Families of Software Systems

Notkin: 3 of 3 lectures on change

Families of software systems

- What are the benefits of considering families of systems during software design?
- Why is the design you get for a system this way different from those achieved through other approaches?
- What is layering? What is the *uses* relation

Families of systems

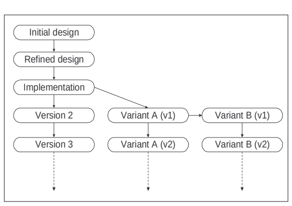
- It is quite common for there to be many related versions of a software system
  - True even omitting new versions intended “just” for adding features and fixing bugs
- Parnas makes the analogy to families of hardware systems
  - The IBM 360 family is a great example
    - One instruction set, many many implementations
    - One goal was to meet distinct price-performance needs; another was to handle upgrading

Software examples

- Windows NT, Windows 98, Windows 2000
- Local language versions of desktop packages
- Federal vs. state versions of TurboTax
- Different Unix versions
- A bazillion others

Common approach ...

- ... to developing members in a family of systems
- Design and build the first member
- Modify the first member to make the next member
  - And so on



Basic problem

- The basic problem is that this is reactive design
  - The design one gets for a later member of the family is based not on the best design, but on the history that led to it
  - Ontogeny recapitulates Phylogeny
- Parnas argues that there are significant benefits to anticipating the family in advance

Premise

- There are collections of software systems in which one benefits enormously from understanding their commonalities before focusing on their differences
  - These are *program families*
- One should explicitly design with this idea in mind
  - Then the design will explicitly account for the family, leading to better designs

Note

- In neither approach will the design for a later member of the family be the same as if it were designed on its own
  - In the evolutionary approach, this is because it's derived from earlier designs
  - In the family approach, this is because it's designed as part of a family
- This is a tradeoff that is likely to have benefits in the long-term

Stepwise refinement: a limited kind of family approach

- This is the top-down style of program design
  - Take your high-level task, decompose it into parts, assuming you can implement each part
  - Then successive apply this technique to each of those parts, until you have a complete program
- Each of the parts that is not fully implemented represents a kind of family

Example: sorting

```
while ∃ x,y ∈ [1..N] | A[x] < A[y] do
   swap(A[x],A[y])
end
```

- You can think of this as capturing the entire family of exchange sorts
  - Bubble sort, insertion sort, shell sort, quicksort, etc.
  - The decisions about the order of indices to compare distinguishes the family members

Stepwise refinement

- Stepwise refinement can reasonably be viewed as a design technique for representing families of systems
- But the top-down nature of the approach yields serious practical limitations
- In particular, one has to replay decisions from whatever node in the design tree is chosen, all the way down
  - In small examples, small deal; in big systems, big deal; in really big systems, really big deal

Parnas' explicit approach

- Anticipate family members and build information hiding modules that support the implementation of those family members
- Doesn't require replay of all decisions from top to bottom
  - Mix-and-match implementations while keeping interfaces stable

Layering

- A focus on information hiding modules isn't enough
- Parnas' also focuses on layers of abstract machines as a way to design families of systems
  - Another view is to design in a way that easily enables the building of supersets (extensions) and subsets (contractions)
  - These are equally important directions to be able to move in software – examples?

Examples

- In a strict layered design, a level can only use the immediately lower level
  - Levels often promote operations through to the next level
  - In the strictest view, recursion would be prohibited
- Other examples of layered systems?

- THE
  [Dijkstra 1960's operating system]
  - Level 5: User Programs
  - Level 4: Buffering for I/O devices
  - Level 3: Operator Console Device Driver
  - Level 2: Memory Management
  - Level 1: CPU Scheduling
  - Level 0: Hardware

The `uses` relation

- Parnas says to layer using the `uses` relation
  - A program `A` uses a program `B` if the correctness of `A` depends on the presence of a correct version of `B`
- Requires `A`'s specification and implementation and `B`'s specification
- What's the specification? Signature? Implied or informal semantics?

`uses` vs. `invokes`

- These relations often but do not always coincide
- Invocation without use: name service with cached hints

```
ipAddr := cache(hostName);
if not(ping(ipAddr))
  ipAddr := lookup(hostName)
endif
```

- Use without invocation: examples?

Parnas' observation

- A non-hierarchical `uses` relation makes it difficult to produce useful subsets of a system
  - That is, loops in the `uses` relation (A uses B and B uses A, directly or indirectly) cause problems
  - It also makes testing difficult
- So, it is important to design the `uses` relation

Criteria for `uses(A,B)`

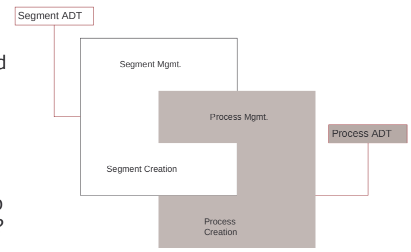
- `A` is essentially simpler because it uses `B`
- `B` is not substantially more complex because it does not use `A`
- There is a useful subset containing `B` but not `A`
- There is no useful subset containing `A` but not `B`

Note again...

- ...Parnas' focus on criteria to help you design

Modules and layers interact?

- Information hiding modules and layers are distinct concepts
- How and where do they overlap in a system?



Language support

- We have lots of language support for information hiding modules
  - C++ classes, Java interfaces, etc.
- We have essentially no language support for layering
  - Operating systems provide support, primarily for reasons of protection, not abstraction
  - Big performance cost to pay for "just" abstraction

Final words

- Design for change isn't easy
- Information hiding and layering are two principles to remember
- There are others, such as separation of concerns
- There are lots of other issues/techniques intended to address change proactively
  - Open implementation
  - Aspect-oriented design/programming
  - ...

Final final words!

- Change in software is a huge issue
- Paying attention to it – even though it's a future benefit more than an immediate one – can produce genuine value