dependences & coupling

Daniel Jackson
6898
April 17, 2003

why this topic?

what is software design?

- › choose syntactic interfaces
- › … to achieve semantic function
- › … in a way that minimizes coupling

despite importance

- › idea of dependence is still vague
- › little research on essential notions
- › tools still primitive

why these papers?

Parnas

- a classic: written in 1979 and still fresh
- a big improvement on its successors

Pfleeger

- Constantine’s coupling and cohesion
- from Yourdon & Constantine’s *Structured Design*
- the idea from SA/SD that lasted

Liskov & Guttag

- both of these ideas in 6170 setting
- specs begin to play more of a role

why these papers? (ctd)

Gang of Four design patterns
- dependences central
- but no notation!

UML Reference Manual
- evidence that good ideas can be blurred and lost

> I have identified some simple concepts that can help programmers design software so that subsets and extensions are more easily obtained. These concepts are simple if you think about software in the way suggested by this paper. Programmers do not commonly do so.
>
> *Parnas*

structured design

approach
- design system as communicating modules
- evaluate using cohesion & coupling metrics

relation to Parnas's uses
- recognizes kinds of coupling that 'uses' doesn't capture
- appealing but slippery ideas; not well-defined

kinds of coupling
- normal: A calls B, B returns to A, all comms by parameters
- data: another module passes data from A to B
- stamp: composite data (ie, must agree on representation)
- control: A passes a flag to B that controls its behaviour
- common: A and B refer to same global data area

parnas's ideas

a family of programs
- every development creates a family
- think about this at outset

bad approaches
- chain of data transformations
- components performing >1 function
- cyclic dependences

a better approach
- identify subsets in requirements
- information hiding
- virtual machines
- design of uses relation

a better approach

identify subsets in requirements

- engage, but don’t trust, the user
- modelling helps a lot here
- like XP’s “the simplest thing that works”

information hiding

- not just about data abstraction
- identify items likely to change: “secrets”
- localize secrets in modules: one secret/module
- design interface to hide secret

virtual machine

- not steps of processing as in SA/SD, top-down design
- basis for SICP (6.001) approach

definition of 'uses'

A uses B = correct execution of B may be necessary for A to complete the task described in its specification

invokes != uses
- A must just invoke B but expect no response
- B may be an interrupt handler that must preserve invariants

elegance vs. independence
- elegant: shared use of subcomponents
- independent: parts duplicate functionality

layered systems

if uses is acyclic, can define levels
- level 0: components use no others
- level K: use at least one component from level K-1 and none from a level higher than K-1

Parnas claims
- each level offers a testable subset

comments
- layers are usually of non-uniform thickness
- often useful to aggregate into packages to see layers
- Parnas says modules do *not* correspond to layers

when may A use B?

criteria

- A is made simpler by using B
- B is not made substantially more complex
- some subset contains B and not A
- no subset contains A and not B

other ideas in the paper

subtyping
- “An AFM can be made compatible with an ASM”

critique of kernel approach to OS design
- not sufficient to bundle key services into tangled kernel
- reminiscent of Kaashoek’s Exokernel

flexibility vs. generality
- generality: can be used without change for many purposes
- flexibility: can be adapted to many purposes
- unlike in mathematics, generality is not always a good thing

No one can tell a designer how much flexibility and generality should be built into a product, but the decision should be a conscious one. Often, it just happens.

problems with ‘uses’

why ‘uses’ is not good enough

- not adequate to describe modern software
  no notion of replaceability, for example
- by definition, uses is transitive!
- uses is binary; no measure of extent of coupling
- certain kinds of coupling not captured

some new ideas

status

- developed for 6170
  - inadequacy of MDD for explaining design patterns
- recently, joint work with Allison Waingold
- influenced by SML, self-updating software
- similar to units
- still in early stages

two key ideas

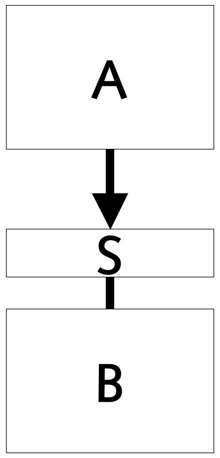
- use mediated by spec
- name dependence

the role of specs

a spec is
- a description of a service provided or required
- not a module with dependences
- may or may not be expressible in programming language
  eg, in Java, some (but not all) specs will be Java interfaces

two relations
- requires: Component -> Spec
- provides: Component -> Spec

‘uses’ becomes
- module A requires a service S
- module B provides a service S



how does this differ from ‘uses’?

makes specs explicit
- actually the key design elements!

not transitive
- module sees service, not module

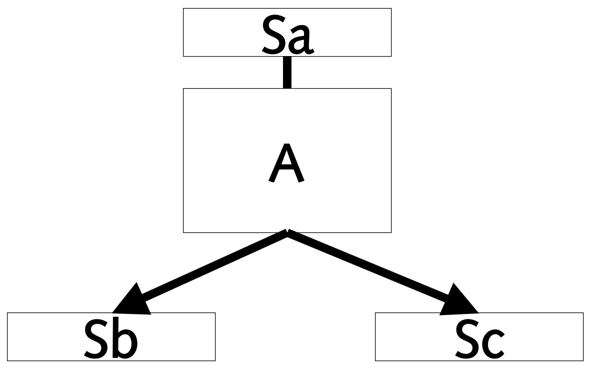
multiple specs
- the same service can be used under different specs
- can explain plugins: module provides different services

module requires service
- does not depend on name of module providing service

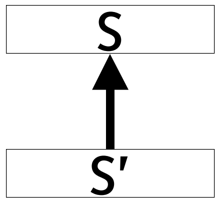
correctness reasoning

argument

- given services Sb and Sc
- code of module A
- correctly provides service Sa



spec ordering



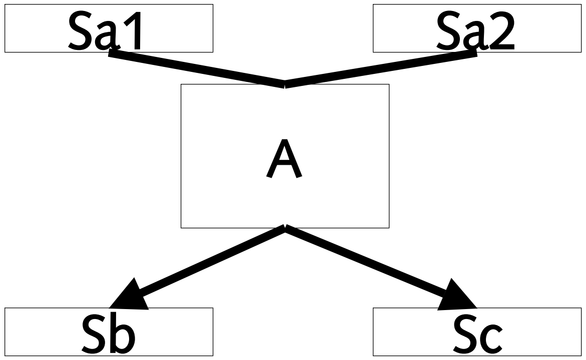
S' extends S iff

- any module that requires S
  will be satisfied
  by a module that provides S'
- any module that provides S'
  provides S

properties

- a partial order
- S extends S
- if S' extends S, and S extends S', S = S'
- if S' extends S, and S'' extends S', S'' extends S

fine structure of dependences

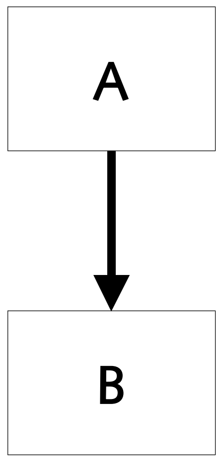


full structure
- deps: Module -> Spec -> Spec
- deps[M][P] is
  set of required specs
  for module M to provide service

requires & provides
- defined in terms of deps

configuration described by
- link: Module -> Spec -> Module
- link[M][R] is the module linked to M
  that provides service that fulfills requirement R
- well-formed iff enough services provided
  and provided services extend required services

name dependence



A has a name dependence on B iff

- module A refers to the name of module B
- so A won't run without presence of B

in languages like Java

- almost all uses have name deps
- dynamic dispatch helps narrow to constructor
- and factory pattern narrows further

challenges for class discussion

polymorphic container

- equality with ==
- element-specific equality
- container as element

standard idiom to reduce coupling

- I x = new C ();

design patterns

- abstract factory
- observer

more …

data abstraction

- rep exposure
- rep independence

inheritance, delegation, etc

- when subclass sees only public interface
- when subclass sees internals
- when superclass relies on subclass

unresolved issues

couplings that don’t follow control
- passing arguments between clients
- read/write file format
- common coupling

relation to requirements
- duplicated functionality
- axiomatic design may help?