Adventures in Groovy – Part 37: Improving The User Experience With AutoFill

To date, we have talked about the performance improvements Groovy introduces, as well as the creative validation we can add. One thing that hasn't been covered yet is the ability to add functionality to make the input easier for a planner. Replicating changes through the months, resetting the values back to the defaults, and many other concepts can be developed to make the user's lives easier.

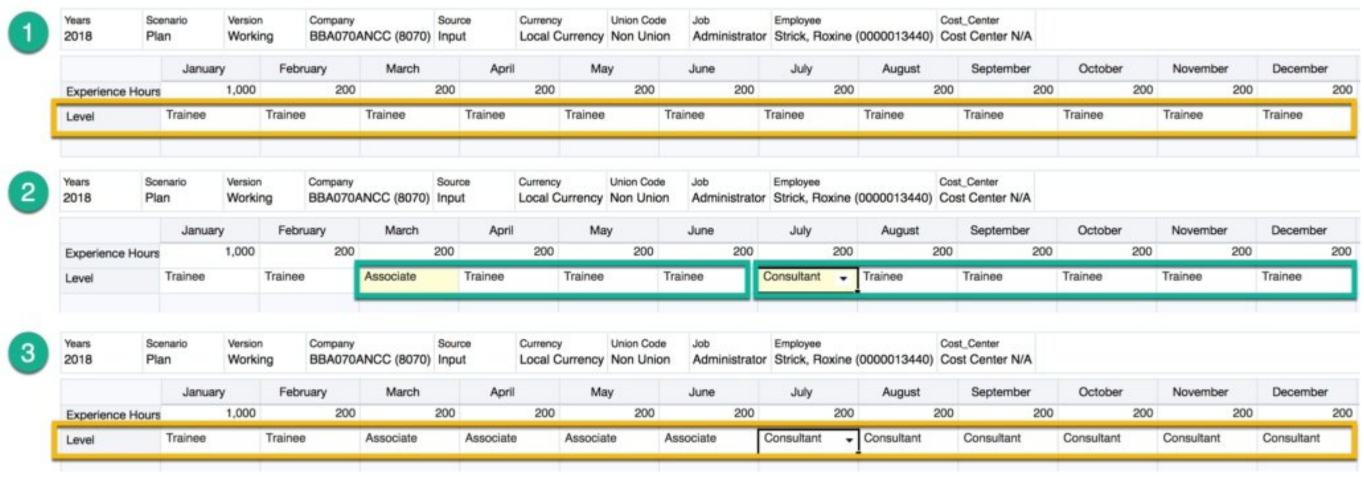
Use Case / Example

The following example is something most applications will encounter, especially in workforce planning and capex. This form has the level of an employee. This functionality would be the same for things like title and employment status. Basically, the user would typically change a month and simulate that change to all remaining months. Or, they might change the status to Maternity Leave for 3 months and then back to Active.

What this functionality will do is allow the planner to change the month from active to Maternity Leave, assume every month after that will be updated for them, until a month is changed/edited. If this is still not clear, I think the following 3 steps will clear it up.

1. The planner opens the form and the employee is set to a Trainee for all months. The level progression is Trainee, Associate, Consultant.
2. The planner changes the person's level from Trainee to Associate in March, and to Consultant in July.
3. The planner saves the form and the result is that the

employee is promoted to Associate in March. The calculation automatically changes April, May, and June to Associate – basically it copies the change to the next month unless the next month is edited. It continues that pattern through the 12 months, so the employee is promoted to a Consultant in July and that is copied through December.


1

Years	Scenario	Version	Company	Source	Currency	Union Code	Job	Employee	Cost_Center
2018	Plan	Working	BBA070ANCC (8070)	Input	Local Currency	Non Union	Administrator	Strick, Roxine (0000013440)	Cost Center N/A

	January	February	March	April	May	June	July	August	September	October	November	December
Experience Hours	1,000	200	200	200	200	200	200	200	200	200	200	200
Level	Trainee	Trainee	Trainee	Trainee	Trainee	Trainee	Trainee	Trainee	Trainee	Trainee	Trainee	Trainee

2

Years	Scenario	Version	Company	Source	Currency	Union Code	Job	Employee	Cost_Center
2018	Plan	Working	BBA070ANCC (8070)	Input	Local Currency	Non Union	Administrator	Strick, Roxine (0000013440)	Cost Center N/A

	January	February	March	April	May	June	July	August	September	October	November	December
Experience Hours	1,000	200	200	200	200	200	200	200	200	200	200	200
Level	Trainee	Trainee	Associate	Trainee	Trainee	Trainee	Consultant	Trainee	Trainee	Trainee	Trainee	Trainee

3

Years	Scenario	Version	Company	Source	Currency	Union Code	Job	Employee	Cost_Center
2018	Plan	Working	BBA070ANCC (8070)	Input	Local Currency	Non Union	Administrator	Strick, Roxine (0000013440)	Cost Center N/A

	January	February	March	April	May	June	July	August	September	October	November	December
Experience Hours	1,000	200	200	200	200	200	200	200	200	200	200	200
Level	Trainee	Trainee	Associate	Associate	Associate	Associate	Consultant	Consultant	Consultant	Consultant	Consultant	Consultant

This may not seem life changing, but it does reduce the effort for a planner and reduce the possibility that they don't know they have to change all the months manually and cause inconsistencies in the budget.

The Code

The following calculation assumes only one employee is on the form. This would need slightly updated to reset when the employee changed, or when the iteration went to the next line in the form. This also needs to be executed before save. Let's jump in.

The first thing we are going to do is set some parameters.

```
def update = false
def runTotal = 0
def change = false
def Months =
["Jan","Feb","Mar","Apr","May","Jun","Jul","Aug","Sep","Oct","
Nov","Dec"]
```

Next, we will loop through the cells. We are only looping through the rows where the account is equal to “Level.” As this goes from January to December, it keeps track of the value the cell should be changed to, or the last edited cell’s value. If it gets to a cell that is edited, it resets the variable so that any cell after that will be updated to the most recent change.

```
operation.grid.dataCellIterator('Level').each { cell ->
  // If the cell is edited, change the variable so it knows that the remaining
  // months need to be changed
  if(cell.isEdited()){
   change = true
  }
  // If the month is not edited and a prior month has been changed, update the
  // value to the prior month's value
  else if(change == true){
        def lastMonth = Months.findIndexOf{it == cell.getPeriodName()} - 1
cell.setFormattedValue(cell.crossDimCell(Months[lastMonth]).formattedValue)
  }
}
```

Not Rocket Science

Like I previous stated, this isn’t going to be the difference between a project success and failure like the performance improvements that have been discussed, but it is a very simple thing that can be added to give your application some polish. Also, I can’t say it enough, little things like this give users confidence and also reduce the possibility of human error, giving the budget more validity and trust.