dependences & coupling

Daniel Jackson
6898
April 17, 2003

dependences & coupling
why this topic?
why this topic?

• In a way that minimizes coupling
• ... to achieve semantic function
• Choose syntactic interfaces

What is software design?
tools still primitive
little research on essential notions
idea of dependence is still vague
despite importance

... in a way that minimizes coupling
... to achieve semantic function
choose syntactic interfaces
what is software design?

why this topic?
why these papers?
Why these papers?

- a classic: written in 1979 and still fresh
- Parnas
- a big improvement on its successors
Why these papers?

- The idea from SA/SD that lasted
- From Yourdon & Constantine's Structured Design
- Constantine's coupling and cohesion
- Pleeger

- A big improvement on its successors
- A classic: written in 1979 and still fresh
- Parnas
why these papers?

- specs begin to play more of a role
- both of these ideas in 6170 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
Why these papers? (ctd)
why these papers? (ctd)

Gang of Four design patterns›dependence central

but no notation!
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)
Programmers do not commonly do so. They think about software in the way suggested by this paper. These concepts are simple if you are more easily obtainable. 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 centred

Gang of Four design patterns

Why these papers? (ctd)
structured design
structured design

approach

> design system as communicating modules
> evaluate using cohesion & coupling metrics
structured design

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

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

structured design
structured design

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

Kinds of coupling

- appealing but slippery ideas; not well-defined
- recognizes kinds of coupling that ‘uses’ doesn’t capture

Relation to Parnas’s uses

- evaluate using cohesion vs. coupling metrics
- design system as communicating modules

approach

structured design
Parnass's ideas
parnas's ideas

think about this at outset

every development creates a family of programs
Parnas’s ideas

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

Think about this at outset
Every development creates a family

A family of programs
Parnas's ideas

- Design of user relations
- Virtual machines
- Information hiding
- Identifying subsets in requirements
- A better approach
- Cyclic dependencies
- Components performing one function
- Chain of data transformations
- Bad approaches

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

Like XP’s “the simplest thing that works”

· engage, but don’t trust, the user
· modelling helps a lot here
· identify subsets in requirements

...
a better approach

- Engage interface to hide secret
- Localize secrets in modules: one secret/module
- Identify items likely to change: “secrets”
- Not just about data abstraction

Information hiding

- Like XP’s “the simplest thing that works”
- Engage, but don’t trust, the user
- Identity subsets in requirements

Identify subsets in requirements

- Design interface to hide secret
- Localize secrets in modules: one secret/module
- Identify items likely to change: “secrets”
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

identify subsystems in requirements
definition of 'uses'
A uses B = correct execution of B may be necessary for A to complete the task described in its specification.
A uses B = correct execution of B may be necessary for A to complete the task described in its specification

B may be an interrupt handler that must preserve invariants

A must just invoke B but expect no response

Invokes i= uses

\textit{Definition of, uses}
A uses B = correct execution of B may be necessary for A to complete the task described in its specification

A must just invoke B but expects no response

B may be an interrupt handler that must preserve invariants

\( \text{Invokes} \neq \text{uses} \)

Elegance vs. independence

\( \text{elegant: shared use of subcomponents} \)

\( \text{independent: parts duplicate functionality} \)

Definition of 'uses'
layered systems
layered systems

If uses is acyclic, can define levels

- Level K: use at least one component from level K-1 and none from a level higher than K-1
- Level 0: components use no others
Parnas claims each level offers a testable subset from a level higher than \( K-1 \): Level \( K \): use at least one component from level \( K-1 \) and none from higher levels. Level 0: components use no others. If uses is acyclic, can define levels.
layered systems

Parnas says modules do not correspond to layers

- 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 levels higher than K-1

Parnas claims

- Each level offers a testable subset

Layers are usually of non-uniform thickness

Comments

- Sometimes useful to aggregate into packages to see layers

It uses is acyclic, can define levels
when may A use B?
When may A use B?

Criteria:
- 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
other ideas in the paper
other ideas in the paper

An AFM can be made compatible with an ASM

Subtyping
other ideas in the paper

Reminiscent of Kaashoek’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
Unlike in mathematics, generality is not always a good thing. Flexibility vs. generality: can be adapted to many purposes; generality: can be used without change for many purposes.

Critique of kernel approach to OS design:

"An AFM can be made compatible with an ASM" subtyping.

Other ideas in the paper:

Reminiscent of Kaashoek's Exokernel.
conscious one. Often, it just happens.

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.

Unlike in mathematics, generality is not always a good thing.

Flexibility: can be adapted to many purposes.

Generality: can be used without change for many purposes.

Flexibility vs. Generality

Reminiscent of Keashook’s Exokernel

Critique of Kernel approach to OS design

“An AFM can be made compatible with an ASM”

Subtyping

Other ideas in the paper
12

problems with 'uses'
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
some new ideas
Some new 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 6T70

status
Some new ideas

- 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
The role of specs is to describe a service provided or required, not a module with dependencies. It may or may not be expressible in a programming language. E.g., in Java, some (but not all) specs will be Java interfaces. A spec is a description of a service provided or required, not a module with dependencies.
The role of specs

A spec is
- a description of a service provided or required
- not a module with dependencies
- may or may not be expressible in programming language

Two relations
- requires: Component \(\rightarrow\) Spec
- provides: Component \(\rightarrow\) Spec

Example, in Java, some (but not all) specs will be Java interfaces.
The role of specs

A spec is a description of a service provided or required, not necessarily a module with dependencies. Some (but not all) specs will be Java interfaces.

Two relations:
- \text{requires: Component} \rightarrow \text{Spec}
- \text{provides: Component} \rightarrow \text{Spec}

In some languages, like Java, some (but not all) specs will be Java interfaces. For example, a module may or may not be expressible in programming language dependencies. A spec is

\[ A \rightarrow S \rightarrow B \]

\[ \text{uses becomes}\]

\[ \text{requires: Component} \rightarrow \text{Spec} \rightarrow \text{provides: Component} \rightarrow \text{Spec} \]
how does this differ from 'uses'?
how does this differ from 'uses'?

makes specs explicit

actually the key design elements!
how does this differ from 'uses'?

module sees service, not module
not transitive

actually the key design element!
makes specs explicit
how does this differ from 'uses'?

- can explain plugins: module provides different services
  - the same service can be used under different specs
  - multiple specs
- module sees service, not module
  - not transitive
- actually the key design element!
  - makes specs explicit

15
how does this differ from 'uses'? does not depend on name of module providing service
module requires service

multiple specs

module provides different services

can explain plugins: module provides different services

the same service can be used under different specs

module sees service, not module
not transitive

actually the key design elements!

makes specs explicit

15
correctness reasoning
correction reasoning

- correctly provides service $S_a$
- code of module $A$
- given services $S_b$ and $S_c$

Argument
spec ordering
S extends S' iff any module that requires S provides S'.

By a module that provides S', any module that provides S will be satisfied.

Spec ordering
spec ordering

properties

provides $S$

any module that provides $S'$

by a module that provides $S'$

will be satisfied

any module that requires $S$

$S$ extends $S'$ iff

$S$ extends $S'$, and $S'$ extends $S'$.

S extends S', S = S'

S extends S

a partial order

$S$ extends $S'$ and $S$ extends $S''$, $S''$ extends $S'$. 

S extends S'}
fine structure of dependences

A

Sb

Sa1

Sc

Sa2
Fine structure of dependences

For module M to provide service set of required specs is [dep[M]|P] is Spec \ dep: Module Spec Spec

Full structure of dependences
The structure of dependencies is defined in terms of `deps`

`requires & provides`

For module \( M \) to provide service

set of required specs

\( \text{deps} \ [\text{M}] \ [\text{P}] \) is

\( \text{deps}: \text{Module} \rightarrow \text{Spec} \rightarrow \text{Spec} \)

Full structure of dependencies
Fine structure of dependences and provided services extend required services and well-formed iff enough services provided. There is a well-formed service if it fulfills requirement $R$. Link $[M][R]$ is the module linked to $M$. Configuration described by:

- Defined in terms of deps
- Requires $R$ provides
- Defined in terms of deps
- Set of required specs
- $[_deps[M][R]]$ is $[deps:M] => Spec$ => Spec

For module $M$ to provide service:

Full structure of dependences
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.
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
challenges for class discussion

- container as element
- element-specific equality
- equality with ==
- polymorphic container
Challenges for class discussion

standard idiom to reduce coupling

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

container as element

\<\text{element-specific equality}\>

\<\text{equality with} \text{polymorphic container}\>
challenges for class discussion

\[ \text{observer} \]
\[ \text{abstract factory} \]
\[ \text{design patterns} \]

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

standard idiom to reduce coupling

\[ \text{container as element} \]
\[ \text{element-specific equality} \]
\[ \text{equality with } \text{polymorphic container} \]
more ...

data abstraction
rep exposure
rep independence
More ...
unresolved issues
unresolved issues

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

- Couplings that don’t follow control
- Read/write file format
- Passing arguments between clients
- Unresolved issues

- Axiomatic design may help?
- Duplicate functionality
- Relation to requirements
- Common coupling