachToHeadlamp) :
      Button(attachToPin),
      lamp(attachToHeadlamp) {
    }
  protected:
    void shortClick() {
      // short click
      lamp.brightnessClick();
    }

    void longClick() {
```

```
      // long click
      lamp.powerToggle();
    }
};

Headlamp headlamp(10, 11);
HeadlampControlButton button(7, headlamp);

void setup() {
  button.setup();
  headlamp.setup();
}

void loop() {
  button.loop();
  headlamp.loop();
}
```

Part II - adding the tail light

Now, one of the the motivations for going to all this trouble is to demonstrate that it's reasonably easy to modify and extend tour sketch. I will add a tail and brake light that flashes intermittently when the headlamp is on and comes on when the brake is hit.

Before I do, stop and think what sort of a challenge this would be with the usual way of coding - one big `loop` method doing the lamp on of, keeping track of the brightness, bringning the brightness up when the lamp is turned in, the whole thing. Coding this way - the OO way - means that adding another thingumajig is mostly more of the same.

Flashing Tail Light

Ok, our tail light has two independent state things - on/off, and braking/not braking - and it's attached to a pin. Lets just code up one of those.

```
class Taillight {
    const byte pin;
    boolean flashing;
    boolean braking;

  public:
    Taillight(byte attachToPin) :
      pin(attachToPin) {
    }

    void setup() {
      pinMode(pin, OUTPUT);
      digitalWrite(pin, LOW);
      flashing = true; // SETTING to true for testing
      braking = false;
    }

    void loop() {
    }
};
```

And we declare one, attach it to pin 12, and give it a setup and a time slice.

```
Headlamp headlamp(10, 11);
HeadlampControlButton button(7, headlamp);
Taillight taillight(12);

void setup() {
  button.setup();
  headlamp.setup();
  taillight.setup();
}

void loop() {
  button.loop();
  headlamp.loop();
```

```
  taillight.loop();
}
```

Great! Now, for logic - the loop. If the brake is on, then the taillight should be on. If the brake is not on and it's day (not flashing), the LED should be off. otherwise, it should flash.

Now, for flashing, I am totally going to cheat. I want it to flash every second. You know what's really close to a second? 1024 milliseconds. Bit 10 of `millis()` . And I want it to be on 1/10th of the time. You know what's really close to 1/10? 1/8. Three bits. So I will look at three bits at position 10 to 8 of millis. If they are all off, then the light is on. That should make it flash like a boss.

I won't bother with "if it was off before and it's on now, then we need to turn on the pin". I'll just unconditionally write the state to the pin. Meh.

```
    void loop() {
      if (braking) {
        digitalWrite(pin, HIGH);
      }
      else if (!flashing) {
        digitalWrite(pin, LOW);
      }
      else {
        digitalWrite(pin, (millis() & 0b1110000000) == 0 ? HIGH : LOW);
      }
    }
```

Flashes just like a bought one! Fantastic. Now to do the rest of the state.

Hooking the taillight to the headlamp

Next thing I need to do is to have the taillight flashing only if the headlamp is on. Now, I don't want to do a lot of surgery to the headlamp. So instead of having the headlamp push its state to the taillight, I will have the taillight pull the data from the headlamp. This means the headlamp needs to expose its isOn state. You can do this by having a method that returns the value (an accessor method), but I will just make the variable public.

This breaks some of the rules, because it means that another object can directly write to that state without the headlamp class knowing about it. This could put the object into an invalid state. YOu would not do this if you were writing a library class, meant to be used by other people.

But in this case, it's just me personally writing this thing. So - meh. I need the taillight to hold a reference to a headlamp, and I actually don't need the 'flashing' variable at all.

So, the variables at the top of the headlamp class now look like this:

```
    const byte powerOutPin;
    ClickQueue brightnessClicker;
    int brightness;

  public:
    boolean isOn;
```

The declarations in the main sketch that create our components look like this:

```
Headlamp headlamp(10, 11);
HeadlampControlButton button(7, headlamp);
Taillight taillight(12, headlamp);
```

And the taillight looks like this:

```
class Taillight {
    const byte pin;
    Headlamp &headlamp;
    boolean braking;

  public:
    Taillight(byte attachToPin, Headlamp &attachToHeadlamp) :
      pin(attachToPin),
      headlamp(attachToHeadlamp) {
    }

    void setup() {
      pinMode(pin, OUTPUT);
      digitalWrite(pin, LOW);
      braking = false;
```

```
    }

    void loop() {
      if (braking) {
        digitalWrite(pin, HIGH);
      }
      else if (!headlamp.isOn) {
        digitalWrite(pin, LOW);
      }
      else {
        digitalWrite(pin, (millis() & 0b1110000000) == 0 ? HIGH : LOW);
      }
    }
};
```

And once again, it works. A long click turns the headlamp on and starts the flashing taillight. Short clicks advance the brightness. A long click turns everything off. Another long click starts the falshing and sends a series of clicks to the brightness to bring it back up to where it was.

Adding the brake switch

You know what? I am not going to bother with making up a brake switch object. I'll just have the tail light read the pin itself. It's justifiable because, well, its a brake light so its ok to have it listen to a brake pin.

Easy as. I'll put the switch on pin 6. I'll reorder the constructor arguments so that inputs come first, then outputs. And since the taillight now is dealing with two pins, I'll rename `pin` to `ledOutPin`

The full, working sketch, with tail light

And here's the final result. It seems to work exactly right.

```
class ClickQueue {
    const byte pin;
    // make these slow for testing
    const unsigned CLICK_DOWN_MS = 250;
    const unsigned CLICK_TOTAL_MS = 1000;

    enum State {
      NOT_CLICKING = 0,
      CLICK_DOWN = 1,
      CLICK_PAUSE = 2
    } state;

    unsigned long clickStartMs;
    int pendingClicks;

  public:
    ClickQueue(byte attachTo) :
      pin(attachTo) {
    }

    void setup() {
      pinMode(pin, OUTPUT);
      state = NOT_CLICKING;
      pendingClicks = 0;
    }

    void loop() {
      switch (state) {
        case NOT_CLICKING:
          if (pendingClicks > 0) {
            pendingClicks --;
            digitalWrite(pin, HIGH);
            clickStartMs = millis();
            state = CLICK_DOWN;
          }
          break;
```

```
        case CLICK_DOWN:
          if (millis() - clickStartMs > CLICK_DOWN_MS) {
            digitalWrite(pin, LOW);
            state = CLICK_PAUSE;
          }
          break;
        case CLICK_PAUSE:
          if (millis() - clickStartMs > CLICK_TOTAL_MS) {
            state = NOT_CLICKING;
          }
          break;
      }
    }

    void addAClick() {
      pendingClicks++;
    }

    void addClicks(int clicks) {
      pendingClicks += clicks;
    }

    void cancel() {
      pendingClicks = 0;
      digitalWrite(pin, LOW);
      state = NOT_CLICKING;
    }
};

class Headlamp {
    const byte powerOutPin;
    ClickQueue brightnessClicker;
    int brightness;

  public:
    boolean isOn;

    Headlamp(byte powerOutAttach, byte brightnessOutAttach) :
      powerOutPin(powerOutAttach),
      brightnessClicker(brightnessOutAttach)
    {
    }

    void setup() {
      brightnessClicker.setup();

      pinMode(powerOutPin, OUTPUT);
      isOn = false;
      digitalWrite(powerOutPin, LOW);

      brightness = 0;
    }

    void loop() {
      brightnessClicker.loop();
    }

    void powerToggle() {
      if (isOn) {
        brightnessClicker.cancel();
        digitalWrite(powerOutPin, LOW);
        isOn = false;
      }
      else {
```

```
        digitalWrite(powerOutPin, HIGH);
        delay(50);
        brightnessClicker.addClicks(brightness);
        isOn = true;
      }
    }

    void brightnessClick() {
      if (isOn) {
        brightnessClicker.addAClick();
        brightness = (brightness + 1 ) % 5;
      }
    }

} ;

class Button {
    const byte pin;
    int state;
    unsigned long buttonDownMs;

  protected:
    virtual void shortClick() = 0;
    virtual void longClick() = 0;

  public:
    Button(byte attachTo) :
      pin(attachTo)
    {
    }


    void setup() {
      pinMode(pin, INPUT_PULLUP);
      state = HIGH;
    }

    void loop() {
      int prevState = state;
      state = digitalRead(pin);
      if (prevState == HIGH && state == LOW) {
        buttonDownMs = millis();
      }
      else if (prevState == LOW && state == HIGH) {
        if (millis() - buttonDownMs < 50) {
          // ignore this for debounce
        }
        else if (millis() - buttonDownMs < 250) {
          shortClick();
        }
        else  {
          longClick();
        }
      }
    }

};

class HeadlampControlButton: public Button {
    Headlamp &lamp;

  public:
    HeadlampControlButton(byte attachToPin, Headlamp &attachToHeadlamp) :
      Button(attachToPin),
```

```
      lamp(attachToHeadlamp) {
    }
  protected:
    void shortClick() {
      // short click
      lamp.brightnessClick();
    }

    void longClick() {
      // long click
      lamp.powerToggle();
    }
};

class Taillight {
    const byte brakeSensePin;
    const byte ledOutPin;
    Headlamp &headlamp;

  public:
    Taillight(byte attachToBrakeSense, Headlamp &attachToHeadlamp, byte attachToLedPin) :
      brakeSensePin(attachToBrakeSense),
      ledOutPin(attachToLedPin),
      headlamp(attachToHeadlamp) {
    }

    void setup() {
      pinMode(brakeSensePin, INPUT_PULLUP);
      pinMode(ledOutPin, OUTPUT);
      digitalWrite(ledOutPin, LOW);
    }

    void loop() {
      if (digitalRead(brakeSensePin) == LOW) {
        digitalWrite(ledOutPin, HIGH);
      }
      else if (!headlamp.isOn) {
        digitalWrite(ledOutPin, LOW);
      }
      else {
        digitalWrite(ledOutPin, (millis() & 0b1110000000) == 0 ? HIGH : LOW);
      }
    }
};

Headlamp headlamp(10, 11);
HeadlampControlButton button(7, headlamp);
Taillight taillight(6, headlamp, 12);

void setup() {
  button.setup();
  headlamp.setup();
  taillight.setup();
}

void loop() {
  button.loop();
  headlamp.loop();
  taillight.loop();
}
```

Final thoughts

Well, that seems like a lot of code. But review what this sketch does - two inputs, one of them timer based, and three outputs. Flashing,

internal state, a little routine that must be done when the light comes on. It's all there, and it all works seamlessly.

Using the "objects where each gets a time slice" pattern, no individual step was particularly difficult to do. At each step, I could check that the object I had written so far worked. And once it worked, I mostly didn't have to go back to revisit things.

Yes, it does require programming skills beyond the cut-and-paste cookbook approach. You actually have to know how to program in C++. I appreciate that many Arduino hobbyists are more interested in hardware and electronics than in programming. What can I say? If you want your program to do something complex, grinding away at a wall of nested if/else-if statements is a recipe for long frustrating hours and at the end of it all, you can't be 100% confident that you really have thought of everything, caught every possible condition.

Writing this stuff is much quicker and easier than writing *about* it. With this approach, your new project works basically the same as your previous one. Which is good. Your code matches the physical components you have. If you decide that your bike doesn't need a taillight, removing it from the sketch above is just a matter of a couple of lines, not of hunting through the code to find every place where it is mentioned.

The sketch has some issues, and I wouldn't sell it like that. The way the taillight depends on the lamp is not good. What I probably need is a 'pushbutton to toggle' class. A software t-flipflop. Both the headlamp and the brake light would read that. I'd have a 'runnable' abstract class for things that have a setup and loop, and manage them with an array rather than hard-coding individual things into the main setup and loop.

But, there you have it. This is one way that a dude who has been making a living programming for 30 years addresses Arduino projects.

Appendix - automatically managing runnables

One of the irritations of this pattern is making sure that you give a time-slice to every component. Your main `setup` and `loop` have to include everything explicitly by name.

As every one of these objects has a `setup` and `loop` method, there's a pattern there. Perhaps there should be an abstract superclass named - let's say `Runnable` - and all these classes should inherit that class:

```
class Runnable {
  public:
    virtual void setup() = 0;
    virtual void loop() = 0;
};

class ClickQueue: public Runnable {
        …

class Headlamp: public Runnable {
        …

class Button: public Runnable {
        …

class HeadlampControlButton: public Button {
        …

class Taillight: public Runnable {
        …
```

That way, you just keep an array of runnables and iterate through them in your `loop` and `setup` .

```
Headlamp headlamp(10, 11);
HeadlampControlButton button(7, headlamp);
Taillight taillight(6, headlamp, 12);

Runnable *runMe[] = {
  &button,
  &headlamp,
  &taillight
};

void setup() {
  for (int i = 0; i < sizeof(runMe) / sizeof(*runMe); i++)
    runMe[i]->setup();
}

void loop() {
  for (int i = 0; i < sizeof(runMe) / sizeof(*runMe); i++)
```

```
    runMe[i]->loop();
}
```

Pretty snazzy. But maintianing that array of pointers is a bit of a wart. What we can do, however, is to have the `Runnable` constructors automagically build a linked list of runnables.

```
class Runnable {
    static Runnable *headRunnable;
    Runnable *nextRunnable;

  public:
    Runnable() {
      nextRunnable = headRunnable;
      headRunnable = this;
    }

    virtual void setup() = 0;
    virtual void loop() = 0;

    static void setupAll() {
      for (Runnable *r = headRunnable; r; r = r->nextRunnable)
        r->setup();
    }

    static void loopAll() {
      for (Runnable *r = headRunnable; r; r = r->nextRunnable)
        r->loop();
    }

};

Runnable *Runnable::headRunnable = NULL;
```

The main section of the sketch now looks like this:

```
Headlamp headlamp(10, 11);
HeadlampControlButton button(7, headlamp);
Taillight taillight(6, headlamp, 12);

void setup() {
  Runnable::setupAll();
}

void loop() {
  Runnable::loopAll();
}
```

A little spooky, because all the runnable objects get put on the queue invisibly. But it works. And you can remove the bit inside `Headlamp` where it calls the `setup` and `loop` of its embedded `ClickQueue` . The embedded `ClickQueue` also automagically adds itself to the list of `Runnables`

Why do this? Well - I like it that way. Sometimes, it just comes down to personal preference. I like the way that, using this pattern, you simply create a runnable object and suddenly it comes alive. You can make an array of them if you want and they *all* come alive.

Now you may be wondering, "Wait! What? What order will things get run in?"

It doesn't matter. Everything gets a time-slice. We could write this thing so that each `loop` , one *random* runnable was selected and run. It would still work. Everything would (most probably) get a time-slice in a reasonable amount of time.

The key to that is that if an object exposes methods that defer things to be done later, during their loop, it must be done in such a way that it can handle any number of calls on those methods before loop gets done.

That is, any object that works this way must either maintain some sort of queue of things-to-do, or expose a "ready" method that the things that use it interrogate. In our case, the clickqueue holds a queue of pending events simply by keeping the number of pending clicks. More complex things that use this model may need to maintain some sort of buffer or use the heap.

Can this result in deadlocks? Sure! If your buffers aren't big enough, if you are asking your arduino to do something that it just plain doesn't have enough memory for, if your serial cable falls out, or (here's the salient bit) if you program it up wrong.

But even then, using objects makes it much easier do draw up dependency and data-flow diagrams to try to work out what's jamming, because you already know what rectangles you need to draw and where the arrows go.

Anyway. Here again is the complete sketch.

```
class Runnable {
    static Runnable *headRunnable;
    Runnable *nextRunnable;

  public:
    Runnable() {
      nextRunnable = headRunnable;
      headRunnable = this;
    }

    virtual void setup() = 0;
    virtual void loop() = 0;

    static void setupAll() {
      for (Runnable *r = headRunnable; r; r = r->nextRunnable)
        r->setup();
    }

    static void loopAll() {
      for (Runnable *r = headRunnable; r; r = r->nextRunnable)
        r->loop();
    }

};

Runnable *Runnable::headRunnable = NULL;

class ClickQueue: public Runnable {
    const byte pin;
    // make these slow for testing
    const unsigned CLICK_DOWN_MS = 250;
    const unsigned CLICK_TOTAL_MS = 1000;

    enum State {
      NOT_CLICKING = 0,
      CLICK_DOWN = 1,
      CLICK_PAUSE = 2
    } state;

    unsigned long clickStartMs;
    int pendingClicks;

  public:
    ClickQueue(byte attachTo) :
      pin(attachTo) {
    }

    void setup() {
      pinMode(pin, OUTPUT);
      state = NOT_CLICKING;
      pendingClicks = 0;
    }

    void loop() {
      switch (state) {
        case NOT_CLICKING:
          if (pendingClicks > 0) {
            pendingClicks --;
            digitalWrite(pin, HIGH);
            clickStartMs = millis();
            state = CLICK_DOWN;
          }
```

```
          break;
        case CLICK_DOWN:
          if (millis() - clickStartMs > CLICK_DOWN_MS) {
            digitalWrite(pin, LOW);
            state = CLICK_PAUSE;
          }
          break;
        case CLICK_PAUSE:
          if (millis() - clickStartMs > CLICK_TOTAL_MS) {
            state = NOT_CLICKING;
          }
          break;
      }
    }

    void addAClick() {
      pendingClicks++;
    }

    void addClicks(int clicks) {
      pendingClicks += clicks;
    }

    void cancel() {
      pendingClicks = 0;
      digitalWrite(pin, LOW);
      state = NOT_CLICKING;
    }
};

class Headlamp: public Runnable {
    const byte powerOutPin;
    ClickQueue brightnessClicker;
    int brightness;

  public:
    boolean isOn;

    Headlamp(byte powerOutAttach, byte brightnessOutAttach) :
      powerOutPin(powerOutAttach),
      brightnessClicker(brightnessOutAttach)
    {
    }

    void setup() {
      pinMode(powerOutPin, OUTPUT);
      isOn = false;
      digitalWrite(powerOutPin, LOW);
      brightness = 0;
    }

    void loop() {
    }

    void powerToggle() {
      if (isOn) {
        brightnessClicker.cancel();
        digitalWrite(powerOutPin, LOW);
        isOn = false;
      }
      else {
        digitalWrite(powerOutPin, HIGH);
        delay(50);
        brightnessClicker.addClicks(brightness);
```

```
        isOn = true;
      }
    }

    void brightnessClick() {
      if (isOn) {
        brightnessClicker.addAClick();
        brightness = (brightness + 1 ) % 5;
      }
    }

} ;

class Button: public Runnable {
    const byte pin;
    int state;
    unsigned long buttonDownMs;

  protected:
    virtual void shortClick() = 0;
    virtual void longClick() = 0;

  public:
    Button(byte attachTo) :
      pin(attachTo)
    {
    }


    void setup() {
      pinMode(pin, INPUT_PULLUP);
      state = HIGH;
    }

    void loop() {
      int prevState = state;
      state = digitalRead(pin);
      if (prevState == HIGH && state == LOW) {
        buttonDownMs = millis();
      }
      else if (prevState == LOW && state == HIGH) {
        if (millis() - buttonDownMs < 50) {
          // ignore this for debounce
        }
        else if (millis() - buttonDownMs < 250) {
          shortClick();
        }
        else  {
          longClick();
        }
      }
    }

};

class HeadlampControlButton: public Button {
    Headlamp &lamp;

  public:
    HeadlampControlButton(byte attachToPin, Headlamp &attachToHeadlamp) :
      Button(attachToPin),
      lamp(attachToHeadlamp) {
    }
  protected:
```

```
    void shortClick() {
      // short click
      lamp.brightnessClick();
    }

    void longClick() {
      // long click
      lamp.powerToggle();
    }
};

class Taillight: public Runnable {
    const byte brakeSensePin;
    const byte ledOutPin;
    Headlamp &headlamp;

  public:
    Taillight(byte attachToBrakeSense, Headlamp &attachToHeadlamp, byte attachToLedPin) :
      brakeSensePin(attachToBrakeSense),
      ledOutPin(attachToLedPin),
      headlamp(attachToHeadlamp) {
    }

    void setup() {
      pinMode(brakeSensePin, INPUT_PULLUP);
      pinMode(ledOutPin, OUTPUT);
      digitalWrite(ledOutPin, LOW);
    }

    void loop() {
      if (digitalRead(brakeSensePin) == LOW) {
        digitalWrite(ledOutPin, HIGH);
      }
      else if (!headlamp.isOn) {
        digitalWrite(ledOutPin, LOW);
      }
      else {
        digitalWrite(ledOutPin, (millis() & 0b1110000000) == 0 ? HIGH : LOW);
      }
    }
};

Headlamp headlamp(10, 11);
HeadlampControlButton button(7, headlamp);
Taillight taillight(6, headlamp, 12);


void setup() {
  Runnable::setupAll();
}

void loop() {
  Runnable::loopAll();
}
```