Whatever Works

Primitive Technology: Tiled Roof Hut
Preliminaries
What Is Construction?

A definition:

Software construction is the creation, assembly, or modification of executable programs typically via modification of the source code.
Abstraction Hierarchy Of A System

Not the only formulation of such a hierarchy!
Architecture Vs. Construction
Coding As Problem Solving

- **Software engineering** is problem solving
  - Hence, the foundational nature of **problem definition**
- **Writing** or modifying code
  - Is also a form of problem solving
    - We hope **smaller** problems.

  *Pro tip: Always know the problem you're solving!*
The Big Four (Plus Two)

- Four primary activities
  1. Creating
     - We need **functionality**
  2. Debugging
     - We need **correctness**
  3. Refactoring (last week!)
     - We need **comprehensibility**
  4. Optimising
     - We need **efficiency** (wrt to some resource)
- Plus two
  - Testing & Reading
Testing Is Everywhere

- All **primary** activities involve testing
  - Whether **formal** or **informal**
  - E.g., **Creation** (whether test first or not)

<table>
<thead>
<tr>
<th>External Qualities</th>
<th>Functional</th>
<th>Non-Functional</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Correctness</td>
<td>Usability</td>
</tr>
<tr>
<td></td>
<td>Accuracy</td>
<td>Efficiency</td>
</tr>
<tr>
<td></td>
<td>Adaptability</td>
<td>Reliability</td>
</tr>
<tr>
<td>For Modification</td>
<td>Maintainability</td>
<td>Integrity</td>
</tr>
<tr>
<td></td>
<td>Flexibility</td>
<td>Robustness</td>
</tr>
<tr>
<td></td>
<td>Portability</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Reusability</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Testability</td>
<td></td>
</tr>
<tr>
<td>Internal Qualities</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>For Comprehension</td>
<td>Readability</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Understandability</td>
</tr>
</tbody>
</table>

3.7
Reading Is Everywhere

- Reading **code** is a key skill
  - **Other** people's code
    - that you are **using**
    - that you are **modifying**
  - **Your** own code!
    - whether **using** or **modifying**
- "Reading" (understanding) **systems** is a key skill
  - **Grasping** the problem, requirements, architecture
  - **Relating** code to those
10 lines of code = 10 issues.

500 lines of code = "looks fine."

Code reviews.

10:58 AM - Nov 5, 2013
Creation
Mindset
What Is (Code) Creation?

Code creation (or coding) is the addition of new functionality by the generation of new code and units of code.

- Key activity!
  - Often directly measured
    - Productivity as LOC/day
    - (Though, deleting code might be better!)
  
- Does not have to be ex nihilo
  - Cut-paste-modify reuse counts
  - Reuse counts!
Prerequisites

- Remember the prerequisites!
  - What's your overall problem definition
    - What part are you tackling
  - What are the pertinent requirements
  - Understand the architecture
    - And how your current code fits in
  - Know the local standards
    - E.g., code formatting style
A good architecture should:
1. help you determine \textit{where} your code should go
2. constrain how functionality is \textit{divvyed up}
3. determine your \textit{communication} channels
4. give you a sense of things \textit{fitting together}
   - that is \textit{shared}

- Code-Architecture \textit{conflicts} indicate
  - A \textbf{problem} with one or the other
  - A \textbf{limit}
Technology Choices

- Different technology choices affect code
  - Language
    - Mono- vs. poly-glott
    - Typing regime
    - "Batteries"
    - Mindshare
    - etc.
  - Frameworks, libraries, OSs, etc.
Tools

- Development environments
  - Editors
  - IDEs
  - Source control
- Dynamic tools
  - Compilers
  - Debuggers & Testers
  - Profilers
- Static tools
  - Lint, syntax, style checkers etc.
Awarenesses

- Situational Awareness
  - Your perception of the current pertinent factors for decision making
  - **Good** situational awareness
    - Tracks all pertinent factors
    - to the right degree
    - in a manner to drive appropriate reactions
    - at low cost
  - Drives tactics and thus action

- Understanding
  - Your systematic grasp of all factors related to decision making
  - Results from sensemaking
  - More cognitive (indirectly drives action)
Getting In The Zone

- Given a problem, our solving can be
  - **focused**
    - we have **tight** situational awareness
    - the "situation" is the problem and solution space
    - we **react** rather than **act**
  - **unfocused**
    - our awareness is **scattered**
      - distracted/multitasking
      - disengaged
      - confused

  *The "zone" is a much higher productivity state*
Admin

- Record-keeping is extremely helpful
  - And sometimes required, e.g., billable hours
- Tracking helps! (a lot can be automated)
  - Time
  - Effort (and sense of effort)
  - What was done (and why, by whom)
  - Mood
  - Discussions and decisions

Some is better than none; enough is better still; there is too much
Programmer Credos

Three Virtues

According to Larry Wall(1), the original author of the Perl programming language, there are three great virtues of a programmer: Laziness, Impatience and Hubris

1. **Laziness**: The quality that makes you go to great effort to reduce overall energy expenditure. It makes you write labor-saving programs that other people will find useful and document what you wrote so you don’t have to answer so many questions about it.

2. **Impatience**: The anger you feel when the computer is being lazy. This makes you write programs that don’t just react to your needs, but actually anticipate them. Or at least pretend to.

3. **Hubris**: The quality that makes you write (and maintain) programs that other people won’t want to say bad things about.

Debugging

—Grace Hopper's Bug Report
Defects Again

Recall:

A **defect** in a software system is a **quality level** (for some quality) that is **not acceptable**.

- We focus on **functional defects**
  - **Correctness** primarily
  - Though **robustness** is also key
    - More **stability**, i.e., doesn't **crash**
What Is Debugging?

*Debugging is the modification of code to remove (or mitigate) correctness defects.*

- We don't count *missing* functionality defects
- Debugging starts *after* a purported *detection*
  - Input: a result of testing or a bug report
- We allow *mitigation*
  - Not *properly* fixing the bug
  - But enough so it's *less damaging*
  - Must still involve *code modification*
    - Other workarounds don't count!
Functional Landscape (Enhanced)
Debug Cycle

- Input: An indication of a defect
  - Stabilise — Make reliably repeatable
  - Isolate (or localise) — To the smallest unit
  - Explain — What's wrong with the code
  - Repair — Replace the broken code
  - Test — Verify the fix

- Check for
  - Regressions
  - Masked bugs
  - Nearby bugs
Indication

An **indication** of a defect is a **tangible record** of a behaviour contrary to the (explicit or implicit) functional specification in a **designated situation**.

- Key parts:
  - Situation
    - Preferably, sufficiently described for replication
  - Expected Behaviour
  - Witnessed Behaviour
    - Typically with some explanation why it's wrong
- Often **very vague**
Indication?

- Often very vague
  - Program crashed sometime during this test
  - Open Office on Ubuntu won't print
    - Actually, only on Tuesdays!

*From John Regehr, "Classic Bug Reports"*
Stabilise

- Bugs are often very situation dependent
  - Precise input + state
    - OS, hardware
    - Sequence of actions
    - Length of operating
- A stabilised bug
  - is reliably repeatable
  - preferably with minimal sufficient conditions
Isolate (Localise)

- Bugs are often very **local**
  - **Single** LOC
  - **Single** routine
  - **Particular** class
- They don’t **have** to be!
  - Communication points are **vulnerable**
- A defect is **isolated** if
  - you have identified the **minimum subsystem necessary** to exhibit the defect
  - for an **trigger input** and **situation**
Explain & Repair

- **Explaining** the bug
  - You can **articulate the mechanism** of the bug
    - Your **bug theory**
  - You can **manipulate** the bug
    - **Trigger** or **avoid** it
    - Produce **variants**
    - **Predict** its behaviour
    - **Fix it**

- **Repairing** the bug
  - Modifying the code so the defect is eliminated
  - May not be possible!
Test

- Post fix
  - You need to verify
    - Your theory
    - Your *execution* of the fix
  - You need to guard against
    - Unintended consequences!
- "New" bugs arise
  - Bugs in the fix
    - The fix is incomplete
    - The fix triggers a regression
  - Masked bugs
Check Nearby

- Bugs come in families
  - Similar mistakes
    - You did it once, you might have done it twice
    - Persistent misunderstanding with multiple manifestations
  - Clustered mistakes
    - Some bugs hidden
      - A crash conceals much
    - Some routines are broken
      - Lots of debt!
- A bug is a predictor of more bugs!
Bug Reports To WONTFIX

- Sometimes, a fix *isn't going to happen*
  - The bug is too *small*
    - Or *insignificant*
    - Or *ambiguous*
  - The bug is too *big*
    - It would *change too much* behavior
      - Which some people *rely* on
      - Other debt increases the *risk*
  - The but is too *hard*
Optimising
Resources

- Size
  - Running space
    - At all levels
  - Persistence and transmission
  - Code
- Time
  - Response vs. throughput
    - Instant vs. Overall
  - Wall/CPU Time/Instructions
What Is Optimisation?

Optimisation is a transformation of code into sufficiently functionally equivalent code that has "better" resource utilisation.

- "Sufficiently functionally equivalent"
  - User observable/desirable behaviour is preserved
  - Up to some point
  - It may be specialised to a certain particular scenario

- Resource utilisation
  - Type and Pattern must be specified
Where?
Tuning Trade-Offs

- **Time** for **Space** (and the reverse)
- **Performance** for **Readability** (and the reverse)
  - And other comprehension qualities
  - Not *always* a trade off for algorithmic improvements
    - Or fat removal
- **Performance** for **Correctness**
- **Performance** for **Cost**
Tuning Alternatives

- Buy More and Faster **Hardware**
- Use the **Optimiser**
- **Better** compilers/frameworks/libraries
- **Input** manipulation
  - "It's slow when I do this" "Don't do that!"
Tuning Safety

- Tuning is **risky**
  - Even optimisation can be risky!
- It’s easy to make code **fast**
  - By making it **incorrect**
- It’s easy to **modify the code** a lot
  - And **not improve** performance much
  - Or **make worse**
Tuning As (Performance) Debugging

- Input: An **indication** of a performance defect
  - **Stabilise** — Make **reliably repeatable**
  - **Isolate** (or localise) — To the **smallest unit**
    - USE A PROFILER! TEST CASES ARE CRITICAL
    - Explain — **What's wrong** with the code
  - **Repair** — **Replace** the "slow" code
  - **Test** — **Verify** the improvements
- Check for
  - **Sufficiency** (Was that enough?)
  - **Trade-offs** (e.g., space consumption)
  - (Correctness) **Bugs**
Boehm’s Evidence

Following slides derived from Making Software, Chapter 10
Reading Papers

- These papers are **challenging**!
  - Even massaged a bit for the practitioner
  - Lots of technical jargon and techniques
  - Summarizing a vast literature
  - Challenging stats and presentations
- Don't panic!
  - These are read and reread
  - First reading should focus on **key points**
  - Later readings should focus on **the evidence**
The Role Of Architecture

- Key challenge (Boehm, Making Software, Chp 10)
  - How much should you **invest** in architecture?
    - Analogy to building
      - We pay the architect **10% of the cost of a building**
      - We should spend **10% of the project budget** on architecture
    - Is this **enough**?
    - How would we **know**?

*Note: statistically general conclusions may not apply in your case!*
Bohem’s Research Questions:

- "By how much should you expect the cost of
  ■ making changes or fixing defects
  ■ to increase as a function of
  ■ project time or product size?"

- "How much should you
  ■ invest in early architecting and evidence-based project reviews
  ■ before proceeding into product development?"
Economies

- Commodity manufacturing exhibits **economies of scale**
  - Making 1 chip may be much more expensive than 1000
  - The **unit cost** diminishes as the **number of units** increases
- Software end-unit costs are (can be) **zero**
  - Cheap to make a copy!
    - Installation & configuration may not be
  - So focus on **lines of code** or **bits of functionality**
- Software exhibits **diseconomies of scale**
  - The **unit cost** rises as the **number of units** increases
    - Potentially exponential! *Pgs 166-167 esp. useful*
Cost Ratios

- What’s the ratio of cost to fix early vs. late?
  - 1970s
    - 1 in requirements to ≈100 post delivery
  - 1981
    - 1:100 for large code bases
      - But 1:5 for small (2,000-5,000 LOC)
  - 1996 survey
    - (70-125):1
  - 2000s
    - Some evidence of reduction from 1:100 to 1:20
    - Or even flat (for 1 million line code base)
Cost Ratios (For Coursework!)

- What’s the ratio of cost to fix early vs. late?
  - Think of your coursework!
  - **Before** deployment (aka submission)
    - Small fixes are cheap
    - Esp. in the currency of the course, i.e., points
  - **After** deployment (aka submission)
    - Even "small" fixes are expensive (or impossible)

- Coursework builds over the semester!
  - So problems can build up
Two Strategies

- **Avoid** late bugs
- Make fixing late bugs **cheaper**
- Failure to do both **kills** the project
  - Failure to do one **may** be mitigated by the other
- **All** our activities should aim for this
  - Thus we want architectures that
    - preclude some bugs
    - confine the effects of all bugs
"20% of the defects account for 80% of the costs"
  ■ "these 20% are...due to inadequate architecture..."

Two sorts of costs
  ■ Direct costs
  ■ Opportunity costs

Two example big failures
  ■ the OS architecture didn't support fail-over when processors failed
    ○ lacked a key functionality
  ■ assuming all messages are short
    ○ thus borking on 1 million character messages
Trade Offs

- More up front arch
  - Costs!
  - Runs risk of overruns
    - Since less time for everything else
- Potentially, getting arch right
  - Reduces rework time

*Note, changing requirements can kill getting it right*
Sweet Spots

Effect of Size on Sweet Spots

- Sweet spot
- Percentage of project schedule devoted to initial architectural and risk resolution
- Added schedule directed to rework
- Total % added schedule

Sweet Spot Drivers:
- Rapid Change: Leftward
- High Assurance: Rightward

Percentage of time for architecture and risk resolution

100KLOC

Sweet Spots

10KLOC

Total

Architecting

Beware

10,000 thousands of equivalent source lines of code (KLOC)
"...the greater the project's size, criticality, and stability, the greater the need for validated architecture feasibility evidence.

"very very small low-criticality projects with high volatility, the architecting efforts make little difference"

Note: There are other cost drivers; check the assumptions!
Creation
Classes
Margaret Hamilton

Next to the Apollo project navigation software code
Abstract Data Types (1)

- A **datatype** is
  - a **set of values**
  - with **associated** operations
- An **abstract** datatype is
  - a **datatype**
  - characterised **entirely by the operations**
    - independent of implementation details
- A **concrete** datatype (or **implementation** of an ADT) is
  - a **representation** of a set of values
  - with **particular** implementations of the operations
ADT Example: Lists

LIST
(sequence of values)
append, memberOf,
at, insert...

Pointer Based
Linked List

Array Based
List
ADT Example: Integers

- Integers
  - Abstract
    - Values: All (or finite subsets of) integers
    - Operations: +, -, *, /, <, =, >
  - Concrete (representation)
    - 32 bit 2s-complement or
    - 64 bit 2s-complement or
    - arrays or
    - floats (Javascript) or
ADT Example: Databases

- CRUD
  - **Create-Read-Update-Delete**
    - Very abstract!
- Some Mappings (from Wikipedia):

<table>
<thead>
<tr>
<th>Operation</th>
<th>SQL</th>
<th>HTTP</th>
<th>DDS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Create</td>
<td>INSERT</td>
<td>PUT / POST</td>
<td>write</td>
</tr>
<tr>
<td>Read (Retrieve)</td>
<td>SELECT</td>
<td>GET</td>
<td>read / take</td>
</tr>
<tr>
<td>Update (Modify)</td>
<td>UPDATE</td>
<td>PUT / PATCH</td>
<td>write</td>
</tr>
<tr>
<td>Delete (Destroy)</td>
<td>DELETE</td>
<td>DELETE</td>
<td>dispose</td>
</tr>
</tbody>
</table>
Why ADTs?

- Separate **interface** and **implementation**
  - The **client** only needs to think about the **interface**
    - Up to a point!
    - Abstractions **leak**
  - Simpler and clearer
  - Focuses on the **semantics** of the type
- Tells you the **critical functionality**
Units Of Creation

A **code unit** is a syntactically complete(able) chunk of a program that **enacts** some **behavior**.

- Line of Code
- Block or control structure
- **Routines**/procedures/functions/methods/operations
- **Classes**/modules
- Packages/modules

*Individual routines and classes are what get unit tested*