Some Useful Engineering Strategies

Seth Teller
RSS II
16 November 2005

My Goals Today

• Discuss engineering from an intellectual and practical standpoint

• Introduce a "toolkit" of ideas and techniques that you can use in your own engineering endeavors

• Solicit your own ideas about useful engineering practices
Caveat Auscultator (Listener beware)

• Some of this material will be new to you; some will be familiar
  – It doesn’t hurt to hear things twice.

• Some things you will probably agree with; some things you probably won't
  – But surely you’re used to this by now.

What is Engineering?

• Engineering (n.)

  (Merriam-Webster Online)

  – a : the application of science and mathematics by which the properties of matter and the sources of energy in nature are made useful to people

  – b : the design and manufacture of complex products

• Do science, math, usefulness, and complexity “capture” engineering?
What is Engineering?

• Engineering (n.)

The process of specifying, designing, implementing, and validating physical artifacts with a desired set of properties

(With “properties” construed broadly to mean material attributes, rigid and articulated DOFS, appearance, behavior, ...)

Process View

• Engineering is a Means ...
  – Specifying: describing what to make
  – Designing: describing how to make it
  – Implementing: realizing actual artifact
  – Validating: convincing yourself (and others) that artifact works as specified

• ... to an End
  – Namely: artifact with desired properties
Human View

- Engineers are people who:
  - Conceive of and execute ways to optimize an underspecified tradeoff between possibly conflicting goals

- Subject to severe constraints:
  - Natural: Laws of physics, i.e., reality
  - Cultural: Legal system, mores, ethics ...

Conception & Execution

- Conception:
  - A mental model of artifact & constraints

- Execution:
  - Putting the mental model into practice
  - Observing whether it predicts behavior under real-world conditions
Essence of Engineering ...

- ... Process is the (typically iterative)
  - Formation of a mental model
  - Implementation of a prototype artifact
  - Observation of its behavior, leading to:
    - Revision of designer’s operative mental model
    - Revision of operative design or implementation
    - (Or both)

- ... Until desired behavior is achieved

Visualization & Inspection

- Idea: graphical analogue of printf
  - Visually expose artifact’s internal state

- Distinct from “algorithm animation”
  - Rendering output of batch computation
Consequences

• If it “looks wrong” to you, two possibilities:
  – A) Artifact state really *is* wrong, in which case:
    • Artifact has deviated from your mental model
    • You can find first place of deviation, and fix it
  – B) Artifact state is *correct*, in which case:
    • Your mental model made it “look wrong” to you
    • Thus your mental model must be revised!

A C M K C E a b d a b C d E

• If it “looks wrong,” it’s an opportunity to
  – Improve the system’s behavior, or
  – Learn something, i.e., improve mental model!

... And if it looks correct?

• Is it correct?

• Sure, it often is. But that doesn’t mean that it always is!

• Can boil these ideas down to an aphorism:
  – “Don’t sweep anomalies under the rug.”
Self-Checking Code

• Idea: make machine work for you
• For each algorithm/module, write a “checker” that inspects output for the properties that it should have
• Convex hull example:
  – Traverse output vertices in order; check orientation of each triplet
• Same idea applies to input
  – Postconditions (A) == Preconditions (B)

Distinction: JavaDocs

• JavaDocs comprise:
  – Declarations
  – Comments

• But teammates’ agreement to make the code implement the *intent* stated in the comments essentially amounts to a *social contract*
Teammate-Checking Code

• Twist: for each module you write, ask a teamwork to write checker
• Multiple benefits:
  – Validates your solution (as before)
  – Decreases chance that checker succeeds due to an invalid assumption (why?)
  – Facilitates agreement of your mental model with your teammate’s model
  – Exploits a natural human characteristic: competitiveness (s/he acts as adversary)

Adversary

• Someone/something that tries to
  – Find holes in your correctness proof (e.g. as A did for R & S of RSA security)
  – Produce inputs that break your code (e.g., by violating your assumptions)
  – Produce conditions that break system (more than just program’s formal input)

• Adversary can be a person, program, or even a contrived environment
Adversary’s Strategies

- Generate challenging *inputs* ...
  - Exhaustively
  - Randomly
  - Qualitatively
  - Deviously (e.g., provoke your teammate to do it)
- ... and environmental *conditions*:
  - Missing or mis-wired connectors
  - Misbehaving sensors
  - Depressed all-stop buttons
  - Undefined environment variables
  - Misconfigured networks, remote hosts, etc.

Benefiting from Adversary

- Implement a “state capture” facility
  - Ensure that it is very easy to invoke
  - ... And that state can be reconstituted
- This makes misbehavior repeatable
- Gives rise to “defensive coding”

- Aphorism: “Chance favors the prepared program”
Self-Reporting Code

• Useful when a subroutine might legitimately succeed or fail
  – Example: path planning among obstacles

• Notion of a “witness” from CS theory community: consists of either a
  – Checkably correct output solution, or an
  – Input subset that “proves” infeasibility of the specific input instance provided

Digression: Line Equations

• Points represented as \( p = (x, y, 1) \)
• Lines represented as \( L = (A, B, C) \)
  – Defines “positive halfspace” \( L \cdot P > 0 \)
  – Defines “negative halfspace” \( L \cdot P < 0 \)

Example:

Line equation \( x + y = 1 \)
\( x + y - 1 = 0 \)
\( (1, 1, -1) \cdot (x, y, 1) = 0 \)

Thus \( L = (1, 1, -1) \)
Linear Separability

- Given point sets \( \{A_i\}, \{B_i\}, i \in [1..N] \)
- Identify line \( L \) s.t. all \( A_i \) lie above \( L \), all \( B_i \) lie below \( L \), or show no such \( L \) exists

Caution

- Make sure your checking, reporting, witness etc. code has no side effects that enable correct algorithm function

- Otherwise, when you remove or suppress self-test, bugs will emerge

- Examples?
Self-Test Summary

• Pit code against itself.

• Aphorism: “Make function prove itself before you trust it.”

Test Harness

• Battery of test cases applied to a system to validate its responses

• We’ve seen these in “software only” systems, with “softcopy only” inputs

• But what about robotics? How can we validate sensors and actuators using only software? ... We can’t!
Robotics is Different!

- Robots are subject to “hard state” fundamentally not under s/w control
- Consider e.g. all-stop button sense question that arose last week
- Or, even harder: sensors. How to force them to behave as you want?
- Actuators have same problem
- Real world is the only way to enforce absolute consistency of env’t, state

Robotics Test Harness

- Place robot in a known environment ... so actions have known outcomes
- For concreteness, imagine harness for:
  - Odometry
  - Motor drivers
  - Bump sensors
  - Visual servoing
  - Arm driver
  - Brick storage
Self-Test Summary (2)

• Pit machine against environment.

• Aphorism (Feynman):
  “You can’t fool Mother Nature.”

General Comments

• You’ve heard it all before
  – “Think before you code”

• My variation on this:
  – “Validate as you design and implement”

• Tangible benefits in rapidity of prototyping & achievable complexity while retaining confidence in correctness
Your Ideas for Next RSS II

• How to promote rapid prototyping, validation?

• More or better tools?

• In-class or in-lab exercises?

Your Ideas for Next RSS II

• How to prompt students to address integration, end-to-end issues earlier?
  – Example: systems group; great idea
  – Move first integration even earlier?
    • I.e., as soon as message formats published?
Summary

• Discussed engineering as an endeavor

• Described several tools/methods for validation and rapid prototyping

• Argued that “robotics is different”