Software components

Service-Oriented Architecture

Jeremy Gibbons

Contents

1 Mass-produced software components

Doug McIlroy, NATO conference on Software Engineering, 1968

1.1 Catalogue of components

- industrialism vs crofting, engineering vs craft
- orthogonal families of interchangeable components
- varying robustness, generality, performance, precision...
- choice tailored to individual use
- no significant penalty for choice

1.2 Component variation

- numerical routines for computing sine function
- varying precision
- floating or fixed point
- argument range (0–$\pi/2$, 0–2π, $-\pi/2$–$\pi/2$, $-\pi$–π, arbitrary)
- validation (out of range, loss of significance...)
- time–space tradeoff by table lookup: table size, quantization
- expected call sequences (eg successive calls for sine and cosine of same angle)

1.3 Parameters

- generality (which parameters adjustable at run-time)
- precision (length, width, size)
- robustness (accuracy versus performance)
- performance (time vs space)
- interface (eg for errors: alternate return, error code, exception)
- algorithm
- data representation

1.4 Pragmatics

- ‘mass production’ of models, not of replicates
- can’t hand-craft each component: they must be generated
- (hence generics, partial evaluation, product lines. . . : systematized parametrization)
- distinction between component suppliers and consumers
- chicken and egg problem with a market
- standardization: neither too early nor too late

2 Modularity

- structured programming: disciplined control flow

  Dijkstra, *Goto Considered Harmful*, 1968

- modules: information hiding

  Parnas, *Criteria for Decomposing Systems*, 1972

- data abstraction: smart data structures and dumb code

  Liskov, *Abstract Data Types*, 1974

- object orientation: polymorphism, subtyping, inheritance

  Dahl, Myhrhaug, Nygaard, *Simula*, 1967

3 Component-oriented programming

- Clemens Szyperski, *Component Software: Beyond OOP*
- *software components* enable practical reuse of software parts
- composing components improves quality, supports rapid development, increases flexibility
- component software expected to be *the* cornerstone of software in the years to come
- . . . but we've been saying that since 1968
- . . . however, recent developments in language technology offer renewed hope

3.1 Engineering components

- use of components is axiomatic in any mature engineering discipline
- by analogy, should be in software engineering too
- sometimes said that software is too *flexible* for components

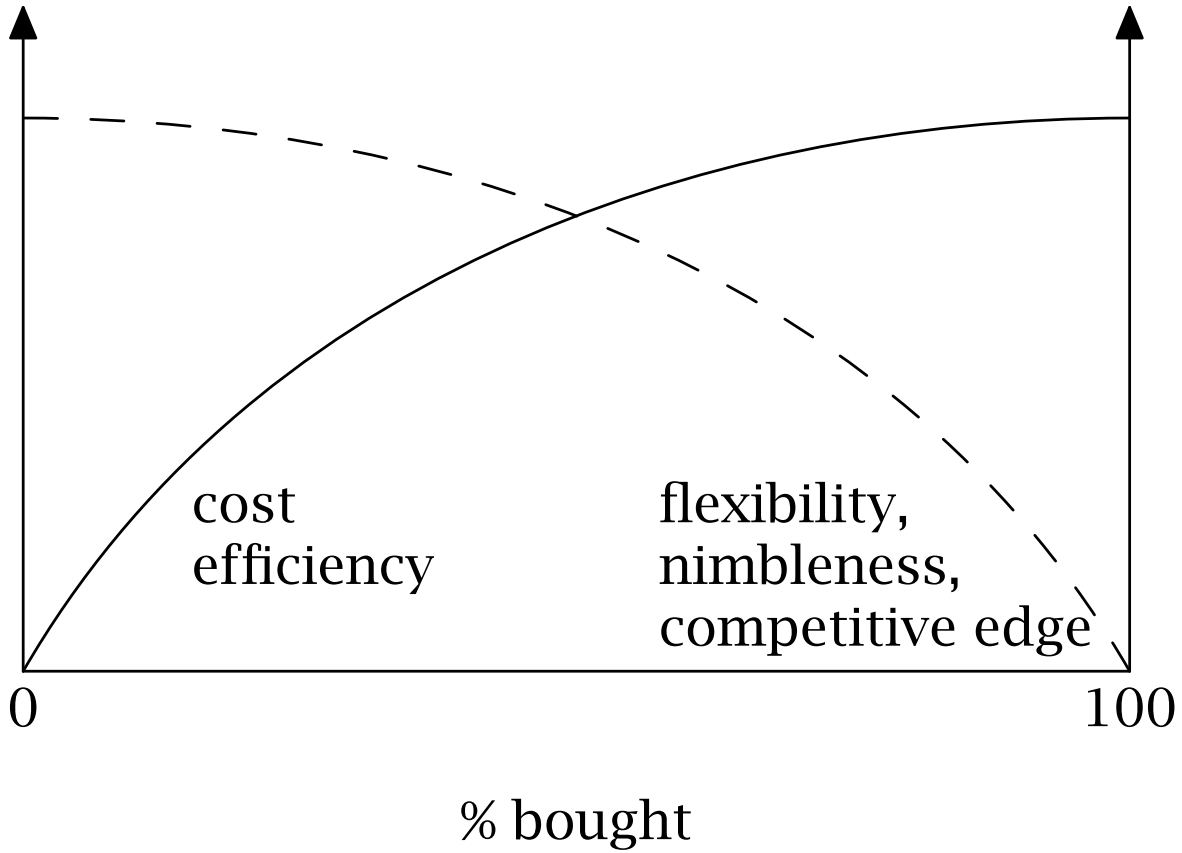
  > *The programmer, like the poet, works only slightly removed from pure thought-stuff.* (Fred Brooks)

- . . . but this just indicates immaturity in the discipline
- Szyperski claims *software engineering now entering maturity*

3.2 Custom-made *vs* standard software

- one extreme: project from scratch, using only programming tools and libraries
- optimally adapted, but expensive, technically suboptimal, usually late, and requires maintenance
- hence major trend towards other extreme: project outsourced, COTS software bought
- may involve business reorganization, cannot confer any competitive advantage, slow to adapt to changing needs
- component software is a ‘third way’: customized assembly of standard components; compromise between cost efficiency and flexibility

3.3 Spectrum between make-all and buy-all



3.4 Critical mass

- technical superiority of component technology is not enough
- *critical mass* also required: variety and quality, apparent benefit, strong or numerous sources
- then use of components becomes inevitable, and proprietary solutions become unsustainable
- if such a vortex is plausible, *preparedness* for an emerging component market is essential
- preparation involves component-friendly (modular) approach, which has its own advantages anyway

3.5 The nature of software

- software components initially considered analogous to hardware components

  software IC, software bus

- also analogy with hi-fi components, mechanical engineering (gears, nuts, bolts), Lego
- but software is different: deliverable is not product, but *blueprint* for product
- computer is *factory*, instantiating blueprint
- *by default, repeated instantiation is trivially easy*

3.6 Deployable entities

- as important to distinguish between software and instances as between plan and building (‘the map is not the territory’)
- confusion between classes and objects
- similar confusions in entity-relationship modelling, CRC cards
- understandable: in software, both plan and building representable by same kind of thing (eg object, bits)
- plans can be parametrized, applied recursively, scaled, repeatedly instantiated, whereas instances cannot
- software is a *generic metaproduct*
- maths a good model of logical structure of software, but not of engineering and market aspects
- moral: SE is a blend of physical and logical engineering

3.7 Lessons learned

- non-incestuous component success stories:
  - modern OSs (co-existing applications)
  - plug-in architectures (eg Firefox, PhotoShop)
  - mobile phone apps
- infrastructure providing *rich foundational functionality*
- components purchased from *independent providers*, *deployed by clients*, with *direct meaning to client*
- different roles for component *construction* and *assembly*
- large enough for reimplementation not to be cost effective
- multiple components from different sources coexist
- . . . but arbitrary combinations may conflict
  (composability need only be highly likely, not guaranteed)

4 Component terminology

- components
- objects
- modules
- whitebox *vs* blackbox
- interfaces
- explicit context dependencies
- component weight
- standardization and normalization

4.1 Components

- *unit of independent deployment*

  encapsulated constituent features; cannot be partially deployed

- *unit of third-party composition*

  self-contained, well specified (interaction through well-defined interfaces)

- *no persistent state*

  indistinguishable from a copy (except perhaps for non-operational attributes like serial number)

- eg database server (but not the database itself)

4.2 Objects

- unit of instantiation (unique identity: Grandpa's axe)
- has state, perhaps persistent
- encapsulates state and behaviour
- construction plan: *class* (instantiated) or *prototype* (cloned)
- initialization: *constructor* (static procedure) or *object factory* (separate object)

4.3 Components vs objects

- typically, component consists of one or more classes or prototype objects, perhaps with immutable objects capturing initial state and other such resources
- but component need not contain classes only, or at all
- might contain global procedures or static variables; might be implemented in functional or assembly language
- (references to) objects created inside component may become visible from outside
- but revealed objects do not tell whether component is ‘all OO’ inside

4.4 Modules

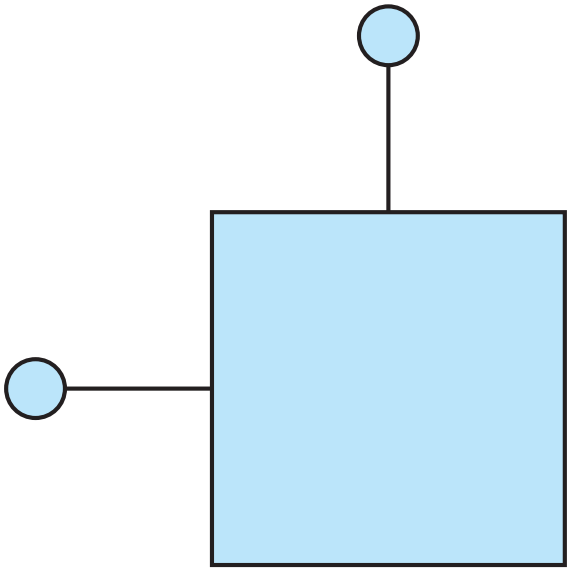
- components close to modules (Modula-2, Ada)
- Eiffel treats class as ‘better’ module
- more recent OO languages (Java, Oberon, etc) keep classes and modules (or packages) separate
- modules are minimal components
- but modules have no persistent immutable resources, apart from hardwired constants, so are not easily reconfigurable
- modularity is necessary but not sufficient for component orientation

4.5 Whitebox vs blackbox

- blackbox abstraction provides only interface signature and specification
- whitebox abstraction reveals also implementation
- blackbox and whitebox reuse of implementations, similarly
- whitebox reuse usually breaks under revision, which retains contract but changes implementation

4.6 Interfaces

- interface defines component's access points
- these allow component's clients (perhaps components themselves) access to services provided
- interface specifies signature and behaviour
- component usually provides multiple interfaces, catering for different client needs
- navigation between interfaces



4.7 Explicit context dependencies

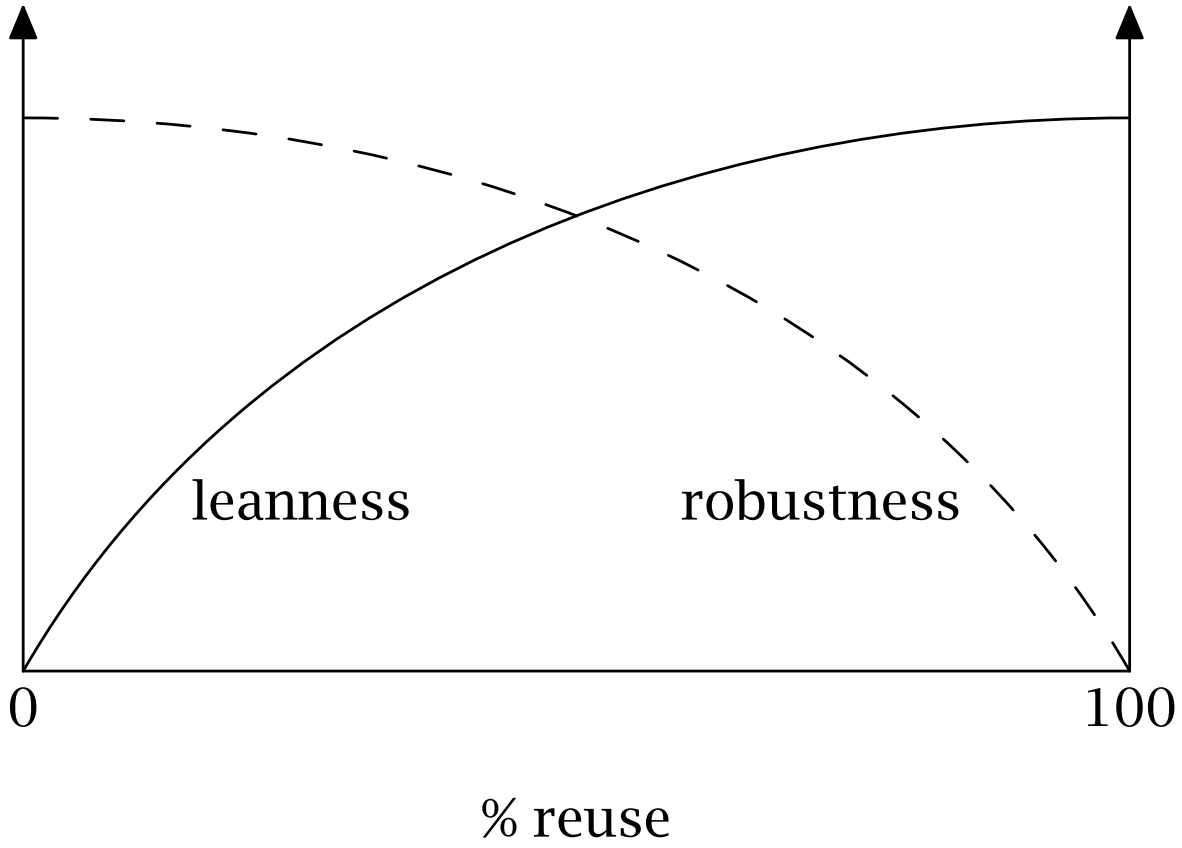
- interfaces describe component's *responsibilities*
- context dependencies describe component's *rights*
- specify what deploying environment must provide
- these too should be explicit

4.8 Component weight

- component may minimize context dependencies by bundling all required services
- defeats purpose of using components
- dually, may maximize reuse by ‘outsourcing’ all but core functionality
- restricts opportunities for use

maximizing reuse minimizes applicability

4.9 Trade-off of leanness against robustness



(note similarity with Slide 3.3: reuse across components here *vs* reuse of components there)

4.10 Standardization and normalization

- shift ‘sweet spot’ from robustness towards leanness by improving degree of standardization and normalization
- more stable and widely supported aspect is safer dependency
- (cf business depending on telephone, now and in past)
- *horizontal standardization*: across domains (eg WWW)
- *vertical standardization*: within a domain (eg medical radiology)
- *normalization* to unify competing standards (especially parallel vertical standards, eg image formats in medical radiology and radio astronomy)

4.11 Software component: a definition

A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties.

- an outcome of *Workshop on Component-Oriented Programming* at ECOOP96
- combines technical and market-oriented aspects

5 Services

It is not the strongest of the species that survive, nor the most intelligent, but the ones most responsive to change. (Darwin)

- flexibility is vital
- processes and systems are becoming more and more complex
- no longer separate systems, but combined
- harmonization and central control does not scale
- we must embrace heterogeneity and decentralization

(Arguments from Josuttis, *SOA in Practice*.)

5.1 Service-oriented architecture

- services

  self-contained, loosely coupled units of functionality; may participate in multiple processes, and be implemented with various technologies on various platforms

- infrastructure

  the *enterprise service bus*, for easy and flexible interoperation of services

- policies

  to deal with the fact that large distributed systems are heterogeneous, continuously under maintenance, have multiple owners

5.2 Large distributed systems

SOA is specifically for dealing with large distributed systems:

- legacy-laden
- heterogeneous
- long-lived
- complex
- under multiple ownership
- imperfect
- redundant (denormalized)
- intolerant of bottlenecks

5.3 Postmodern software engineering

Lots of software is already written and can solve your problems, if you can only get it to do what you need. *Postmodern programming* is about using code that already exists, and not writing much code yourself. This is usually in the form of gluing together or configuring other people's code. This is the reality of 'enterprise' software today. Postmodern programmers recognise this fact and work with it rather than pretend that it isn't the case and come up with, for example, a programming language or paradigm that would solve all programming problems once and for all if only everybody does everything properly—that way—the one true way. **There is no One True Way.**

`http://www.elvis.ac.nz/brain?PostmodernProgramming`

PoMoPro principles

- examples over documentation
- source code over binary components
- loosely structured data over highly structured data
- dynamic typing over static typing
- focused components over frameworks
- composing components over modifying existing applications
- rich component library over programming tools
- actual capabilities over intended use (POSIWID)
- simplify the problem over functional areas

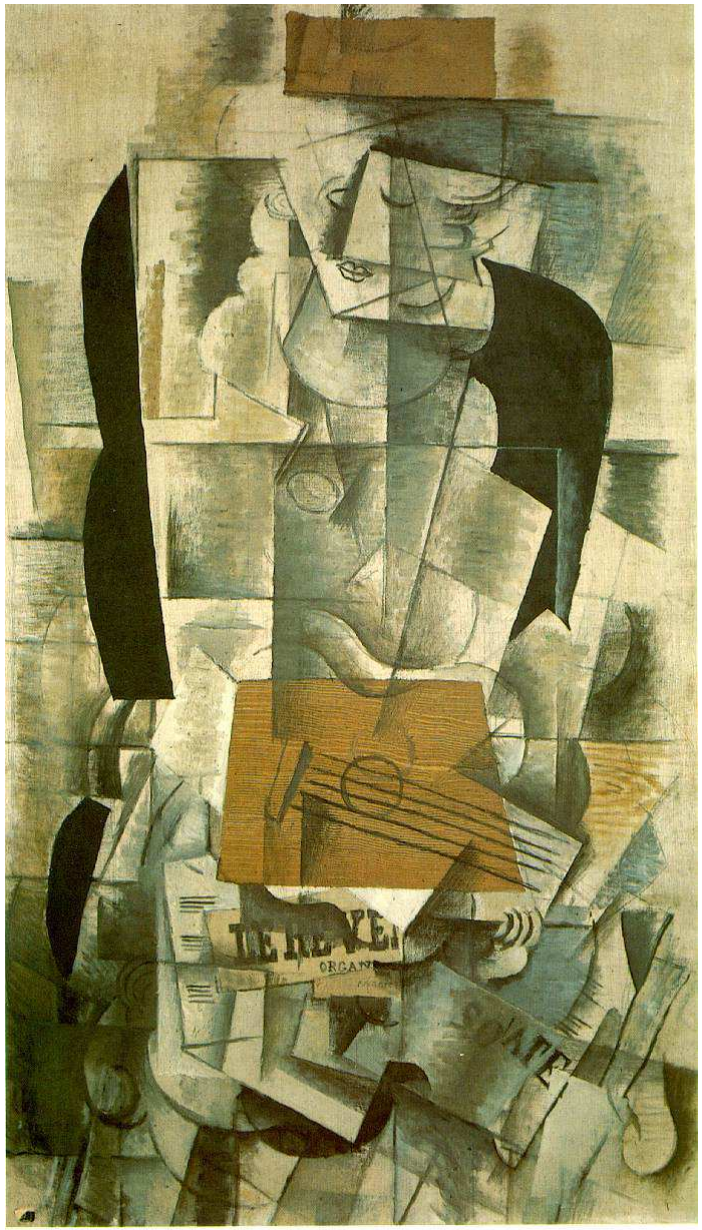
(Nat Pryce, Ivan Moore, *Scrapheap Challenge*, OOPSLA2005)

Modernist perspective

- absolute truth, right and wrong answers, one big story
- correctness is everything
- specification, stepwise refinement, proof

Postmodern perspective

- ‘incredulity towards meta-narratives’
- no one privileged view
- toleration of conflict and contradiction
- it’s all relative
- ‘good enough software’, ‘worse is better’



5.4 Some more definitions of services

loosely coupled software components that interact with each other dynamically via standard Internet technologies (Gartner)

a piece of business logic accessible via the Internet using open standards (Microsoft)

encapsulated, loosely coupled, contracted software functions, offered via standard protocols over the Web (DestiCorp)

a software application identified by a URI, whose interfaces and binding are capable of being defined, described and discovered by XML artifacts and supports direct interactions with other software applications using XML-based messages via Internet-based protocols (W3C)

5.5 Observations

- services are ideally *stateless*
- *reuse* has been subordinated to modularity
- *interfaces* are the crux

  (a Gartner analyst said that ‘SOA would be better named *interface-oriented architecture*’)

- signature → behaviour → service level

5.6 Service principles

- reusable
- contracted
- loosely coupled
- abstract
- composable
- autonomous
- stateless
- discoverable

(After Erl, *SOA*.)

5.7 SOA vs Web Services

- WS = WSDL + UDDI + SOAP (roughly speaking)
- service description, discovery, invocation
- using standard XML and Internet technologies
- just one approach to realizing the technical aspects of SOA
- SOA is not a technology!

- is service-orientation the same as component-orientation?

Index

Service-Oriented Architecture

Monday	Tuesday	Wednesday	Thursday	Friday
Introduction Components	REST	Composition	Architecture	Engineering
coffee	coffee	coffee	coffee	coffee
Components	REST	Composition	Architecture	Conclusion
lunch	lunch	lunch	lunch	lunch
Web Services	Qualities	Objects	Semantic Web	
tea	tea	tea	tea	
Web Services	Qualities	Objects	Semantic Web	