Title Page	
Project	Portrait Tree
Chapter	02
Content	First Refactoring of original stand-alone code

First Refactoring of Production Code

The working code has been successfully tested. The following explanation outlines what it does. The code comes from the file currently called **C2_classFlowBrick.php**.

Within the class is a single property called, “Work Rectangle”. At any point in the process, this represents a single level of spreadsheet-style output. So, the Work Rectangle represents a single level; that level has members and members may have children. Every child is a member of the next (future) level. For a number of reasons, all levels beyond 2 are handled by the <regular loop>.

1. Function “*alter Number Update”* gives the new Row Number; this is later available for reference.

2. Function “*data To Send*” takes $children of the last level used, from the Work Rectangle.

3. The same *data To Send* also calls a function called “*data Completion*”. That function enriches the Work Rectangle , by updating two more attributes for each child:

   There is the "row attribute" set to the number mentioned previously. Beyond this, a "column attribute" is a letter unique to the member; when a child comes from the same member as its neighbor it has a consecutive letter; otherwise, a gap is left.

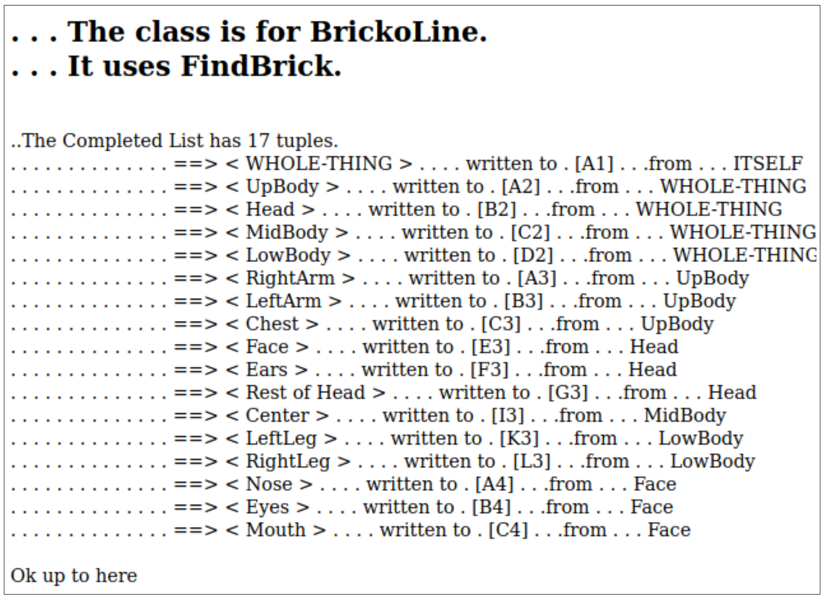
4. These enriched items within the Work Rectangle forms the new $children, adding each of them is to the waiting list.

5. The function “discover Future Parents” is called within “Rebuild Using Parents”. Thus the “$parent List” is made up of every child which itself has children:

   That *Rebuild Using Parents*, overwrites the the Work Rectangle according to that $parent List.

Looking at the first Auto-Output

The lowest level work is ready and since it passed “tests”, the question remains: “Can it be used for human-readable output?” One of the test-series answers this directly:

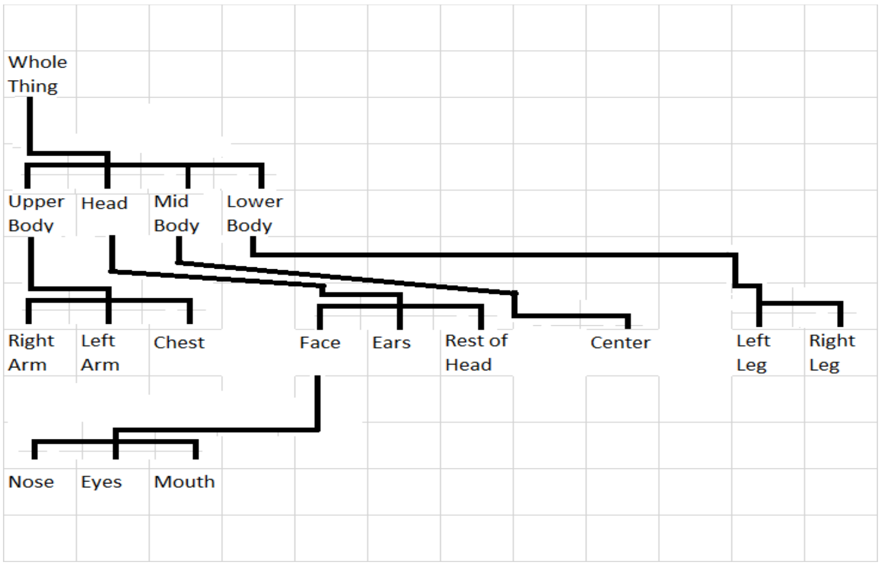


```
. . . The class is for BrickoLine.
. . . It uses FindBrick.

..The Completed List has 17 tuples.
. . . . . . . . . . . . . . ==> < WHOLE-THING > . . . . written to . [A1] . . .from . . . ITSELF
. . . . . . . . . . . . . . ==> < UpBody > . . . . written to . [A2] . . .from . . . WHOLE-THING
. . . . . . . . . . . . . . ==> < Head > . . . . written to . [B2] . . .from . . . WHOLE-THING
. . . . . . . . . . . . . . ==> < MidBody > . . . . written to . [C2] . . .from . . . WHOLE-THING
. . . . . . . . . . . . . . ==> < LowBody > . . . . written to . [D2] . . .from . . . WHOLE-THING
. . . . . . . . . . . . . . ==> < RightArm > . . . . written to . [A3] . . .from . . . UpBody
. . . . . . . . . . . . . . ==> < LeftArm > . . . . written to . [B3] . . .from . . . UpBody
. . . . . . . . . . . . . . ==> < Chest > . . . . written to . [C3] . . .from . . . UpBody
. . . . . . . . . . . . . . ==> < Face > . . . . written to . [E3] . . .from . . . Head
. . . . . . . . . . . . . . ==> < Ears > . . . . written to . [F3] . . .from . . . Head
. . . . . . . . . . . . . . ==> < Rest of Head > . . . . written to . [G3] . . .from . . . Head
. . . . . . . . . . . . . . ==> < Center > . . . . written to . [I3] . . .from . . . MidBody
. . . . . . . . . . . . . . ==> < LeftLeg > . . . . written to . [K3] . . .from . . . LowBody
. . . . . . . . . . . . . . ==> < RightLeg > . . . . written to . [L3] . . .from . . . LowBody
. . . . . . . . . . . . . . ==> < Nose > . . . . written to . [A4] . . .from . . . Face
. . . . . . . . . . . . . . ==> < Eyes > . . . . written to . [B4] . . .from . . . Face
. . . . . . . . . . . . . . ==> < Mouth > . . . . written to . [C4] . . .from . . . Face

Ok up to here
```

The program up to here did NOT consider (in detail) what comes next. So here, we start **Imagining** what comes **Next**:

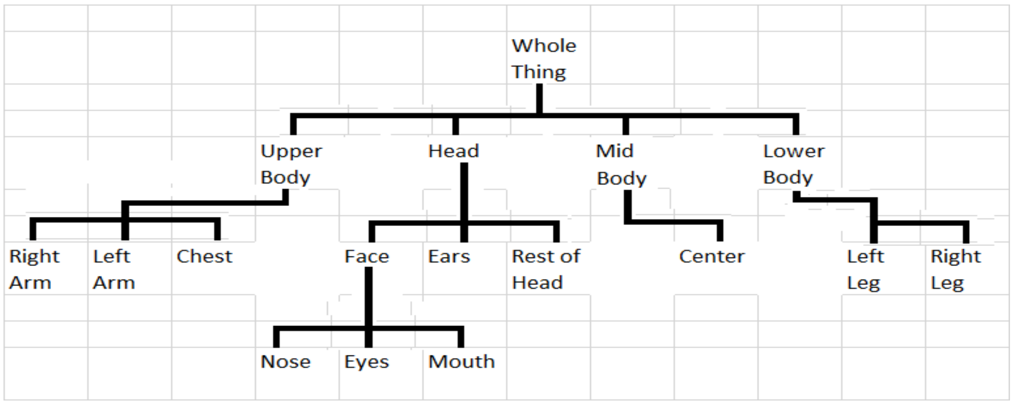
Imagine-Next WITHOUT tricks

Without altering the output in any way, the result may be very hard to join together:



Imagine-Next WITH exactly two tricks

Exactly the same output with two two types of post-output operation possible makes life far easier. The example below shows most of the rows shifted to the right and the 2nd row with double-spacing.



Imagine-Next USER tries to read it

Without the graphics problem caused by the first example, the remaining problems are easier to see:

1. Expect “Head” before “Upper Body” (when thinking top-down).
2. Expect “Chest” between the two “Arms” (in any horizontal sequence).
3. Expect logic of legs to be same as arms; Left Leg BEFORE Right Leg demands Left Arm BEFORE Right Arm.

Imagine-Next Concluding the Priorities

The point of this discussion has shown that the logical/visual order is an OBLIGATION and at least two of the other "tricks" are serious candidates. Together, we can call these the "Core Priorities".

Considering Auto-Output for Database Work.

By tracing it through, the program did what it was asked to do; however, there are clearly ways to pass it valid data which produces an unacceptable result. In other words, the resultant output depends significantly on the order in which lines are fed to the program.

The obvious work-round to this feature, is editing that “Feed Table”; unfortunately, that would be forcing **raw data** to conform to limitations of the computer-program. In the database world, things are far simpler, there is a single Feed Table which has two data columns and more than one sorting column. This directly supports alternative views of the same data.

In short, the database world, provides answers our input requirements, because the possible alternative views of the pseudo-spreadsheet could provide a new computer program with the result in any order required.

Since graphics is NOT yet part of the solution, the obvious destination for the final version of the “WaitingList” is writing to a database table. That leaves us with an immediate policy of database at both ends.

Containing the objectives of Auto-Output

This document started with **code-quality** and continued to database at **both ends**. Although there is still work to do, it is now becoming very clear what is being built right now. To use the acceptable computer-term, this item has a **single responsibility**.

It generates a data-table in spreadsheet-style format of the table-view of “owner child” which is fed to it. Using even more familiar computer terminology this becomes “Populate the waiting list of spreadsheet cells in a hierarchy-tree from one view of the owner-child table.”

In contrast, to what has happened until now, the schedule of *what’s needed* has emerged:

Improvement Schedule for Chapter-03

1. The class "Flow Brick", reaches a final form valid for all future chapters of this project.
2. Outside of "Flow Brick", the database engine has a query that feeds the rectangle of "owner child" data in a known order.
3. Also Outside of "Flow Brick", the database engine has a data-writer that takes the "waiting list" data into cells of "Child, row, column, parent".
4. The remaining work is INSIDE the two classes.
5. Functionality for tracing / explicit debugging and generating test data is removed.
6. The process of internal code naming closer to the "clear algorithm" continues for "Flow Brick".
7. Some functions within the private class are not used, so they are removed.

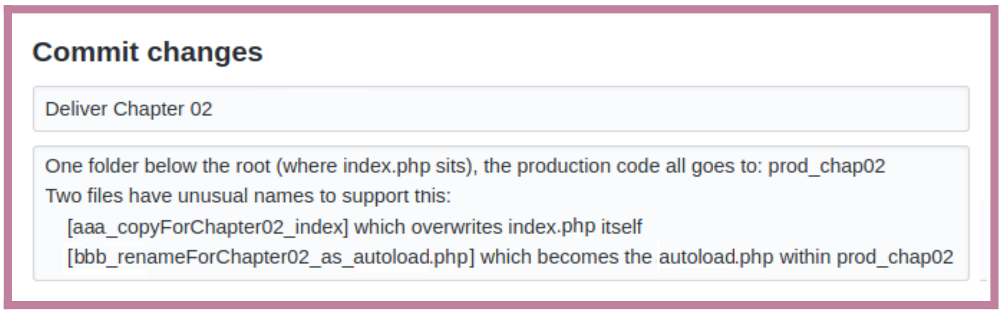
Technical Points

Those wishing to assemble the code, need to know “what goes where”:

These form the Technical sub-Sections that follow. I also want to give an idea of the later tweaking stages in which tested but low-readability code was carefully replaced by the more acceptable versions.

In a very simple way, the software produced so far is object-oriented. Beginners to php need to know about autoload which defines how source-file names are generated directly from the object-names that the running-code demands.

Before getting to all that, this represents the saving of all (this chapter's) production code to git Hub:



Commit changes

Deliver Chapter 02

One folder below the root (where index.php sits), the production code all goes to: prod_chap02
Two files have unusual names to support this:
  [aaa_copyForChapter02_index] which overwrites index.php itself
  [bbb_renameForChapter02_as_autoload.php] which becomes the autoload.php within prod_chap02

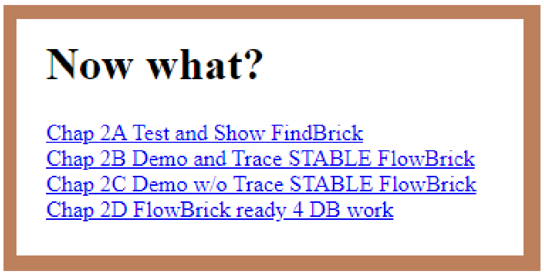
Mechanism for full debug-tracing

The following convention was used by all the modules for switching full-code tracing on/off: The class has a __construct method so that creating the object has at least one parameter. One of those parameters includes true or false for built-in code-tracing.

Meaning of ABCD

In fact, when work began on the final round of this chapter, the inner object "Find Brick" was already working and tested. The older version of "Lino Brick" was now locked with two implementations produced one with tracing and the other without. The final round was getting the (partially) refactored version called "Flow Brick" which was convenient to test against those two implementations of "Lino Brick".

In other words, there was a menu made within "default .php" for three files that would NOT change, so that results of the one being changed could be compared with them:

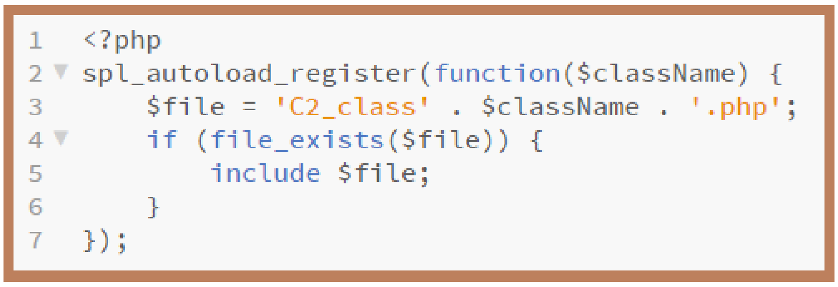


Now what?

Chap 2A Test and Show FindBrick
Chap 2B Demo and Trace STABLE FlowBrick
Chap 2C Demo w/o Trace STABLE FlowBrick
Chap 2D FlowBrick ready 4 DB work

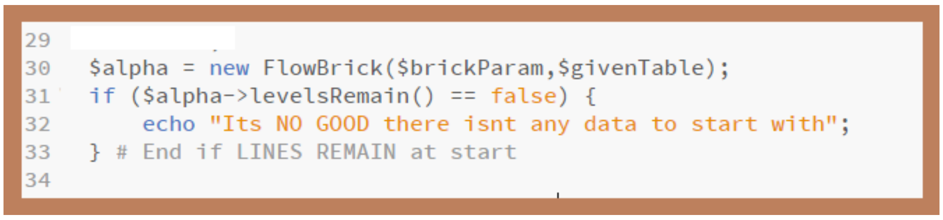
Autoload – objects on Demand

The autoload.php file was the mechanism used by the language processor to build objects from the appropriate source code:



```php
<?php
spl_autoload_register(function($className) {
    $file = 'C2_class' . $className . '.php';
    if (file_exists($file)) {
        include $file;
    }
});
```

In the following example, the object in use is called “**FlowBrick**”; thus the code for it comes from the file: **c2_classFlowBrick.PHP**



```
$alpha = new FlowBrick($brickParam,$givenTable);
if ($alpha->levelsRemain() == false) {
    echo "Its NO GOOD there isnt any data to start with";
} # End if LINES REMAIN at start
```

Chapter Summary

This chapter should have started with “white box testing”. That term means running tests when the algorithm is completely understood; it follows that unreadable code makes this impossible.

From here, the first refactoring made the code sufficiently readable. In that state, it became worthwhile studying the behavior of working-code. The study helped reach a better strategic model for what was needed both before and after our current area of focus.