Abstraction: The Hardcore of Software Engineering

Dewayne E Perry
Motorola Regents Chair in Software Engineering
Electrical and Computer Engineering
The University of Texas at Austin

What is Software Engineering (SE)?

• SE is concerned with building and evolving software systems that have a practical effect in the world
  - Fundamentally, SE is a set of problem solving skills, methods, techniques and technology applied in a variety of domains
  - Programming is just one of these basic problem solving skills
Basic job of a Software Engineer

• Discover, create and build/evolve abstractions and behaviors
• Effectively evaluate and decide among alternative abstractions/solutions

To do this we use

- Theories or models that we use or create
  • From standard well understood domains
  • For new domains we do not understand well

Basic job of a Software Engineer

- Experience
  • Feedback, either directly or from users
  • Experimentation, engineering or scientific

- Process
  • Problems solving methods and techniques
  • Technologies appropriate to the product and to the methods and techniques
  • Organizational and cultural structures -
Software Engineering Research

• Discover, create and build new abstractions to help software engineers
• Evaluate the effectiveness and utility of these abstractions for engineering software systems
• Discover, create and evaluate effective measures for comparing and evaluating abstractions, behaviors and solutions

Abstractions

• SE, as any engineering discipline, is a discipline of design
  - Brooks' goal: conceptual integrity of design
  - Abstractions are our fundamental intellectual tool for design
    • Simplification
    • Generalization
    • Codification
    • Satisficing
Abstractions

• For the SEs, abstractions
  - Are the primary means of managing complexity
  - Provide basic domain specific concepts

• For SE Researcher, abstractions
  - Provide the primary means to help SEs to think about how to architect, design, build and evolve software systems
  - Remove accidental underbrush and simplify SW development

Whither Structured Programming?

• Basic set of abstractions for programming
  - Basic set of programming actions
    • Complete
    • Orthogonal
  - Well defined semantics, composition and proof rules
  - Disciplined control flow
  - Static structure reflects dynamic structure
  - Basis for additional needed abstractions
    • Eg, separation of normal and abnormal (exceptions)
Some Useful Abstractions

- Problem versus solution space
- Virtual machines
- Product families
- Essential versus accidental characteristics
- Components and connectors
- Computations versus behaviors

Problem vs Solution Space

- SE’s often too focused on solutions
- Emphasis on problem domain
  - What rather than how
  - Problem discourse
  - Domain abstractions
- In the world rather than in the machine
- Jackson: shape of the solution should reflect the shape of the problem
Virtual Machines

- Interfaces as little languages
- Coherent and related set of abstractions
- Layering
  - with increasingly rich concepts
  - with increasingly higher level languages
- Domain specific machine and language
- Encapsulated implementations - changeable

Product Families

- Parnas: Planning for change
- Commonality vs variability
- System as a sequence of family members
  - Some parts are invariant
  - Some change
- Exploit commonality to reduce maintenance
- Basis for things to come
  - Product line architecture
Essential versus Accidental

• Brooks: Critical distinction
• Essential - need to be managed
  - Basic facts of life
• Accidental - need to be remedied - Eg,
  - Inadequate abstractions, expressions
  - Inadequate modes of expression
  - Inadequate support and resources
  - Inadequate knowledge

Components and Connectors

• Perry/Wolf: Logical rather physical distinction
• Connectors represent interactions
  - Communication
  - Coordination
  - Mediation
• Components represent computations and behaviors to be composed
• Connectors relate and regulate the behavior of the composed components
• Possible: composable non-functional properties
Computations vs Behaviors

- Turski: critical distinction
- Computations
  - Bounded, neat problems
  - Underlying theory available
  - Admit of clean, theoretically nice solutions
    - Eg, Misra’s composition of concurrent programs
- Behaviors
  - Unbounded, messy problems
  - Little theory available - often make it up as we go
  - Harder to formally describe and reason about

Evaluation

- Few agreed on measures
  - Still at level of naïve art critic
    - I know what I like
    - I’ll know it when I see it
  - Tend to rely on experience (anecdotal)
  - Poor construct validity
    - Eg, cohesion
  - Little agreement on effectiveness and reliability of metrics
Creating Effective Evaluations

- Tends to be *I have a dream* paradigm
- Too little theory to go on
- Experimental side of SE (and CS) very immature
  - Lack of understanding of basic experimental issues: design, validity, analysis
  - Lack of standard designs and measures
- Virtually no replicated experiments
- A long way to go yet

Conclusions

- Abstractions and evaluations fundamental to software engineering
- Abstractions, evaluations and the creation of effective measures fundamental to software engineering research.
- Doing well in the abstractions department
- Doing poorly in the evaluations department
- Much worse in creating effective measures