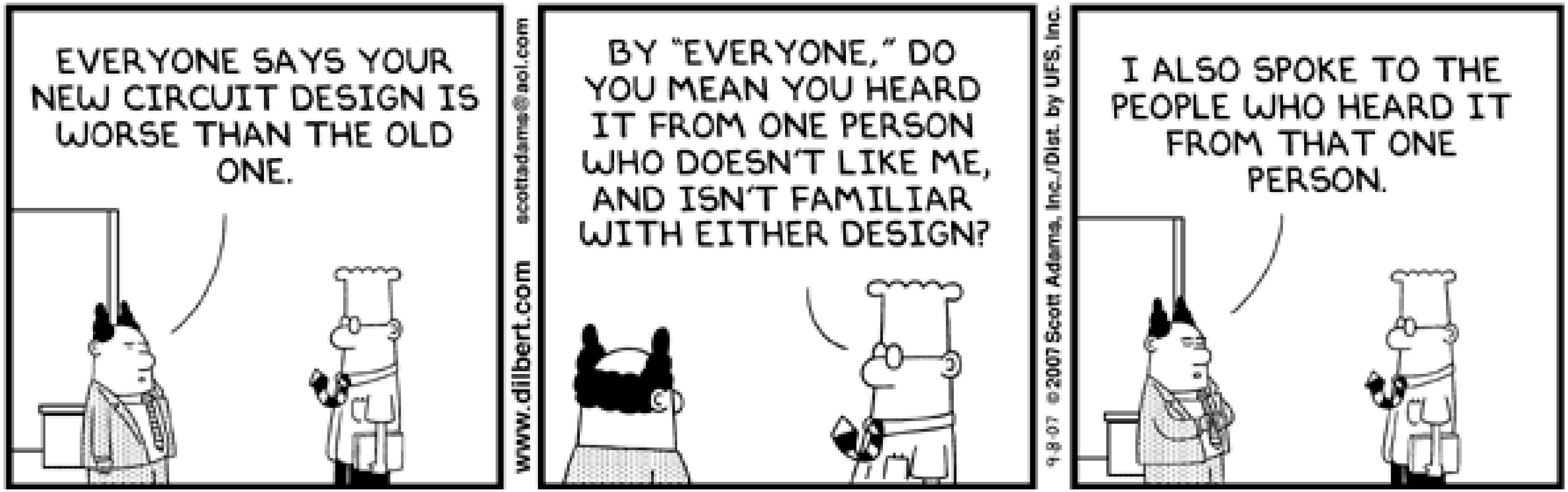
Design Evaluation according to Dilbert


EVERYONE SAYS YOUR NEW CIRCUIT DESIGN IS WORSE THAN THE OLD ONE.
scottadams@aol.com
www.dilbert.com
BY "EVERYONE," DO YOU MEAN YOU HEARD IT FROM ONE PERSON WHO DOESN'T LIKE ME, AND ISN'T FAMILIAR WITH EITHER DESIGN?
9-8-07 ©2007 Scott Adams, Inc./Dist. by UFS, Inc.
I ALSO SPOKE TO THE PEOPLE WHO HEARD IT FROM THAT ONE PERSON.

© Scott Adams, Inc./Dist. by UFS, Inc.

Common Problems

- Remember
  - Systems will change and evolve
  - Not because not done right the first time
    - Though sometimes we don't
  - But because of change in the world and use of the system
- Misleading wisdom from Mathematics
  - Prove a more general theorem to satisfy several related problems
  - Often too general and hence too expensive
  - Push for reuse tends to argue for more generalized components
  - Useful solution: limit the domain but extend beyond single use

Common Problems

- Problem with typical CS view relative to programs
  - Have a specific, unique problem
  - Specify *THE* task to be performed by *THE* program
  - Unique problem solved by one program
- Not a useful or productive view of what needs to be done – often have:
  - Different HW, OS, platforms
  - Different data formats
  - Different data structures, algorithms due to difference resources
  - Different size input data sets, frequency of events
  - Different reliability/performance/security constraints
  - Different standards for different customers (eg, telephony)

Common Problems

- Often want to do the following but cannot:
  - Deliver a subset of functionality
    - But cannot because everything has to be there to work at all
  - Add a capability
    - But cannot without completely rewriting the entire system
  - Remove a capability
    - But cannot with out significantly rewriting the system
  - Want to tailor for specific customers
    - But cannot because the system isn't flexible enough

Common Problems

- How monolith programs/systems come about
  - Excessive information distribution
    - Dependency on whether or not a given feature is present or not
      - ✓Eg, an OS supporting three languages
      - ✓Add a 4th → large amount of code change
      - ✓Difficult to reduce to 2 languages
  - A chain of data transforming components
    - Intermediate data formats
      - ✓Eg, remove one intermediate component → incompatible data formats
      - ✓Eg, data unsorted then gets sorted
  - Components that perform more than one function
    - Common to combine several functions into one unit
      - ✓Eg, runtime checking at call time
  - Loops in "use" relation
    - Often duplicated common functions
    - Problems with program usage: nothing works till everything does
    - Have to worry about dependencies
    - Eg, OS where scheduler depends on the file system

Design

- Intellectual tools to manage complexity
  - Modularization
  - Encapsulation
  - Abstraction
  - Virtual machine
- Modularization
  - Decompose into manageable pieces
  - Basic building blocks
    - Bases for composition into higher level modules
  - General strategy: do one thing well
  - Practical strategy: module per page
    - Easily readable and understandable
- Encapsulation
  - Localizes related data, functions etc
  - Useful strategy: localize things expected to change

Design

- Abstraction
  - Basic form: function/procedure with parameters
  - Information hiding
    - Provide logical interface
      - ✓ Changes infrequently
    - Hide implementation details
      - ✓ Isolates changeable parts
    - Facade pattern
  - Abstract object
    - Abstract interface
    - Encapsulated object
    - Eg, the abstract syntax tree in the parallel compiler example
  - Abstract type
    - Abstract interface
    - Separate implementation
    - Can declare objects of the abstract type
  - Forms of abstraction
    - Data abstraction (ie value) – parameters
    - Type abstraction (ie structure) – abstract data types
    - Procedural abstraction (ie, processing) – parameters, generics

Design

- Virtual machines
  - Don’t think of programs as components that correspond to steps in processing
  - Think of a system as layers of functionality
    - Like an onion
    - Separates levels of concerns
  - Begin with basic machine (eg, OS + programming language)
    - Basic abstractions, basic vocabulary for developing the system
  - Build layers of abstractions
    - Each layer provides a higher level of abstraction
      - ✓ Concepts and constructs appropriate to that level
      - ✓ A higher level language
    - Each layer provides just the right abstractions for an easy implementation of the next layer
  - Can then change implementation details of lower layers without affecting the upper layers
  - Eg, array, vector, binary tree, heap, priority queue

Design

- Fundamental design trade-off
  - Generality
    - Don't need to change
    - Generally larger components
    - Often more complex
  - Flexibility
    - Easy to change
    - Generally, small building blocks
    - Often simpler
- Basic design goals
  - Finding useful and appropriate data structures
  - Finding useful and appropriate algorithms
  - Finding useful and appropriate modularizations, encapsulations and abstractions to provide
    - Ease of maintenance and evolution
    - Simplicity and correctness
    - understandability