CS125 : Introduction to Computer Science

Lecture Notes #11
Procedural Composition and Abstraction

©2005, 2004 Jason Zych

Lecture 11 : Procedural Composition and Abstraction

Solving a problem...with help

Imagine I have a math problem I would like to solve. We'll use an arithmetic arithmetic problem, just to keep things simple, even if it will seem a little silly.

So, let's imagine that I am faced with calculating the following value, but don't know how, i.e. perhaps I have no idea how to do certain arithmetic operations. Specifically, we will assume that I know how to do addition, but that I do not know how to do subtraction, multiplication, or division.

```
(((2 + 5) * ((3 + 10) - 1)) / 6) + 7
```

Since I can perform addition, I can evaluate the values of the first two additions in the problem, thus giving me:

```
I have: (((2 + 5) * ((3 + 10) - 1)) / 6) + 7

[I calculate]

I have: ((7 * (13 - 1)) / 6) + 7
```

but I cannot perform the third addition, since I first need to know the value of the expression

```
((7 * (13 - 1)) / 6)
```

since that value is one of the operands of the addition. Since I do not know how to perform subtraction, multiplication, or division, I cannot evaluate that expression, and thus cannot obtain the first operand of the remaining addition.

So, now it's time to call on friends for help. Let's imagine I have a friend, named `FriendA`, who knows how to do subtraction. Let's send `FriendA` the subtraction part of our problem. The chain of events so far is as follows:

```
I have: (((2 + 5) * ((3 + 10) - 1)) / 6) + 7

[I calculate]

I have: ((7 * (13 - 1)) / 6) + 7

             --------> I send FriendA: (13 - 1)

             FriendA has: (13 - 1)
```

`FriendA` isn't seeing the entire problem, but `FriendA` doesn't need to! All `FriendA` cares about, is that I have sent a subtraction problem. `FriendA` doesn't need to know *why* I need the subtraction done, in order to do the subtraction. `FriendA` can simply calculate the result, and send it back to me. Likewise, I don't need to know how `FriendA` does the work – I just need to send the problem to `FriendA`, and receive the solution in return.

```
I have: (((2 + 5) * ((3 + 10) - 1)) / 6) + 7

[I calculate]

I have: ((7 * (13 - 1)) / 6) + 7

               -------> I send FriendA: (13 - 1)

               FriendA has: (13 - 1)

               [FriendA calculates]

               FriendA has: 12

               <------- FriendA sends back 12
```

and now, thanks to `FriendA`, I can replace the expression `(13 - 1)` with the expression `12`:

```
I have: (((2 + 5) * ((3 + 10) - 1)) / 6) + 7

[I calculate]

I have: ((7 * (13 - 1)) / 6) + 7

               -------> I send FriendA: (13 - 1)

               FriendA has: (13 - 1)

               [FriendA calculates]

               FriendA has: 12

               <------- FriendA sends back 12

I have:  ((7 * 12) / 6) + 7
```

Now, we've still got a multiplication and a division to deal with. I have no idea how to do those (or so we are pretending), but I have a friend, `FriendB`, who *claims* to know how to do multiplication and division. So, I send `FriendB` that part of the expression.

```
I have: (((2 + 5) * ((3 + 10) - 1)) / 6) + 7

[I calculate]

I have: ((7 * (13 - 1)) / 6) + 7

               -------> I send FriendA: (13 - 1)

               FriendA has: (13 - 1)

               [FriendA calculates]

               FriendA has: 12

               <------- FriendA sends back 12

I have:  ((7 * 12) / 6) + 7

               -------> I send FriendB: ((7 * 12 / 6)

               FriendB has: ((7 * 12) / 6)
```

Now, perhaps `FriendB` knows how to do multiplication, but not division. `FriendB` lied to us!! But fortunately for `FriendB`, we don't need to find that out. This is because we've asked `FriendB` to perform a task for us, but we *don't care* how that task gets done. All we want is the correct answer returned to us. So, `FriendB` could likewise ask for help, completely unbeknownst to us.

First of all, `FriendB` performs the multiplication, since `FriendB` knows how to do multiplication:

```
I have: (((2 + 5) * ((3 + 10) - 1)) / 6) + 7

[I calculate]

I have: ((7 * (13 - 1)) / 6) + 7

               -------> I send FriendA: (13 - 1)

               FriendA has: (13 - 1)

               [FriendA calculates]

               FriendA has: 12

               <------- FriendA sends back 12

I have:  ((7 * 12) / 6) + 7

               -------> I send FriendB: ((7 * 12 / 6)

               FriendB has: ((7 * 12) / 6)

               [FriendB calculates]

               FriendB has: (84 / 6)
```

Next, since `FriendB` doesn't know how to do division, `FriendB` asks another friend, `FriendC`, for help:

```
I have: (((2 + 5) * ((3 + 10) - 1)) / 6) + 7

[I calculate]

I have: ((7 * (13 - 1)) / 6) + 7

               -------> I send FriendA: (13 - 1)

               FriendA has: (13 - 1)

               [FriendA calculates]

               FriendA has: 12

               <------- FriendA sends back 12

I have:  ((7 * 12) / 6) + 7

               -------> I send FriendB: ((7 * 12 / 6)

               FriendB has: ((7 * 12) / 6)

               [FriendB calculates]

               FriendB has: (84 / 6)

                        ------> FriendB sends FriendC:  (84 / 6)

                        FriendC has: (84 / 6)
```

Note a few things here. Firstly, just as `FriendA` never saw the entire problem, likewise, `FriendB` does not get to see the entire problem, and `FriendC` sees even less. That does not matter! All I need from `FriendB` is to solve the part I sent to `FriendB`, just like all I needed from `FriendA` was to solve the part I sent to `FriendA`. If `FriendA` is asked to solve `(13 - 1)`, `FriendA` does not need to know why I want that answer – `FriendA` just needs to give me the answer I am asking for. And likewise, `FriendB` does not need to know why I need the answer to `((7 * 12) / 6)` – `FriendB` just needs to give me the answer I am asking for. And likewise, `FriendC` doesn't need to know why `FriendB` wants the answer to `(84 / 6)` – `FriendC` just needs to give `FriendB` the answer that `FriendB` is looking for.

But likewise, this "secrecy" works in reverse as well! When I sent `FriendA` part of the original problem, I didn't care how `FriendA` got the answer. I just wanted the answer. So maybe `FriendA` did all the work, or maybe `FriendA` asked someone else for help. Eitehr way, I don't need to care. I just need to ask `FriendA` for an answer to a problem, and then wait to receive that answer. And similarly, when I ask `FriendB` for help, I don't care how `FriendB` got the answer, as long as `FriendB` sends the answer back to me. Whether `FriendB` can do all the work, or has to ask someone (such as `FriendC`) for help, does not concern me. All I care about is that I sent a problem to `FriendB` and that I get the answer back.

So, finishing our example, `FriendC` is able to return the answer `14` to `FriendB`:

```
I have: (((2 + 5) * ((3 + 10) - 1)) / 6) + 7

[I calculate]

I have: ((7 * (13 - 1)) / 6) + 7

               -------> I send FriendA: (13 - 1)

               FriendA has: (13 - 1)

               [FriendA calculates]

               FriendA has: 12

               <------- FriendA sends back 12

I have:  ((7 * 12) / 6) + 7

               -------> I send FriendB: ((7 * 12 / 6)

               FriendB has: ((7 * 12) / 6)

               [FriendB calculates]

               FriendB has: (84 / 6)

                       ------> FriendB sends FriendC:  (84 / 6)

                       FriendC has: (84 / 6)

                       [FriendC calculates]

                       FriendC has: 14

                       <------ FriendC sends back 14

                FriendB has: 14
```

and now `FriendB` can return that result back to me:

```
I have: (((2 + 5) * ((3 + 10) - 1)) / 6) + 7

[I calculate]

I have: ((7 * (13 - 1)) / 6) + 7

               -------> I send FriendA: (13 - 1)

               FriendA has: (13 - 1)

               [FriendA calculates]

               FriendA has: 12

               <------- FriendA sends back 12

I have:  ((7 * 12) / 6) + 7

               -------> I send FriendB: ((7 * 12 / 6)

               FriendB has: ((7 * 12) / 6)

               [FriendB calculates]

               FriendB has: (84 / 6)

                       ------> FriendB sends FriendC:  (84 / 6)

                       FriendC has: (84 / 6)

                       [FriendC calculates]

                       FriendC has: 14

                       <------ FriendC sends back 14

                FriendB has: 14

                <------ FriendB sends back 14

I have:  14 + 7
```

at which point I can finally complete the last addition and finish the evaluation of the original expression:

```
I have: (((2 + 5) * ((3 + 10) - 1)) / 6) + 7

[I calculate]

I have: ((7 * (13 - 1)) / 6) + 7

               -------> I send FriendA: (13 - 1)

               FriendA has: (13 - 1)

               [FriendA calculates]

               FriendA has: 12

               <------- FriendA sends back 12

I have:  ((7 * 12) / 6) + 7

               -------> I send FriendB: ((7 * 12 / 6)

               FriendB has: ((7 * 12) / 6)

               [FriendB calculates]

               FriendB has: (84 / 6)

                       ------> FriendB sends FriendC:  (84 / 6)

                       FriendC has: (84 / 6)

                       [FriendC calculates]

                       FriendC has: 14

                       <------ FriendC sends back 14

                FriendB has: 14

                <------ FriendB sends back 14

I have:  14 + 7

[I calculate]

I have: 21
```

This concept we are illustrating, is known as *procedural abstraction*. It is a kind of abstraction in which you hide away *the details of how a particular task is done*. Something/someone else worries about how the details of the work are done, and all you need to do is ask that something or someone, "please do this work for me" and then wait for the work to get done (and if you are waiting for an answer of some kind, wait for that something or someone to return your answer).

Because of the idea of procedural abstraction, it is not necessary for you to worry about every single detail of a computation. In the example above, I didn't worry about *any* of the details of the subtraction, multiplication, or division – for each of those tasks, I sent those problems to someone else and awaited my answer. And even though I did a little bit of the work in our example above (the addition work), there were details that I didn't worry about – I sent sections of the problem to other friends and waited for their answers, without worrying about how those friends were doing the work. In this way, I was able to get the value of a complex arithmetic expression, without knowing any arithmetic other than addition.

You have already used this same idea in your own MPs. When you have printed something to the screen, you sent the value you wanted to print, to a `System.out.println` or `System.out.print` statement. You didn't worry about the specifics of how that `System.out.println` or `System.out.print` statement did its work – you just typed the statement and trusted that the statement would do what it was supposed to do. Similarly, you didn't need to type a lot of code in order to input a value. You just typed `Keyboard.readInt()` or `Keyboard.readDouble()` or `Keyboard.readChar()`, and trusted that that expression would work as we said it would, and would do the work of obtaining your input value and sending it back to you, even if you have no idea how that task is actually working behind the scenes.

Again, this is how abstraction helps us. Since you are freed from having to worry about every little detail, you have more time to focus on the "big picture" and develop that further.

We can also relate this example to how the machine implements procedural abstraction. To do this, let's go through the example again, but let's pretend that there is a common sheet of scratch paper that `FriendA`, `FriendB`, `FriendC`, and I are using:

Column 1	Column 2	Column 3	Column 4

I receive the initial problem, and start working in column 1 of the scratch paper:

Column 1	Column 2	Column 3	Column 4
(((2 + 5) * ((3 + 10) - 1)) / 6) + 7			

After I perform the two additions I can do, I have reduced the problem further:

Column 1	Column 2	Column 3	Column 4
(((2 + 5) * ((3 + 10) - 1)) / 6) + 7 ...some work done... ((7 * (13 - 1)) / 6) + 7			

At this point, I needed to ask `FriendA` for help. So I will write the problem I need `FriendA` to solve, at the top of column 2 – thus showing `FriendA` where that scratch work should be done – and then I will *pass the scratch paper to* `FriendA`.

Column 1	Column 2	Column 3	Column 4
(((2 + 5) * ((3 + 10) - 1)) / 6) + 7 ...some work done... ((7 * (13 - 1)) / 6) + 7	(13 - 1)		

Note now that I cannot do anymore work for the time being. This is because `FriendA` has the scratch paper! There wasn't any other arithmetic I could do anyway, but even if there was, I am forced to sit around waiting for `FriendA` to finish, since `FriendA` has the scratch paper and I do not. This is similar to how, in your programs, when you call `Keyboard.readInt()` or `System.out.println()`,

your program sits there and waits for the `Keyboard.readInt()` call or the `System.out.println()` call to finish, before continuing. Likewise, I cannot continue until `FriendA` finishes and gives me back the scratch paper.

Likewise, we will assume `FriendA` does NOT work in column 1, and we will further assume that `FriendA` does not even look at the work I have written in column 1. We have told `FriendA` to work in column 2, and so that is where `FriendA` works. And what `FriendA` does is to perform whatever scratch work is needed to figure out that `(13 - 1)` is `12`:

Column 1	Column 2	Column 3	Column 4
(((2 + 5) * ((3 + 10) - 1)) / 6) + 7 ...some work done... ((7 * (13 - 1)) / 6) + 7	(13 - 1) ...some work done... 12		

Now that `FriendA` has completed the calculation, `FriendA` must do three things. First, `FriendA` must notify us of the final answer, by writing the answer in our column:

Column 1	Column 2	Column 3	Column 4
(((2 + 5) * ((3 + 10) - 1)) / 6) + 7 ...some work done... ((7 * (13 - 1)) / 6) + 7 12 `<---` answer from `FriendA`	(13 - 1) ...some work done... 12		

Second, `FriendA` should *erase column 2*. After all, I might want to give this scratch paper to someone else later on. If `FriendA` is not using that area of the scratch paper anymore, than erasing all the remarks there, so that the area is free for someone else to use, seems to be the right thing

to do. To leave all the now-useless remarks there, taking up space, would be wasteful:

Column 1	Column 2	Column 3	Column 4
(((2 + 5) * ((3 + 10) - 1)) / 6) + 7 ...some work done... ((7 * (13 - 1)) / 6) + 7 12 `<---` answer from `FriendA`			

Finally, `FriendA` should give the scratch paper back to me! Now, finally, I can do some work again, just like when `Keyboard.readInt()` finishes its job and reports the input integer to your program. In that situation, your program can then finally resume work where it had paused while waiting for the `Keyboard.readInt()` call to finish, and likewise, now that I have the scratch paper again, I can make use of the value that `FriendA` gave me, to reduce my expression further. This work is of course done in column 1, since that's where I do my work:

Column 1	Column 2	Column 3	Column 4
(((2 + 5) * ((3 + 10) - 1)) / 6) + 7 ...some work done... ((7 * (13 - 1)) / 6) + 7 12 `<---` answer from `FriendA` ((7 * 12) / 6) + 7			

Note that not only did `FriendA` *not* mess with anything in column 1 (aside from writing in the result value in the end, and noting it was the result value), but in addition, I cannot move to column 2 and mess with any of the work `FriendA` did. First of all, that work has been erased, and is thus completely gone; secondly, I am supposed to stay in column 1 anyway.

Now, the next step was to pass the multiplication and division to `FriendB`. I will again use column 2 for this, since it is the column right next to where I am working, and it is (and should be) free and available for this purpose. So, I will write the problem I am handing to `FriendB`, at the top of column 2, and then will hand the scratch paper to `FriendB`. As with `FriendA`, when `FriendB` is working on the problem, I do not have the scratch paper, and so I cannot do anything. And similarly, as with `FriendA`, `FriendB` will know to only do work in the designated column, and

will know to *NOT* mess with my work in column 1.

Column 1	Column 2	Column 3	Column 4
(((2 + 5) * ((3 + 10) - 1)) / 6) + 7 ...some work done... ((7 * (13 - 1)) / 6) + 7 12 `<---` answer from `FriendA` ((7 * 12) / 6) + 7	((7 * 12) / 6)		

So what we see here is that *our columns are available for general use.* Just because `FriendA` was using column 2 for scratch earlier, doesn't mean we can't use it now for `FriendB`. After all, `FriendA` was finished with the subtraction work, and with column 2, and thus erased everything from column 2! So if column 2 is sitting there unused, why *shouldn't* `FriendB` go ahead and use it? There's no reason not to! If I am working in column 1, then when it's time to have a friend help me out, I just have that friend work in the open column to my right – column 2, in this case. Regardless of whether that scratch work is scratch work for `FriendA` who does subtraction, or `FriendB` who does multiplication and division, if column 2 is the next open column, then when I pass work off to a friend, that's the column my friend will use.

`FriendB` will then do some multiplication work in an attempt to reduce the expression to something simpler:

Column 1	Column 2	Column 3	Column 4
(((2 + 5) * ((3 + 10) - 1)) / 6) + 7 ...some work done... ((7 * (13 - 1)) / 6) + 7 12 `<---` answer from `FriendA` ((7 * 12) / 6) + 7	((7 * 12) / 6) ...some work done... (84 / 6)		

And now `FriendB` is stuck. This is where `FriendB` had to turn for help to `FriendC`, and that is exactly what happens here. `FriendB` will do the same thing I did – write the problem for which help is needed, at the top of the *next open column*, and then pass the scratch paper to someone

who will work on that problem – in this case, `FriendC`:

Column 1	Column 2	Column 3	Column 4
(((2 + 5) * ((3 + 10) - 1)) / 6) + 7 ...some work done... ((7 * (13 - 1)) / 6) + 7 12 <--- answer from `FriendA` ((7 * 12) / 6) + 7	((7 * 12) / 6) ...some work done... (84 / 6)	(84 / 6)	

What has happened here is, `FriendC` needs a scratch column, just as I did, and just as `FriendA` and `FriendB` did. But right now, not only is column 1 taken, but column 2 is taken as well. *This does not matter!*. `FriendC` can use *any* column for scratch work; it just so happened before, that column 2 was always the next available open column. Right now, column 3 is the next available open column, so that is the column that gets used by `FriendC` instead. Either way, `FriendC` still gets a scratch area to work in. Whether it is me doing work, `FriendA` doing work, `FriendB` doing work, or `FriendC` doing work, none of us care which *particular* column we do our work in. We only care that we get *some* column – whichever one it might be – to do our work in. So the easiest thing to do is to just use the leftmost column that is not in use; I used column 1, people who I asked for help used column 2, and anyone *they* ask for help, uses column 3. If `FriendC` were to ask for help, then that person would use column 4, since that is the next available column.

This is an important concept! Someone can use any column that is convenient as their scratch column. `FriendA`, for example, is not bound to always use column 2; `FriendA` only used column 2 since the person using column 1 (me) was the one who asked for help. If `FriendC` had subtraction to do, and asked `FriendA` for help, `FriendA` would proceed to work in column 4 – and yet doing subtraction, the same kind of work `FriendA` had done for me in column 2. Similarly, when you actually write Java code, any collection of Java instructions such as the instructions to make `Keyboard.readInt()` work or the instructions to make `System.out.println()` work, will use whatever area of memory is to the right of where the program had just been working. In the case of your programs so far, you called `Keyboard.readInt()` from `main()`, and so whatever area of memory `main()` had used for its variables, if you then call `Keyboard.readInt()`, `Keyboard.readInt()` will use the area of memory *next to* the area being used by `main()`, to do any scratch work that `Keyboard.readInt()` needs to do. It is the same when you call `System.out.println()`. Every time you ask another piece of code to do some work for you, that piece of code just uses the next available memory to the right of where you were working, to do its job – and then releases that memory once the code is done. Whatever area of memory `Keyboard.readInt()` is using, `Keyboard.readInt()` erases that memory and gives it back to the system when it is done and returning back to `main()`, just like `FriendA` erased column 2 just before handing the scratch paper back to me.

That is, your Java code will not be bound to run in a particular area of memory, but rather, will run in whatever area is most convenient. (You don't need to worry about that level of detail, but it will help you if you realize that concept.)

Continuing on with our example, `FriendC` will perform the work needed to figure out the result

of the division:

Column 1	Column 2	Column 3	Column 4
(((2 + 5) * ((3 + 10) - 1)) / 6) + 7 ...some work done... ((7 * (13 - 1)) / 6) + 7 12 <--- answer from `FriendA` ((7 * 12) / 6) + 7	((7 * 12) / 6) ...some work done... (84 / 6)	(84 / 6) ...some work done... 14	

And now `FriendC` will do the same thing that `FriendA` did earlier. First, `FriendC` will return the result, to the person who had asked for help – namely, `FriendB` in column 2:

Column 1	Column 2	Column 3	Column 4
(((2 + 5) * ((3 + 10) - 1)) / 6) + 7 ...some work done... ((7 * (13 - 1)) / 6) + 7 12 <--- answer from `FriendA` ((7 * 12) / 6) + 7	((7 * 12) / 6) ...some work done... (84 / 6) 14 <--- answer from `FriendC`	(84 / 6) ...some work done... 14	

Next, `FriendC` will erase all the scratch work in column 3, since now `FriendC` is done, and there is no sense cluttering up a column someone else could use, with scratch work that no longer serves a purpose:

Column 1	Column 2	Column 3	Column 4
(((2 + 5) * ((3 + 10) - 1)) / 6) + 7 ...some work done... ((7 * (13 - 1)) / 6) + 7 12 <--- answer from `FriendA` ((7 * 12) / 6) + 7	((7 * 12) / 6) ...some work done... (84 / 6) 14 <--- answer from `FriendC`		

And finally, `FriendC` will return the scratch paper back to `FriendB`, and `FriendB` can resume work now that both the scratch paper, and the needed result value, have been returned.

Now, all `FriendB` has left to do, is to recognize that the result given by `FriendC`, is the final

answer that `FriendB` needed:

Column 1	Column 2	Column 3	Column 4
(((2 + 5) * ((3 + 10) - 1)) / 6) + 7 ...some work done... ((7 * (13 - 1)) / 6) + 7 12 <--- answer from `FriendA` ((7 * 12) / 6) + 7	((7 * 12) / 6) ...some work done... (84 / 6) 14 <--- answer from `FriendC` 14		

And now `FriendB` can prepare to return back to me. First, `FriendB` tells me what the result is:

Column 1	Column 2	Column 3	Column 4
(((2 + 5) * ((3 + 10) - 1)) / 6) + 7 ...some work done... ((7 * (13 - 1)) / 6) + 7 12 <--- answer from `FriendA` ((7 * 12) / 6) + 7 14 <--- answer from `FriendB`	((7 * 12) / 6) ...some work done... (84 / 6) 14 <--- answer from `FriendC` 14		

Note that I got my result from `FriendB`, not `FriendC`. In fact, since I have no idea what `FriendB` did to get this result, I don't even have any idea `FriendC` was ever involved.

Second, `FriendB` erases column 2, since now `FriendB` is done and so there's no point in keeping column 2 cluttered up with completed work when someone else could use that column in the future:

Column 1	Column 2	Column 3	Column 4
(((2 + 5) * ((3 + 10) - 1)) / 6) + 7 ...some work done... ((7 * (13 - 1)) / 6) + 7 12 <--- answer from `FriendA` ((7 * 12) / 6) + 7 14 <--- answer from `FriendB`			

And finally, `FriendB` hands the scratch paper back to me, and I can resume my own work where I

left off. Since I now know the result of the first addition operand, I can make a substitution:

Column 1	Column 2	Column 3	Column 4
(((2 + 5) * ((3 + 10) - 1)) / 6) + 7 ...some work done... ((7 * (13 - 1)) / 6) + 7 12 <--- answer from `FriendA` ((7 * 12) / 6) + 7 14 <--- answer from `FriendB` 14 + 7			

And after some calculation work, I can complete the final addition – and with it, finish the evaluation of the expression:

Column 1	Column 2	Column 3	Column 4
(((2 + 5) * ((3 + 10) - 1)) / 6) + 7 ...some work done... ((7 * (13 - 1)) / 6) + 7 12 <--- answer from `FriendA` ((7 * 12) / 6) + 7 14 <--- answer from `FriendB` 14 + 7 ...some work done... 21			

Now that I know the result of the expression, I can note it somewhere, and erase column 1 so that column 1 can be used for a different bunch of scratch work the next time I need to do some calculation work in the future:

	Column 1	Column 2	Column 3	Column 4
<--- result : 21				

So that is an example of how different areas of memory (different columns, in our example) can get used by various sets of instructions. Any particular set of instructions (such as `main()` or `Keyboard.readInt()`) can make use of any empty area of memory. And as the instructions run, when a new area of memory is needed in order to run a new set of instructions, we just move over

to the right, to the closest-available empty area and use that. And when any set of instructions has finished, the memory that was being used by those instructions is erased and can be re-used by a different set of instructions in the future.