CSE P503: Principles of Software Engineering

David Notkin
Spring 2009

Tonight's agenda

- Bounded model checking: Alloy
  - Why might we care?
  - Slides from elsewhere – a quick-ish run through, focusing on what Alloy can do and why we might care more than on how it does it
  - The next assignment
- Software design: history and (semi-)free-for-all
- One-minute paper (post on wiki or email to me by close of business tomorrow): Key point? Open question? Mid-course correction?

Remember

- You are not yet done with the first assignment
- "one-fifth of your grade for the essay will depend on comments you make (as of Wednesday April 22, 2009 at 6PM) via the wiki on essays written by your classmates"
  - Mark them using the wiki's "Your signature with timestamp" stamp
  - Be constructive

From last week

Bounded model checking

- The TCAS/EPD work avoided most abstraction by starting with finite state specifications
- SLAM/SDV and other model checkers that work on source code must abstract the program to get to a finite state model
- Bounded model checking instead accepts an infinite state machine along with a formula to check – and then truncates the search space
  - Guaranteed to find errors within the bound
  - Errors outside the bound are not found
  - Small scope hypothesis: a high proportion of bugs can be found by testing a program for all test inputs within some small scope

Why might we care?

- See Daniel Jackson's very recent article for this week's class for his view – dependable software at lower cost

Class discussion

My general view

- There are numerous important situations in which we neglect clarity and pay a long-term cost – and many of these situations are recognizable early on
  - What is a Metro bus route? An airplane trip?
  - How do pages, paragraphs, etc. interact with one another in terms of formatting? Style sheets and web pages?
  - Which users are authorized to perform what operations on what directories and files?
- In such situations, an investment in clarity is almost surely worthwhile – and Alloy-like systems can help achieve the needed understanding and clarity

Designations: Michael Jackson

- A *designation* defines a term using a rule
- Does a phenomenon satisfy the designation?
- Allows refutable statements to be made about the requirements
- Should define as few as possible

`x is a human being`	`Human(x)`
`x is male`	`Male(x)`
`x is female`	`Female(x)`
`x is the genetic mother of y`	`Mother(x,y)`
`x is the genetic father of y`	`Father(x,y)`

$\forall x,y \bullet (((Human(x) \wedge Mother(x,y)) \Rightarrow (Female(x) \wedge Human(y)))$

A refutable statement

Definitions: Michael Jackson

- Definitions define terms in terms of existing designations: they are macros, in essence
- They can simplify what you can talk about but don't fundamentally change what you can talk about
- Definitions can't be right or wrong, just well-formed (or not) and useful (or not)

- $Brother(x,y) \cong Male(x) \wedge \exists f \bullet (Father(f,x) \wedge Father(f,y) \wedge \exists m \bullet (Mother(m,x) \wedge Mother(m,y)) \wedge x \neq y$

Clarity

- Part of conceptual integrity?
- Absence a contributor to software disasters?

Alloy tutorial (from Alloy site)

- Session 1 - Intro & Logic (PDF)
- Session 2 - Language & Analysis (PDF)
- Session 3 - Static Modeling (PDF)
- Session 4 - Dynamic Modeling (PDF)

- (Only about 130 slides, so it'll be quick :-)

Your Alloy assignment: HIPAA

- Health Insurance Portability and Accountability Act of 1996 (HIPAA) "provides federal protections for personal health information held by covered entities and gives patients an array of rights with respect to that information. At the same time, the [it] is balanced so that it permits the disclosure of personal health information needed for patient care and other important purposes."
- Figuring out precisely what is and is not permitted is a complicated issue facing software developers who are dealing with HIPAA regulations – failure can be costly both to organizations that can face penalties for non-compliance and also to individuals whose personal health information is misused.
- Can Alloy help induce clarity among key aspects of HIPAA?

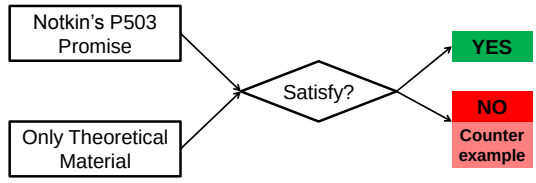
Your Alloy assignment: model

- Individuals
- Personal representatives of individuals (for example, parents of minors)
- Patient information that your health care providers can share with each other
- Patient information that your health care providers can share externally only with your explicit authorization
- …see assignment for details
- There is much more to HIPAA – for instance, handling subpoenas, information used for research, employment information, etc. – but these are not required to be dealt with in your model
- Neither are you required to provide a dynamic model (that is, with operations)

Your Alloy assignment: odds & ends

- Suporn is the "go to" person for Alloy questions, etc. (although I'll help, too, of course)
- Use the wiki and/or mailing list to help each other with Alloy issues
- Can work in groups
- Must turn in not only the final model, but also some intermediate models that characterize your progress and a brief assessment of Alloy
  - This assessment can be done independently by members of a group but combined in the final version that is turned in

Will the formalisms ever stop…?



DESIGN!

Word association: in groups

- Generate words that you think of when you hear the term (software) "design"

Groups of 3-4

What is *design*?

- OED
  - Noun: nine definitions, 1462 words
  - Verb: 16 definitions, 2165 words
- Does your organization have a clear definition?
  - Do you?
- Does your organization have an identifiable design phase?
  - Do you?
- Does this matter?

Software design: some key points

- Addresses complexity – that is, design is primarily for people rather than for computers
- Occurs at multiple levels – that is, design decisions of different sorts are made frequently throughout the lifecycle
- Selection criteria are multifaceted, often hard to capture and difficult to tradeoff – performance, modifiability, reliability, safety, understandability, compatibility, robustness, …

Complexity

- Software entities are more complex for their size than perhaps any other human construct, because no two parts are alike (at least above the statement level). If they are, we make the two similar parts into one… In this respect software systems differ profoundly from computers, buildings, or automobiles, where repeated elements abound [Brooks].
- …as soon as the programmer only needs to consider intellectually manageable programs, the alternatives he is choosing from are much, much easier to cope with [Dijkstra].
- The complexity of the software systems we are asked to develop is increasing, yet there are basic limits upon our ability to cope with this complexity. How then do we resolve this predicament [Booch]?

Conceptual integrity

- Brooks and others assert that conceptual integrity is a critical criterion in design
  - It is better to have a system omit certain anomalous features and improvements, but to reflect one set of design ideas, than to have one that contains many good but independent and uncoordinated ideas [Brooks].
- Such a design often makes it far easier to decide what is easy and reasonable to do as opposed to what is hard and less reasonable to do – it reduces complexity
  - May not please management

Rationalism vs. empiricism

- Brooks' 1993 talk "The Design of Design"
- rationalism — the doctrine that knowledge is acquired by reason without resort to experience [WordNet]
- empiricism — the doctrine that knowledge derives from experience [WordNet]

Examples

- Life
  - Aristotle vs. Galileo
  - France vs. Britain
  - Descartes vs. Hume
  - Roman law vs. Anglo-Saxon law
- Software (Wegner)
  - Prolog vs. Lisp
  - Algol vs. Pascal
  - Dijkstra vs. Knuth
  - Proving programs vs. testing programs

Brooks: empiricist

- A "thoroughgoing, died-in-the-wool empiricist"
- "Our designs are so complex there is no hope of getting them right first time by pure thought. To expect to is arrogant."
- "So, we must adopt design-build processes that incorporate evolutionary growth …"
  - "Iteration, and restart if necessary"
  - "Early prototyping and testing with real users"
  - "Plan to throw one away, you will anyway"

Divide and conquer

- The technique of mastering complexity has been known since ancient times: Divide et impera (Divide and Rule) [Dijkstra]
- We have to decompose large systems to be able to build them – decrease size of tasks, support independent testing and analysis, separate work assignments, ease understanding, …
- For software, decomposition techniques are distinct from those used in physical systems – fewer constraints are imposed by the material

Composition

- Divide and conquer. Separate your concerns. Yes. But sometimes the conquered tribes must be reunited under the conquering ruler, and the separated concerns must be combined to serve a single purpose [M. Jackson]
- Jackson's view of composition as printing with four-color separation
- Composition in programs is not as easy as composition in logic

How do we select a decomposition?

- Determine the desired criteria and select a decomposition (design) that will achieve those criteria
  - Whence the potential decomposition?
- In practice, it's hard to
  - Determine the desired criteria with precision
  - Tradeoff among various conflicting criteria
  - Figure out if a design satisfies given criteria
  - Find a better one with respect to the criteria
- In practice, it's easy to build something designed pretty much like the last one (or at least a recent one)

Semi-continuous

- High-level ("architectural") design
  - What pieces?
  - How connected?
- Low-level design
  - Should I use a hash table or binary search tree?
- Very low-level design
  - Variable naming, specific control constructs, etc.
  - About 1000 design decisions at various levels are made in producing a single page of code

Alan Perlis quotations

- If you have a procedure with 10 parameters, you probably missed some.
- One man's constant is another man's variable.
- There are two ways to write error-free programs; only the third one works.
- When someone says "I want a programming language in which I need only say what I wish done," give him a lollipop.
- Simplicity does not precede complexity, but follows it.

Change: a key criterion

- …accept the fact of change as a way of life, rather than an untoward and annoying exception [Brooks].
- Software that does not change becomes useless over time [Belady and Lehman].
- It is generally believed that to accommodate change one must anticipate possible changes
  - Counterpoint: Extreme Programming
- By anticipating (and perhaps prioritizing) changes, one defines additional criteria for guiding the design activity
- It is not possible to anticipate all changes

Structure: a design keystone

- The focus of most software design approaches is structure
- What are the components and how are they put together?
- Behavior is important, but largely indirectly

Traditional properties of design

- Cohesion
- Coupling
- Complexity
- Correctness
- Correspondence

Cohesion

- The reason that elements are found together in a module
  - Ex: coincidental, temporal, functional, …
- The details aren't critical, but the intent is useful
- During maintenance, one of the major structural degradations is in cohesion
  - Need for "logical remodularization"

Coupling

- Strength of interconnection between modules
- Hierarchies are touted as a wonderful coupling structure, limiting interconnections
  - But don't forget about composition, which requires some kind of coupling
- Coupling also degrades over time
  - "I just need one function from that module…"
  - Low coupling vs. no coupling

Unnecessary coupling hurts

- Propagates effects of changes more widely
- Harder to understand interfaces (interactions)
- Harder to understand the design
- Complicates managerial tasks
- Complicates or precludes reuse

It's easy to...

- ...reduce coupling by calling a system a single module
- …increase cohesion by calling a system a single module

- No satisfactory measure of coupling
  - Either across modules or across a system

Coupling and cohesion

- Do you think about these? Explicitly?
- Any tools?

Class discussion

Complexity

- Few if any useful measures of design/program complexity exist
- There are dozens of such measures; e.g., McCabe's cyclomatic complexity = E - N + p
  - E = the number of edges of the CFG
  - N = the number of nodes of the CFG
  - p = the number of connected components
- My understanding is that, to the first order, most of these measures are linearly related to "lines of code"
- No way to distinguish accidental from essential complexity

Complexity

- Do you think about this? Explicitly?
- Any tools?

Class discussion

Correctness

- Well, yeah
- Even if you "prove" modules are correct, composing the modules' behaviors to determine the system's behavior is hard
- Leveson and others have shown clearly that a system can fail even when each of the pieces work properly – this is because many systems have "emergent" properties
- Arguments are common about the need to build "security" and "safety" and … in from the beginning

Correspondence

- "Problem-program mapping"
- The way in which the design is associated with the requirements
- The idea is that the simpler the mapping, the easier it will be to accommodate change in the design when the requirements change

Functional decomposition

- Divide-and-conquer based on functions
  - `input; compute; output`
- Then proceed to decompose compute
- This is stepwise refinement (Wirth, 1971)
  - In essence, refining until implementable directly in a programming language (or on an architecture)
- There is an enormous body of work in this area, including many formal calculi to support the approach
  - Closely related to proving programs correct
- More effective in the face of stable requirements

Information hiding

- What do you think it is?

Groups of 3-4

Information hiding

- Information hiding is perhaps the most important intellectual tool developed to support software design [Parnas 1972]
  - *Makes the anticipation of change a centerpiece in decomposition into modules*
- Provides the fundamental motivation for abstract data type (ADT) languages
  - And thus a key idea in the OO world, too
- The conceptual basis is key

Basics of information hiding

- Modularize based on anticipated change
  - Fundamentally different from Brooks' approach in OS/360 (see old and new MMM)
- Separate interfaces from implementations
  - Implementations capture decisions likely to change
  - Interfaces capture decisions unlikely to change
  - Clients know only interface, not implementation
  - Implementations know only interface, not clients
- Modules are also work assignments

Anticipated changes

- The most common anticipated change is "change of representation"
  - Anticipating changing the representation of data and associated functions (or just functions)
  - Again, a key notion behind abstract data types
- Ex:
  - Cartesian vs. polar coordinates; stacks as linked lists vs. arrays; packed vs. unpacked strings

Information hiding: issues

- Can we effectively anticipate changes?
- What is the underlying cost model and is it reasonable?
- The semantics of the module remain unchanged when implementations are changed: the client should only care if the interface is satisfied
  - But what captures the semantics of the module? The signature of the interface? Performance? What else?
- One implementation should satisfy multiple clients, which should only care if the interface is satisfied

Representation change less common

- We have significantly more knowledge about data structure design than we did 25 years ago
- Memory is less often a problem than it was previously, since it's much less expensive
- Therefore, we should think twice about anticipating that representations will change
  - This is important, since we can't simultaneously anticipate all changes

Other anticipated changes?

- Information hiding isn't only ADTs
- Algorithmic changes
  - (These are almost always part and parcel of ADT-based decompositions)
  - Monolithic to incremental algorithms
  - Improvements in algorithms
- Replacement of hardware sensors
  - Ex: better altitude sensors
- …

Best to change implementation?

- Usually, perhaps, but not always the lowest cost

- Changing a local implementation may not be easy
- Some global changes are straightforward: mechanically or systematically
- Rob Miller's simultaneous text editing
- Bill Griswold's work on information transparency

Information hiding reprise

- It's probably the most important design technique we know
- And it's broadly useful
- It raised consciousness about change
- But one needs to evaluate the premises in specific situations to determine the actual benefits (well, the actual potential benefits)

Dependence on implementation

- Gregor Kiczales: open implementation
- Clients indeed depend on some aspects of the underlying implementations in a broad variety of domains
- Decompose into base interface (the "real" operations) and the meta interface (the operations that let the client control aspects of the implementation)
- Arose from work in (roughly) reflection in the Meta-Object protocol (MOP) and led to the development of aspect-oriented programming

Information Hiding and OO

- Are these the same? Not really
  - OO classes are chosen based on the domain of the problem (in most OO analysis approaches)
  - Not necessarily based on change
- But they are obviously related (separating interface from implementation, e.g.)
- What is the relationship between sub- and super-classes?

Layering [Parnas 79]

- A focus on information hiding modules isn't enough
- One may also consider abstract machines
  - In support of program families, which are systems that have "so much in common that it pays to study their common aspects before looking at the aspects that differentiate them"
- Still a focus on anticipated change

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 specification and implementation of **A** and the specification of **B**
- Again, what is the "specification"? The interface? Implied or informal semantics?

uses vs. **invokes**

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

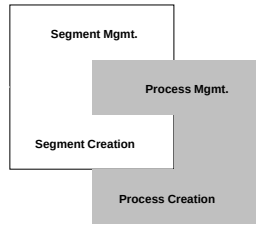
```
ipAddr := cache(hostName);
if wrong(ipAddr,hostName) then
   ipAddr := lookup(hostName)
endif
```


Parnas' observation

- A non-hierarchical uses relation makes it difficult to produce useful subsets of a system
- So, it is important to design the **uses** relation using these criteria
  - **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**

Modules and layers interact?

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



A key point

- Not all boxes in a design are the same thing
- Not all arrows in a design are the same thing
- Imprecision in communication about these boxes and arrows can add significant confusion to a software design process and the resulting design
- Oh, that's the issue of clarity again

Language support?

- We have lots of language support for information hiding modules
  - C++ classes, Ada packages, 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

Design questions/topics/insights?

Class discussion

Don't forget…

- One-minute paper (post on wiki or email to me by close of business tomorrow): Key point? Open question? Mid-course correction?