Out of the Tar Pit

JADE ALLEN

JADEALLENX@OUTLOOK.COM
Overview: Complexity is the problem!

- Complexity is cause of the vast majority of problems with software.
- The unfortunate truth: Simplicity is Hard
- Complexity makes informal reasoning about system behavior difficult (i.e., a “white box” inspection of code and using that to inform a mental model of its behavior.)
- “Testing is hopelessly inadequate” – Dijkstra – Why?
- Simplicity > Testing
Overview: Causes of Complexity

- "State"
  - What is “State?”
  - Impact on testing
  - Impact on informal reasoning

- “Control”
  - Sequence of events/concurrency
  - Implicit sequence of program execution
Overview: More causes of complexity

- Complexity caused by code base size
- “Complexity breeds complexity”
- “Power corrupts”
Approaches to manage complexity

- Object orientation
  - "suffer[s] greatly from state-derived and control-derived complexity"

- Functional programming
  - "goes a long way towards avoiding the problems of state-derived complexity"

- Logic programming
  - "offers the tantalizing promise... to escape from complexity problems"
**Essential and accidental complexity**

- **Essential complexity** is inherent in and the essence of “the problem” as perceived by users.
  - Important implication: complexity the user doesn’t know about/care about are not *essential!* (It may be necessary for the sake of efficiency, but for the purposes of this paper, it’s not *essential*.)

- **Accidental complexity** is “all the rest” of complexity.
  - "Complexity with which developers would not have to deal in the ideal world."
Recommended General Approach

- A thought experiment in “the ideal world”
  - Informal specification ➔ formal specification
  - “no relevant ambiguity”

- State management
  - State (data) directly input by users
  - Derived from input

<table>
<thead>
<tr>
<th>Data Essentiality</th>
<th>Data Type</th>
<th>Data Mutability</th>
<th>Classification</th>
</tr>
</thead>
<tbody>
<tr>
<td>Essential</td>
<td>Input</td>
<td>-</td>
<td>Essential State</td>
</tr>
<tr>
<td>Essential</td>
<td>Derived</td>
<td>Immutable</td>
<td>Accidental State</td>
</tr>
<tr>
<td>Essential</td>
<td>Derived</td>
<td>Mutable</td>
<td>Accidental State</td>
</tr>
<tr>
<td>Accidental</td>
<td>Derived</td>
<td>-</td>
<td>Accidental State</td>
</tr>
</tbody>
</table>

Table 1: Data and State
Required Accidental Complexity

- Performance
- Ease of expression (of logic/business rules)
How to deal with complexity

Avoid it

Separate it

<table>
<thead>
<tr>
<th>Complexity</th>
<th>Type</th>
<th>Recommendation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Essential Logic</td>
<td>State</td>
<td>Separate</td>
</tr>
<tr>
<td>Essential Complexity</td>
<td>State / Control</td>
<td>Separate</td>
</tr>
<tr>
<td>Accidental Useful Complexity</td>
<td>State / Control</td>
<td>Separate</td>
</tr>
<tr>
<td>Accidental Useless Complexity</td>
<td>State / Control</td>
<td>Avoid</td>
</tr>
</tbody>
</table>

Table 2: Types of complexity within a system
Functional Relational Programming

- Draws on the work of E.F. Codd
- Relational algebra has 8 operators:
  - Restrict
  - Project
  - Product
  - Union
  - Intersection
  - Difference
  - Join
  - Divide
Constructing a model in FRP

- (Essential) state expressed as relations between entities
- (Essential) logic (business rules) expressed as relational algebraic operations.

Concepts:
- Feeders: turn input into entities with associated relationships
- Observers: generate output in response to changes of relational values.
Conclusion

- Complexity causes more problems than anything else.
- Only by means of a concerted effort to avoid or separate complexity can it be tamed.
- In cases where separation cannot be achieved, strive at all costs to get rid of code.
- “So, what is the way out of the tar pit? What is the silver bullet?”

Simplicity!
Let’s talk about our experiences as developers?

Do you agree or disagree with the premise of this paper?
  Why or why not?

Have you worked in a language/framework which you felt encouraged “simplicity” as a top-level language feature? Tell us about it.

Have you worked in a language/framework where you felt “complexity” was inherent to the language design? Did your system design suffer from complexity? How?