# COSC 1P03 — Assignment 3

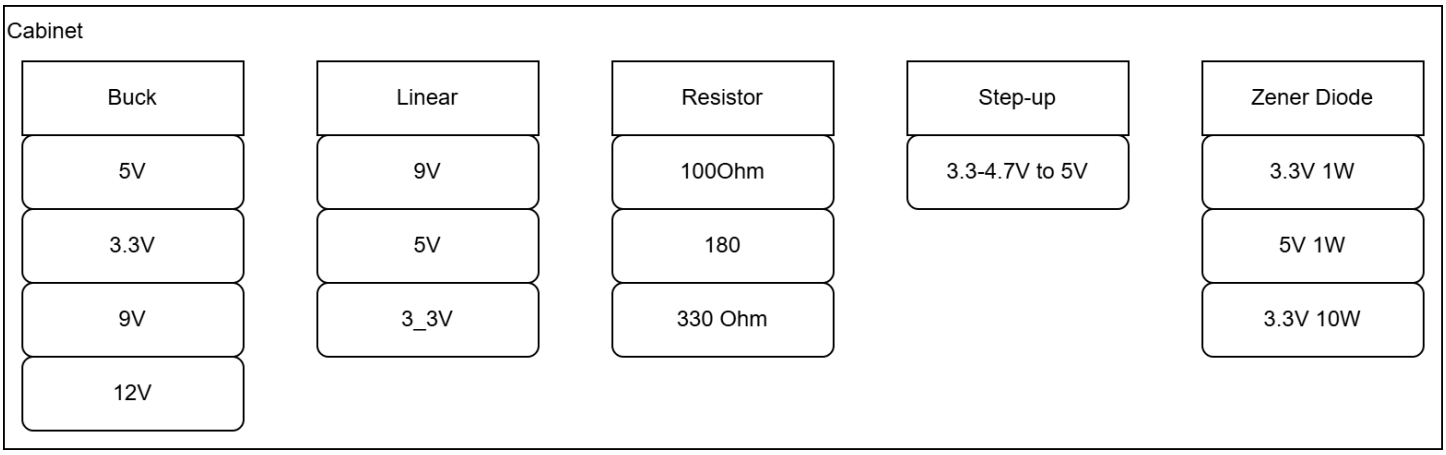
I need to make a trip to Ikea some time soon

## Background

There are myriad ways to store and organize records, and there's always value in exploring and inventing new mechanisms for doing so. In this case, consider a **Cabinet**. There are Parts Cabinets, Filing cabinets, etc., and most of them can even be repurposed for holding other types of things. They have multiple drawers (or *bins*), and if you're organizing your stuff well, each bin will represent a different category.

Consider, for example, the Parts Cabinet on the left from Canadian Tire's site.

In the example below, I have five categories for voltage regulators, with 1–4 examples of each.




For Java, we can call it `Cabinet`, and have it hold *any* type (so long as they're all the *same type*), organized into `String`-identified `Bin`s. This type supports an indefinite number of `Bin`s, which are organized lexicographically within the cabinet (i.e. sorted by the labels attached to each `Bin`). Each `Bin` itself has a different means of organization within it: since we always pile new crap onto the old crap already there, the assumption is that whenever we take anything out, it'll be the most-recently added item for that label.

## Specifications: `Cabinet`s and `Bin`s

Refer to the included interfaces to see how a `Cabinet` and a `Bin` should work.

Note that, while although a `Bin` *in general* has no prescribed sequencing, for the version you'll be implementing it's expected that it will always yield the most-recently added element. For the scope of this task, the only way a client program can get a `Bin` is by asking the `Cabinet` to create and return one.

## Client Programs

You are required to have two client programs, both within the `client` package.

First, write a very simple `Demonstration` program to let the user preload a data file for the contents of a `Cabinet`, and then enter very basic commands for testing out (*most* of) the required methods:

- Adding/removing individual members, of some requested category label
  - Must correctly handle trying to remove from an empty category (you *should* understand this now!)
- Removing an entire `Bin` altogether, and displaying its contents
  - You don't need to worry about *adding a* `Bin` back in here, as the second program will include that
- Displaying the `Bin` labels, and number of entries stored at each label

One such data file is already included (that, if loaded into a correctly-written `Cabinet`, will be stored as in the diagram above).

The second program (in the same project!) is *nostalgia time* for 1P02!

- Your `Cabinet` will hold 5 samples each of `Pictures`/colour swatches for different colour categories
- A simple `BasicForm` will be used to 'check out' a `Bin` corresponding to a colour of the user's choice, to cycle through displaying them to the user. The Bin is then re-added to the `Cabinet`
- This loops until the user quits

Note: there's a draft (`Wacky`). You can comment/document it, and change it to match the name of your `Cabinet`'s implementation.

## Tips

A few things to note:

- The specifications dictate how client software interacts with the interfaces; it does not (necessarily) indicate how concrete classes should — or must — be implemented
    - For example, though a `Cabinet` may return a `Bin`, there's no expectation that the Cabinet be implemented as *using* `Bin`s for its internal representation. A `Bin` can simply be created and populated by the `Cabinet` when the client needs one.
    - Also, since the `Cabinet` itself yields elements in effectively the opposite sequence in which they were added, and the `Bin` *also* does that, that means (in a correct implementation) a client may have access to elements in either direction of sequence: remove directly for the reverse sequence, or request a `Bin` and deplete *that* to get it in the original sequence
- You might have noticed there's no *direct* way to see the contents of the `Cabinet` without removing them. For example, though `Bin`s are `Iterable`, the `Cabinet` is *not*. This is by design: you ***must*** understand your organization *before* you start coding or else something will fail. Spectacularly. (Conversely, working out enough illustrations first can potentially make this the simplest task all term)
- Access modifiers *matter*, including the *lack* of them, where appropriate
- If it's not *painfully* obvious, this is "the linked list" assignment
- (This *should* be obvious, but if anything anywhere within your `storage` package performs *any* form of output at all, expect a zero. that is obvious by now, right? yes? good.)
- Similarly, it should go without saying that you may not modify the provided interface files **at all**. This is an instant zero, and so easily avoided. Please don't make me add a little bit extra to this disclaimer each time this type of assignment is offered

## Submission

You'll be submitting a **.zip** (and only .zip) file to Brightspace.
Your submission must include:

- The complete IntelliJ project you wrote for this assignment
    - Which, remember, is to be written on a *single* computer (if this isn't possible, include a note explaining)
        - Going to need to actually enforce this one this time
- Whatever the marker might need to easily grade your submission:
    - A file (e.g. .txt or .pdf; something readily readable) explaining how to use your program
    - Sample executions
    - Whatever one would need to understand it *quickly*
    - (This one's not part of the marking scheme; it's just so your submission will be as easy to grade as possible)