Coupling

Dependences
why this topic?

what is software design?
choose syntactic interfaces

to achieve semantic function

do that minimizes coupling
despite importance

idea of dependence is still vague

tools still primitive

little research on essential notions

despite importance

... in a way that minimizes coupling

... to achieve semantic function

what is software design?
why these papers?

 specs begin to play more of a role
 both of these ideas in 6179 setting
 Liskov & Guttag

 the idea from SA/SD that lasted
 From Yourdon & Constantine's Structured Design
 Constantine's coupling and cohesion
 Pfleeger

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

Programmers do not commonly do so.
think about software in the way suggested by this paper. These concepts are simple if you are more easily obtained. These concepts are simple if you have identified some simple concepts that can help evidence that good ideas can be blurred and lost.

UML Reference Manual

but no notation;

dependence central

Gang of Four design patterns

Why these papers? (ctd)
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

stamp: composite data (i.e., must agree on representation)

data: another module passes data from A to B

control: A passes a flag to B that controls its behaviour

common: A and B refer to same global data area

Kinds of Coupling

apparently
Parnas' ideas

- Design of uses relation
- Virtual machines
- Information hiding
- Identity subsets in requirements
- A better approach

- Cyclic dependencies
- Components performing 1 function
- Chain of data transformations
- Bad approaches

- Think about this at outset
- Every development creates a family
- A family of programs
a better approach

basis for SICP (6.001) approach

not steps of processing as in SA/SD, top-down design

virtual machine

design interface to hide secret

localize secrets in modules: one secret/module

identity items likely to change: "secrets"

not just about data abstraction

information hiding

"like XP's "the simplest thing that works"

modeling helps a lot here

engage, but don't trust, the user

identity subjects in requirements
invokes \neq uses

\begin{itemize}
\item independent: parts duplicate functionality
\item elegant: shared use of subcomponents
\item elegance vs. independence
\end{itemize}

B may be an interrupt handler that must preserve invariants

A must just invoke B but expect no response

invokes \neq uses

complete the task described in its specification

\begin{itemize}
\item A uses B = correct execution of B may be necessary for A to
\end{itemize}

definition of \textit{uses}
Parnas says modules do not correspond to layers

often useful to aggregate into packages to see layers

Layers are usually of non-uniform thickness

Comments

each level offers a testable subset

Parnas claims from a level higher than K-1

Level K: use at least one component from Level K-1 and none

Level 0: components use no others

if uses is acyclic, can define levels

Layered systems
when may A use B?

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

Criteria
Other ideas in the paper

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

unlike in mathematics, generality is not always a good thing
Flexibility vs. Generality

reminiscent of Keashoek's Exokernel
not sufficient to bundle key services into tangled kernel
critique of kernel approach to OS design

An AFM can be made compatible with an ASM
subtyping
Problems with ‘uses’

- Certain kinds of coupling not captured
- ‘Uses’ is binary; no measure of extent of coupling
- By definition, ‘uses’ is transitive
- No notion of replaceability, for example
- Not adequate to describe modern software
- Why ‘uses’ is not good enough

uses is not good enough
some new ideas

name dependence

use mediated by spec

two key ideas

still in early stages

similar to units

influenced by SML, self-updating software

recently, joint work with Allison Waingold

inadequacy of MDD for explaining design patterns

developed for 6170

status
the role of specs

Spec

module A provides a service S
module B requires a service S
uses becomes 'uses' becomes

module A requires a service S
module B provides a service S

Spec

Component

two relations

eg, in Java, some (but not all) specs will be Java interfaces
may or may not be expressible in programming language

a spec is

Spec

Component

a description of a service provided or required
how does this differ from 'uses'?
Correctness reasoning
Spec ordering

S' extends S, and S extends S', S'' extends S', S = S''

If S extends S', and S extends S', S = S''

S extends S

Providing S

Any module that provides S

By a module that provides S

Will be satisfied

Any module that requires S

S extends S iff
and provided services extend required services
well-formed iff enough services provided
that provides service that fulfills requirement R
\[ \text{link}[M][R] \text{ is the module linked to } M \]
\[ \text{link: Module } \rightarrow \text{ Spec } \rightarrow \text{ Module} \]
configuration described by

\[ \text{defined in terms of } \text{deps} \text{ requires } \text{provides} \]

for module M to provide service
set of required specs
\[ \text{deps}[M][P] \text{ is } [\text{deps}: \text{Module } \rightarrow \text{ Spec } \rightarrow \text{ Spec} \]
full structure
fine structure of dependencies
A has a name dependence on B iff

- module A refers to the name of module B
- A has a name dependence on B iff
  - almost all uses have name deps
  - dynamic dispatch helps narrow to constructor
  - and factory pattern narrows further

in languages like Java

so A won't run without presence of B
challenges for class discussion

- observer
- abstract factory
- design patterns

\[ I \ x = \text{new C}() \]

Standard idiom to reduce coupling

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

- polymorphism
data abstraction
• rep exposure
• rep independence
• inheritance, delegation, etc.

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

more ...
unresolved issues

axiomatic design may help?

duplicated functionality

relation to requirements

common coupling

read/write file format

passing arguments between clients

couplings that don’t follow control