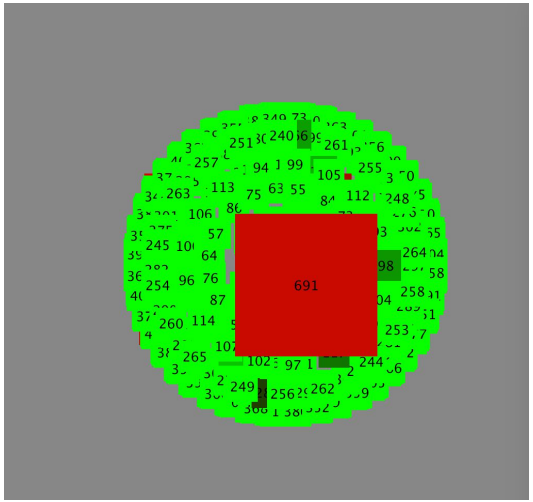
01 - Visual Process Manager

Dan Bye, Dan Brill, & James Draper

Project Proposal

- The Visual Process Manager is an interactive process manager like top, but with a graphical interface where processes are visualized as nodes in an undirected graph
- Each node is a button which is dynamically created from each process currently running by polling a periodically updated database
- The user can manage their processes by selecting a node in the graph and then further selecting a software interrupt to send to that process

Details



- The button is labeled by the Process I.D.
- The size of the button is based on ram usage.
- The color of the button represents the CPU usage of each process.
  - This ranges from green (low usage) to red (high usage).
- The locality of each button is based on the scaled size of the largest node, pushed out radially with an increasing number of nodes at each layer

Design Pattern: Flyweight

- Because we're generating so many nodes for process it would be viable to architect a class like NodeProc that refactors it's major methods by way of extraction and puts them into a class like GraphNode

- This way we can instantiate many NodeProc objects without putting an overly large load on our system

Design Pattern: Observer

- We could've used the observer pattern to monitor the database
- Each time the database changed the observer could

  invoke our graph class

  and re-render the visualization,
- Right now, both are timed updates.

Design Pattern: FactoryMethod

- Each NodeProc which represents a process could create a GraphNode

```
public class GraphNode extends NodeProc implements Comparable<GraphNode>{
    private NodeProc proc;
    private Color color;
    private float locality;
    private float size;

    GraphNode(){
        this.color = new Color(0, 0, 0);
        this.locality = 0;
        this.size = 27;
    }
    GraphNode(NodeProc newProc){
        this.proc = newProc;
        setColor();
        setSize();
        this.color = getColor();
        this.size = getsize();
    }
    GraphNode(NodeProc newProc, Color newColor, float newLocality, float newSize){
        this.proc = newProc;
        this.color = newColor;
        this.locality = newLocality;
        this.size = newSize;
```

Design Pattern: Singleton

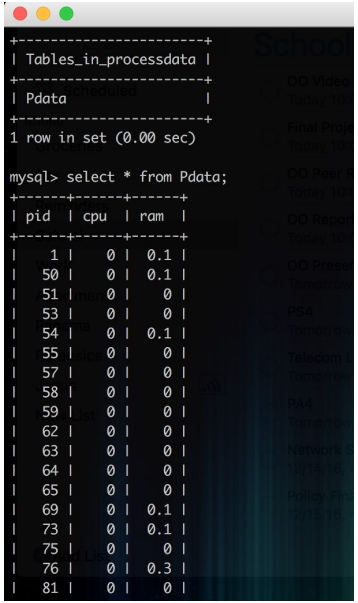
- We only want one connection to the database to exist at a given time

```java
public class DBConnection {

    public Connection openConnection() throws ClassNotFoundException {
        Connection conn = null;
        try {
            String myDriver = "com.mysql.jdbc.Driver";
            String url = "jdbc:mysql://localhost:3306/ProcessData?useSSL=false";
            Class.forName(myDriver);
            conn = DriverManager.getConnection(url, "root", "dumb_password");
        } catch (SQLException e) {
            e.printStackTrace();
        }
        return conn;
```

Use Cases and Requirements

- US-01: Able to review processes visually.
- US-02: Be able to manage processes by interacting with the visualization.
- Database Implementation
  - FR-06: Process Data can be stored in a Database
  - NF-05: Processes updated periodically



```
+-----------------------+
| Tables_in_processdata |
+-----------------------+
| Pdata                 |
+-----------------------+
1 row in set (0.00 sec)

mysql> select * from Pdata;
+------+------+------+
| pid  | cpu  | ram  |
+------+------+------+
|    1 |    0 |  0.1 |
|   50 |    0 |  0.1 |
|   51 |    0 |    0 |
|   53 |    0 |    0 |
|   54 |    0 |  0.1 |
|   55 |    0 |    0 |
|   57 |    0 |    0 |
|   58 |    0 |    0 |
|   59 |    0 |    0 |
|   62 |    0 |    0 |
|   63 |    0 |    0 |
|   64 |    0 |    0 |
|   65 |    0 |    0 |
|   69 |    0 |  0.1 |
|   73 |    0 |  0.1 |
|   75 |    0 |    0 |
|   76 |    0 |  0.3 |
|   81 |    0 |    0 |
```

Demo