Design and Definition of Data Abstractions

Mary Shaw
January 2005

Overview

A central problem in software design and development is managing intellectual complexity of the product

- Techniques for managing complexity:
  - Abstraction
  - Information hiding
  - Separation of concerns, divide and conquer
- Abstraction: suppressing details
  - Good abstraction: suppressing the right details
  - Right details: the ones that don’t matter at the moment
- Today’s papers:
  - Data structures, information hiding, programming language support
  - Formal support for abstract data types

Timeline

Growth in Abstraction Granularity
Understanding data abstraction

Building on recognition of common data structures, languages acquire new constructs for describing those structures

- First round of papers:
  - First formulations of data structures
    - (supplemental) Knuth 68
  - First shift from procedures to information hiding
    - Parnas 72 (*2): organize software around data structures and other "secrets", not around calculations (functions)
  - Experience to validate information hiding
    - Parnas & 85: pre/post specs of modules
    - Wulf/London/Shaw 76: models can be incorporated in languages
  - Early integration of data abstraction and inheritance as objects
    - Booch 86: object-oriented development
  - More mature, reusable objects
    - Booch 90 (supplemental): Booch components

Growth in Specification Power

<table>
<thead>
<tr>
<th>Year</th>
<th>Architectural Chunks</th>
<th>Abstract Data Types</th>
<th>Packages</th>
<th>Inheritance</th>
<th>Formal Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>1960</td>
<td>Concrete Complexity</td>
<td>Formal Specifications</td>
<td></td>
<td></td>
<td>Formal Syntax</td>
</tr>
<tr>
<td>1970</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1980</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1990</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2000</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Software Technology Maturation Points

<table>
<thead>
<tr>
<th>Major Technology Area</th>
<th>Technology Concepts</th>
</tr>
</thead>
<tbody>
<tr>
<td>Software Engineering</td>
<td>Structure Based</td>
</tr>
<tr>
<td></td>
<td>Technology</td>
</tr>
<tr>
<td></td>
<td>Verification</td>
</tr>
<tr>
<td></td>
<td>Metrics</td>
</tr>
<tr>
<td></td>
<td>Formal Specifications</td>
</tr>
<tr>
<td></td>
<td>Abstract Data Types</td>
</tr>
<tr>
<td></td>
<td>Methodology Technology</td>
</tr>
<tr>
<td></td>
<td>Architecture</td>
</tr>
<tr>
<td></td>
<td>Cost Models</td>
</tr>
<tr>
<td></td>
<td>Senior Mod</td>
</tr>
<tr>
<td></td>
<td>Midl</td>
</tr>
</tbody>
</table>

Formal models for data abstraction

Building on early information hiding results, formal models for data abstraction enable a new class of programming languages

- Second round of papers:
  - First formulations of program verification
    - (supplemental) Floyd 67, Hoare 69
  - First formal models for abstract data types
    - Hoare 72: relation between abstraction and implementation
    - Liskov/Zilles 74: ADTs (not assigned, 75: data types as algebras)
  - Integration of formal model with programming language
    - (supplemental) Parnas 72: pre/post specs of modules
    - Wulf/London/Shaw 76: models can be incorporated in languages
  - Second-generation formalism
    - Guttag/Horning/Wing 85: address some of design problems of algebras
    - Coupling of utility and formalism
Transition Points for Abstract Data Types

- Basic research ==> concept formation
  > 1968: formulation of information hiding
- Concept formation ==> development & extension
  > 1973: abstract data type models
- Development & extension ==> internal exploration
  > 1977: incorporation in programming languages
- Internal exploration ==> external exploration
  > 1980: incorporation in other technologies
- External exploration ==> Popularization
  > late 80’s: object models, C++, Java

Finding the Thread of the Argument

- Real World:
  - Practical problem: Does the product help to solve the practical problem?
- Solution to practical problem:
  - Research setting: Does the result help to solve the idealized problem?
- Idealized problem: Does the product solve the idealized problem?
- Solution to idealized problem:
  - Research produce (technique, method, model, system, …)

Approaching a Technical Result

- What kind of problem is it solving?
- What’s the real-world setting or motivation?
- What’s the research model?
  > How is it related to the real world?
- What is the research hypothesis?
- What is the research strategy?
  > What’s the result?
  > How does it solve the problem in the research model?
  > How does it satisfy the research hypothesis?
- Do you believe the result, and why? On what evidence?
- How does the result map to the real-world setting?
- How mature is the idea?

Parnas: On the Criteria (1972)

- Real-world problem: Modular decomposition allows modules to be written independently and systems to be composed from parts. This should lead to shorter development time, product flexibility, and better understandability. However, criteria for partitioning a system into modules are unstated/unclear; but the criteria matter.
- Research problem: KWIC (Keyword in context) index: Accept a set of lines, create the interesting “circular shifts”, and sort the result.
- Experiment: Create two modular decompositions, one conventional and one based on information hiding. Compare properties of the two.
- Result: Information hiding wins on properties examined, but it may lose efficiency if tool support isn’t provided.
- Impact: Example suggests strategy applicable to system design — localize design decisions. Also, indicates need to minimize intrusion of implementation in specifications. Note: this sets the stage for abstract data types (localize representation decisions) and lazy evaluation.
**Parnas: On the Criteria**

Key elements of the result

- **Information hiding**: Instead of organizing modules around processing steps, organize them around design decisions that are likely to change.
- **Module interfaces**: Simplify them -- functional interfaces are easier to maintain than data representations (which are likely to change).
- **Specification content**: Resist temptation to say unnecessary things in specification -- they often constrain implementation.

**Problem type (question):** Means of development (Is there a better way to do module decomposition?)
- **Research result**: Technique/method. “Modularization” is considered to be an assignment of responsibility, not a subprogram. Basis for comparison includes understanding and changing code as well as running it. Note: these are significant shifts in point of view from the norms of the period.
- **Hypothesis**: The strategy used to select software system substructure significantly affects ability to develop modules independently and to change and understand the system.
- **Strategy**: Present an example and comment on it.
- **Validation**: Qualitative observation and persuasive narration. Paper refers to use in class project but doesn’t give details. This sort of argument depends on credibility of “conventional” example as faithfully representing practice.

**Summary of your summaries (Parnas 72)**

- Newman type: radical solution (2), enhanced tool (2), enhanced solution (2), enhanced method, ? (5)
- Question: development method (9), criteria (2), ?
- Result: technique/method (10), specific solution, ?
- Validation: example (8), persuasion (2), evaluation, ?

**Parnas et al: Modular Structure (1985)**

- **Real-world problem**: Software engineering research ideas don’t get used.
- **Research problem**: As nearly as possible like real world -- the A7E flight program, regarded as a good example.
- **Experiment**: Replicate existing system using academic ideas, especially information hiding. This is preliminary report, on the design.
- **Result**: Information hiding was applicable, after refinement.
- **Impact**: Realism of example, detail of specification, analysis of practical issues show relation to practice
A7E Module Structure

- **Hardware-Hiding Module**
  - Extended Computer Module: processors, instr set, concurrency
  - Device Interface Module: characteristics of peripherals

- **Behavior-Hiding Module**
  - Function Driver Module: rules governing values of outputs
  - Shared Services Module: behavior that must be shared

- **Software Decision Module**
  - Application Data Type Module: data reps and code to manipulate them
  - Physical Model Module: physical models and implementations
  - Data Banker Module: how values are produced

- **System Generation Module**: decisions postponed for late binding

Parnas et al: Modular Structure

Key elements of the result

- **Three structures**: Programmers must deal with 3 structures.
  - Module structure: decomposition into work assignments (groups of related programs) and assumptions each team may make about other modules
  - Uses structure: which programs require the presence of others
  - Process structure:

- **Keep secrets**: Localize decisions that may change; access functions provide more flexibility than data formats.

- **Lessons**:
  - Scalability: Many modules => need hierarchy and structured documentation
  - Info can’t always be hidden => “restrict” interfaces that expose info
  - Boundary between specification and implementation is slippery

Parnas et al: Modular Structure

- **Problem type (question)**: Means of development (Is there a better way to do module decomposition?)
- **Research result**: Technique. Improvement on earlier information hiding
- **Hypothesis**: Research results are slow to be adopted because it’s too risky to use unproven techniques; research examples are too unlike real ones; research ideas are not fully worked out.
- **Strategy**: Step up to a real example, close to practical setting.
- **Validation**: demonstrate feasibility of method on real problem
  - Criteria: simple modules; independent implementations; cost of change related to likelihood; major changes should be sets of independent module changes; understandable specs; ability to find relevant modules; sets in hierarchy small enough to reason about.
  - Evidence from the experiment: Authors find method helps satisfy criteria.

Summary of your summaries (Parnas 85)

- **Newman type**: Experience/heuristic (4), Enhanced tool (2), enhanced solution, ? (5)
- **Question**: Development method (6), how to apply method in practical case (3), feasibility (2), ?
- **Result**: Technique (3), tool, specific solution (2.9), qualitative model (.1), ? (5)
- **Validation**: Experience (8), evaluation, ? (3)
Booch: Object-Oriented Development (1986)

- Real-world problem: Functional development doesn’t support information hiding, natural concurrency, response to problem changes.
- Research problem: Sea buoy: Collect and provide weather and navigation data. Also, cruise control: maintain speed of car.
- Experiment:
  > Review history and generalize
  > Work out example in object-oriented style. [For motivation, sketch two designs, one functional and one object-oriented, and compare the two.]
- Result: (a) synthesized model of objects; (b) example of object-oriented design.
- Impact:

Booch: Object-Oriented Development

- Common properties of object concept
  - State: Object has state.
  - Actions: Object is characterized by actions that it suffers and that it requires of other objects
  - Constructors, selectors, iterators
  - Classes: Object is an instance of some (possibly anonymous) class
  - Name: Object is denoted by name
  - Visibility: Object has restricted visibility of and by other objects
  - Specification and implementation: Object may be viewed either by its specification or by its implementation

Booch: Object-Oriented Development

- Problem type (question):
  Means of development (Is there a better way to do module decomposition?)
  Characterization (What, exactly, do we mean by object?)
- Research result: Technique, object-oriented design and development
- Hypothesis:
  > Object-oriented design is superior to functional decomposition,
  > Objects draw on prior results, and draw credibility from those results
- Strategy: Present an example and comment on it. Relate history to see how prior work contributes to object orientation.
- Validation: Qualitative observation and persuasive narration. For synthesis, paper cites other sources re the benefits of object-oriented design.

Hoare: Correctness of Data Reps (1972)

- Real-world problem: Stepwise refinement suggests a method but does not help with determining whether successive steps are correct.
- Research problem: Find a relation between specification and implementation that allows verification that the implementation will be faithful to the specification.
- Experiment: Develop formal model based on an abstraction function that maps between representation and abstraction and an invariant that expresses integrity constraints. Prove that it shows correctness as above
- Result: Method proved formally
- Impact:
  > Set rules for this sort of verification: explicit abstraction function, invariant
  > Established “commutative diagrams” as technique for reasoning about SW
Composable Software Research at Carnegie Mellon

**Hoare: Correctness of Data Reps**

1. The concrete function is correct
2. Abstract precondition & Invariant imply concrete precondition & Invariant
3. Concrete precondition & Invariant imply concrete postcondition & Invariant
4. Therefore the abstract function is correct

**Summary of your summaries (Hoare 72)**

- Newman type: enhanced tool [includes enhanced method] (6), ? (2)
- Question: correctness proof [analysis] (3.5), analysis/analysis method (2), development method (1.5), method
- Result: procedure/technique (5), analytic model, proof, ?
- Validation: example (5), proof (2), analysis and example

... and four others late

**Liskov/Zilles: Programming with ADTs (1974)**

- This paper is like Parnas72 and Booch86
  > “We have a better idea”
  > “Here’s an example”
- You can see the transition from the idea of information hiding to programming language support
  > Global variables were still an issue then
- The paper was influential
Composable Software Research at Carnegie Mellon

Liskov/Zilles: Specs for Data Abstraction (1975)
- Real-world problem: Formal specification and verification is promising, but it’s still too hard.
- Research problem: For a particular class of problems -- data abstractions -- and a matching specification technique, and show that formal reasoning is practical for this class of problems.
- Experiment:
  > State criteria for evaluation
  > Classify approaches: abstract models vs implicit definitions
  > Using stack as example, evaluate several classes of approaches
- Result: Algebraic axioms best satisfy criteria
- Impact:
  > The 1974 precursor paper introduced algebraic axioms for specification
  > This paper helped gain support for algebraic axioms

Liskov/Zilles: Criteria for Specification Methods
- Formality, to allow mathematical reasoning
- Constructibility, so specifications can be written
- Comprehensibility, so a reader can understand them
- Minimality, so no extra information interferes and so what is done isn’t confounded with how it’s done
- Wide range of applicability, to maximize utility
- Extensibility, so change in concept has similar change in spec

Liskov/Zilles: Specs for Data Abstraction
- Problem type (question): Generalization (What are the varieties of X, and how are they related?) and implicitly (Is X always true of Y?)
- Research result: Analytic (formal) theory, that algebras can specify data abstractions
- Hypothesis: Algebraic axioms best satisfy criteria.
- Strategy: Compare specifications of stack.
- Validation: Careful argument based on stacks
- The other story …
- Algebraic axioms can capture essential functionality of data abstractions
  > the big result of the 1974 paper
  > established by developing the algebras for some examples

Wulf & al: Construction and Verification (1976)
- Real-world problem: Poor language support for constructing understandable, maintainable programs whose specifications can be shown to match their implementations.
- Research problem: Design a programming language that both captures abstractions as integral part of the program and supports verification.
- Experiment:
  > Design programming language with data abstraction as central element
  > Address Hoare-style verification as integral part of language
  > Using stack as example, show that the language restrictions match the verification requirements
- Result: Verification support can be integrated in language design
- Impact:
  > Helped show deficiency of Algol-ish nested scope, interaction with verification
  > One of the major precursors of O-O languages
Wulf & al: Construction and Verification

- Problem type (question): Means of development (How do I accomplish X?)
- Research result: Technique. Hoare-style specification and verification of abstractions
- Hypothesis: Synergy between language design can specification.
- Strategy: Design language based on requirement to capture abstraction information for verification and future maintenance, dry-run it on “stack”.
- Validation: Full verification of example (stack). Careful argument about language and about interaction among method, language, verification
- The other story …
  > Set up counterpoint to algebraic axioms for modeling data abstraction

Guttag & al: Larch (1985)

- Real-world problem: This is a second generation specification language for data abstractions.
- Research problem: Address shortcomings of first generation: monolithic nature of a complete algebra, mismatch between formalism and language mechanism.
- Experiment:
  > Develop a theory in which specification fragments exist independently
  > Show how these can be combined to obtain full specifications
  > Separate pure functionality from specs related to language mechanisms
- Result: This partitioning is possible and promising
- Impact:
  > Consolidated prior work on specifications for data abstractions

Guttag & al: Larch

- Problem type (question): Means of development (What is a better way to accomplish X?)
- Research result: Technique for composing specification fragments. Various container abstractions, algebraic axioms for abstractions
- Hypothesis: Specification-writing can be made easier by supporting composable fragments.
- Strategy: Design new language and theory for specification of data abstraction, in which specification fragments are the primitive elements
- Validation: Demonstration of theory on a set of examples

Maturity: Progressive Codification Cycle

- Ad hoc solutions
- Improved practice
- Models & theories
- Codification
- Folklore
- New problems
### Current interests

<table>
<thead>
<tr>
<th>Who</th>
<th>Topic 1</th>
<th>Topic 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Marwan Abi-Antoun</td>
<td>Adoption of SW arch desc languages</td>
<td>Tools for code gen and synch of spec &amp; code</td>
</tr>
<tr>
<td>Kathryn Bergmann</td>
<td>Safety cases for SW</td>
<td>Fault analysis techniques</td>
</tr>
<tr>
<td>Kevin Bierhoff</td>
<td>Role and handling of state in SE</td>
<td>Requirements engineering</td>
</tr>
<tr>
<td>Tudor Dumitra</td>
<td>SE for real time</td>
<td>SW metrics</td>
</tr>
<tr>
<td>Ihe Golden</td>
<td>SE education</td>
<td>SE economics</td>
</tr>
<tr>
<td>Greg Hartman</td>
<td>Reusing knowledge &amp; str that differs from subr blys</td>
<td></td>
</tr>
<tr>
<td>Thomas LaToza</td>
<td>Empirical SE</td>
<td>SW maintenance and evolution</td>
</tr>
<tr>
<td>Larry Maccherone</td>
<td>SW process</td>
<td>Metrics</td>
</tr>
<tr>
<td>Theresa Monino</td>
<td>Real-time systems</td>
<td>Reliability/dependability for embedded systems</td>
</tr>
<tr>
<td>Jennifer Morris</td>
<td>SW safety</td>
<td>SW dependability</td>
</tr>
<tr>
<td>Justin Ray</td>
<td>Reliability: SW fault tolerance</td>
<td>SW specification and modeling</td>
</tr>
<tr>
<td>Christopher Scaffidi</td>
<td>End user program/g envs</td>
<td>Spreadsheets</td>
</tr>
</tbody>
</table>