# Table Oriented Programming

Abbreviated "TOP" in some places.
Languages or techniques that tend to view data as relational tables (or relational-influenced) and behavior as operations on such tables. Sometimes code is put into the table "cells". In many ways it is a kind of CollectionOrientedProgramming.
Some RelationalWeenies, such as TopMind, feel that TableOrientedProgramming is a direct competitor to ObjectOrientedProgramming. Others feel that they are orthogonal. Past debates to try to settle this issue never came to a consensus.
Relationship with SQL: Although few seem to believe SQL is the ideal query language to build TOP or anything else around (see SqlFlaws), its entrenchment in the industry suggests that TOP standards or tools either be built around SQL, or at least support it as an option.

---

**Relational tables** are not required, just a decision table. This is nothing more than an Arbitrary redefinition of language to hide the fact these are decision tables which are as old as programming it's self.
{Nobody claimed it's brand new. See bottom illustration re Ada Lovelace. And it's not just about "decision tables". Decision tables are one component/technique of it, as described below. Also note that by some measures, the existing RDBMS are not "true" relational either. If that was a key goal, then one may want to focus on "Relational Oriented Programming" instead. However, I don't believe that would require overhauling most the general TOP techniques. If you believe otherwise, I'd like to see an example scenario. Hopefully that wouldn't rekindle the "bag" debate (BagVersusSetControversyRoadmap).}

---

**Discussion**
**Is there anything new here? Isn't this standard, Microsoft favored, RecordSet based Programming? (See Fowler's PoEAA)**
One open issue is whether the "tableness" should be directly built into a language, or instead be an add-on library to existing languages.
*It's not really an open issue, since what it should be isn't relevant... what's relevant is what it is, and most major languages do not have table support built in, it's always in the libraries. Arguing about what should be is pointless.*
It is a name given to programming using a database and some procedural code. The term is an invention of TopMind ("top"), and as such there are no communities, conferences, journals,

publications, associations, consortia or well-known papers devoted to TableOrientedProgramming.

- I have seen a book with something like "table oriented programming" in the title, and it talked about using logic tables for programming, somewhat similar to ControlTables, but with a heavier reliance on Booleans. However, I have not been able to find it online anywhere so far. It was published approximately in 1970 and the author was something like "Goldstein". But, I do kind of consider myself the "AlanKay of T.O.P." in that although I did not invent it, I did (attempt to) define, describe, and evangelize it. And I have seen it fairly often in the ExBase community when I did ExBase consulting at multiple companies; it just never had a clear name/identity. Real programmers were "doing it" even though they did not call it T.O.P. --TopMind

*I still have no idea what TableOrientedProgramming is. If it is simply code that uses RelationalAlgebra, well I don't get why the syntax matters (i.e. procedural or object-oriented).*
What it is, is a name for using a database and some procedural code to put up crud screens, while doing most of the work with SQL. Nothing special, basic first year programming stuff. {I don't know, most colleges tend to teach code-centric techniques, not data-centric techniques regardless of "year". And I see only a weak relationship with CrudScreens, which is mostly a UI issue, not an underlying structure issue (although TOP tends to blur the tools used by both ends). And, Microsoft's approach is not very table-oriented in my observation. It tends to follow the flavor of the day. For example, the built-in "objects" of Microsoft Access (queries, macros, reports, etc.) could have been stored in a table-oriented way; in which case they could be queried just like any other table instead of for-each object iteration to access them via code. FoxPro partially followed the TOP path, I would note. -t}
*OO tends to do things in code that would be in tables under TOP. Generally one is more likely to store taxonomies, relationships, and structures in tables rather than in code under TOP. Inheritance is putting a taxonomy in code, for example. In TOP thinking this is a no-no because taxonomies are relative to use.*
It sounds like Prolog Language would be an example of Table Oriented Programming: everything is represented as nary relations (tables) and code is represented as logical statements (with some procedural pollution) about those relations.
*PrologLanguage and RelationalLanguages share many things in common.*
What is it that distinguishes a RelationalLanguage and TableOrientedProgramming from PrologLanguage?

- Prolog has recursive queries, and hence is Turing Equivalent, whereas Relational Languages do not and are not. Further, in Prolog relations are free form and highly dynamic; one need not create table headers before creating data tuples, and related to that, tables with many rows are comparatively rare in Prolog, while tables with single row are rare in Relational Languages. Prolog has much more general support for relations, but is vastly slower for the kinds of applications that RDBMS systems are typically used for. The two areas have an important theoretical link, but are largely non-overlapping in pragma. (Related: Dynamic Relational)

---

Characteristics of Table Oriented Programming:

- Heavy use of DataDictionarys
- Heavy use of ControlTables and table editors (TableBrowsers) to store and manage business rules, categories, logic, configuration, and meta-data.
- Closely related to ControlTables (above) is the belief that often it's best to manage code in tables **instead of files**. This often leads to criticism of file systems and file-oriented development tools. TOP proponents tend to find file systems "primitive". For examples, see InternationalUiExample and SeparationAndGroupingAreArchaicConcepts (source-code management discussion).
- A reliance on SQL or query languages to do a large part or majority of the processing.
- Languages that are well-integrated with query languages and possibly table editors and TableBrowsers.
- Languages that support "local" and internal tables and/or query systems so that the boundary or conversion effort between local and RDBMS is relatively small.
- Tends to view tables as a near-universal data structure and de-emphasizes arrays, linked lists, pointer graphs, object graphs, etc. (See AreTablesGeneralPurposeStructures.)
- A belief that tables bring about uniformity
- A belief that tables better match human physiology and/or psychology (or at least a large enough percent of the population and/or application to make it a viable tool).
- A belief that APIs or their equivalent would be simpler and better if they used existing table-oriented standards and conventions. Example: KissWebServices is merely a web wrapper around SQL.
- Variations-on-a-theme tend to be represented as rows in tables or ControlTables, often one row per variation, instead of code, such as where sub-classing in OOP and case/switch statements would otherwise be used. Blanks or nulls may represent "parent" or "default".
- Values HigherOrderFunctions used with collection-oriented idioms (although existing tools often don't support this well.) Imagine SQL similar to:

```
/* execute mySubroutine for each result row */
SELECT mySubroutine(st.columnA, st.ColumnB)
FROM stuff AS st
WHERE foo = bar
```

It is not intended to be a "Boolean concept", but rather a continuum. And it may not embrace SQL as the "ultimate" relational query language, but perhaps respects it as a "good enough" standard. Also, adherence to "strict" relational (as interpreted) may vary widely between supporters.
The industry sometimes uses the term "data oriented" or "data-driven" for systems and tools tied heavily to SQL and RDBMS.

---

**Q:** How do you justify that 'beliefs' are characteristic to a programming style?
*Good point. Some of the above perhaps belong below, in the "Belief Policies". I'll put refactoring on my to-do list.*

---

**Belief policies behind T.O.P.**
1. DataAndCodeAreTheSameThing. Programming code is simply a manifestation (view) of structures (CodeAvoidance). To an interpreter, code is just data. LittleLanguage is a productivity and abstraction technique to define a mini domain-specific language. With TOP, one is essentially doing the same kind of thing by creating a "little interpreter", a domain-specific interpreter. (However, it will usually have a more declarative feel than actual interpreter code.)
*No one seems interested that code-centric approaches emphasize something different than data-centric approaches. I am having trouble putting this into words, but my experience tell me data-centric approaches yield data driven programs. TOP smells (to me) like data driven programming. In my experience, placing an emphasis on the data over the code created simpler, elegant designs that worked better than the structured, code-centric approaches. May the wiki please speak to this? I also note that DataDriven is a link into a sister site (which speaks to DataDriven testing); why such a vacuum here?* - jme

- When you say "data driven", do you mean http://foldoc.org/foldoc.cgi?query=data+driven? It doesn't sound like it.
- *Data driven could also mean CollectionOrientedProgramming. There are many other "types" of collections (data structures) besides tables.*
  *Thus, TableOrientedProgramming is more specific.*

2. If we focus on data structures, then we should focus on finding the best structuring methods.

- *Which raises the question: Best at what, exactly?*
- Perhaps this relates to AreTablesGeneralPurposeStructures
- It could also be said to relate to the age-old "navigational" (NavigationalDatabase) versus relational structuring techniques. Codd and Bachman (sp?) battled it out in the mid 70's, and it still seems to be simmering with OO/XML versus relational today.

3. The best data structuring method appears to be relational in most cases.

(In practice, existing tools are not up to a full-blown table-centric approach, so compromises are in order.)
4. There are more pre-package-able data-centric idioms than behavioral-centric idioms DataIdiomAndBehaviorIdiomQuantity. And thus if we shift our design to be data-centric, we can take advantage of these existing idioms to avoid reinventing the wheel and avoid relearning the idiom systems.
5. **The Eyes Have It** - Information in tabular form is easier for most humans to relate to and digest. For example, patterns can be visually spotted in tables that would be more difficult in other approaches. Some kind of TableBrowser and/or query language can be also used to transform tables (or views) to emphasize certain aspects. Contrast this with textual code which is pretty much stuck in the format that the original author provided. You are thus at the mercy of the originator's view. (Code browsers are essentially graph browsers, which sort of fill in the same kind of need, but "navigational" structures are generally more difficult to "re-project", and this is one of the reasons why NavigationalDatabases fell out of favor.)

- I disagree with your premises / assumptions. (1) "Navigational" is about **API**, not about **structure**. One can certainly represent, transform, and query even DirectedGraphs - which are about as 'navigational' as a 'structure' can feasibly get - in non-navigational manners. NavigationalDatabase fell out of favor not so much because they used graphs/trees, but rather because they didn't much support whole-graph queries, joins, updates, and views. Navigational APIs view and surgically manipulate a database through a straw. (2) There are some interesting ways of organizing code in GraphicalProgrammingLanguages, Smalltalk IDEs, etc. that should be found among your "other approaches" but that seem to have been ignored, which makes your claim about tabular approaches quite dubious. Organizing code such that it's in much smaller chunks with implicit context such that they can be organized and rearranged at convenience is certainly feasible and worthy of pursuit, but you seem to suggest a dichotomy between 'tabular' vs. 'textual in files'.
- *Regarding (2), They have not really caught on, and I'm generally comparing the more common techniques to TOP. Feel free to create a topic to sing the praises of GPL's. Regarding (1), see TableOrientedProgrammingDiscussion.*

6. Having MoreThanOneWayToPresentIt is a good thing. With query tools and table browsers, it's relatively easy to transform one's view of table-ized info into a form more conducive to the task at hand. Contrast this with text files where you are more or less limited to the author's grouping and presentation of code. (Fancy IDE's can be more flexible, but they are essentially confirming GreencoddsTenthRuleOfProgramming.)
7. Tables are easier for most **power-users** to grok compared to linguistic-centric approaches, in part because of the ubiquity of spread-sheets. See CompilingVersusMetaDataAid.

---

**One area where OO and procedural languages do not operate as well as relational are in operations done over collections or sets of the same object.**
The ways that OO, procedural, and functional languages support operations over collections are:

- Write a separate loop (typically a for loop, but other variants are also used) for each operation and manually verify each loop is identical. A unique function or set of steps is embedded within the loop.
- Write a single loop but pass it a function as a parameter; each function must have the same signature. This can be done directly with functions in assembly, C, and C++ although the syntax is relatively obscure; one can use the "..." syntax or K&R C to avoid much or all of the signature restrictions. For OO languages, the function may be embedded in a function class and class member variables my be used in lieu of function parameters using a variant of the Visitor pattern.
- Use an InternalIterator or ExternalIterator to iterate over the collection; the logic to be applied is inlined in the code.
- Use HigherOrderFunctions (like mapcar/apply, fold/foldr/reduce) to iterate over the container (or reduce it), passing in a function to perform the necessary operation. For example, suppose we could execute a function for each record by having constructs borrowed from SQL such as (PageAnchor: exec_1):

```
EXECUTE myRoutine(columnA, columnB) FROM myTable WHERE x > y

// This version makes references clearer:

EXECUTE myRoutine(m.columnA, m.columnB) FROM myTable AS m WHERE m.x > m.y
```

- (In practice, most systems don't directly support the above, but we can rely on similar constructs by convention.)

In comparison, SQL treats data inherently as a collection. All operations have the loop built into the language, compiler, or library so that the programmer does not need to explicitly code the loop nor take any unusual steps to pass a function to a loop.
*Unfortunately, SQL isn't sufficient as a programming language for most tasks, so one must pull the data into memory and process with a real language, which puts us back to using loops and functions, or HigherOrderFunctions to process the data. Keep in mind that SQL is probably far from the ideal TOP language.*
Language constructs and examples that support passing a function into a common loop:

- C++ STL for_each template

- JavaScript [Note: Using "for in" in JavaScript loops over the properties of the object, not over it's elements, so for(each in aList) won't work.]

```
Array.prototype.map=function(toRun){
for(var index=0;index<this.length;index++)
        toRun(this[index]);
}

Array.prototype.filter=function(isCondition){
var result = new Array();
for(var index=0;index<this.length;index++)
        if(isCondition(this[index]))
        result.push(this[index]);
return result;
}
```

To run over a collection, use the following:

```
aList.map(function(each){each.DoSomething()});
```

or

```
var theMatches = aList.filter(function(each){return each == "some condition"});
```

- HigherOrderFunctions in FunctionalProgrammingLanguages and MultiParadigmLanguages
- Smalltalk blocks

```
aList do: [:each | each doSomething].
```

or

```
|theMatches|
theMatches := aList select: [:each | each = 'some condition'].
```

- Java InnerClasses
- foreach in CsharpLanguage and JavaLanguage 1.5

Either way, they remove the need to ever have to manually write loops. Since you are passing functions to these methods, no other parameters are needed.

*Why can one not merely derive a user collection class from some base collection class and from the class to be contained in the collection and have the user collection class then expose all methods from the base collection class? Without writing any additional code, one could call a method on the user collection class and it would automatically iterate through its collection, calling the same method on each member. For example, I could create a collection of text boxes and then call a SetBackgroundColor(newColor) method on the collection and it would set the background color on all text boxes in the collection. -- WayneMack*

[*Consider that that is easily done by a generic foreach or do method like so....*]
[*aTextBoxCollection.forEach(function(each){each.backGroundColor="newColor"});*]
[*You don't really want specific methods as you suggest, you want generic ones that can take functions as parameters to specialize their behavior, they are much more flexible and require much less code to be written over the long run. A collection should not be concerned with the particulars of an operation, only that the operation needs to be applied to all it's members or some of it's members. Therefore we leave the operation open for extension by making the operation a parameter, and only write the loops on the collection.*]
[One can and many do. I wouldn't inherit implementation from both the collection and the member classes, though. I might inherit implementation from the collection and interface from the member if that interface makes sense in the context of a collection. Usually I just use a collection that has a forEach() method and pass it a closure/block/inner class instance that performs the desired behavior.]
Would you care to expand on why it is advantageous to implement methods on a collection outside of a collection class? (I guess the little light bulb just didn't go off in my head when I read the above comments.)
[I assume you mean the methods passed to "forEach()"? If so, those methods don't operate on a collection. They operate on members of a collection. "forEach()" makes them operate on every member of a collection.]
(Aside, I'm going to start to refactor the top of this section to eliminate bulk. If I lose any important content, please refactor back in.)

---

Just an idea. It would be interesting to see a language that could do something like:

```
  tableX.doStrategy() where [condition]
```

This would be more or less equivalent to:

```
  r = sqlQuery("select doStrategy from tableX where [condition]")
  while (row = getNext(r)) {
    execute(row['doStrategy']);
  }
```

Based roughly on concepts discussed in EvalVsPolymorphism.

> That's why people rave about smalltalk's collections: that's exactly what you can do. Likewise, there are several libraries which wrap java's collections which let you do the same thing, and the whole thing is trivially easy with Lisp. Add ListComprehensions to the mix, and you even get the set algebra: "[ doStrategy x y | x <- tableX, y <- tableY, condition x y]".

---

This might be a good place to start a discussion on what BuckyPope of IbmThomasJayWatsonResearchCenter used to call "class codes" - maybe it has since reappeared under a new name. He noticed that you very often find code like:

```
  IF PROVINCE = 'NB' OR PROVINCE = 'PEI' OR PROVINCE = 'NS'...
```
in programs. If you have many occurrences of this kind of thing, it becomes very hard to maintain. He advocated storing the attributes that you are interested in in tables, so this code becomes something like:
```
  LOOK UP PROVINCE
  IF PROVINCE IS MARITIME ...
```
While this looks obvious in hindsight, especially to TableOrientedProgramming people, it is surprising how often you see statements like the former in business applications. They just seem to grow, and it's hard to stamp them out once they take root!
I have just come across the ControlTable page - it sounds very similar.

---

If someone were looking to design a new TopBasedLanguage, what would you consider to be necessary, what would you consider desirable, and what would you wish to avoid?
First-class relational operators would seem an obvious one, but would you implement them as a RelationalAlgebra or a relational calculus, or a combination of both?
What else would you want? Transaction management? List comprehensions? Shaped arrays? First-class StoredProcedures, including possibly RecursivelyStoredProcedures (i.e., inner functions)? Higher-order functions (presumably operating on the StoredProcedures)? would functions be both represented and implemented as tables, homoiconically (see HomoiconicLanguages)? What sort of syntax and semantics would you want?
It also occurs to me that if you can have RecursivelyStoredProcedures, and if you save a stored procedure's lexical environment in a table along with the code, and invoke it at need, you would have a LexicalClosure - which, since ClosuresAndObjectsAreEquivalent, shows that the two paradigms must, on at least a theoretical level, be compatible. Similarly, if you can store a function's successor for later invocation, you could have continuations, in which case you wouldn't necessarily need the inner functions. Of course, using a table for this purpose implies the existence of persistent continuations, which is JustWrong...
*One has to be careful to design something that can be both big and small. For example, one should perhaps be able to use such a language or kit with MinimalTable abilities, but also plug in DB2 or Oracle if needed without code overhaul. I also think that as much as possible should be in libraries instead of hard-wired into the language. Thus, ideally the language would offer meta-language abilities so as to easily extend it. I would like to see Lisp reworked to be more palatable to the "masses". Java grew popular partly because it borrowed from people's C/C++ familiarity (for good or bad).*
Goddess Eris, it just occurred to me! TOP isn't a paradigm - it's a MetaObjectProtocol! TopMop! ;) -- JayOsako
*Well, I suppose everything could be defined in terms of MetaObjectProtocols, or any other TuringComplete paradigm for that matter. OO just has a psychological tilt toward behavior instead of declarative techniques and is not "bound" to relational rules. In the end it is all about psychology. -- top*

Related: MaspBrainstorming

---

**Spreadsheet-Influenced Ideas**

I once got an email from a fellow table fan who suggested using spreadsheets, or at least a spreadsheet-like interface for programming. Initially I was not very warm to the idea, but looking for better ways to combine nimble tables with code, I am now warming up to the idea. Most spreadsheets allow text to flow into adjacent cells as long as there is nothing in those adjacent cells. Thus, long programming text will not cause problems. And, the cells can be used for indentation, which can avoid the TabMunging problem that pure text keeps facing. If a section of code is indented 3 cells, then it is always indented 3 cells regardless of what edits it. Tabs versus spaces is no longer an issue.

But the biggest benefit is the ability to embed and define smaller tables inside "code". Think of them as "table closures".

Here is an example. First the non-spreadsheet version of the code:

```
  // define column headers

  addColumnHeader("Name");
  addColumnHeader("Student ID");
  if (authorized) {
    addColumnHeader("SSN");
  }
  addColumnHeader("Grade - GPA");

  // loop for each student

  while (row = getNextStudent()) {
     outNewRow();
     outColumn(row['name'], left, '');
     if (authorized) {
       outColumn(row['SSN'],'center','');
     }
     outColumn(row['studentID'],'right','');
     outColumn(row['GPA'],'right','#9.999');
  }
```

Spreadsheet version:

```
 A..B.......C...........D.........E.......F
 ...............................................
 table rptCols // report column definitions
 ....FldName..Descript.....Alignment..Format...HideOption
 ....name.....Name.........left
```

```
....SSN..................center.............Yes
....studentID............center
....GPA.....Grade.-.GPA..right......#9.999
end.table
.
loop on table rtpCols where not HideOption and authorized
....addColumnHeader(Descript)
end loop
.
loop on table Students
....outNewRow()
....loop on table rptCols where not HideOption and authorized
............outColumn(&FldName, &Align, &Format)
....end loop
end loop
```

(Dots only to prevent TabMunging)
This is rough pseudo-code and not necessarily meant to promote a certain style of syntax. Also, in a real spreadsheet we would be able to see the cell grid, which would make the table much more clear. The "loop on table" structure is similar to ColdFusion's <CFloop query="foo"> tag, which allows column scope within the tag. (However, perhaps such a feature should be allowed to be turned off if desired.) The instantiation of result set "Students" is not shown in either example. We may be able to factor the "where" criteria to one spot. Benefits:

- No quotes needed in table
- Stuff is aligned to help see patterns
- Adding new report fields can be done at one spot instead of two.
- I personally find it better SeparationOfConcerns and cleaner. Stuff about what is displayed is not mixed in with details on how to display it. The first half is declarative and the second half imperative for the most part.

I imagine some adjustments or additions would have to be made to spreadsheets to make them more useful for code editing and viewing. For example, one may have to keep resizing the cell (column) widths in order to view portions of the current screen. Maybe this can be automated. for example, pressing F6 may stretch all cells to fit all the information in the current screen, and pressing shift-F6 puts it back the way it was. Even in the example it is obvious that the indentation (cell width) for viewing tables tends to be different than the best for viewing code. Keep in mind that most spreadsheets display the current cell's contents at top, so that if any single cell is too large to show, it is relatively easy to put the cursor on it in order to see its full contents at top.

Maybe some way can be devised to alternate between a grid view and text mode in the same module (virtual or real). Maybe the browser can have "start grid"..."end grid" and "start text"..."end text" markers of some kind, and the editor would display that section in accordance. In an editor, it could kind of resemble the "bands" found in many report writers. One could slide the bands larger or smaller as needed, and a check-box in the band bar would determine whether it is a grid or text band. This would allow text where it is best used and grids where it is best used, but have them intermixed in the same view rather than going back and forth between the grid screen and the text screen that most approaches currently require. -- top

---

**MultiThreading**

TableOrientedProgramming reduces the need for multi-threaded programming. One can launch independent processes or sub-processes that communicate entirely via tables. Most table engines and RDBMS have concurrency management built in, either with row and table locks, and/or with transactions. Since these are available in the table engine, adding them to a language is kind of a violation of OnceAndOnlyOnce. However, this may not be effective for certain embedded or timing-sensitive applications. See AreRdbmsSlow for more on timing issues.

- *WOW, that quite a claim. You do realize that Mulithreading is about avoiding process overhead right? And you know that not all threads need to share data, or that when they do it is always best to avoid threads right? An you are aware that message passing via queues tends to do the same thing right? And you know about pipes and commercial messaging products like MQSeries right? If you are using tables only to pass non persistent data there is something wrong with your design. If you are doing it to ensure delivery then you should consider if your protocol really needs to be stateful.*
- I'm not sure what you are getting at. Perhaps this discussion belongs under TableOrientedSynchronization.

*This seems to be assuming that the database is necessarily provided by an external process. Integrating it with the language is likely to be both more efficient and more convenient, and personally I'd much rather do this in a language with good support for concurrency (say ErlangLanguage using the MnesiaDatabase), than try to avoid MultiThreading. -- DavidSarahHopwood*

Perhaps, but then you lose multi-language ability. If you "share", every language then does not have to invent and perfect it all from scratch.

*Suppose we use n programming languages in a project, one of which acts as a hub to integrate components written in the other languages and to access a database. If the database is external, then the hub language is effectively SQL. I'm just suggesting using a real programming language as the hub instead. The number of inter-language interfaces is the same: n-1 in each case. -- dh*

Why would it be called the "hub" then?
*The central node in a star (a.k.a. HubAndSpoke) network is usually called the hub. "Pivot" would be another possible term. We use a star network because we don't want n*(n-1)/2 inter-language interfaces.*
{So the only difference between that and a network is that the edge nodes are not allowed to talk directly to each other? Is that the only distinguishing feature you are suggesting? If not, what are the other distinguishing features?}
Or use FlowBasedProgramming, which can link together modules written in different languages, as well as reusable components that drive different services, e.g. SQL. In the first FBP implementation we had a reusable stream-to-stream component that drove the standard IBM Sort (or any vendor Sort that used the same call interface) - it was *much* more friendly than using the vanilla Sort because you could change the key on the fly, route data around it, etc. Similar comments apply to a reusable SQL component. -- PaulMorrison
Related, and perhaps a merge candidate: TableOrientedSynchronization

---

**Tables All The Way Down**
From WhenDoSchemasAndClassesDeviate:
Re: *nobody has managed to design a TableOrientedProgramming language that's "tables all the way down", like SmallTalk is objects.*
Tables are best used as a larger-scale structure than objects. Rather than everything *be* a table, a "pure table" arrangement would probably look more like everything is *in* a table one way or another. For example, every variable might be a row in a Variables table, every function defined in a Function table, etc.
Things like Microsoft's semi-generic bytecode interpreter (CommonLanguageRuntime) is essentially a database, just not a relational database. A TOP approach would change that fact and toss the underlying NavigationalDatabase for such a thing. But, MooresLaw has not quite caught up yet. Source code is only one view among many possible of programs. It is about divorcing presentation from meaning, as described in CodeAvoidance.
Further, DynamicRelational may be required to keep it flexible enough to be practical. So far there are no completely dynamic implementations. After all, Smalltalk is dynamic. Relational can be too.
Another issue is that maybe a yin-yang kind of relationship may be better than a single atomic "type". See YinYangVersusSinglism and FormulasPlusAttributes.
-- top
*But I would be interested in seeing an implementation of a tables-all-the-way-down system. How could you represent functions? Functions lend themselves to hierarchical containment. Mathematically, a function is a mapping from a set to a set, and a table is a set, so could you map a table to another table? Could you have virtual tables that have an infinite number of rows (say for real-numbered values) and implement this mapping? I'm just pontificating here... this would be interesting to figure out.*

Using the mathematical definition of a function, a TOP language is almost trivial. A function is really a set of ordered pairs <a, b>, where a is a member of the domain (a set) and b is a member of the range (another set). This can be represented as a table with the two columns (domain, range). Functions of arity > 1 would just be represented as a table (arg1, arg2, arg3, ..., result).
The problem is, the SetTheory definition of a function is very difficult to work with on a daily basis. You don't want to have to explicitly specify outputs for every input; that defeats the purpose of a computer. Go use a paper-and-pencil spreadsheet.
Instead, most functions are defined as compositions - possibly recursive - of other functions. This is fairly easy to see in Lisp notation: (factorial n) = (cond ((= n 1) 1) (T (* n (factorial (- n 1))))). There're some tricky bits, like the conditional and the use of recursion, but these are all covered by DenotationalSemantics.
Something like this is very hard to represent as a table. It's fundamentally hierarchical: functions are made of compositions of functions, which are themselves made of compositions of functions, etc. And for reasons of practicality, most of these intermediate compositions are unnamed. They're also denoted positionally by their position in the argument list. When you try to stuff this into a table, you basically end up with a TreeInSql.
*There is no need to* implement *a function as a table by listing every (argument, result) pair. A function could be represented like a table, i.e. every object* looks *like a table. After all SQL views look like a table, but are none (being views into one). Looking at a function as a table it would be easy (and difficult in other approaches) to get the domain and range of a function. Of course no every query on an function-table could be computed effectively let alone efficiently.*
*I find it a worthwhile idea to combine such an approach with e.g. a SetOrientedProgramming approach based on PrologLanguage. Handling small set (in memory) and large sets (database) uniformly (possibly annotated with hints about size and atomicity) could provide a very elegant and safe way to implement critical financial applications.* -- GunnarZarncke
PS. *It seems, this (same approach for large and small sets) has already been done: See HaskellDb.* -- .gz PPS. See PrologForMassiveData
There is one exception: ContinuationPassingStyle. In CPS, every function call must appear in tail position, so no argument is itself a composition of function applications. The tree has been flattened out and labels assigned to every intermediate node, which is just what we need to store the full composition in a table. I'm very curious about a table-based CPS intermediate representation: it looks like there's a full mapping from CPS to relational tables, and this also encodes the complete call graph of the program. Closure analysis may be tricky though; most CPS representations store environment information in the lexical structure of the CPS representation, which would be lost on conversion to tables. If that can be encoded too, however, then you'll have the complete liveliness graph of all registers encoded in the

intermediate representation. This would let you do really efficient inter-procedural register allocation. -- JonathanTang

---

Keep in mind, a table is just one of several WaysToExpressRelations.
*Everything is probably equivalent to everything, if we want to go that route.*

---

I wonder if the compile-time-type-checking view is in direct conflict with TOP. A good portion of TOPs challengers are pro-StaticTyping it seems. Heavy pre-run checking requires finding ways to limit the variations of "queries" or dispatching in order for the compiler to "reason" about them. Such tools lean toward hierarchies in order to limit the search tree on "types" and related variations. However, if one finds trees inherently limiting, the alternatives don't seem to offer enough regularity compared to trees to do pre-run pass-or-fail analysis. My view is that this simply reflects the real unescapable nature of the world (LifeIsaBigMessyGraph) and that tree-based checking is a false and messy security. You can't compile the whole world. -- top
*It's not in conflict or required; it's orthogonal. This is obvious if you have a sufficiently expressive type system. See also FourOutOfFiveRule. -- DavidSarahHopwood*
I doubt there is such as thing as "sufficiently expressive type system", as described in ThereAreNoTypes.
*By "sufficiently expressive type system", I was talking about type systems that exist in practice, for example SoftTyping in MrSpidey.*

---

Looks like someone's trying to cash-in: http://www.tablecode.com/founders.htm. And it's not TopMind.
You guys can fight over it. Let me know who wins.
*The patents often talk about "objects in tables". As long as nobody calls them objects, no problem. However, they also talks about "code in tables". Does this mean that "Eval(myColumn)" has been patented?*

- Currently, the "patent" is merely a patent *application*, not a granted patent. And it is, as far as I can tell, utter BullShit; TopMinds stuff appears to be PriorArt for much of it.

Anyway, the guy claims to have "invented" TableOrientedProgramming. While I can find references to him on the web that are earlier than 2004; I can't find anything older than last year on TOP.
ThePragmaticProgrammer reportedly has content on the topic as well.
JoelOnSoftware has a thread about this: http://discuss.joelonsoftware.com/default.asp?joel.3.166265
The Diabs have a paper up on ACM; the list of references is embarrassing: http://portal.acm.org/citation.cfm?id=1094855.1094930#references

---

While it isn't TableOrientedProgramming per se (and I imagine that there is much prior art in CommonLispObjectSystem for this), a paper that TopMindmight find interesting: http://www.informatik.uni-ulm.de/rs/mitarbeiter/ch/publ/Heinlein:Goerigk:Kiel.Inf:2004.pdf *Perhaps this link could go under EvalVsPolymorphism.*

- A better location would be a more general page on dispatch techniques. Like many other things, the stuff in that paper is largely orthogonal to eval.

---

Top,
I'm curious: Did you edit the following article on WikiPedia? http://en.wikipedia.org/wiki/Table-Oriented_Programming
I didn't see any of your usual IPs (dslextreme) in the edit list, and you didn't post with a recognizable username. I *did* notice, however, your fingerprints on http://en.wikipedia.org/wiki/Design_pattern_%28computer_science%29 and I'm curious: Why do you think that design patterns have anything to do with the OO-vs-relational debate? Were one to embrace TOP whole-hog; I'm *certain* that higher-level patterns would emerge. Many view design patterns as only relevant to OO; in my experience they are applicable to almost any programming paradigm. -- ScottJohnson (who edits Wikipedia as http://en.wikipedia.org/wiki/User:EngineerScotty, btw...)
*I actually did not create the Table Oriented Programming topic over there. However, I do remember encountering it and making changes quite a long while back. I forgot all about it until you mentioned it. -- top*
I see you did delete some vandalism from the page (vandalism which uttered your name in vain)... do you still prefer to not have your real name published? -- sj

- He's been stating that preference for many years; there's no reason to assume that his policy has some built-in time limit that will expire at some random point. :-) -- Doug
  - If you go to the tablizer website, and look at the bottom of the pages, you'll see the following:
  - "...Material © Copyright 1998...2005 by Findy Services and [Top's First Initial and Last Name]"
  - Plus, there seem to be quite a few guys on the net who try and "out" topmind wherever they can... if Top wants to keep his real name a secret, he does a rather poor job of it. :) -- *First letter S...* :-)
  - *That's not the point. We all know what his real name is, and he knows that, and isn't trying to keep it a secret as such, he just does not like to be addressed by his real name, because it makes it overwhelmingly obvious who he is, as opposed to it simply being possible to find out his real name. Since his given rationale is to avoid threatening his career, this seems reasonable to me; a potential employer*

*will search on his real name, **not** on TopMind, so his approach works perfectly well for that kind of thing. -- Doug*

  - Although, typing Top's real name into google produces the tablizer page as the first couple of results... -- sj
    - Discussion moved to RealNamesPlease.

---

For those of us sick of writing and manually managing essentially tabular data in an object-oriented language like C++, I'm interested in pursuing a TemplateMetaprogramming approach to auto-building tuples, tables of tuples, and indices within the C++ runtime, and maybe a TemplateMetaprogramming supported query language, too. Might be something boost-worthy. However, I ain't tacklin' that one alone. My main use of tables is when I need to index the same data N-ways from Sunday, I almost always end up writing them by hand, and I rarely use joins (though I know how to go about it after writing the C++ version for EveryCombinationInManyProgrammingLanguages).
*The SqLite libraries are C-based. Perhaps you can incorporate them into your code.*

---

When I coined "table-oriented programming", this is NOT what I had in mind: BefungeLanguage (http://en.wikipedia.org/wiki/Befunge)
That is more like "grid-oriented programming". Grids are like 2D positional arrays and tables are like 2D associative arrays.

---

Re: Quote: "A belief that tables better match human physiology and/or psychology (or at least a large enough percent of the population to make it a viable tool)."
Psychology has nothing to do with tables. Human psychology is oriented around depression, happiness, food, replication, sadness. The relational model was created to organize data efficiently and sensibly, not to tie in psychology of the human brain. in fact one problem with bad programmers is that they let their emotions get in the way of truth - and truth is what computers are all about. If you want to model the human brain you'd be best looking into artificial intelligence and fuzzy logic. The relational model is very un-human and un-animal. It is infact extremely "true or false" oriented, centered more around the CPU "bit" where there is **0** or **1**. Bringing psychology and anthropomorphisms into programming is extremely dangerous, because databases are not about human psychology. Databases are about correct data retrieval (would you want erroneous data?) that can be accessed efficiently and did I already say it, **correctly**. Tables (relations) were invented so that data would be accessible in an efficient and correct manner. And by efficient I don't mean that your database is "fast", I mean efficient organization of data where one can look up how many customers live in so and so state or province without having to write tons of code (normalization helps significantly there). It has nothing to do with psychology. It's about accessing the data in the most efficient manner. And once again, by efficiency I do not mean the clock time.

*"Psychology" involves lots of things. Emotions is one of them. But there's also the "perceptual" side, that borders on physiology. I wish there was a better name for the aspect that we are considering here, but I haven't found it. Most people would agree that "psychology affects grokkability". Somewhere on this wiki is a discussion about whether relational is a "natural" property of our universe, like Pi, or whether it's somehow tied to human nature and human qualities. I suspect it's the latter. Machines and math don't "care" either way. Relational is merely a UsefulLie for humans. As far as "correctness", I'd use some form of relational even if it didn't have referential integrity etc. That's a nice bonus, but not the single magical factor. -t*

You ought to do some research as to why the relational model was invented. It was invented to organize data, especially repetitive similarly structured data. The relational model (tables) are tied to data, not some natural property of humans, but the natural properties of data. The natural properties of data are: often you will see repetitive entries (by repetitive I mean they all fit into a table - for data that doesn't have a common structure, that's where relational fails.. for example a GUI with a button, a panel, and a window - these don't all fit easily into a table and are better as single structures or objects (unless you had 3400 buttons on the screen that needed to be searched)).

*I agree about the factoring part, but navigational (NavigationalDatabase) can also be used to factor out repetition. Thus, relational has no monopoly there. I disagree that a table-oriented GUI is not possible. I even gave an example in NonOopGuiMethodologies. Hardware wasn't up to the task when GUI's started, but may be now. Further, DynamicRelational may be a better fit for such anyhow rather than the Oracle clones we have now. Further, a lot of "kinds" of widgets can be rolled up into just two "types" of widgets: singular and compound. The difference between a label, link, click-able image, and button are not sufficient to make them separate "kinds" of things. If I want to make an image click-able, I don't want to have to change it's type, I merely want to be able to be able to add clicking behavior/attributes to it. Viewing them through the smorgasbord model [see...damn-i-forgot-name] may give us more flexibility and combinations. I agree that hard "types" may be a UsefulLie, especially for newbies, but does not add to flexibility and reuse.*

---

MicrosoftAccess is "proof by popularity" that TableOrientedProgramming works: entire small and medium-sized applications are often written in Access without using a single line of code (other than expressions, such as ">id"), or at least very little. Many Access programmers don't even know VBA. That being said, I would do many things different if designing a better TOP C.R.U.D. thing-a-matic, the primary one being making all the application configuration, queries, settings, macros, etc. be full-blown tables that can be queried and analyzed by the DBMS like any other table can, not the proprietary hidden binary structures that MS often uses. It may have originally been done for efficiency reasons. For example, to allow lots of open-ended "cells", a Memo type would have to be used often, and these tend to be slow. More efficient auto-sizing would have to be explored. Existing RDBMS are tuned for data usage patterns, not config attributes. Table-based management of query parameters would also help. -t

*At best, MicrosoftAccess is "proof by popularity" that desktop DBMSes using QueryByExample and reasonably-usable built-in form and report painters plus an included implementation of VisualBasic for scripting and marketed by MicroSoft can be popular. It says nothing about TableOrientedProgramming, whatever that is. Arguably, what is being called "TableOrientedProgramming" here is nothing more than using the RelationalModel, or something resembling it, to build applications.*

I'm not talking about form and report painters, for basic (and esthetically ugly) apps can be built without ever touching them. Perhaps what I'm getting at is AttributeOrientedProgramming, which is a close relative of TOP. An entire app can be built by merely filling in attributes and attribute tables. Most if not all of the things in Access could be turned into tables or queries on tables, and many of them are. For example, the "switchboard" (menu panel) wizard creates a switchboard ControlTable, which is internally used to control the switchboard. The macro commands are also presented to the user as a table/grid (although I don't know if it's implemented as a table under the hood). And I've built basic ControlTable based report writers for ExBase with sub-grouping/totalling back in the ASCII days. Basic forms can also be done that way, if you don't mind meek aesthetics. Let's review:

- Queries – Designed primarily via a QueryByExample-based grid interface. These can be Select, Update, Insert, etc. queries. One can also edit queries as direct SQL if desired.
- Macros – Attribute-driven in macro designer grid. Could be or perhaps is table-ized under-the-hood. (Conditional expressions are also possible, but awkward in Access.) It's almost like a CRUD-oriented Assembler language, which is fine for short lists of commands.
- Menu Panels ("switchboard") – Table-driven under the hood. A wizard puts a GUI dialog interface over this table.
- Data entry forms – One can use "raw" edit table views, or the drag-and-drop VB-like painter tools. However, these can also be done via table-driven interfaces, although MS-Access doesn't. (You'll just have to take my word for it.)
- Report writer – Access can "auto-guess" from edit table views, but otherwise uses a report painter. I built a table-driven report-writer back in the DOS days. It used a DataDictionary with additional attributes for sub-grouping options, which somewhat resembled Access's query-builder when the "totaling" row is switched on.

Thus, all the basics of typical CrudScreen apps can be table- and query-based. Application programming code and screen/form painters are *not* necessary. (But even painter tools can store the design in tables. FoxPro did this more or less under the hood.) Again, I'm not suggesting one always take the 100% TOP road. I'm only pointing out it's possible. A good many apps can be around 70%. Generally the low-use and internal forms and reports are made using TOP, and the high-use critical ones are more carefully tweaked and micro-managed using or assisted by application code. -top

---

It is exceedingly unclear to me what TableOrientedProgramming really 'means'. Proposed:

- Definition 1: Tables, Sets, or Relations are FirstClass, and so are relational operations on these (union, join, select, etc.), and relations are used (idiomatically) instead of CompositePattern for data. By FirstClass, I mean: table-values may be anonymous, stored to variables in the language (RelVars), communicated as arguments, and returned. Consequences of using TOP under this definition:
    - TableOrientedProgramming is fully compatible with ObjectOrientedProgramming, FunctionalProgramming, FunctionalReactiveProgramming, DataflowProgramming, etc. Some extensions are required mostly to support relational operations, and the language-implementation might do well to provide some automated optimizations (both query-optimizations, and storage/indexing optimizations for the
    - Need for VisitorPattern is diminished.
    - With tables being anonymous, it can be somewhat difficult to define relationships between 'tables' variables, such as ForeignKeys and CascadingDelete and consistency requirements. This is suitable for DynamicTyping, but one may wish to expand to something even broader for StaticTyping: FirstClass 'database' objects where the TypeSystem recognizes as a primitive values that can encompass *many* tables along with the relationships between them.
    - PersistentLanguage feature is possible, but occurs more along the lines of persistent ObjectOriented language.
    - Integration with external RDBMS is unlikely, or at best 'indirect' (i.e. loading queries from remote tables into a local table-value, ideally lazily or at-need). It will likely require some mapping effort on the edge of the system (e.g. via a ForeignFunctionInterface). But if PersistentLanguage feature allows automatic re-establishment of communications, this problem will be greatly reduced.
    - Tables are subject to ObjectCapabilityModel security. That is, tables can be "hidden" within the process, fully encapsulated, inaccessible to DBAs and such unless made available through an IDE or debugger. Even if they are accessible, such tables will have semantics that depend heavily on context and who is using the table, so a DBA would need much detailed knowledge of the application to make sense of the multitude of tables.
    - Mutable-state communication can be avoided (KillMutableState) in favor of passing about immutable table values between components and processing them using pure functions. This can reduce need for ACID transactions, though some

sort of SoftwareTransactionalMemory is still appropriate for the little mutable-state remaining. Whatever solution is used to protect mutable state for other variables is likely to protect mutable state for tables.

- Definition 2: Tables, Sets, or Relations are SecondClass (or at least used as such by TableOrientedProgramming). That is, you can name these RelVars in a "global" space, as part of a program definition. You might be able to create temporary views or tables for processing (i.e. functions that take tables as arguments), but either you cannot return tables, or such FirstClass use of tables (i.e. creating a process that loops forever passing tables as arguments and assigning result-tables to local variables) is idiomatically discouraged by TableOrientedProgramming in favor of communicating between components by shared-state mutation on globally identified tables. Consequences of using TOP under this definition:
    - Communication between components primarily by shared state in common tables closely matches the BlackboardMetaphor and LindaTupleSpaces designs. Process-composition occurs as a flat architecture attached to a common database. Tables are used for RemoteProcedureCall, InterProcessCommunication, etc. Related: TableOrientedSynchronization.
    - Since all tables are named, their number doesn't vary wildly during program execution, and the names are in a global space, it is easy to describe and establish constraint-relationships between RelVars (including ForeignKey, CascadingDelete, etc.) that are often associated with RDBMS.
    - Multiple instances of the same application need semantics for whether all instances share the same tables, or each application gets its own table. One might annotate this sort of information as part of defining the global RelVars (even going so far as to say a particular RelVar corresponds to a particular UniformResourceIdentifier for an RDBMS). It may be that some tables are marked 'volatile' - gone with the wind the moment the application is killed.
    - Easy integration with RDBMS and local DBs (like SqLite). Easy persistence via such integration.
    - DBAs can get easy access to all the tables. Since they're "global", the table semantics are pretty much independent of context. This allows DBAs to easily understand and maintain tables.
    - Challenged to handle two or more databases or data sources - distributed queries and distributed transactions to work with more than one RDBMS will be a serious challenge for implementors. This encourages the 'store everything into one uber-massive organization-global RDBMS' design. Ideally, it would even be world-global, but the security issues of this architecture forbid that.
    - Security-challenged architecture. By default, everyone has access to any table they can name. This is problematic given the above-described propensity to

grow into truly organization-global tables. People have attempted to erect various forms of egg-shell security - i.e. passwords and such - to protect tables and data. If any component of a TOP application is compromised, in general all tables it has authority to interact with are compromised.

- The use of tables for TableOrientedSynchronization purposes creates considerable extra 'cleanup' and burden for the applications. It is rare that resources are sufficient to keep a complete history of everything every process does. More relevantly, general forms of GarbageCollection won't be able to take out tables in a global space, so it will be up to the processes to selectively delete rows from the global tables after taking the appropriate actions. All this exacerbates the egg-shell security problem, since any compromised process will have access to various shared IPC tables and can therefore compromise communications between all other processes.
- Transactions and persistence over tables, as they are, do not include the processes or communications. This can create some challenges if one wishes to perform *behaviors* atomically and persistently - all at once or not at all. This leads to patterns as seen in MnesiaDatabase, where one creates tables that essentially track what each process is doing, what it plans to do next. This requires a great deal of explicit effort and explicit interactions with tables. Further, if persistence is the goal, then the table in question cannot be local and volatile to the application - it must be part of an RDBMS. (For security, this table at least could be local to an application to avoid granting other processes the ability to directly compromise program behavior.)
- It is not possible for two or more independent processes or applications to coordinate a behaviors atomically: communication occurs through the database, but process A cannot see what process B is attempting to do atomically until AFTER process B commits its intent to the database. Because processes do not compose or coordinate atomically, each process will need to be monolithic in nature - i.e. process A will need to *include* the features of process B. Since monolithic processes need much more authority than fine-grained processes, they contribute to egg-shell security issues.

It has been my impression that what TopMind really means when he discusses 'TableOrientedProgramming' is more along the lines of Definition 2. It isn't particularly compatible with ObjectOrientedProgramming, and it's a closer fit to the sort of situations he describes (i.e. relying on external RDBMS systems, procedural+relational, using RDBMS for persistence, etc.).
*It has been my impression that what TopMind really means when he discusses 'TableOrientedProgramming' is ExBase.*

- ExBase is a "taste" of a TOP-friendly language. I honed many of my TOP skills in it and I saw other ExBase fans do the same. However, it still has a lot of warts in my opinion. I once kicked around a re-worked version of ExBase, but realized that a flexible enough general language with good libraries could do almost the same without having to hard-wire DB idioms into it. -t

Maybe so. Either way, it seems to be a relatively limited vision of a programming model.
*Indeed. It's more of a coding style than a programming model.*
{The programming model can make TOP simpler and smoother, but I agree that it's not really about programming languages. However, one of you is a "linguistical thinker" I believe, and to such a person, everything may be "about language" because they think in language terms. I'm not sure how to translate this into linguistic-speak. If I was going to make the "ultimate TOP language", first we'd have to settle on whether we maximize for integration with existing RDBMS and SQL, or focus on an "ideal" (or alternative) relational system. And this includes deciding on compiler/type-heavy or dynamic-friendly designs. Ideally, a language or libraries would support "local tables" to supplement RDBMS usage that have language scoping rules more or less like arrays. (They can be emulated using nested maps and some API's that support a query language.) This is closer to the "first class" description above. As far as "sharing", at this point, I am happy to stick with TableOrientedSynchronization for concurrency issues. -t}

---

After reading this page, I still do not really understand TOP. Thus, I will as a comparitive question. Programming only with functions is possible. Functions can be expressed as an ordered pairs, for example f(i)=i*i where 0<i<4 is the set {{1,1}, {2,4}, {3,9}}, which fits nicely into a table of two columns and three rows. Is this a trivial example TOP?
--EdwinEarlRoss
*Rather than strive for a hard-bordered definition, which will probably result in an endless LaynesLaw battle, I'll try to present more examples in topics such as ViewingAlgorithmsAsCollectionProcessing.*

---

AdaLovelace, the first Table Oriented Programmer?:

---

---

See: KayLanguage, ArrayOrientedLanguage, ControlTable, DataDictionary, RelationalDatabase, FoxPro, SetTheory, RelationalAlgebra, SourceCodeInDatabase, BusinessRulesMetabase, SetOrientedProgramming, CollectionOrientedProgramming, ProgrammingParadigm, PayrollExample, TableOrientedProgrammingDiscussion, EmbraceSql

---

CategoryProgrammingLanguage, CategoryInfoPackaging, CategoryDatabase, CategoryTable CategoryDataOrientation

---

Last edit November 25, 2014, See github about remodeling.