Arquitetura de Software
Parte I - Introdução

Marco Túlio Valente

mtov@dcc.ufmg.br

DCC - UFMG

Who needs and architect?

Martin Fowler,

IEEE Software,

July/August 2003

Introduction

- “Architect” and “architecture” are terribly overloaded words
- I define architecture as a word we use when we want to talk about design but want to puff it up to make it sound important
- IEEE: architecture is the highest level concept of a system.
  - The architecture of a software system is its organization or structure of significant components interacting through interfaces

Ralph Johnson's Definition

- I was a reviewer on the IEEE standard and I argued uselessly that this was clearly a completely bogus definition.
- There is no highest level concept of a system.
- <u>Customers</u> have a different concept than <u>developers</u>.
- An architecture is the <u>highest level concept that developers</u> have of a system in its environment.
- Let's forget the developers who just understand their little piece.
- Architecture is the highest level concept of the <u>expert developers</u>.

Ralph Johnson’s Definition

- So, a better definition would be:
  - The expert developers working on software projects have a <u>shared understanding</u> of the system design
  - This shared understanding is called architecture

- This understanding includes:
  - How the system is divided into <u>components</u>
  - How the components interact through <u>interfaces</u>

Ralph Johnson’s Definition

- Whether something is part of the architecture is entirely based on whether the <u>developers think it is important</u>
- People who build “enterprise applications” tend to think that persistence is crucial
  - They will mention “and we use Oracle for our database and have our own persistence layer to map objects onto it.”
- But a medical imaging application might include Oracle without it being considered part of the architecture
  - Fetching and storing images is done by one little part of the application and <u>most of the developers ignore it</u>

To Conclude

- “Tell us what is important.”
- Architecture is about the <u>important stuff</u>. Whatever that is.

The Architect's Role

- So if architecture is the important stuff, then:
  - The architect is the person (or people) who worries about the important stuff.

- And here we get to the essence of the difference between two "species" of architect

Architect’s Role #1

- The architect is the person who makes the <u>important decisions</u>
- The architect does this because:
  - A single mind is needed to ensure <u>conceptual integrity</u>
  - Perhaps the team members are not sufficiently skilled to make those decisions
- Often, such decisions must be made <u>early</u>
  - So that everyone else has a <u>plan</u> to follow

Architect’s Role #2

- This kind of architect must be very aware of what’s going on in the project, looking out for important issues and tackling them before they become a serious problem

- The most noticeable part of the work is the intense collaboration:
  - In the morning, the architect programs with a developer, trying to harvest some common locking code
  - In the afternoon, the architect participates in a requirements session, helping explain to the requirements people the technical consequences of some of their ideas

- The most important activity is to mentor the development team

What do software architects really do?

Philippe Kruchten

Journal of Systems and Software

Volume 81, Issue 12, December 2008

What do architects really do?

- "— Mr. Beck, what is software architecture?" asked a participant at an OOPSLA workshop in Vancouver, 1992.
- "— Software architecture?" Replied Kent: "well, it is what software architects do." (Chuckles in the audience.) "
- — So then, what is an architect?"
- — 'Hmm, 'software architect' it's a new <u>pompous title</u> that programmers demand to have on their business cards to justify their sumptuous emoluments."

Architects design the architecture

- Software architects should design, develop, nurture, and maintain the architecture of the software-intensive systems

- Architects' responsibilities are the part of both the design and the design decisions that have long-lasting impact on some of the major quality attributes of a software-intensive system:
  - Cost, evolution, performance, decomposability, safety, security etc

Responsibilities of architects

- Defining the architecture of the system
- Maintaining the architectural integrity of the system
- Assessing technical risks. Working out risk mitigation strategies.
- Participating in project planning.
- Consulting with design, implementation, and integration teams
- Assisting product marketing and future product definitions

Allocating time

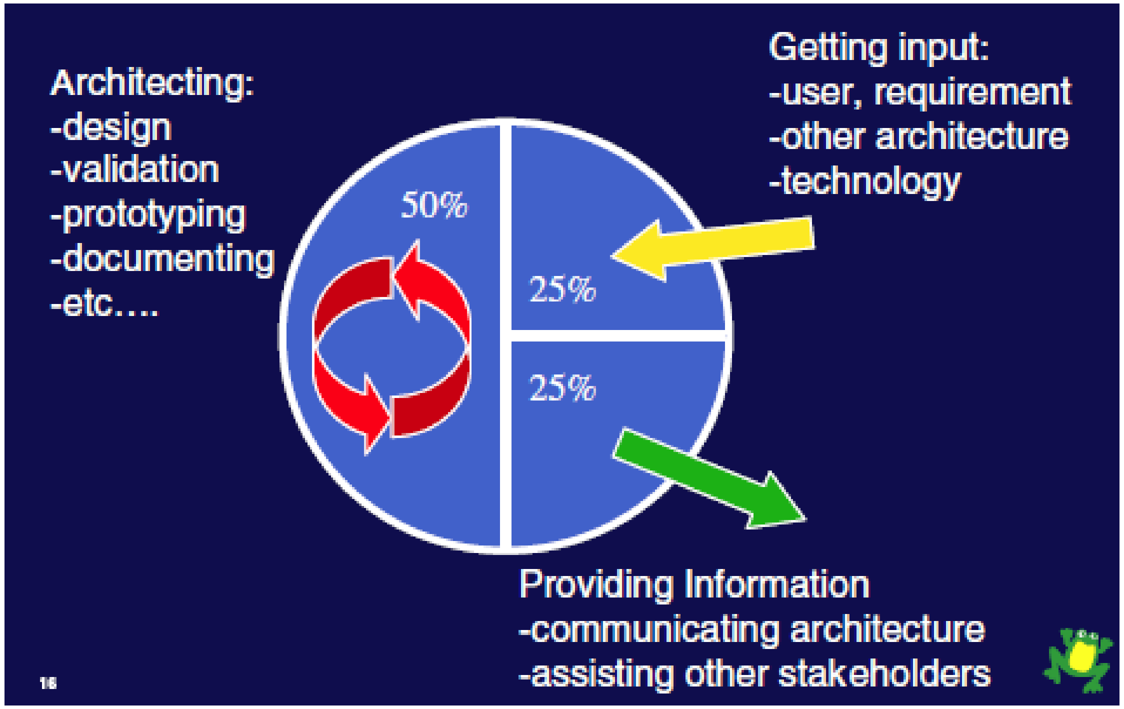


Fig. 1. What do architect really do?

Allocating time

- Internal focus (50%): focused on <u>architecting per se</u>: architectural design, prototyping, evaluating, documenting, etc.
- External focus (50%): <u>interacting</u> with other stakeholders:
  - Inwards (25%): getting input from the <u>outside world</u>, <u>listening</u> to customers, users, and other stakeholders.
  - Outwards (25%): <u>providing information</u> or help to other stakeholders or organizations

Antipattern: Goldplating

- A software architect who is <u>not communicating</u> regularly with the customer, the end users, or their representatives
- They are probably doing a <u>good technical job</u>, as they are getting <u>plenty of input</u>, but if they do not regularly provide value to their environment, their input will be too late and be ignored

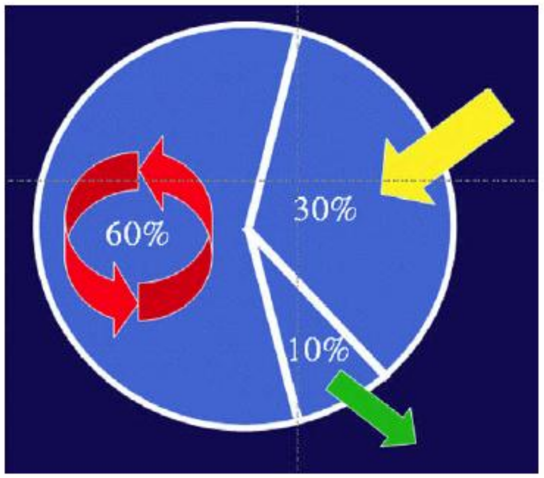


Fig. 2. The [60:30:10] antipattern – goldplating.

Antipattern: Ivory Tower

- Architecture team that lives isolated in some other part of the organization — another floor, another building — and who comes up after some months with a complete architecture

- They are not getting enough input from the users and developers, and they are not providing enough value to their organization (advocating the architecture, providing assistance to other teams)

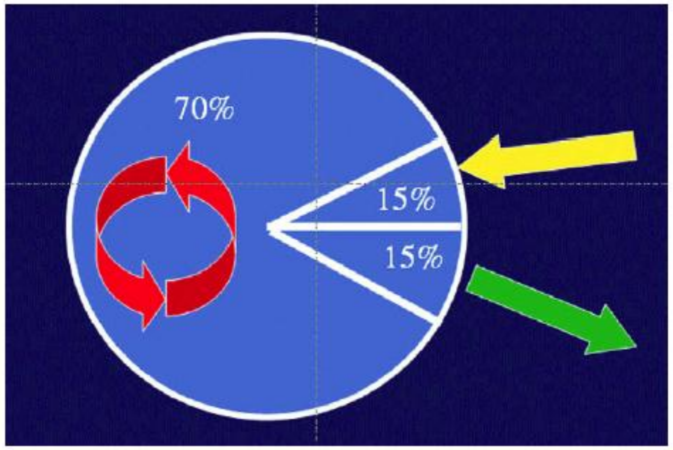


Fig. 3. The [70:15:15] antipattern – ivory tower.

Antipattern: The Absent Architects

- No or little architecture design progress is made: the architects are always away doing fascinating things or fighting fires

- This is a software architecture team that is spending far too much time traveling the world. Unless this is a very mature system, they will run into architectural difficulties.

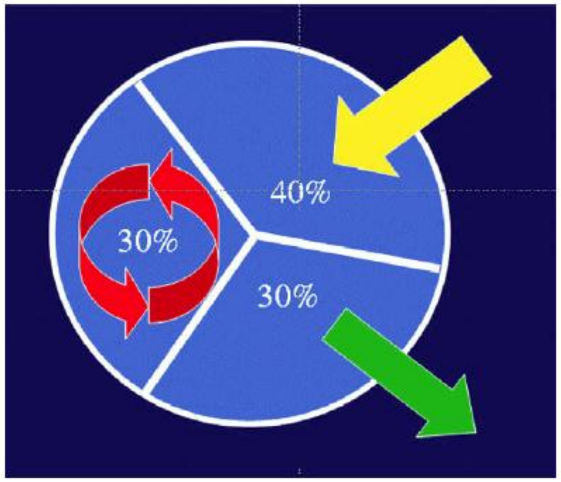


Fig. 4. The [30:40:30] antipattern – absent architect.

Antipattern:Just Consultants

- This is a software architecture team that is acting more as an <u>internal consulting shop</u>

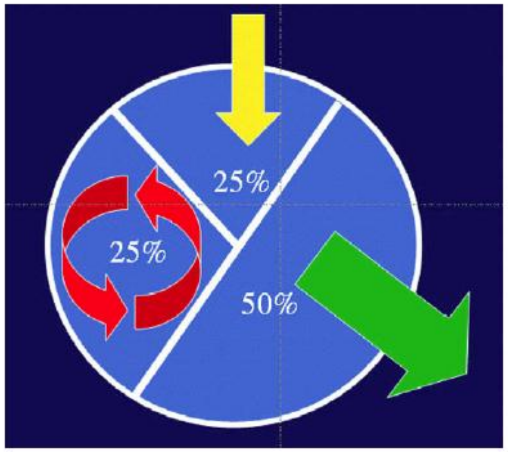


Fig. 5. The [25:25:50] antipattern – just consultants.

Tanenbaum-Torvalds Debate: Microkernel x Monolithic Systems

Open Sources: Voices from the Open Source Revolution (Appendix A)

O'Reilly - 1st Edition, January 1999

http://oreilly.com/catalog/opensources/book/appa.html

Tanenbaum’s Mail - 29 Jan 92

- Most older operating systems are monolithic:
  - The whole OS is a single a.out file that runs in 'kernel mode.'
  - This binary contains the process management, memory management, file system and the rest.
  - Examples: UNIX, MS-DOS, VMS, MVS

- The alternative: microkernel-based system
  - Most OS runs as separate processes, outside the kernel
  - They communicate by message passing
  - Kernel: interrupt handling, process management, and I/O
  - Examples: Amoeba, Mach

- The OS the debate is essentially over: microkernels have won

Torvalds' Reply - 29 Jan 92

- True, linux is monolithic...From a <u>theoretical</u> standpoint linux looses.
- If the GNU kernel had been ready last spring, I'd not have bothered to even start my project: the fact is that it wasn't and still isn't. Linux wins heavily on points of being <u>available now</u>
- If this was the only criterion for the "goodness" of a kernel, you'd be right
- What you don't mention is that minix doesn't do the micro-kernel thing very well, and has problems with multitasking
- If I had made an OS that had problems with a multithreading filesystem, I wouldn't be so fast to condemn others

Tanembaum’s Reply - 30 Jan 92

- I still maintain the point that designing a monolithic kernel in 1991 is <u>a fundamental error</u>.
- Be thankful you are not my student.
- You would not get a high grade for such a design :-)

Torvalds’ Reply - 30 Jan 92

- That's ok. Einstein got lousy grades in math and physics.

Ken Thompson’s Mail - 3 Feb 92

- I agree that microkernels are probably the wave of the future
- However, it is easier to implement a monolithic kernel
- It is also easier for it to turn into a mess in a hurry as it is modified

Torvalds’ Talk – LinuxCon 2009

- We're getting <u>bloated and huge</u>. Yes, it's a problem
- I mean, sometimes it's a bit sad that we are definitely not the streamlined, small, hyper-efficient kernel that I envisioned 15 years ago...
- And whenever we add a new feature, it only gets worse.

No Silver Bullet: Essence and Accidents of Software Engineering

Frederick P. Brooks, Jr

Computer Magazine, April 1987

Introduction

- Of all the monsters that fill the nightmares of our folklore, none terrify more than werewolves. For these, one seeks bullets of silver that can magically lay them to rest.

- The familiar software project has something of this character; it is usually innocent and straightforward:
  - But is capable of becoming a monster of missed schedules, blown budgets, and flawed products.
  - So we hear desperate cries for a <u>silver bullet</u>

- But, <u>we see no silver bullet</u>.
- There is no single development, in <u>either technology or in management technique</u>, that promises even one order-of-magnitude improvement in productivity, in reliability, in simplicity.

Essential and Accidental Difficulties

- Not only are there no silver bullets now in view, the very nature of software makes it unlikely that there will be any
  - No inventions that will do for software productivity, reliability, and simplicity what large-scale integration did for hardware.
- First, the anomaly is not that software progress is so slow, but that <u>hardware progress is so fast</u>
- Second, to see what rate of progress one can expect in software technology, let us examine the difficulties of that technology
- Following Aristotle, I divide them into:
  - <u>Essence</u>, the difficulties inherent in the nature of software
  - <u>Accidents</u>, difficulties attending its production but not inherent

Essential Difficulties

- Inherent properties of this irreducible essence of software:
  - Complexity
  - Conformity
  - Changeability
  - Invisibility

Complexity

- Software entities are **more complex** for their size than perhaps any other human construct
  - No two parts are alike (at least above the statement level).
  - If they are, we make the two similar parts into a subroutine.
  - In this respect, software differ profoundly from computers, buildings, or automobiles, where repeated elements abound

- A scaling-up of a software entity is not merely a repetition of the same elements in larger sizes;
  - It is an increase in the number of different elements.
  - The complexity of the whole increases more than linearly

Complexity

- Hence, descriptions of a software entity that abstract away its complexity often abstract away its essence.

- Mathematics and the physical sciences made great strides by constructing <u>simplified models</u> of complex phenomena
  - Deriving properties from the models
  - Verifying those properties by experiments

- This paradigm worked because the complexities ignored in the models were not the essential properties of the phenomena

- It does not work when the complexities are the essence

Conformity

- Much of the complexity that software must master is arbitrary complexity, forced by the many human institutions and systems to which his interfaces must conform.

- In many cases, the software must conform because it is the most recent arrival on the scene.

- It must conform because it is perceived as the most conformable

Change

- The software entity is constantly subject to pressures for change
  - So are buildings, cars, computers.
  - But manufactured things are infrequently changed after manufacture; they are superseded by later models

- In part it is because software can be changed more easily -- it is pure thought-stuff, infinitely malleable.
  - Buildings do in fact get changed, but the high costs of change serve to dampen the whims of the changes.

- All <u>successful</u> software gets changed:
  - First, as a software is found useful, people try it in new cases
  - Second, successful software survives beyond the normal life of the machine vehicle for which it is first written

Invisibility

- Geometric abstractions are powerful tools to tackle complexity
  - The floor plan of a building helps both architect and client evaluate spaces, traffic flows, views.

- However, software is not inherently embedded in space

- When we attempt to diagram software structure, we find it to constitute not one, but several, general directed graphs superimposed one upon another, representing:
  - The flow of control
  - The flow of data
  - Patterns of dependency

Architectural Blueprints: The “4+1” View Model of Software Architecture

Philippe Kruchten (Rational Soft., now UBC, Canada)
IEEE Software, 1995

Introduction

- We have seen many books where <u>one diagram</u> attempts to capture the gist of the architecture of a system

- Their authors have struggled hard to represent more on one blueprint than it can actually express
  - Are the boxes representing running programs? Or chunks of source code? Or physical computers?
  - Are the arrows representing compilation dependencies? Or control flows? Or data flows?
  - Usually it is a bit of everything

- As a remedy, we propose to organize the description of a software architecture using several <u>concurrent views</u>
  - Each one addressing one <u>specific set of concerns</u>

4+1 Views

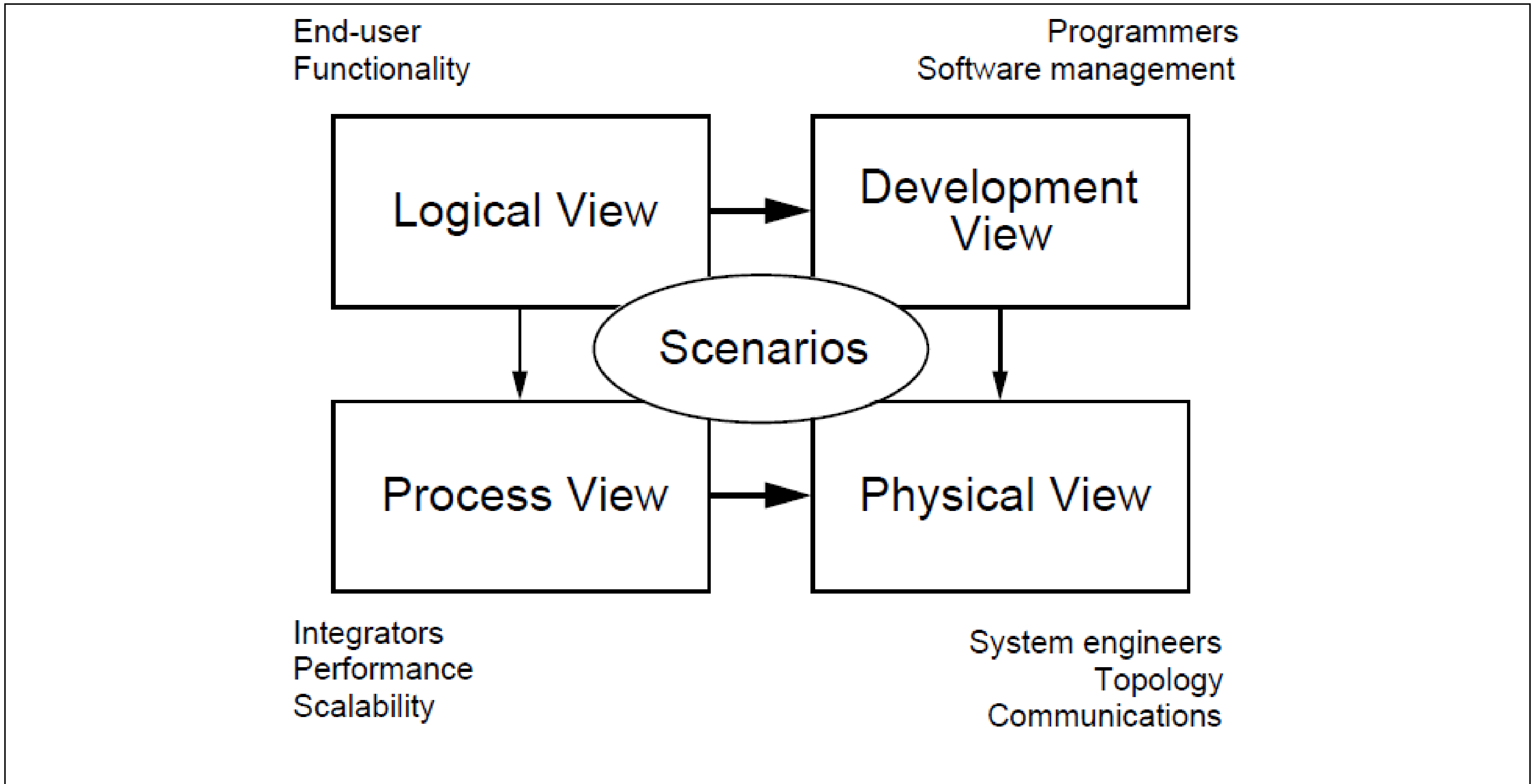


Figure 1 — The “4+1” view model

4+1 Views

1. **Logical**: what the system should provide in terms of services

2. **Process**: concurrency and synchronization aspects of the design

3. **Physical**: mapping(s) of the software onto the hardware

1. **Development**: static organization of the software

- The description of an architecture can be organized around these four views
  - And then illustrated by a few selected **scenario**s (fifth view)

Logical View

- The logical architecture primarily supports the functional requirements — what the system should provide in terms of services to its users

- The system is decomposed into a set of key abstractions, taken from the problem domain

- Class diagrams are used to model the logical architecture
    - Considering only items that are architecturally significant

Logical View

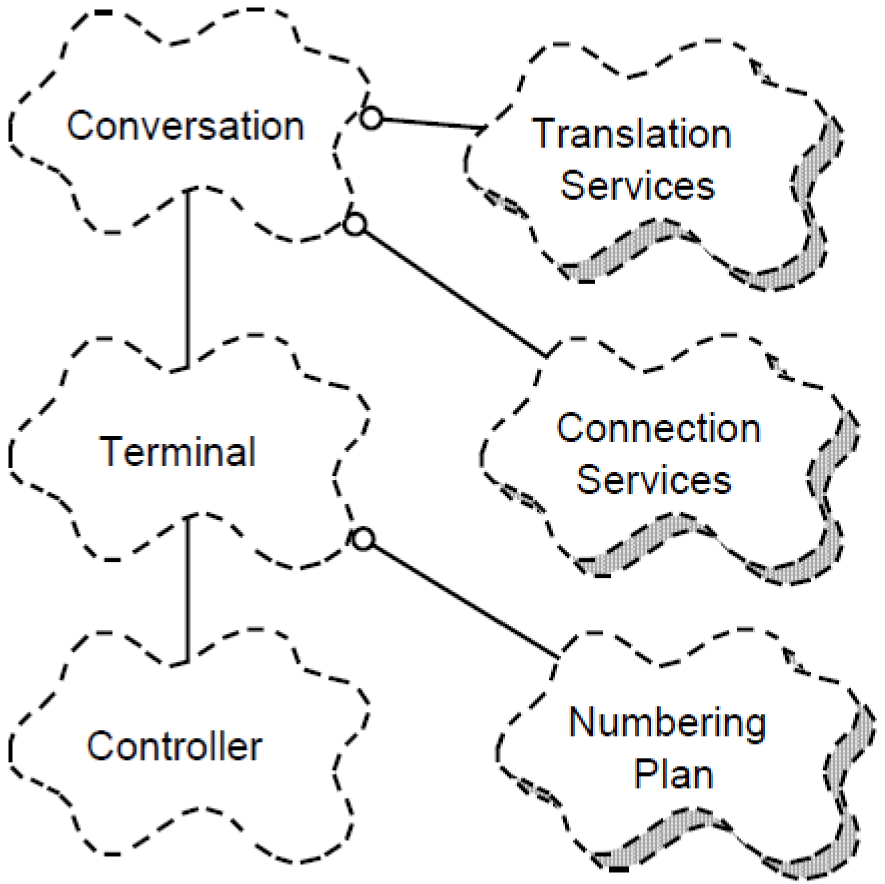


Figure 3— a. Logical blueprint for the Télic PABX

A PABX establishes commmunications between terminals

The controller object decodes and injects all the signals on the line interface card

The terminal object maintains the state of a terminal, and negotiates services on behalf of that line

For example, it uses the services of the numbering plan to interpret the dialing in the selection phase

The conversation represents a set of terminals engaged in a conversation.

The conversation uses translation services (directory, logical to physical address mapping, routes), and connection services to establish a voice path between the terminals

Process View

- The process view takes into account non-functional requirements
  - Performance, concurrency, distribution, fault-tolerance

- The process architecture can be viewed as a set of independently logical “processes”, distributed across a set of hardware resources

- A process is a grouping of tasks that form an executable unit

Process View

Example of a Process blueprint

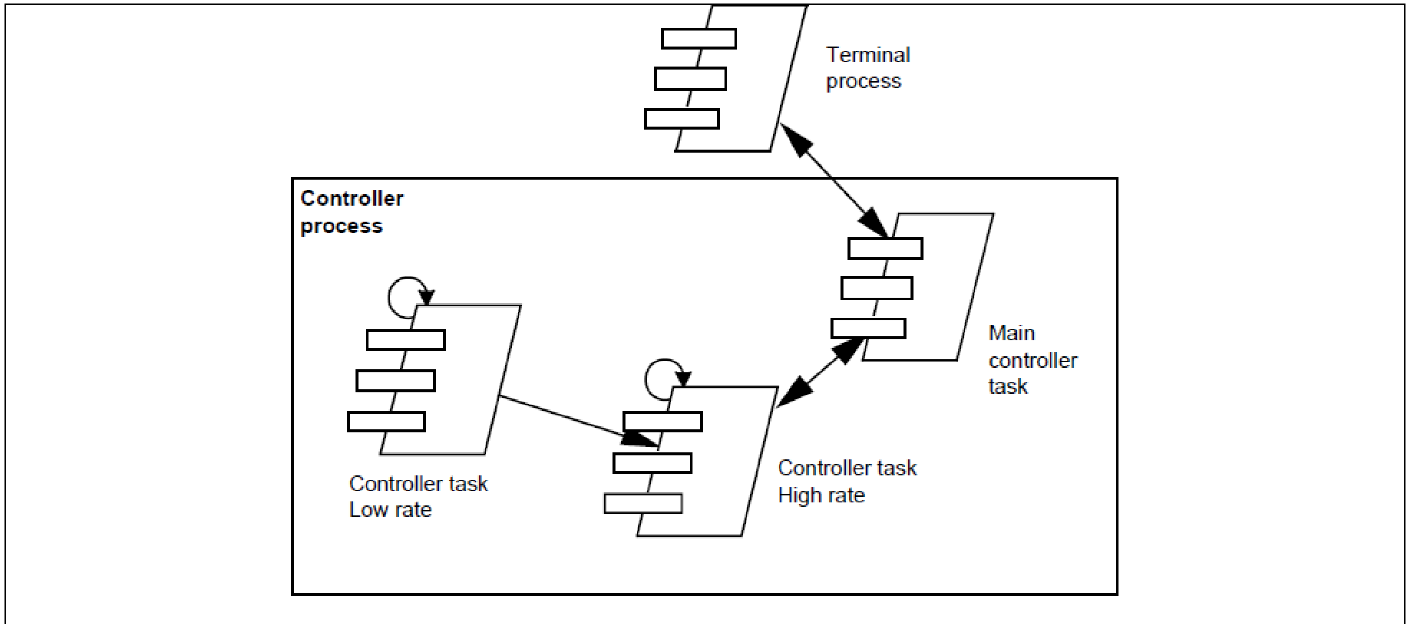


Figure 5 — Process blueprint for the Télic PABX (partial)

All terminals are handled by a single *terminal process*. The controller objects are executed on one of three tasks that composes the controller process: a *low cycle rate task* scans all inactive terminals (200 ms), puts any terminal becoming active in the scan list of the *high cycle rate task* (10ms), which detects any significant change of state, and passes them to the *main controller task*

Development View

- The development architecture focuses on the actual software module organization
- The software is packaged subsystems that can be developed by one or a small number of developers
- Subsystems are organized in a hierarchy of layers, each layer providing a well-defined interface to the layers above it
- The development view serves for:
  - Requirement allocation
  - Allocation of work to teams
  - Monitoring the progress of the project
  - Reasoning about software reuse, portability and security

Development View

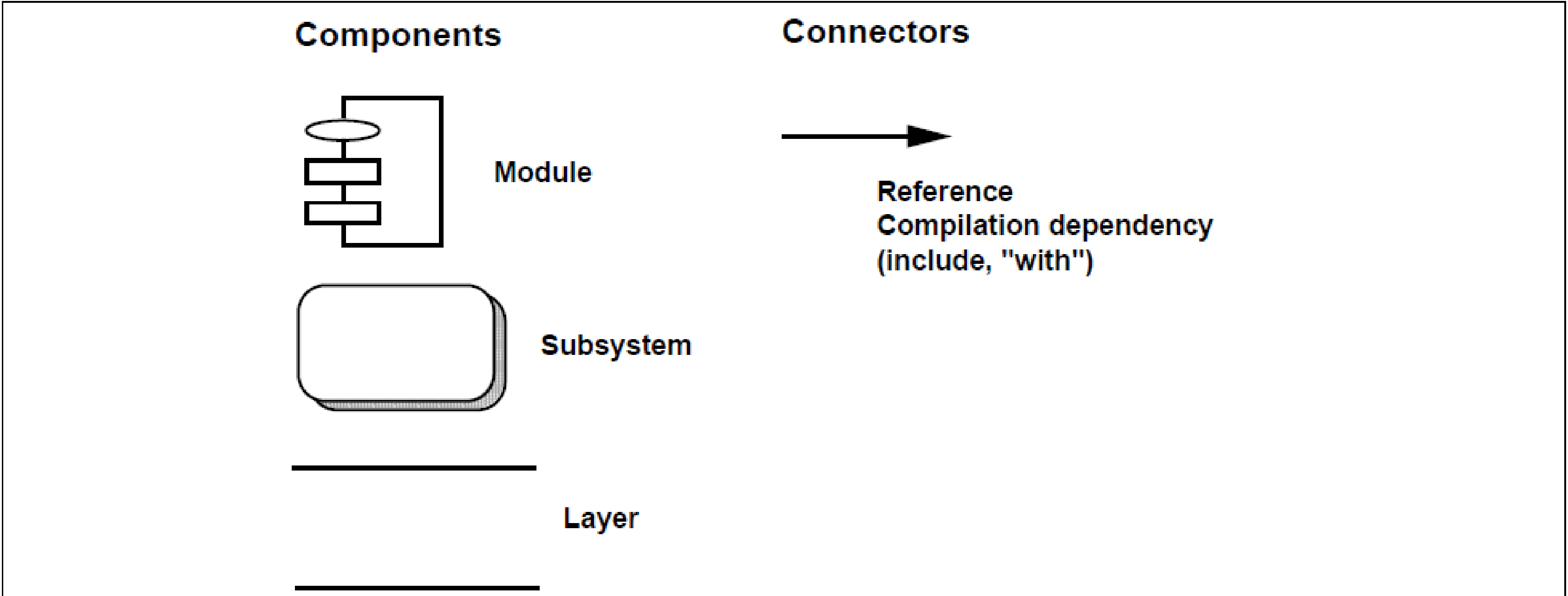


Figure 5 — Notation for the Development blueprint

Physical View

- The software executes on a network of computers, or processing nodes (or just nodes for short)
- The various elements identified — networks, processes, tasks, and objects — need to be mapped onto the various nodes

Physical View

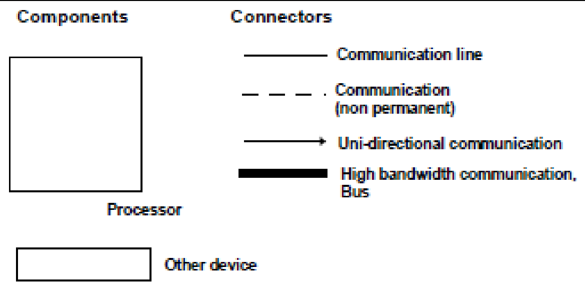

Components
Connectors
Communication line
Communication (non permanent)
Uni-directional communication
High bandwidth communication, Bus
Processor
Other device

Figure 7 — Notation for the Physical blueprint

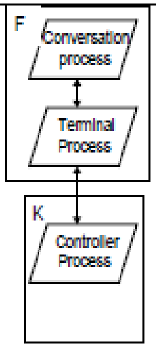


Figure 9 — A small PABX physical architecture with process allocation

Physical View

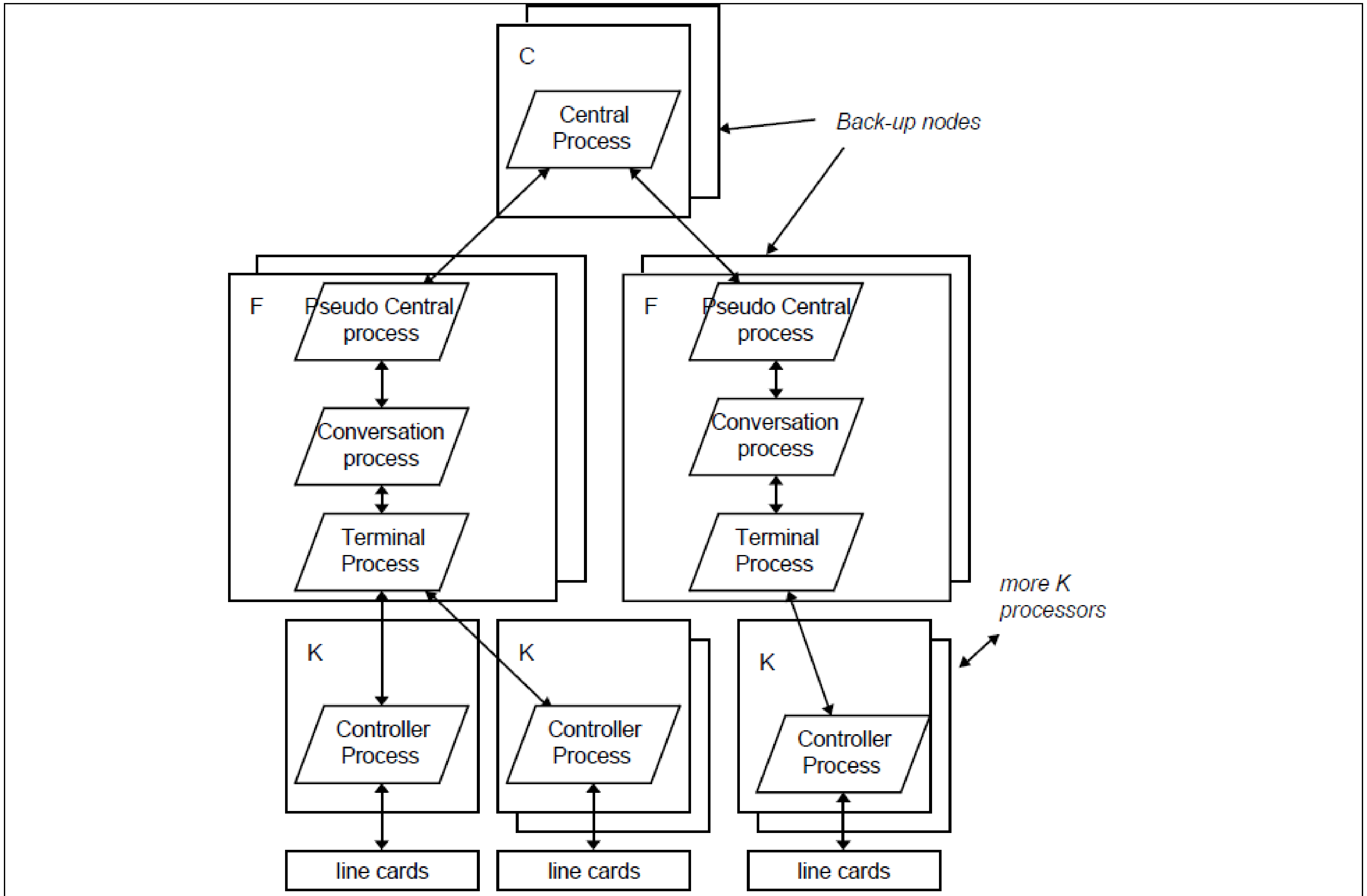


Figure 10 — Physical blueprint for a larger PABX showing process allocation

Summary

- Architecture Views:
  - **Logical**: object-oriented decomposition (functionality)
  - **Process**: process decomposition
  - **Development**: subsystem decomposition
  - **Physical**: mapping the software to the hardware

Scenarios: Putting it all together

- The elements in the four views are shown to work together by the use of a small set of important scenarios

- This view serves two main purposes:
  - As a driver to discover the architectural elements during the architecture design
  - As a validation and illustration role after this architecture design is complete

Scenario Example

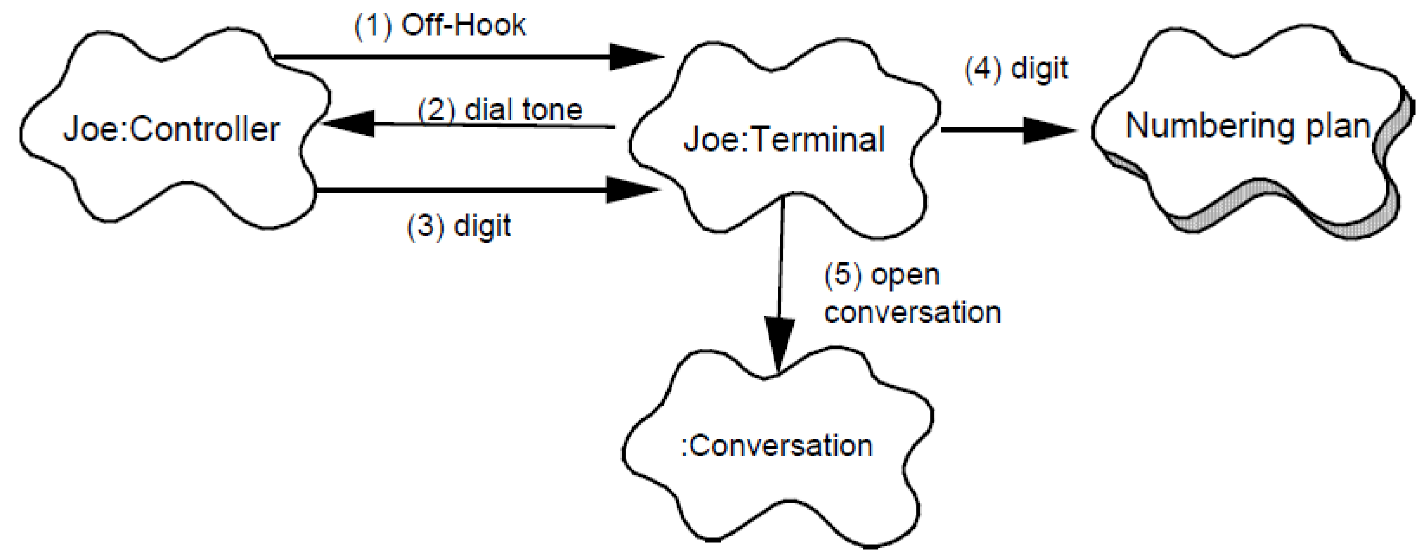


Figure 11 — Embryo of a scenario for a local call—selection phase

1. The controller of Joe's phone detects and validate the transition from on-hook to off-hook and sends a message to wake up the corresponding terminal object.
2. The terminal allocates some resources, and tells the controller to emit some dial-tone.
3. The controller receives digits and transmits them to the terminal.
4. The terminal uses the numbering plan to analyze the digit flow.
5. When a valid sequence of digits has been entered, the terminal opens a conversation

Tailoring the Model

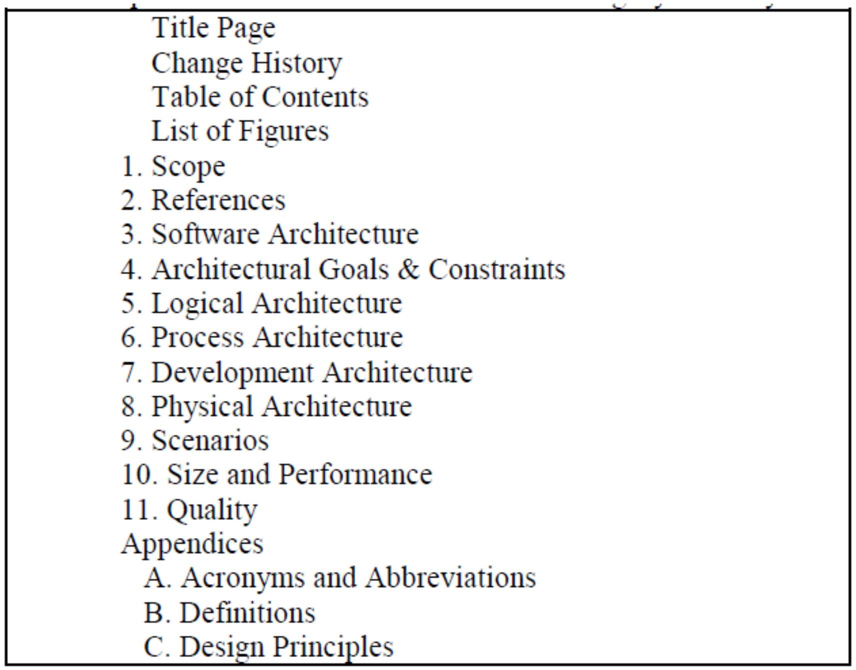
- Not all software architecture need the full “4+1” views.

- Views that are useless can be omitted from the description
  - Such as the physical view, if there is only one processor
  - The process view if there is only process or program
  - For very small system, it is even possible that the logical view and the development view are so similar that they do not require separate descriptions.

- The scenarios are useful in all circumstances.

Documenting the Architecture

- The documentation produced during the architectural design is captured in two documents:
  - A Software Architecture Document, whose organization follows closely the “4+1” views



Title Page
Change History
Table of Contents
List of Figures
1. Scope
2. References
3. Software Architecture
4. Architectural Goals & Constraints
5. Logical Architecture
6. Process Architecture
7. Development Architecture
8. Physical Architecture
9. Scenarios
10. Size and Performance
11. Quality
Appendices
A. Acronyms and Abbreviations
B. Definitions
C. Design Principles

Figure 13 — Outline of a Software Architecture Document

Conclusions

- This “4+1” view model has been used with success on several large projects

- It actually allowed the various stakeholders to find what they want to know about the software architecture.
  - Systems engineers approach it from the Physical view, then the Process view.
  - End-users, customers, data specialists from the Logical view
  - Project managers, software configuration staff see it from the Development view

Jeff Bezos (Amazon’s CEO) Mandate

From: Stevey's Google Platforms Rant

https://plus.google.com/112678702228711889851/posts/eVeouesvaVX

Jeff Bezos Mandate

1. All teams will henceforth expose their data and functionality through service interfaces.
2. Teams must communicate with each other through these interfaces.
3. There will be no other form of interprocess communication allowed: no direct linking, no direct reads of another team's data store, no shared-memory model, no back-doors whatsoever. The only communication allowed is via service interface calls over the network.
4. It doesn't matter what technology they use. HTTP, Corba, Pubsub, custom protocols -- doesn't matter. Bezos doesn't care.

Jeff Bezos Mandate

5. All service interfaces, without exception, must be designed from the ground up to be externalizable. That is to say, the team must plan and design to be able to expose the interface to developers in the outside world. No exceptions.

6. Anyone who doesn't do this will be fired.

7. Thank you; have a nice day!

Stevey's Google Platforms Rant

- Google+ is a prime example of our complete failure to understand platforms
- The Golden Rule of Platforms, "Eat Your Own Dogfood", can be rephrased as "Start with a Platform, and Then Use it for Everything."
- Certainly not easily at any rate -- ask anyone who worked on platformizing MS Office.
- If you delay it, it'll be ten times as much work as just doing it correctly up front.

Stevey's Google Platforms Rant

- You can't cheat.
  - You can't have secret back doors for internal apps to get special priority access, not for ANY reason.
  - You need to solve the hard problems up front.

- I'm not saying it's too late for us, but the longer we wait, the closer we get to being Too Late.

Modularization of a Large-Scale Business Application: A Case Study

Santonu Sarkar (InfoSys) et al.

IEEE Software, March/April 2009

Motivation

- Throughout their evolution, software systems are subject to repeated debugging and feature enhancements.

- Consequently, they gradually <u>deviate from the intended architecture</u> and <u>deteriorate into unmanageable monoliths</u>

- To contend with this, practitioners often:
  - Rewrite the entire application in a new technology
  - Invest considerable time in documenting the code and training new engineers to work on it

- However, for very large systems, such approaches are typically impossible to carry out

Goal

- We adopted a modularization approach to:
  - A banking application that was developed in the late ’90s to serve a single bank’s requirements
  - Had expanded to power more than 100 large installations across more than 50 countries

- The application — which grew from 2.5 million to 25 million LOC (MLOC) — has endured more than 10 mainline releases and is supported by several hundred engineers

- The original application was programmed in C, with functional enhancements in Java, JavaScript, and JavaServer Pages

- In this case study, we describe the modularization approach we adopted to address this situation

Problem Analysis

- We observed the following maintenance problems:
  - Learning time for new employees was continuously growing, more than doubling — from three months to almost seven months — over the past five years
  - Despite code reviews and testing, bug fixes invariably introduced new problems due to incomplete impact analysis
  - Product extendability for even simple features was taking an inordinately long time.
  - Even when we enhanced only one functional area, we had to test the entire application before the release. Consequently, partial deployment was often impossible

Root Causes

- RC1. Over time, developers had created shared libraries as assorted collections of business functions irrespective of their domain modules.
  - Almost 60% of the shared library code had this problem
- RC2. Developers had mixed up presentation and business logic in the code; presentation logic (about 2% to 3% of the code) was spread across the entire code base
- RC3. Lower-granularity functions, such as date validation, existed in the same library — and sometimes in the same source-code file — as complex domain functions, such as interest calculation.
  - That is, the system didn’t have a layered architecture

Root Causes

- RC4. Code wasn’t organized by functional domain, such as interest calculation, loan, or user account
  - One directory, named “sources,” had more than 13,000 C files and 1,500 user-interface-related files

- RC5. Functional modules with clearly defined module APIs were nonexistent.
  - A quick manual inspection of a part of the application showed that 5% of potential API functions were duplicates

Previous Attempts

- Prior to adopting our solution, management had aimed for operational efficiencies by offering:
  - Product-specific training to newcomers
  - Creating extensive documentation
  - Using pair programming
  - And so on.

Solution Overview

- The solution task force therefore proposed a long-term solution:
  - Restructure the entire system by creating a set of independently compilable and deployable domain modules
  - While keeping the same programming language and platform

- Proposed solution:
  - Modular Design Guidelines
  - Intermodule Interaction
  - Layered Module organization

Modular Design Guidelines

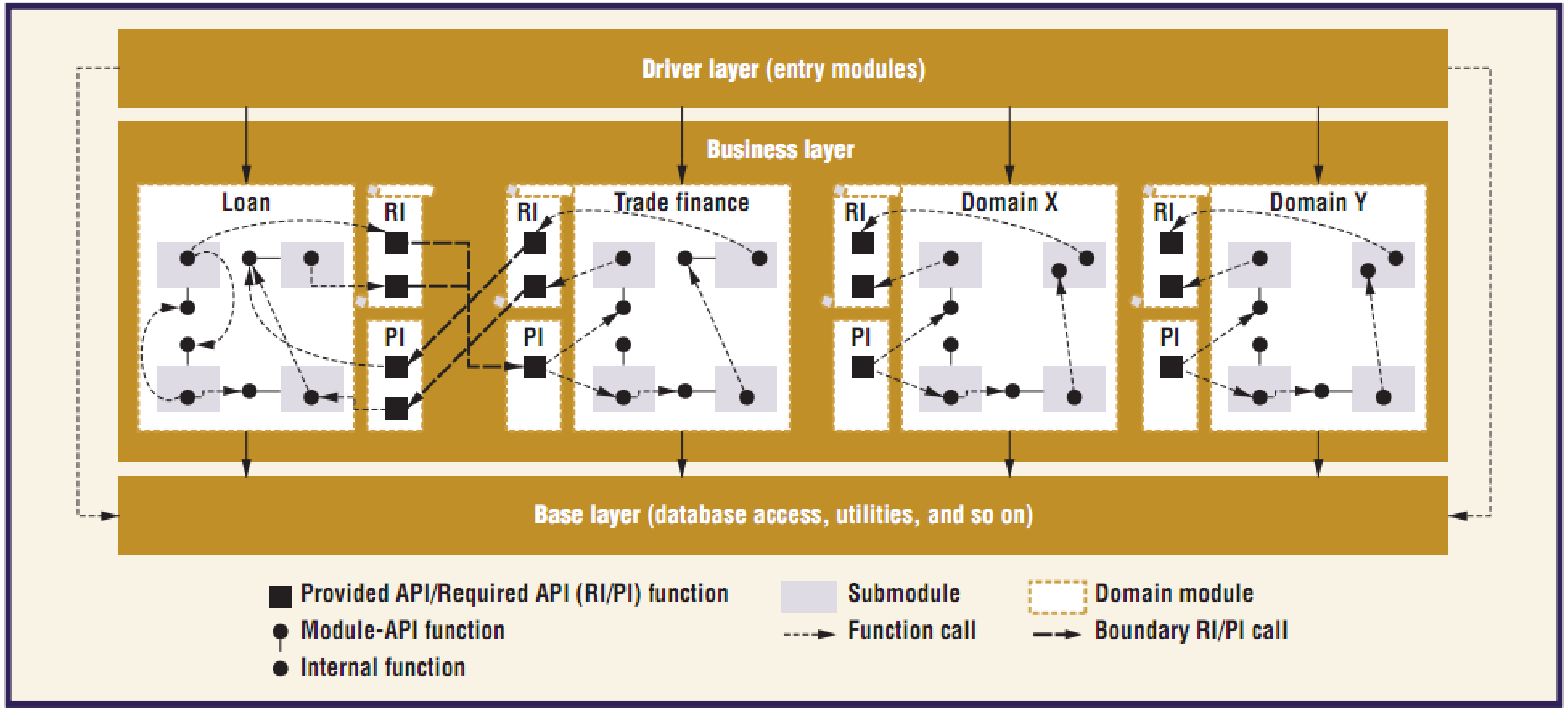
- The task force defined an overall guideline to identify modules on the basis of domain concepts.
  - Such as interest calculation, loans, account, calendar, etc
- The task force also suggested two additional guidelines:
  - Developers <u>shouldn’t create a module consisting of logically unrelated</u> services, such as date format conversion, credit-card issue, and user profile creation
  - Developers <u>should limit the sharing of data structures and function definitions across modules</u> so that they can build and test modules more independently

Intermodule Interaction

- The solution task force recommended that each domain module define two types of interfaces:
  - Provided API (PI): services that the module implements
  - Required API (RI): services that the module needs

- Modules residing:
  - In the same layer can communicate with each other through their PI–RI infrastructure
  - Layers at upper levels can communicate directly with the layers below
  - Layers at lower levels should not communicate with modules in higher layers

Three Layer Architecture



The driver layer (3% of the system) contains modules that offer entry points into the application. The business layer (64%) is divided into three domain sublayers. The base layer (33%) consists of modules that provide infrastructure support, such as database access and general purpose utilities

Modularization Approach

- We used domain-related naming conventions for file names
    - Example: files loanXXX.cxx

- We decomposed a domain module's artifacts into submodules — such as securitization and corporate loans in the loan domain

Intermodule Interaction

- We identified the PI for each module as follows.
- First, we identified all the intermodule calls
- Next, we classified a function as PI if it
  - Supports domain-specific queries by fetching domain data
  - Validates domain data
  - Supports domain-specific processing
- In one loan-related module we found that a particular date-formatting functionality became accessible to other modules
  - Such a functionality shouldn’t exist in a loan related module
  - We removed this functionality from the loan-related module and put it into a date module in the lower architectural layer

- After we’d identified a domain module’s PI, we identified a set of RI functions.

Complex Scenarios

1. Many modules were strongly coupled through global variables.
   - We encapsulated the global variables in functions to ensure that data was passed only through function parameters

2. A single function containing the logic of multiple domains.
   - When a function, such as **open_account()**, contained business logic pertaining to different domains — such as opening a deposit account and opening a loan account — we split the function into **open_deposit_account()** and **open_loan_account()** and assigned them to the respective domains
   - When domain logic was intertwined with a utility functionality (such as audit trail or calendar manipulation), we moved the utility functionality to functions in the lower utility layer

Complex Scenarios

3. Generic business operations such as interest calculation are applicable to multiple domains, such as loan, trade finance, etc
   - Consequently, functions implementing interest calculation contained all the applicable domain-specific nuances
   - We split these generic operations into multiple domain-specific operations

4. We unearthed a complex intertwining of business logic and external environment integration logic.
   - For example, the loan domain’s interest calculation logic was checking whether the interest calculation had been invoked from the batch environment or online
   - We therefore introduced a driver layer in the architecture
   - We then restructured the interest calculation functionality and delegated infrastructure-related tasks to the driver layer

Modularization Quantitative Analysis

- To manage the modularization, we considered <u>only 12.5 MLOC</u> from the entire product for the first phase.

- Implementing the design guidelines and modularizing 7 MLOC (56 percent of the 12. 5 MLOC) required nearly two years:
  - About 520 person-days for design
  - 2,100 person-days for coding and preliminary testing.
  - At times, the project went dormant; at its peak, it had 13 staff members

- The modularized system comprises:
  - 10 newly created/extracted domain modules
  - about 52 submodules.

Benefits

- Fault localization:
  - 50% effort reduction in localizing simple faults
  - 20% to 25% reduction for complex ones
- Testing: 5% reduction in detected defects
- Memory requirements: 43% less memory to load
- Load-time: 30% improvement
- Build time: 30% reduction

Conclusions

- Overall, our modularization project has been a success.
    - The application has now existed in a stable state in the production environment for one year

- Our next modularization phase aims to extract modules from the remaining 44% of the system's code

On the Criteria to Be Used in Decomposing Systems into Modules

David L. Parnas

Communications of the ACM,

December 1972

Introduction

- Usually nothing is said about the criteria to be used in dividing the system into modules.

- This paper suggest some criteria which can be used in decomposing a system into modules.

- What is modularization?
  - “module" = responsibility assignment
  - (rather than a subprogram)

Benefits of Modular Programming

- **Managerial**: development time should be shortened because separate groups would work on each module with little need for communication

- **Product Flexibility:** it should be possible to make drastic changes to one module without a need to change others;

- **Comprehensibility:** it should be possible to study the system one module at a time.

Example: KWIC System

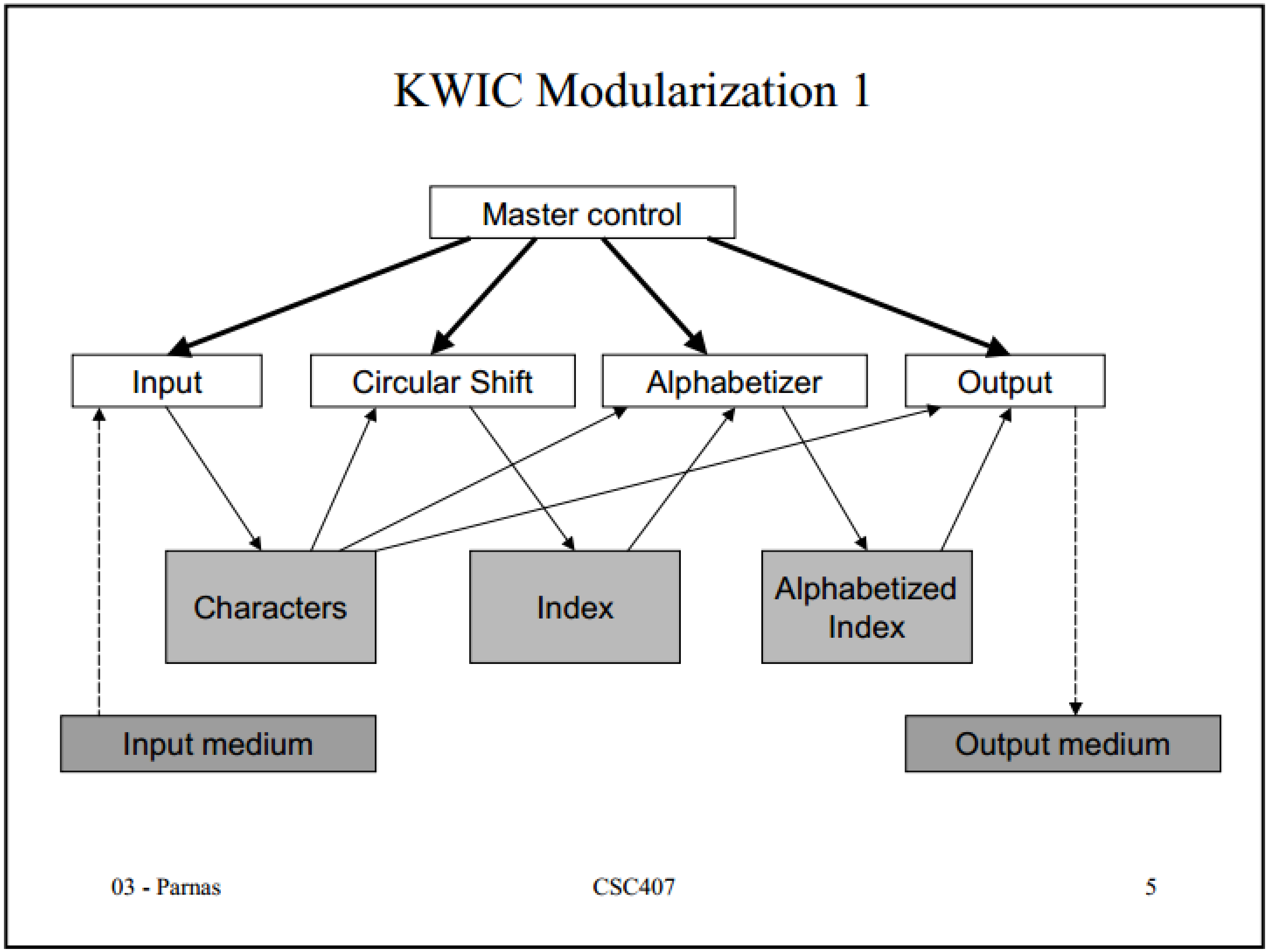
- Input: an ordered set of lines
    - Each line is an ordered set of words
    - Each word is an ordered set of characters.
- Any line may be "circularly shifted" by repeatedly removing the first word and appending it at the end of the line.
- Output:
    - A listing of all circular shifts of all lines in alphabetical order

KWIC Example

- Input:
  - Pattern-Oriented Software Architecture
  - Software Architecture
  - Introducing Design Patterns

- Output
  - Architecture Software
  - Architecture Pattern-Oriented Software
  - Design Patterns Introducing
  - Introducing Design Patterns
  - Patterns Introducing Design
  - Pattern-Oriented Software Architecture
  - Software Architecture
  - Software Architecture Pattern-Oriented

Modularization #1

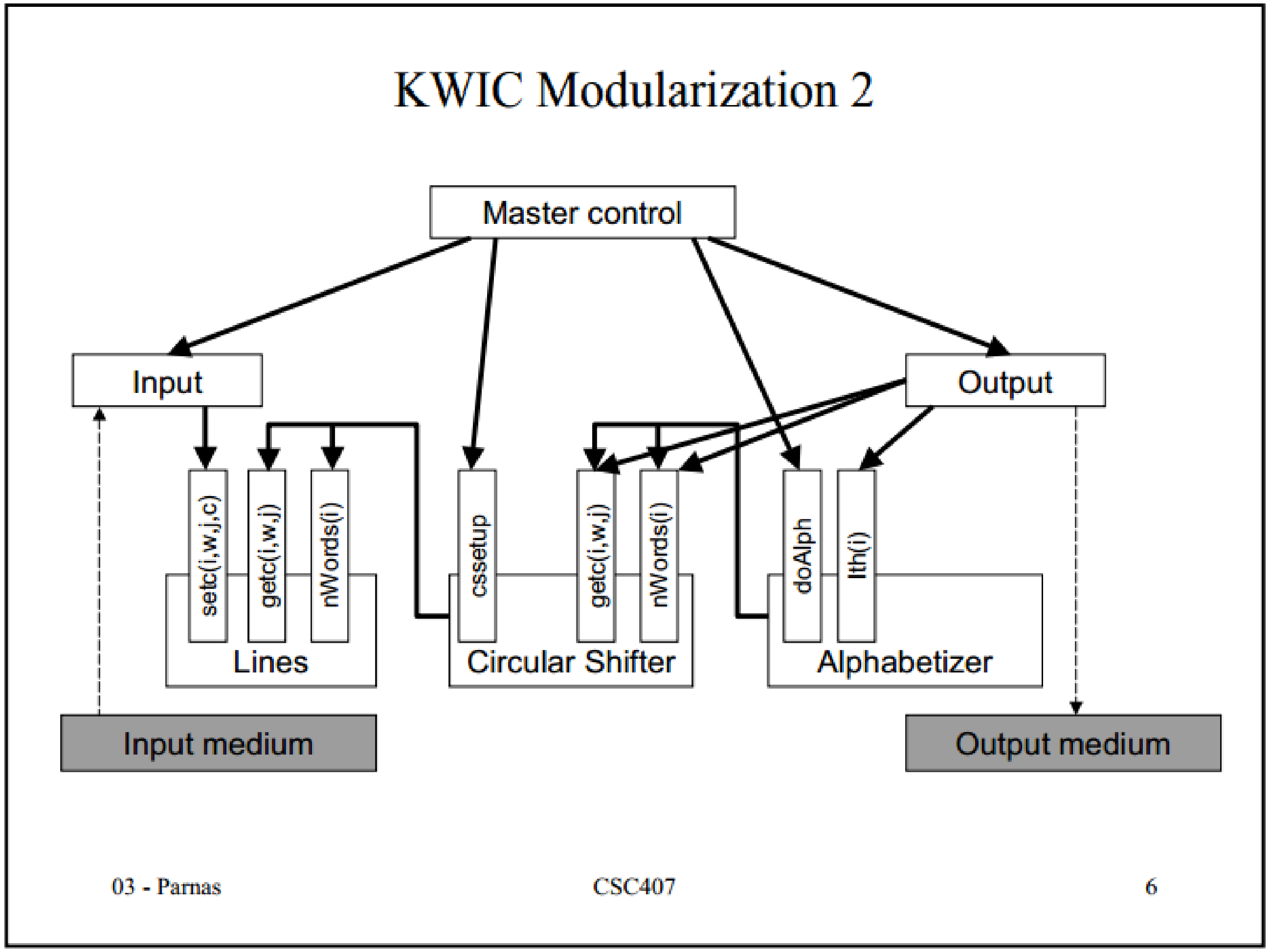

KWIC Modularization 1

03 - Parnas CSC407 5

Modularization #1

- **Input.** This module reads the data lines from the input medium and stores them in core for processing by the remaining modules
- **Circular Shift.** This module prepares an index which gives the address of the first character of each circular shiff
- **Alphabetizing.** This module produces an array in the same format as that produced by module 2. In this case, however, the circular shifts are listed in another order (alphabetically).
- **Output.** Using the arrays produced by module 3 and module 1, this module produces a formatted output listing all of the circular shifts

Modularization #2



Modularization #2

- **Lines Storage**:
  - *CHAR(r,w,c):* an integer representing the c-th character in the r-th line, w-th word
  - *SETCHAR(rpv,c,d):* causes the c-th character in the w-th word of the r-th line to be the character represented by d
  - *WORDS(r)* returns as value the number of words in line *r*.

- **Circular Shifter**:
  - The module creates the impression that we have a line holder containing all the circular shifts of the lines.
  - *CSCHAR(l,w,c)* provides the value representing the c-th character in the w-th word of the l-th circular shift

Comparison

- Modularization #1:
  - Each major step in the processing was a module

- Modularization #2: **information hiding** / abstract data types
  - Each module has one or more "secrets”
  - Each module is characterized by its knowledge of design decisions which it hides from all others.

Changeability

- Design decisions likely to change under many circumstances.
  1. Input format
  2. The decision to have all lines stored in core
  3. The decision to pack the characters four to a word
  4. The decision to make an index for the circular shifts rather that actually store them as such

- Differences between the two modularizations:
  - Change #1: confined to one module in both decompositions.
  - Change #2: for mod #1, changes in every module!
  - Change #3: for mod #1, changes in every module!
  - Change #4: confined to the circular shift module in the 2nd decomposition, but in the 1st decomposition the alphabetizer and the output routines will also change.

Independent Development

- In the first modularization the interfaces between the modules are the fairly complex formats and table organizations.

- In the second modularization the interfaces are more abstract.
  - They consist primarily in the function names and the numbers and types of the parameters.
  - These are relatively simple decisions and the independent development of modules should begin much earlier.

Comprehensibility

- To understand the output module in the first modularization, it will be necessary to understand something of the alphabetizer, the circular shifter, and the input module.

- The system will only be comprehensible as a whole.

- It is my subjective judgment that this is not true in the second modularization.

Conclusion

- We have tried to demonstrate by these examples that it is almost always incorrect to begin the decomposition into modules on the basis of a flowchart.

- We propose instead that one begins with a list of difficult design decisions or design decisions which are likely to change.

- Each module is then designed to hide such a decision from the others.

Later Comments

- To a man with a hammer, everything looks like a nail.
- To a Computer Scientist, everything looks like a language design problem.
- Languages and compilers are, in their opinion, the only way to drive an idea into practice.
- My early work clearly treated modularisation as a design issue, not a language issue. A module was a work assignment, not a subroutine or other language element.
- Although some tools could make the job easier, no special tools were needed to use the principal, just discipline and skill.