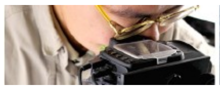
Software Architecture

2016

Software Architecture

- The quality and longevity of a software-reliant system is largely determined by its architecture.
- Recent US studies identify architectural issues as a systemic cause of software problems in government systems (OSD, NASA, NDIA, National Research Council).

Architecture is of enduring importance because it is the right abstraction for performing ongoing analyses throughout a system's lifetime



Software Architecture Thinking

- High-level system design providing system-level structural abstractions and quality attributes, which help in managing complexity
- Makes engineering tradeoffs explicit

Quality Attributes

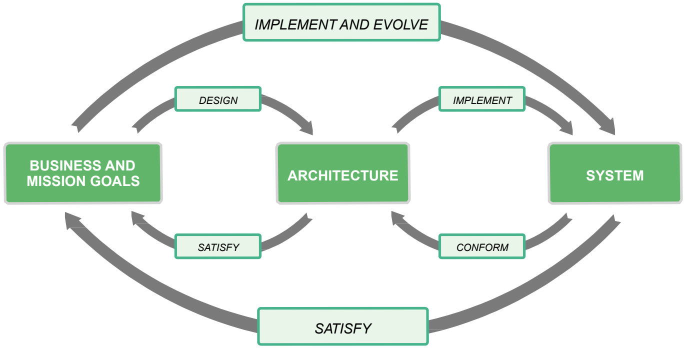
Quality attributes

- properties of work products or goods by which stakeholders judge their quality
- stem from business and mission goals.
- need to be characterized in a system-specific way

Quality attributes include

- Performance
- Availability
- Interoperability
- Modifiability
- Usability
- Security
- Etc.

Central Role of Architecture



One View: Architecture-Centric Engineering

- *explicitly focus on quality attributes*
- *directly link to business and mission goals*
- *explicitly involve system stakeholders*
- *be grounded in state-of-the-art quality attribute models and reasoning frameworks*

Advancements Over the Years

- Architectural patterns
- Component-based approaches
- Company specific product lines
- Model-based approaches
- Frameworks and platforms
- Standard interfaces

What HAS Changed?

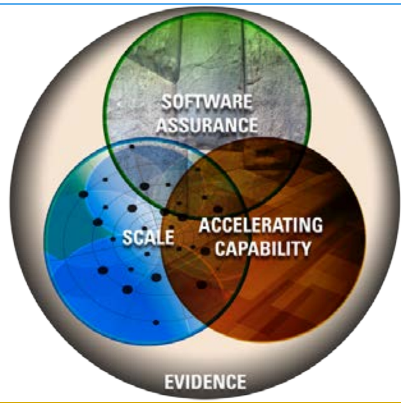
- Increased connectivity
- Scale and complexity
  - decentralization and distribution
  - “big data”
  - increased operational tempo
  - inter-reliant ecosystems
  - vulnerability
  - collective action
- Disruptive and emerging technologies

Technology Trends

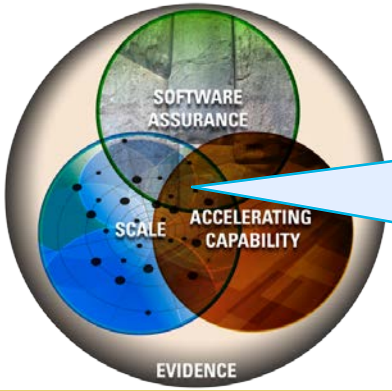
Software Development Trends

- Application frameworks
- Open source
- Cloud strategies
- NoSQL
- Machine Learning
- MDD
- Incremental approaches
- Dashboards
- Distributed development environments
- DevOps

Technical Challenges



The Intersection and Architecture



At the intersections there are difficult tradeoffs to be made in structure, process, time, and cost.

Architecture is the enabler for tradeoff analyses.

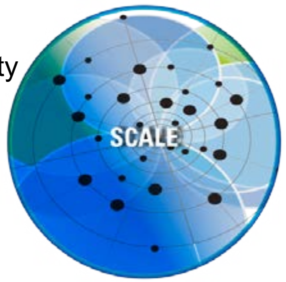
State of the Practice

Focus is on

- culture and teaming
- process and practices
  - value stream mapping
  - continuous delivery practices
  - *Lean* thinking
- tooling, automation, and measurement
  - tooling to automate repetitive tasks
  - static analysis
  - automation for monitoring architectural health
  - performance dashboards

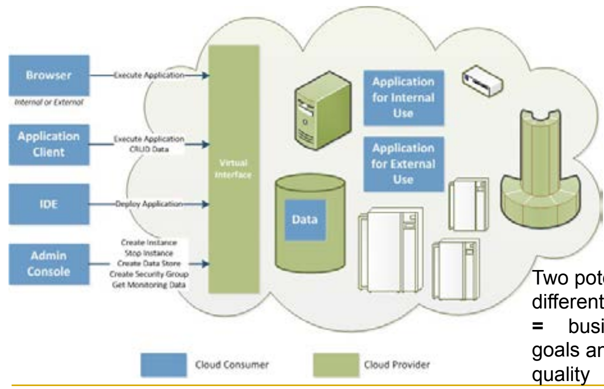
Architecture and Scale

- Cloud strategies
- Cloud strategies for mobility
- Big data



"Scale Changes Everything"

Two Perspectives of Software Architecture in Cloud Computing



Two potentially different sets of = business goals and quality attributes

Mobile Device Trends

Architecture Trends: Cyber-Foraging

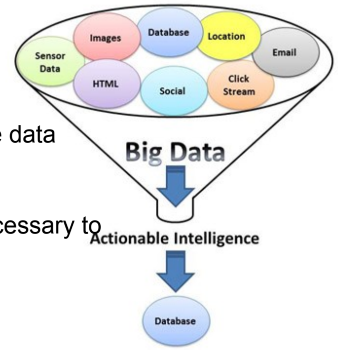
- Edge Computing
- Using external resource-rich surrogates to augment the capabilities of resource-limited devices
  - code/computation offload
  - data staging
- Industry is starting to build on this concept to improve mobile user experience and decrease network traffic.
- Our research: cloudlet-based cyber-foraging
  - brings the cloud closer to the user

nsn
Nokia Siemens Networks
Liquid Applications

CISCO
Cisco Systems
Fog Computing

Big Data Systems

- Two very distinct but related technological thrusts
  - Data analytics
  - Infrastructure
- Analytics is typically a massive data reduction exercise – "data to decisions."
- Computation infrastructure necessary to ensure the analytics are
  - fast
  - scalable
  - secure
  - easy to use



Big Data – State of the Practice "The problem is not solved"

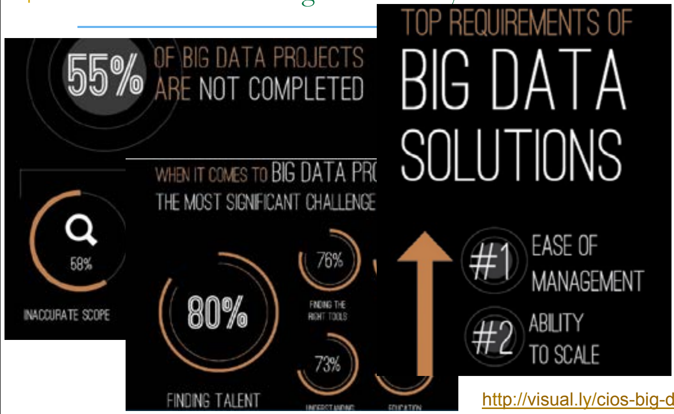
Building scalable, assured big data systems is hard.

HealthCare.gov Take health care into your own hands
Google
NETFLIX
amazon
NASDAQ

Building scalable, assured big data systems is expensive.

Google
facebook
amazon

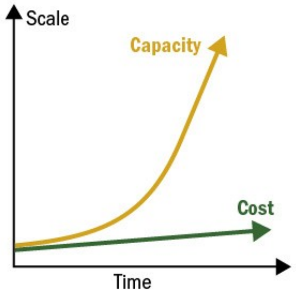
Big Data Survey



http://visual.ly/cios-big-data

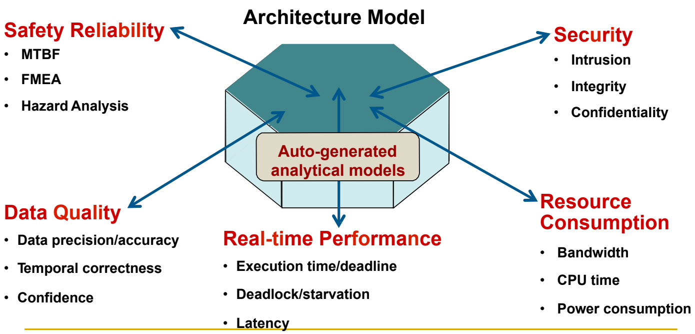
Architecture and Big Data

- System costs must grow more slowly than system capacity.
- Approaches
  - scalable software architectures
  - scalable software technologies
  - scalable execution platforms
- Scalability reduces as implementation complexity grows.
- NoSQL models are not created equal.
- You can't manage what you don't monitor.

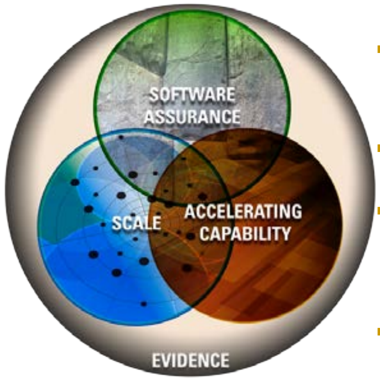


Architecture-Centric Quality Attribute Analyses

Single Annotated Architecture Model Addresses Impact Across Operational Quality Attributes



Summary



- Software architecture principles and their importance persist.
- Change brings new challenges.
- Software architecture practices and research are key to meeting these challenges.
- Much remains to be done.

What to do with the data

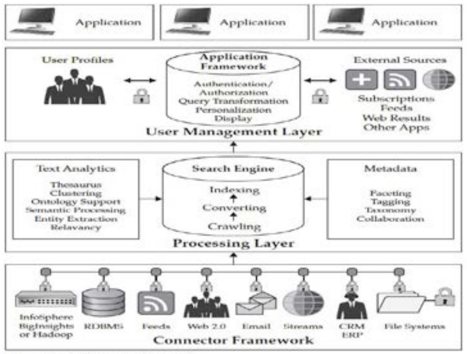
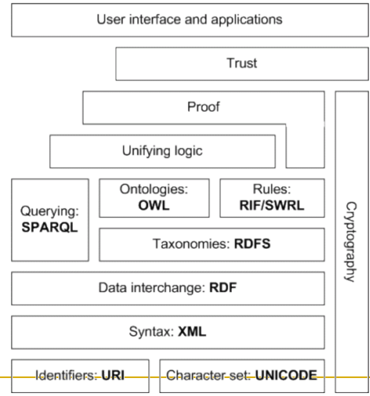
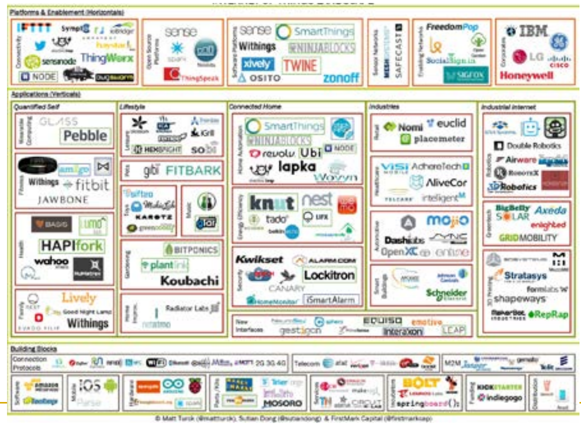


Figure 7-1 *Data Explorer architecture*

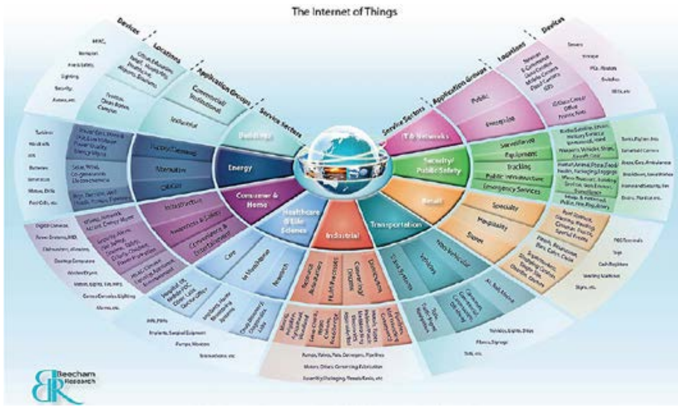
SW Stack: Architecture,Standards



IoT Landscape



EVERY Where



Class Book & References

- Classics:
  - Mary Shaw & David Garlan, Software Architecture: Perspectives on an Emerging Discipline;, Prentice Hall, 1996.
  - Many IEEE, ACM, ELSEVIER, and Open Source papers
- Other references actively used:
  - Evaluating Software Architectures: Methods & Case Studies, Addison-Wesley, ISBN 020170482X.
  - Design & Use of Software Architectures: Adopting and Evolving a Product Line Approach: ISBN: 0201674947
  - SEI Selected Papers: http://www.sei.cmu.edu/architecture/
  - USC Selected Papers: http://sunset.usc.edu
  - UCI Selected Papers: http://www.ics.uci.edu
- Other References & Standards discussed:
  - IEEE 1471, ISO RM-ODP
    - http://www.dstc.edu.au/Research/Projects/ODP/ref_model.html
  - IEEE 1220, EIA-632, EIA-731, DO-178B, J-STD-016-1995

29

Other Referenced Sites

- Primary References:
  - http://portal.acm.org/dl.cfm?coll=portal&dl=ACM&CFID=3943453&CFTOKEN=26992569
    - (ACM Copyrighted Papers)
  - http://ieeexplore.ieee.org/Xplore/DynWel.jsp
    - (IEEE Copyrighted papers)
  - http://www.sciencedirect.com/
    - (Elsevier Copyrighted material)
  - http://citeseer.nj.nec.com/cs (Lots of open source materials)
  - http://www.sei.cmu.edu/str/ (Lots of open source materials)
  - http://www-2.cs.cmu.edu/~able/publications/ (Lots of open source materials)
  - http://www.sei.cmu.edu/staff/rkazman/SE-papers.html (Lots of open source materials)
  - http://www.cs.ukc.ac.uk/people/staff/cr3/bib/bookshelf/Modules.html
    - (A ton of open source materials)
  - http://www2.umassd.edu/SECenter/SAResources.html (Links to other sites)
  - http://www.cgl.uwaterloo.ca/~rnkazman/SA-sites.html (Links to other sites)
  - http://www-ksl-svc.stanford.edu:5915/ (Ontologies)
  - http://www.sei.cmu.edu/architecture/adl.html (Architecture Definition Languages)

30

Other Referenced Sites

- Secondary References Used:
  - http://www-2.cs.cmu.edu/afs/cs/academic/class/17655-s02/www/
    (Garlan’s Class on SA)
  - http://www.softwaresystems.org/architecture.html
  - http://www-old.ics.uci.edu/pub/arch/
  - http://cora.whizbang.com/
  - http://selab.korea.ac.kr/selab/resources/se/architecture.htm#Web
  - http://www.cebase.org/
  - http://www.afm.sbu.ac.uk/
  - http://www.wwisa.org
  - http://www.dacs.dtic.mil/databases/url/key.hts?keycode=71:2024&islowerlevel=1
  - http://www.cs.colorado.edu/serl/arch/Papers.html
  - http://www.cetus-links.org/

31

Other Referenced Sites

- Tools:
  - http://www.isr.uci.edu/projects/xarchuci/ (xADL 2.0)
  - http://www.isr.uci.edu/projects/xarchuci/tools-overview.html
  - http://www.isr.uci.edu/projects/archstudio/setup-rundevelop.html (xADL)
  - http://www.isr.uci.edu/projects/archstudio/
  - http://www.kc.com/products/iuml/lite.html (xUml)
  - http://www-2.cs.cmu.edu/~able/software.html (ACME SW)
  - http://www-2.cs.cmu.edu/~Compose/html/tools.html
  - http://pavg.stanford.edu/rapide/ (RAPIDE)
  - http://www.afm.sbu.ac.uk/z/ (Z)

32

Further Reading

- *Design and Use of Software Architectures : Adopting and Evolving a Product-Line Approach, Jan Bosch, 2000*
- *Applied Software Architecture, Hofmeister Christine, 1999*
- *The Art of Software Architecture: Design Methods and Techniques*, S. T. Albin, 2003
- *Software Architecture in Practice*, L. Bass, P. Clements and R. Kazman, 2003.

Further Reading

- *Evaluating Software Architectures: Methods and Case Studies,* P. Clements, R. Kazman and M. Klein, 2001.
- *Documenting Software Architectures: Views and Beyond,* Clements, Paul, Felix Bachmann, Len Bass, David Garlan, James Ivers, Reed Little, Robert Nord and Judith Stafford, 2002.
- *Software Architecture: Organizational Principles and Patterns,* Dikel, David, D. Kane and J. Wilson, 2001.
- *Large-Scale Software Architecture : A Practical Guide using UML*, Jeff Garland and Richard Anthony, 2002.
- *Designing Software Product Lines wtih UML: From Use Cases to Pattern Based Software Architectures,* H, Gomma, 2005.
- *Essential Software Architecture*, Ian Gorton, 2006.

Further Reading

- Software Architecture: Foundations, Theory, and Practice, Richard N. Taylor, Nenad Medvidovic and Eric Dashofy
- Essential Software Architecture, Ian Gorton, 2006.
- 97 Things Every Software Architect Should Know, Richard Monson-Haefel
- Documenting Software Architectures: Views and Beyond, Paul Clements, Felix Bachmann, Len Bass and David Garlan, 2010.
- The Process of Software Architecting, Peter Eeles and Peter Cripps, 2009.

Contents

Topic 0: Overview

Build Software or Build Airliners

- True Feats of Engineering
  - The Boeing 747 (the biggest beast)
  - Software Systems
- The Boeing 747
  - 1970: 4.5 million parts, 3 million pins, rivets
  - 2003: 6 million parts, 3 million pins, rivets
- Windows ™ Operating System
  - Windows XP: 40 Million Lines of Code
  - Windows 2000: 20 Million Lines of Code
- Linux Operating System
  - Kernel ver. 2.6: 5.7 Million Lines of Code
- There is no "Silver Bullet"

Engineering Common Ground

- Reliability
  - Specification Owner
  - Frozen Requirements
  - Verification
- Budgets
  - Schedule
  - Complexity
- Innovation
  - Near-term Requirements
  - In-place Evolution
- People
  - Management
  - Personalities

No "Silver Bullet"

- Innovation in aerospace, not the 747!
  - Complexity per Year: 1% vs. 20% in Software
    - Boeing 747 35yrs Reliable WW2 technology
    - Software 49yrs Constant Innovation
- Reinventing Engineering
  - Reliability follows Innovation
  - Being First means Learning the Hard Way
- Build Software, not Airliners

The Core Triad for Success

- People
- Process
- Technology

- People + Process + Technology !!!

Software Evolution

- People
  - Position Descriptions of the 70s:
    - Programmers
  - 80s
    - Programmer/analysts
  - 90s
    - Developer
    - Architect
    - Business Analyst
  - 00s ?
    - UI Designer
    - Bean Provider
    - Assembler
    - Deployer

+ Architect
+ Modeler
+ Implementer
+ Project Manager

+ System Analyst
+ Config. Mgr.
+ System Tester
+ Test Designer

Software Evolution (cont'd)

- Processes
  - Ad Hoc
  - Waterfall
  - Spiral
  - Incremental
  - Iterative
  - RAD/JAD
  - Unified Process
  - Extreme Programming (XP)
  - Feature-Driven Development
  - etc

Software Evolution (cont'd)

- Technology
  - Languages: Assembler , Procedural, Structured, Object-Oriented
  - 3GL/4GL/CASE
  - Life Cycle Tools
    - Requirements, Architecting, Building, Testing
    - Configuration Management/Version Control
    - Round-trip Engineering (manual steps)
    - **Simultaneous** round-trip tools
  - Modeling:
    - Structured, DFD,
    - Coad, OMT, Booch,
    - UML

Some Fundamental Issues

- Software is very complex today
  - Hard for one to understand it all
  - Difficult to express in terms all stakeholders understand
- Business drivers add pressure
  - Shrinking business cycle
  - Competition increasing
  - Ever rising user expectations
- "Soft" Requirements
  - A common threat to schedule, budget, success
  - Too much change can cause failure

Fundamental Issues (ii)

- Flexibility and resilience to change is key
  - Ability to adapt to sudden market changes
  - Design is solid enough that change does not impact the core design in a destabilising way
  - Willingness to re-architect as required
- Most projects are unpredictable
  - Lack of knowing where and what to measure
  - Lack of yardsticks to gauge progress
  - Requirements creep is common
  - Scrap and rework is common
  - Interminably 90% done

Fundamental Issues (iii)

- Lack of quality people results in failure
  - …in spite of best processes and tools!
  - Managing people is difficult
- Major change is organisationally difficult
- Reusing software artifacts is rare
  - Architectures/Designs/Models
  - Estimating processes
  - Team processes
  - Planning & Tracking procedures and reports
  - Software construction & management
  - Reviews, testing
  - *etc.*

The Pioneering Era (1955-1965)

- New computers were coming out every year or two.
- Programmers did not have computers on their desks and had to go to the "machine room".
- Jobs were run by signing up for machine time. Punch cards were used.
- Computer hardware was application-specific. Scientific and business tasks needed different machines.
- High-level languages like FORTRAN, COBOL, and ALGOL were developed.
- No software companies were selling packaged software.
- Academia did not yet teach the principles of computer science.

The Stabilising Era (1965-1980)

- Came the IBM 360.
- This was the largest software project to date.
- The 360 also combined scientific and business applications onto one machine.
- Programmers had to use the job control language (JCL) to tell OS what to do.
- PL/I, introduced by IBM to merge all programming languages into one, failed.
- The notion of timesharing emerged.

The Stabilising Era (1965-1980)

- Software became a corporate asset and its value became huge.
- Academic computing started in the late 60's.
- Software engineering discipline did not yet exist.
- High-hype disciplines like Artificial Intelligence emerged.
- Structured Programming burst on the scene.
- Standards organisations became control battle grounds.
- Programmers still had to go to the machine room.

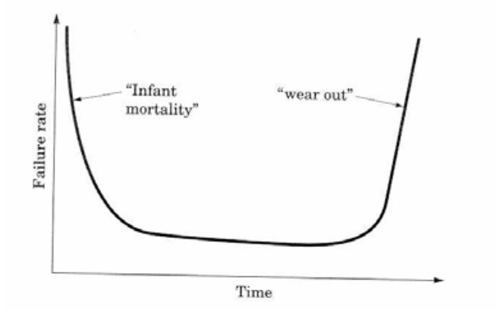
The Micro Era (1980-Present)

- The price and size of computers shrunk. Programmers could have a computers on their desks.
- The JCL got replaced by GUI.
- The most-used programming languages today are between 15 and 40 years old. The Fourth Generation Languages never achieved the dream of "programming without programmers".

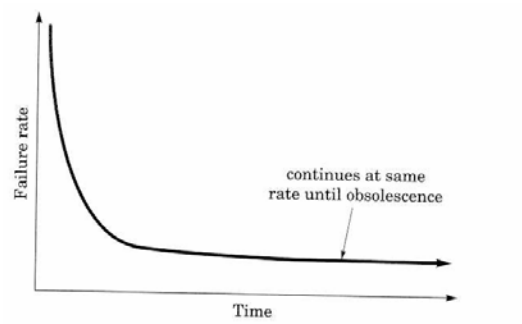
Software Characteristics

- software is engineered, not manufactured
  - no manufacturing phase which introduces quality problems
  - costs concentrated in engineering
- software does not wear out
  - does deteriorate
  - no spare parts
- most software is custom built rather than being assembled from components

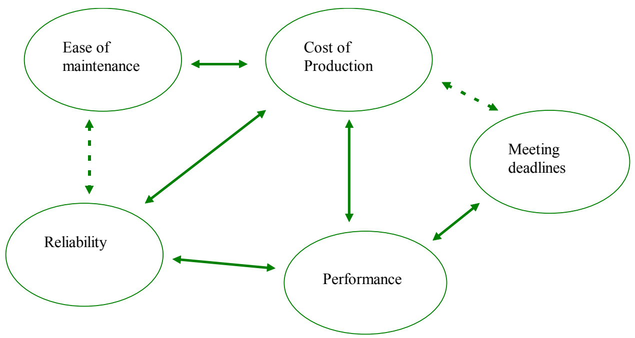
Hardware Characteristics



Software Characteristics



Software attributes - conflicting aims



Digital Economy

- E-Commerce
  – the online exchange of value (goods, services, and/or money) within firms, between firms, between firms and their customers, and between consumers.
- Digital Economy
  – An economy that is empowered by IT, Internet and digital technologies.
  – Individuals and enterprises create wealth by applying knowledge, networked human intelligence, and effort to manufacturing, agriculture and services
- Impact of the Digital Economy
  – New Products
  – New Business Opportunities
  – New Business Relationship

BITS INSTEAD OF ATOMS

- **Digital Representation**
  - The physical world *(atoms)* is modeled with 1's and 0's *(bits)*
- **Enabling Technology**
  - Allows high capacity data storage *(bit memory)*
  - Allows large system integration *(bit regeneration)*
  - Allows large distance communications *(noise immunity)*
  - Allows inexpensive, low power hardware *(CMOS)*
  - Allows data compression *(Internet images & video)*
  - Allows data encryption *(e-commerce)*

Summary

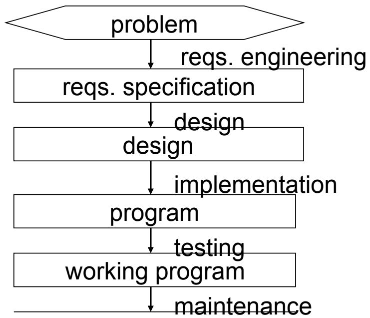
- **Changing Society & Changing Software**
- **A Day in the Life of the Digital Consumer in the Future**
  - People move freely from one environment to another
  - They use whatever devices are most convenient at the time
  - They automatically connect using the best network available at the time based on their personal profile
  - They have access to the same personalised services automatically scaled to the device and connection they are using
  - They have one service provider that manages and optimises their service and account based on their unique needs and resources.

Topic 1: History and Definition of Software Architecture

Software Life Cycle Revisited

- software development projects are large and complex
- a phased approach to control it is necessary

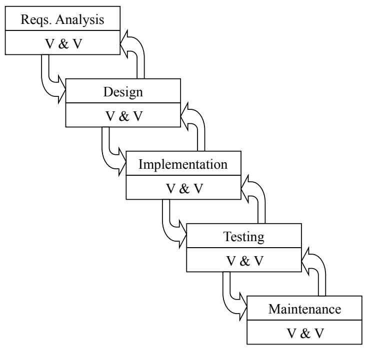
Simple life cycle model



Simple Life Cycle Model

- document driven
- milestones are reached if the appropriate documentation is delivered (e.g., requirements specification, design specification, program, test document)
- problems
  - feedback is not taken into account
  - maintenance does not imply evolution

Waterfall Model



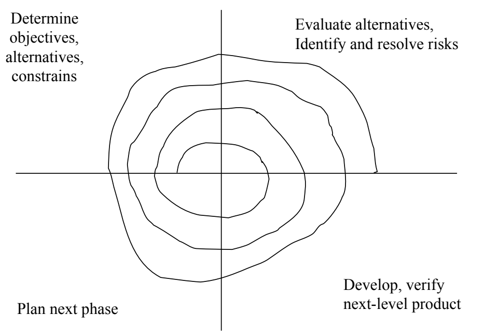
Waterfall Model (cntd)

- includes iteration and feedback
- validation (*are we building the right system*?) and verification (*are we building the system right*?) after each step
- user requirements are fixed as early as possible
- problems
  - too rigid
  - developers cannot move between various abstraction levels

Spiral Model

- all development models have something in common: reducing the risks

Spiral Model (cntd)



Towards a Software Factory

- developers are not inclined to make a maintainable and reusable product, it has additional costs
- this viewpoint is changed somewhat if the product or product family is the focus of attention rather than producing the initial version of the product

Towards a Software Factory (cntd)

- reuse becomes important
- progress has been made in the areas of: reusable design (e.g., software architecture and design patterns) and software components
- the idea of a software factory
  - share people’s knowledge
  - share reusable products/components

Software Architecture

- High-level abstraction of system - Programming in the large

Benefits of Studying Software Architecture

- Greater understanding
- Reuse
- Evolution
- Analysis
- Management

Fundamentals

Architecture should establish a common vision of the project/product development. Acts as glue:

Conceptual models of the application domain

Relatively formal models that are the basis for implementation

Classical Building Architecture Analogy

Points of commonality:

- notations and representations (*building:* blueprints, physical models, perspective drawings; *software:* structural graphics, coupled with behavioral and informational views)
- architectural styles (*building:* Romanesque, Gothic, Victorian; *software:* distributed, layered, client/server)
- constraints (*building:* acoustics, air/water flow, lighting; *software:* throughput, timing, fault tolerance, ease of use)
- standards/practices (*building:* building codes & inspections; *software:* interface standards & walk-throughs/reviews)

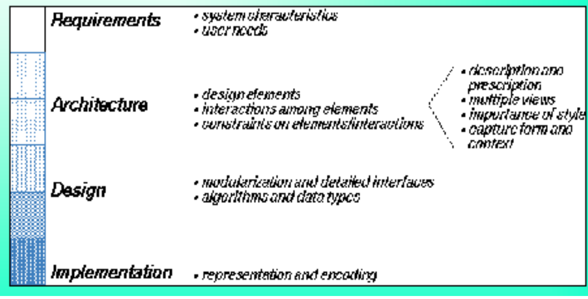
Chemical (Unit Process) Architecture Analogy

Points of commonality

- components (*chemical:* unit operations [e.g., heat exchanger]; *software:* architectural components [e.g., DBMS])
- connectors (*chemical:* pipes, conveyors; *software:* procedure calls, pipes)
- architectural styles (*chemical:* operation [e.g., batch, continuous]; *software:* topological styles [e.g., pipe & filter, blackboard])
- constraints (*chemical:* precedence of elements in a process; *software:* precedence of elements in an execution sequence)
- notation (*chemical:* process flowcharts; *software:* ADLs)
- product (*chemical:* chemical plant; *software:* application system)

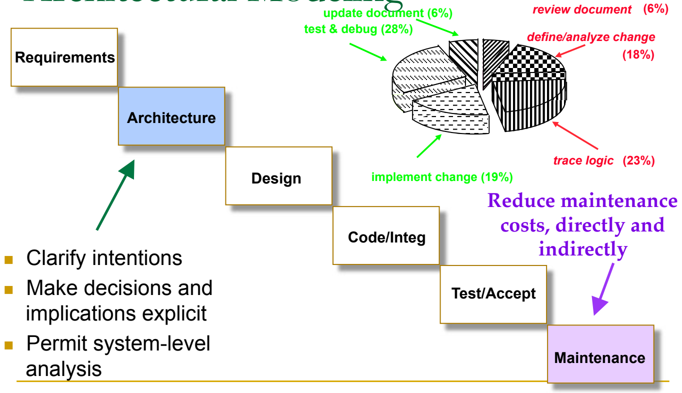
73

Where it Fits . . .



74

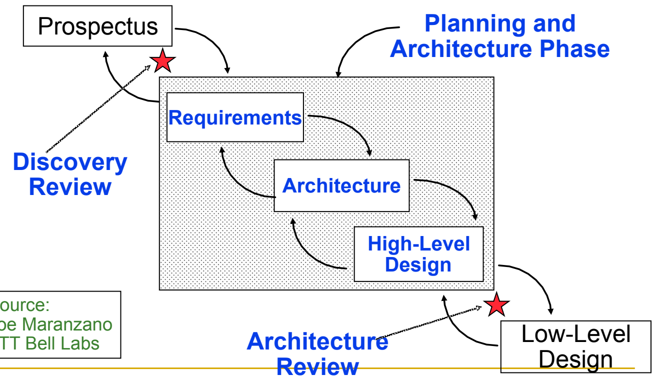
Promised Benefits of Architectural Modeling



- Clarify intentions
- Make decisions and implications explicit
- Permit system-level analysis

75

Architectural Design Reviews



Source:
Joe Maranzano
ATT Bell Labs

76

Architecture in the Life-Cycle (1)

- Views of a Software System

"An important objective of software architecture is the production of a consistent set of views of the system and its parts presented in a structure that satisfies the needs of both the end-user and later designers."(Witt et al.)

77

Architecture in the Life-Cycle (2)

- Customer
  - Concern
    - Schedule and budget estimation
    - Feasibility and risk assessment
    - Requirements Traceability
    - Progress Tracking

78

Architecture in the Life-Cycle (3)

- User
  - Concern
    - Consistency with requirements and usage scenarios
    - Future requirement growth accommodation
    - Performance, reliability, interoperability, etc.

79

Architecture in the Life-Cycle (4)

- Architect
  - Concern
    - Requirements Traceability
    - Support of tradeoff analyses
    - Completeness, consistency of architecture

80

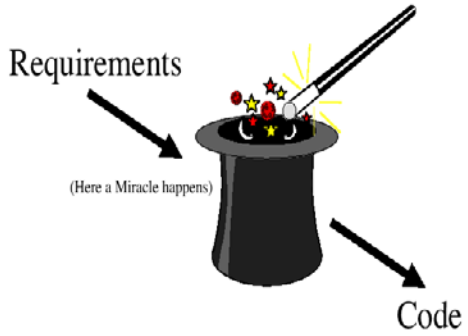
Architecture in the Life-Cycle (5)

- Developer
  - Concern
    - Sufficient detail for design
    - Reference for selecting/assembling components
    - Maintain interoperability with existing systems

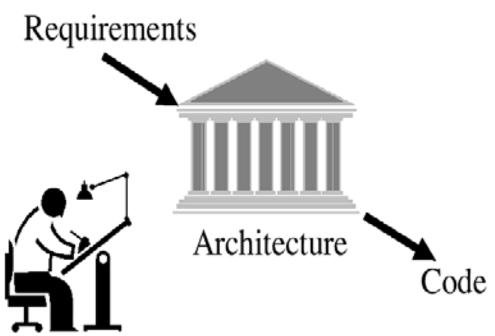
Architecture in the Life-Cycle (6)

- Maintainer
  - Concern
    - Guidance on software modification
    - Guidance on architecture evolution
    - Maintain interoperability with existing systems

"Magician Coder" View of Development



A Professional View



Architecture of Buildings

- Types (Domains): office building, shepherd's shelter, detached home, apartment building, aircraft hanger
  - Domain-specific software architectures
- Styles: colonial, Victorian, Greek revival, Mediterranean, Bauhaus
  - Software system organization paradigms
- Building codes: electrical, structural, ...
  - Constraints on how the building can be legally built
- Blueprints and drawings
  - Formal specification of supporting details

Architectural Design

Buildings

Elements
- Floors
- Walls
- Rooms

Types
- Office building
- Villa
- Aircraft hanger

Styles
- Colonial
- Victorian
- Southwestern

Rules and regulations
- Electrical
- Structural

Software

Elements
- Components
- Interfaces
- Connections

Types
- Office automation
- Game
- Space shuttle control

Styles
- Pipe and filter
- Layered
- Implicit invocation

Rules and regulations
- Use of interfaces
- Methods of change

Design

- Architectural design
  - High-level partitioning of a software system into separate modules *(components)*
  - Focus on the interactions among parts *(connections)*
  - Focus on structural properties *(architecture)*
    - "How does it all fit together?"
- Module design
  - Detailed design of a component
  - Focus on the internals of a component
  - Focus on computational properties
    - "How does it work?"

Software Architecture Definition

- The software architecture of a system is the structure or structures of the system, which comprise software components, the externally visible properties of those components, and the relationships among them.
- Len Bass, Paul Clements, and Rick Kazman. Software Architecture in Practice. SEI Series in Software Engineering. Addison-Wesley, Reading, Massachusetts, 1998.

Software Architecture Definition (cntd)

- Important issues raised:
  - multiple system structures;
  - externally visible (observable) properties of components.
- The definition does not include:
  - the process;
  - rules and guidelines;
  - architectural styles.

Software Architecture

- The IEEE Architecture Working Group (P1471), the Recommended Practice for Architectural Description, has established the following definition of architecture and related terms:

- Architecture is the fundamental organisation of a system embodied in its components, their relationships to each other and to the environment and the principles guiding its design and evolution.

Software Architecture (cntd)

IEEE architecture definition rationale:

- To avoid the inclusion of the term "structure" which is often associated with the physical structure of a system. An architecture is a property or concept of a system, not merely its structure.
- The phrase ``highest-level'' is used to abstract away from low-level details.
- An architecture can not be viewed in isolation, its environment in which it is embedded should be taken into account.

Software architecture (cntd)

Other IEEE related architecture definitions:

- **Architect:** the person, team or organisation responsible for systems architecting.
- **Architecting:** the activities of defining, maintaining, improving and certifying proper implementation of an architecture.
- **Architecture:** the highest-level conception of a system in its environment.
- **Architectural description:** a collection of products to document an architecture.

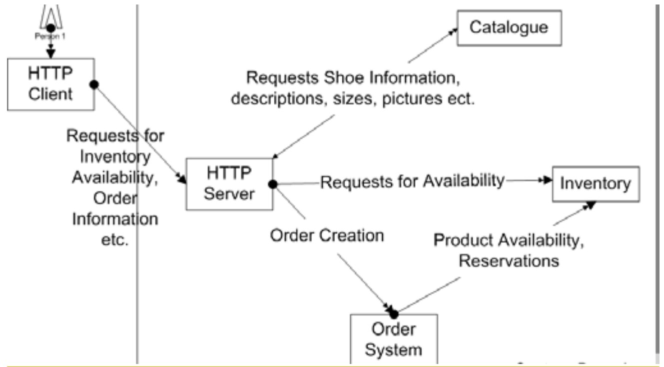
Software architecture

Other IEEE related architecture definitions (continued):

- **System stakeholder:** an individual, team, or organisation (or classes hereof) with interests in, or concerns relative to, a system.
- **View:** a representation of a whole system from the perspective of a related set of concerns.
- **Viewpoint:** a pattern or template from which to construct individual views. A viewpoint establishes the purposes and audience for a view and the techniques or methods employed in constructing a view.

93

A Simple Architecture



94

Early Notions of Software Architecture

- The earliest pioneers of what we now refer to as Software Architecture were Edgar Dijkstra, Fred Brooks Jr., and David Lorge Parnas
- In programming the term *architecture* was first applied to descriptions covering more than one computer system
  - i.e. "families of systems"
- Brooks and Iverson (1969) called architecture the "...conceptual structure of a system...as seen by the programmer"

95

Edgar Dijkstra 1968

- Dijkstra stressed as early as 1968 that how software is partitioned and structured is important
  - Not merely simply programming a "correct" solution
  - He introduced the idea of "layered structures" for operating systems
    - Resulted in ease of development, maintenance

96

Fred Brooks Jr. on System Architecture (1975)

- "By the architecture of the system I mean the complete and detailed specification of the user interface...."
- "The architect of a system, like the architect of a building, is the user's agent." ("Aristocracy, Democracy and System Design" in The Mythical Man-Month, 1975)

Brooks: Simplicity and Straightforwardness

- "...It is not enough to learn the elements and rules of combination; one must also learn idiomatic usage, a whole lore of how the elements are combined in practice. Simplicity and straightforwardness proceed from conceptual integrity. Every part must reflect the same philosophies and the same balancing of desiderata....Ease of use, then, dictates unity of design, conceptual integrity"

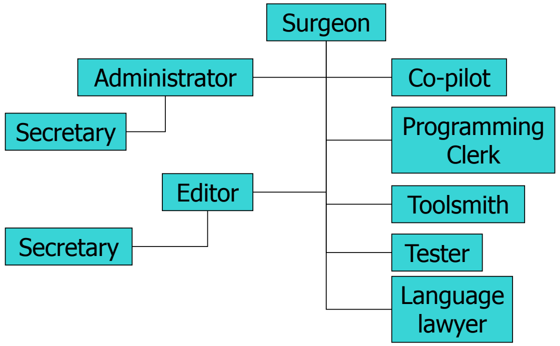
'One Mind, Many Hands'

- Conceptual integrity must proceed from one, or a small number of minds
  - e.g., Reims Cathedral's Jean d'Orbais
- But schedule pressures demand many hands
- Two techniques proposed:
  - Separation of architectural effort from implementation
  - New structuring of software development teams
    - "The Surgical Team"

The Surgical Team

- The problem
  - The "small, sharp" team is ideal...
    - Ten or less excellent programmers
  - ...but too slow for really big systems
- The solution
  - The 'surgical team': one cutter, many supporters

Communication Patterns



How the Surgical Team works

- 10 people, seven professionals, work to a system which is the product of a single (or maybe two) mind.
- Not a democracy of equals. The surgeon rules. No "division of problem"
- Division of labour permits radically simpler communication patterns.

Brooks on Blaauw

- Blaauw says total creative effort involves three distinct phases
  - Architecture
    - Writing all the external specifications
  - Implementation
  - Realisation
- Three "common objections" noted by Brooks
  - The specifications will be too rich and costly
  - Creativity is confined to the architects
  - Implementors will sit idle waiting for the architecture

Refuting Common Objections

- "...architects will get all the creative fun"
  - An illusion. Implementation is a creative activity that is undiminished by external specification: "Form is a liberator"
- "...implementors will sit idly by..."
  - Another illusion. A matter of timing and phasing, i.e. project management. The three activities can "be begun in parallel and proceed simultaneously"
- "The specifications will be too rich..."
  - A serious issue(dealt with by Brooks in "The Second-System Effect")

"Interactive Discipline for the Architect"

- In building, contractors' bids most often exceed the budget
- Architect has two possible responses
  - Cut the design
  - Challenge the bid by suggesting cheaper implementations
- The latter involves interactive dialogue with the builder

Architect vs. Builder

- To be successful, the architect must
  - Suggest, not dictate, an implementation
    - "The builder has creative and inventive responsibility for the implementation"
  - Always be able to suggest *a* way of implementing a specification
    - Be prepared to accept alternatives
  - Deal privately and quietly in such suggestions
  - Be ready to forego credit for suggested improvements
- "Often the builder will counter by suggesting changes to the architecture. Often he is right"

David Parnas 1971-79

- Parnas developed these 'architectural' concerns and turned them into fundamental tenets of Software Engineering. The main principles included:
  - Information Hiding as the basis of decomposition for ease of maintenance and reuse [72]
  - The separation of Interface from implementation of components [71, 72]
  - Observations on the separate character of different program elements [74]

David Parnas 1971-79 (contin.)

- Main principles (continued):
  - The "uses" relationship for controlling the connectivity between components [79]
    - To increase extensibility
  - Principles for the detection and handling of errors [72, 76]
    - i.e., exceptions
  - Identifying commonalities in "families of systems" [76]
    - To provide coarse-grained, stable common structures
  - Recognition that structure influences non-functional 'qualities' of a system [76]

- Parnas D. 1972. "On the Criteria for Decomposing Systems into Modules". *Communications of the ACM*. 15(12): pp.1053-8
- Parnas D. 1974. "On a 'Buzzword': Hierarchical Structure". *Proceedings of the IFIP Congress. 74* pp.336-390
- Parnas D. 1976. "On the Design and Development of Program Families". *IEEE Transactions of Software Engineering*, SE-2(1):pp. 1-9
- Parnas D. 1979. "Designing Software for Ease of Extension and Contraction". IEEE Transactions on Software Engineering. SE-5(2) pp.128-137

Fundamental Understanding

- Architecture is a set of principal design decisions about a software system
- Three fundamental understandings of software architecture
  - Every application has an architecture
  - Every application has at least one architect
  - Architecture is not a phase of development

109

Wrong View: Architecture as a Phase

- Treating architecture as a phase denies its foundational role in software development
- More than "high-level design"
- Architecture is also represented, e.g., by object code, source code, …

110

Context of Software Architecture

- Requirements
- Design
- Implementation
- Analysis and Testing
- Evolution
- Development Process

111

Requirements Analysis

- Traditional SE suggests requirements analysis should remain unsullied by any consideration for a design
- However, without reference to existing architectures it becomes difficult to assess practicality, schedules, or costs
  - In building architecture we talk about specific rooms…
  - …rather than the abstract concept "means for providing shelter"
- In engineering new products come from the observation of existing solution and their limitations

112

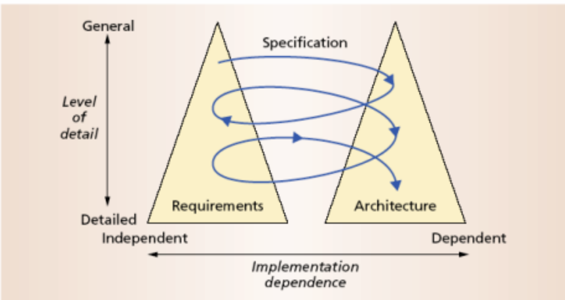
New Perspective on Requirements Analysis

- Existing designs and architectures provide the solution vocabulary
- Our understanding of what works now, and how it works, affects our wants and perceived needs
- The insights from our experiences with existing systems
  - helps us imagine what might work and
  - enables us to assess development time and costs
- → Requirements analysis and consideration of design must be pursued at the same time

Non-Functional Properties (NFP)

- NFPs are the result of architectural choices
- NFP questions are raised as the result of architectural choices
- Specification of NFP might require an architectural framework to even enable their statement
- An architectural framework will be required for assessment of whether the properties are achievable

The Twin Peaks Model



Design and Architecture

- Design is an activity that pervades software development
- It is an activity that creates part of a system' s architecture
- Typically in the traditional Design Phase decisions concern
  - A system' s structure
  - Identification of its primary components
  - Their interconnections
- Architecture denotes the set of principal design decisions about a system
  - That is more than just structure

Architecture-Centric Design

- Traditional design phase suggests translating the requirements into algorithms, so a programmer can implement them
- Architecture-centric design
  - stakeholder issues
  - decision about use of COTS component
  - overarching style and structure
  - package and primary class structure
  - deployment issues
  - post implementation/deployment issues

Design Techniques

- Basic conceptual tools
  - Separation of concerns
  - Abstraction
  - Modularity
- Two illustrative widely adapted strategies
  - Object-oriented design
  - Domain-specific software architectures (DSSA)

Object-Oriented Design (OOD)

- Objects
  - Main abstraction entity in OOD
  - Encapsulations of state with functions for accessing and manipulating that state

Pros and Cons of OOD

- Pros
  - UML modeling notation
  - Design patterns
- Cons
  - Provides only
    - One level of encapsulation (the object)
    - One notion of interface
    - One type of explicit connector (procedure call)
      - Even message passing is realized via procedure calls
  - OO programming language might dictate important design decisions
  - OOD assumes a shared address space

Implementation

- The objective is to create machine-executable source code
  - That code should be faithful to the architecture
    - Alternatively, it may adapt the architecture
    - How much adaptation is allowed?
    - Architecturally-relevant vs. -unimportant adaptations
  - It must fully develop all outstanding details of the application

Faithful Implementation

- All of the structural elements found in the architecture are implemented in the source code
- Source code must not utilise major new computational elements that have no corresponding elements in the architecture
- Source code must not contain new connections between architectural elements that are not found in the architecture
- Is this realistic?
  Overly constraining?
  What if we deviate from this?

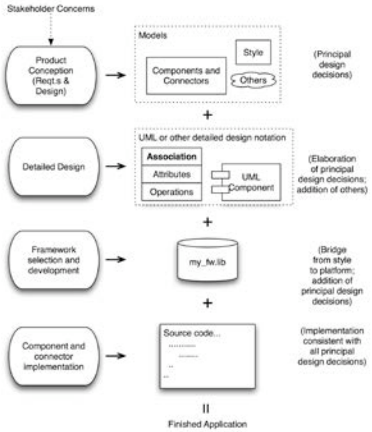
Unfaithful Implementation

- The implementation does have an architecture
  - It is latent, as opposed to what is documented.
- Failure to recognise the distinction between planned and implemented architecture
  - robs one of the ability to reason about the application's architecture in the future
  - misleads all stakeholders regarding what they believe they have as opposed to what they really have
  - makes any development or evolution strategy that is based on the documented (but inaccurate) architecture doomed to failure

Implementation Strategies

- Generative techniques
  - e.g. parser generators
- Frameworks
  - collections of source code with identified places where the engineer must "fill in the blanks"
- Middleware
  - CORBA, DCOM, RPC, …
- Reuse-based techniques
  - COTS, open-source, in-house
- Writing all code manually

How It All Fits Together



Analysis and Testing

- Analysis and testing are activities undertaken to assess the qualities of an artifact
- The earlier an error is detected and corrected the lower the aggregate cost
- Rigorous representations are required for analysis, so precise questions can be asked and answered

Analysis of Architectural Models

- Formal architectural model can be examined for internal consistency and correctness
- An analysis on a formal model can reveal
  - Component mismatch
  - Incomplete specifications
  - Undesired communication patterns
  - Deadlocks
  - Security flaws
- It can be used for size and development time estimations

Analysis of Architectural Models (cont'd)

- Architectural model
  - may be examined for consistency with requirements
  - may be used in determining analysis and testing strategies for source code
  - may be used to check if an implementation is faithful

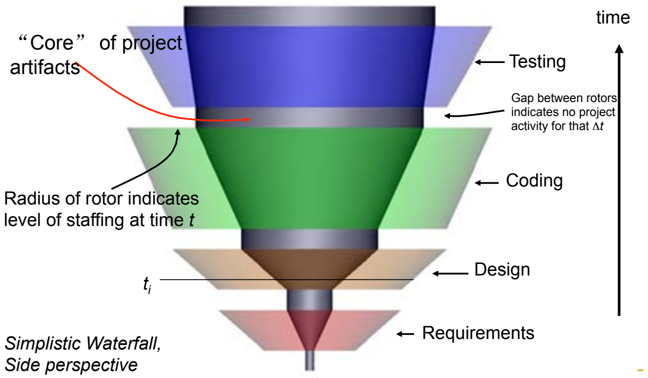
Evolution and Maintenance

- All activities that chronologically follow the release of an application
- Software will evolve
  - Regardless of whether one is using an architecture-centric development process or not
- The traditional software engineering approach to maintenance is largely ad hoc
  - Risk of architectural decay and overall quality degradation
- Architecture-centric approach
  - Sustained focus on an explicit, substantive, modifiable, faithful architectural model

Turbine – A New Visualisation Model

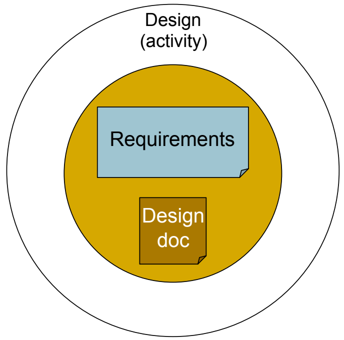
- Goals of the visualisation
  - Provide an intuitive sense of
    - Project activities at any given time
      - Including concurrency of types of development activities
    - The "information space" of the project
  - Show centrality of the products
    - (Hopefully) Growing body of artifacts
    - Allow for the centrality of architecture
      - But work equally well for other approaches, including "dysfunctional" ones
  - Effective for indicating time, gaps, duration of activities
  - Investment (cost) indicators

The Turbine Model

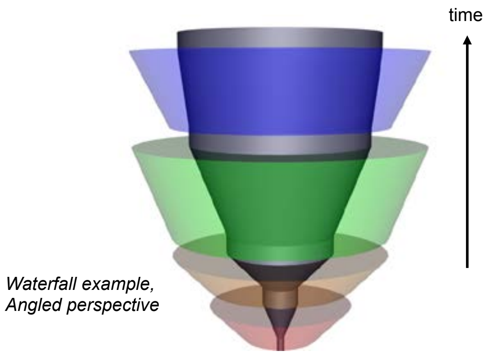


Simplistic Waterfall, Side perspective

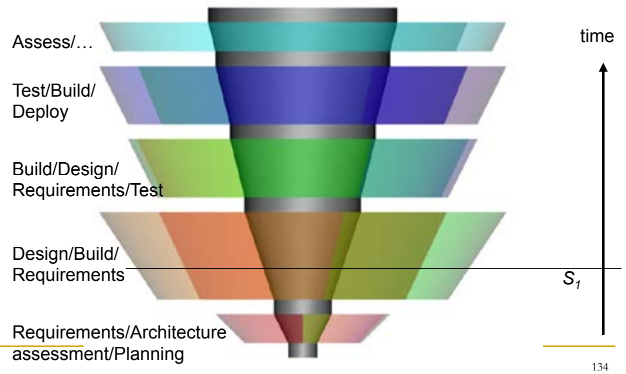
Cross-section at time t_i



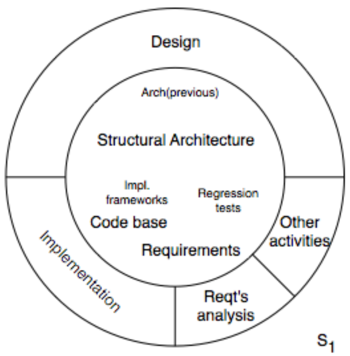
The Turbine Model



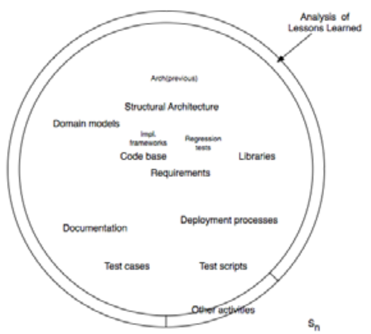
A Richer Example



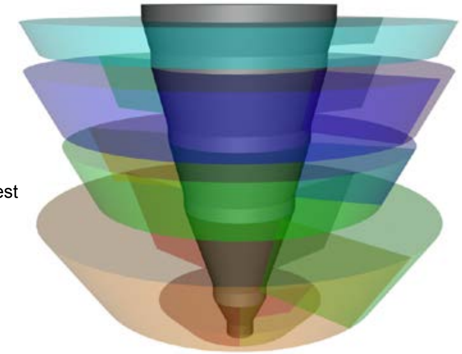
A Sample Cross-Section



A Cross-Section at Project End

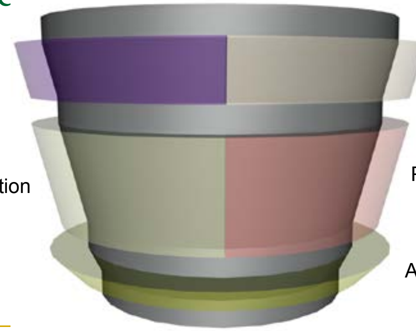


Volume Indicates Where Time was Spent



137

A Technically Strong Product-Line Project



138

Visualisation Summary

- It is illustrative, not prescriptive
- It is an aid to thinking about what's going on in a project
- Can be automatically generated based on input of monitored project data
- Can be extended to illustrate development of the information space (artifacts)
  - The preceding slides have focused primarily on the development activities

139

Summary (1)

- A proper view of software architecture affects every aspect of the classical software engineering activities
- The requirements activity is a co-equal partner with design activities
- The design activity is enriched by techniques that exploit knowledge gained in previous product developments
- The implementation activity
  - is centered on creating a faithful implementation of the architecture
  - utilises a variety of techniques to achieve this in a cost-effective manner

140

Summary (2)

- Analysis and testing activities can be focused on and guided by the architecture
- Evolution activities revolve around the product' s architecture.
- An equal focus on process and product results from a proper understanding of the role of software architecture

Software architecture: milestones

1968: *The inner and outer syntax of a programming language* (Maurice Wilkes)

1968-1972: Structured programming (Edsger Dijkstra); industrial applications (Harlan Mills & others)

1971: *Program Development by Stepwise Refinement* (Niklaus Wirth)

1972: David Parnas 's articles on information hiding 1974: Liskov and Zilles 's paper on abstract data types

1975: *Programming-in-the-large vs Programming-in-the-small* (Frank DeRemer & Hans Kron)

1987: *Object-Oriented Software Construction*, 1st edition

1994: *An introduction to Software Architecture* (David Garlan and Mary Shaw)

1995: *Design Patterns* (Erich Gamma et al.)

1997: UML 1.0

Topic 2: Modern Software Architecture

Leading Contributors

- The leading contributors of the modern discipline of Software Architecture to date are:
  - Dewayne Perry and Alexander Wolf
  - Mary Shaw and David Garlan
  - Len Bass, Paul Clements, Rick Kazman and Linda Northrup
  - Frank Buschmann et al.,
  - James O. Coplien

Foundations of Study

- The seminal work is a 1992 paper by Dewayne E. Perry and Alexander L. Wolf
  - "Foundations for the Study of Software Architecture". ACM SIGSOFT Software Engineering Notes 17(4) pp.40-52
- Constructed a model of Software Architecture consisting of 3 components:
  - Elements
  - Form
  - rationale

Perry and Wolf, 1992

- "We use the term 'architecture' rather than 'design' to evoke notions of codification, of abstraction, of formal training (of software architects), and of style"
- Benefits sought
  - Architecture as a framework for satisfying requirements
  - Architecture as the technical basis for design
  - Architecture as an effective basis for reuse
  - Architecture as the basis for dependency and consistency analysis

Basis of the Intuition

- Perry and Wolf examined other "architectural disciplines" for lessons
  - Computing hardware architecture
    - Small number of design features
    - Scale achieved by replication of the design elements
  - Network architecture
    - Two kinds of components – nodes and connections
    - Only a few topologies to be considered
  - Building architecture
    - Multiple views
    - Architectural styles
    - Style and engineering
    - Style and materials

The Context of Architecture

- *Requirements* are concerned with the determination of the information, processing and characteristics of that information needed by the user of the system
- *Architecture* is concerned with the selection of architectural elements, their interactions, and the constraints on those elements and their interactions necessary to provide a framework in which to satisfy the requirements and serve as a basis for design
- *Design* is concerned with the modularisation and detailed interfaces of the design elements, their algorithms and procedures, and the data types needed to support the architecture and to satify the requirements; and
- *Implementation* is concerned with the representations of the algrithms and data types that satisfy the design, architecture and requirements.

The Purpose of Architectural Specification

- Architectural specifications are required to be of such a character that we can
  - Prescribe the architectural constraints to the desired level
  - Separate aesthetics from engineering
  - Express different aspects of architecture in an appropriate manner
  - Perform dependency and consistency analysis

The Model: elements

- Software Architecture = {elements, form, rationale}
- Elements:
  - Processing elements
  - Data elements
  - Connecting elements

The Model: form

- Form
  - Consists of weighted properties and relationships
  - Weighting is either:
    - Importance of property or relationship
    - Or necessity of selecting among alternatives
  - Properties
    - Define the minimum constraints on the choice of architectural elements
  - Relationships
    - used to constrain the "placement" of architectural elements and how they interact

The Model: rationale

- Rationale:
  - Is underlying, but integral
  - Captures the motivation for the choice of style, elements and form
- In building architecture
  - Rationale explicates underlying philosophical aesthetics
- In software architecture
  - Instead explicates the satisfaction of system constraints
    - Functional and non-functional

Architectural Style

- Perry and Wolf noted that, distinct from other architectural disciplines, software architecture had no *named* styles
- They proposed that architectural styles be used as constraints on an architecture
- "The important thing about an architectural style is that it encapsulates important decisions about the architectural elements and emphasises important constraints on their elements and their relationships"

153

Garlan and Shaw

- David Garlan and Mary Shaw, both of Carnegie Mellon University, wrote a book, Software Architecture: Perspectives on an Emerging Discipline in 1996
- Identified three levels of software design
- Introduced four categories of research/development on software architecture
- Presented a number of common "architectural styles"

154

Levels of Software Design

- The three levels of software design identified by Garlan and Shaw are:
  1. Architecture
     - Issues involve the overall association of system capability with components
     - Components are modules
       - their interconnections can be handled in different ways
     - Operators guide the composition of systems from subsystems

155

Levels of Software Design

2. Code
   - Issues involve algorithms and data structures
   - Components are programming language primitives
   - Composition mechanisms include records, arrays, procedures etc.,

156

Levels of Software Design

3. Executable
   - Issues involve memory maps, data layouts, call stacks and register allocations
   - Components are bit patterns supported by hardware
   - Composition and operations described in the machine code

Architectural Styles

- Garlan and Shaw identify a number of common *architectural styles*, characterised by their respective *components* and *connectors*
- These styles include:
  - Dataflow systems
    - Batch Sequential
    - Pipes and Filters
  - Call-and-Return Systems
    - Main program and subroutine
    - OO Systems
    - Hierarchical systems

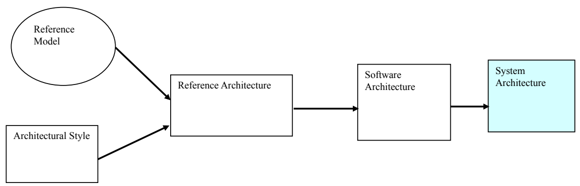
Architectural Styles

- Independent Components
  - Communicating processes
  - Event systems
- Virtual Machines
  - Interpreters
  - Rule-based Systems
- Data-Centred Systems
  - Database
  - Hypertext Systems
  - Blackboards

Bass, Clements and Kazman

- Len Bass, Paul Clements and Rick Kazman wrote a book Software Architecture in Practice in 1998
  - From the Software Engineering Institute, based at Carnegie Mellon University
    - Presented a taxonomy for “architecture”
    - Introduced an “Architectural Business Cycle (ABC)
    - Explained the Software Architecture Analysis Method (SAAM)
    - Described some Architectural Description Languages (ADLs)

A Taxonomy of "Architecture"

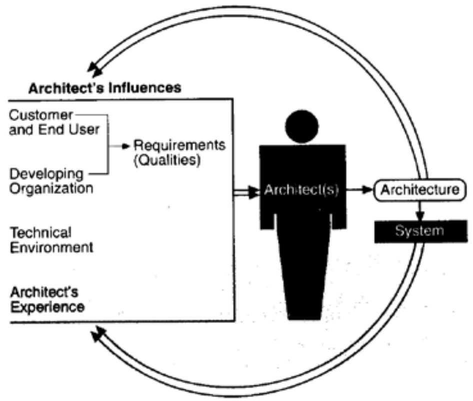


"Reference models, architectural styles and reference architectures are NOT architectures: they are useful steps towards architectures....At each successive phase in this progression, more requirements are addressed, and more design and development have taken place"

Taxonomy

- Architectural Style
  - "A description of component types and a pattern of their run-time control and/or data transfer"
    - Effectively a set of constraints on the architecture that define a family of architectures
- Reference Model
  - "A division of functionality together with dataflow between the pieces"
    - A standard decomposition of a known problem into parts that co-operatively solve the problem
- Reference Architecture
  - "A reference model mapped onto components ...and the data flow between the components"

The Architectural Business Cycle



The Architectural Business Cycle

1. The architecture affects the structure of the developing organisation
   - Software units prescribed correspond to work assignments
2. The architecture can affect the enterprise goals of the developing organisation
   - E.g., open up market opportunities, aid efficient development of product families etc.
3. The architecture can effect customer requirements for the next system
   - E.g., through upgradeability etc.,
4. The architecture adds to the corporate base of experience
5. The architecture may actually change software engineering culture

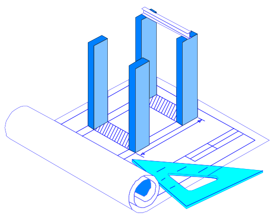
Topic 3: Software Architecture and the Built Environment

Architectural Knowledge

- **ADLs and Notions of 'Software Architecture Styles' help us analyze structure better… but how do they help us create architectures?**
- **The built environment has a notion of architecture that goes back to Ancient Egypt**
  - And recently the Royal Institute of British Architects (RIBA) has tried to define what an architect needs to know
  - Perhaps architecture offers real lessons, not just a metaphor?
- **It is interesting and important to examine the fundamentals of building construction**
  - Derive a notion of "architectural knowledge" as distinct from "vernacular design"

Structure

- An architecture defines the arrangement of structural elements in a system
  - Relates to form, function and characteristics
  - Architectural style is the underlying structuring principle and philosophy
- But any structure contains a distribution of responsibility
  - In complex structures this is often a sociological as much as a technical choice



Space

- Construction is both a physical and spatial transformation of a pre-existing situation

> At the most elementary level, a building is a construction of physical elements or materials into a more or less stable form, as a result of which space is created which is distinct from the ambient space.
>
> **[Hillier1996]**

Boundaries

- Building has a logical aspect too
  - Separates notions of "inside" and "outside"
- Architecture addresses the complex whole of interrelationship between such domains

> The drawing of a boundary establishes not only a physical separateness, but also the social separateness of a domain – the protected space – identified with an individual or collectivity which creates and claims special rights in that domain.
> [Hillier1996]

Neighborhoods as Domains

- The logical structure of an architecture is based on *spatial* domains…
  - Buildings, rooms, alcoves, etc.
- And *connection* domains between them...
  - Streets, alleys, hallways, corridors, doors, etc.
- And on how they are *configured* as a whole
  - Christopher Alexander strongly reflects this idea of a configuration based on logical coupling, cohesion, and connections in some patterns

Neighborhood Boundary

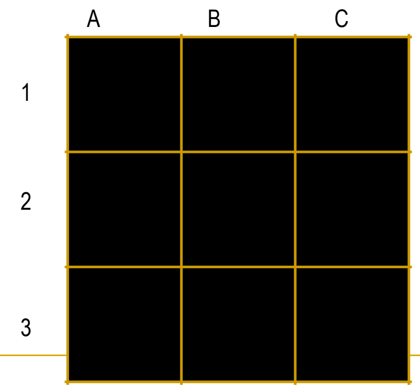
" The strength of the boundary is essential to a neighborhood. If the boundary is too weak the neighborhood will not be able to maintain *its own identifiable character….*"

" … Encourage the formation of a boundary around each neighborhood, to separate it from the next door neighborhoods. Form this boundary by closing down streets and limiting access to the neighborhood…."

" ... Place gateways at those points where the restricted access paths cross the boundary; and make the boundary zone wide enough to contain meeting places for the common functions shared by several neighborhoods."

[Alexander+1977] (pp89-90)

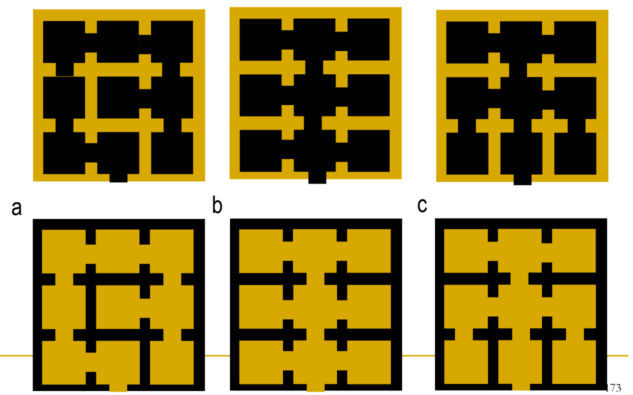
Exercise: Configurational Knowledge



Consider the squares in this grid to be spaces; the yellow lines to be walls.

At B3 is an external entrance; use exactly 8 other *internal* entrances to connect the rooms so that every room is accessible

Configuration of Space



Configuration of Space

- In the previous slide 3 notional courtyard buildings are shown
  - Same basic physical structures and cell division
  - Same number of internal, external openings
  - Lower figure highlights space as against normal view of 'structure' above
- 'Only' difference is the location of cell entrances
  - But this radically changes the patterns of movement through the buildings
  - Which offers more opportunities for "private" space?

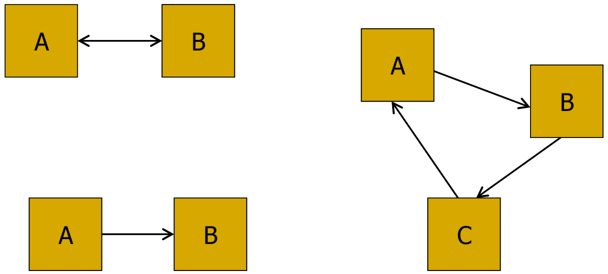
Dependencies

- Consider rooms A2, B2 and C2 in each of the configurations a, b and c and the routes by which they can be reached etc.
- Which other rooms is each directly dependent on?
- Which other rooms is each indirectly dependent upon?
- Which other rooms directly depend on A2,B2 and C2?
- Are there any parallels with software?

Software and "Space"

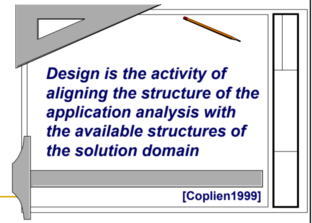
- Software does not deal with physical spaces
- But space is not merely a *physical* construct in architecture of the built environment
  - Also embodies notions of *logical* and *social* spaces
- We can consider modules, packages, components etc., to occupy virtual spaces in software
  - And connectors to be access paths to these spaces which make them interdependent
- Therefore the knowledge of how to put modules and connectors together appropriately is configurational

Software, Space and Dependency



Architecture Enables Creative Design

- Design is a creational and intentional act
  - Conception and construction of a structure on purpose for a purpose
- 'Good' architecture provides forms which enables creativity rather than dictate design
  - Form is liberating



Significance of Office Buildings

- Icons of the 20th Century
  - Office towers dominate the skylines of cities on every continent
  - At least 50% of the population of industrialised countries work in offices
- In function they stand closest to computing
  - As foci for administrative and information generating work
    - "Computer" originated as a term for number crunching clerks!

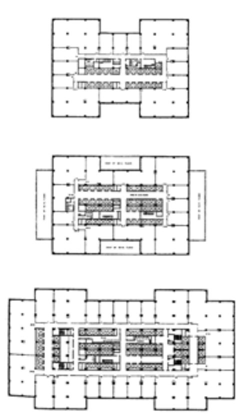
Office Architecture: 2 Traditions

- North American vs. North European
  - Skyscrapers in Chicago, ground hugging buildings in Stockholm
  - City centre in NA, suburban in NE
  - Tall and deep in NA, short and narrow in NE
  - Space efficient in NA, rambling in NE
  - Corporate domination of requirements in NA, workers' requirements better reflected in NE

The North American Tradition

- Forces driving the North American tradition
  - Late 19th century economic boom in Chicago and New York
  - Rapid exploitation of new technology
    - Steel frame, elevator, electricity, air conditioning
  - Advances in real estate financing, city planning
    - Buildings as multipliers of land value, speculator and regulator struggle, "form follows finance"
  - Taylorist management theory

Exemplar: Empire State Building



Exemplar: Empire State Building

- Architectural features of the Empire State Building 1930-31
  - Standardised construction elements to maximise efficiency
  - Highly service central core on each floor
  - Surrounded by continuous "race track" of subdivisible, rentable space
  - Relations between landlord and tenant dominate other (e.g. functional) concerns

The North European Tradition

- Forces driving the North European tradition
  - North European cities have long histories
    - Historically established patterns of land ownership
  - Financing
    - Direct Bank loans, rather than share issues, dominate the financing of construction
    - Offices often customised for specific uses by client
  - Social democratic political climate
    - Statutory based negotiating procedures for working conditions, etc.

Exemplar: Ninoflex Building

Exemplar: Ninoflex Building

- Architecture of the Ninoflax Building, 1962
  - "Office landscaping", open plan interiors, wall to wall carpeting, break areas
  - Complex, dynamic, "more organic" geometry
  - Based on a theoretical understanding of office processes and communication, rather than needs of syndicated investors/speculators

Office Architecture vs. Design

- Complex forces shaping the two traditions reveal themselves only in retrospect
  - Most office architects took (at least some) of the prevailing forces for granted
  - The essentially architectural knowledge was hidden, and transmitted culturally

It is clear from this analysis that architecture does not depend on architects, but can exist within the context of what we would normally call the vernacular.
[Hillier1996]

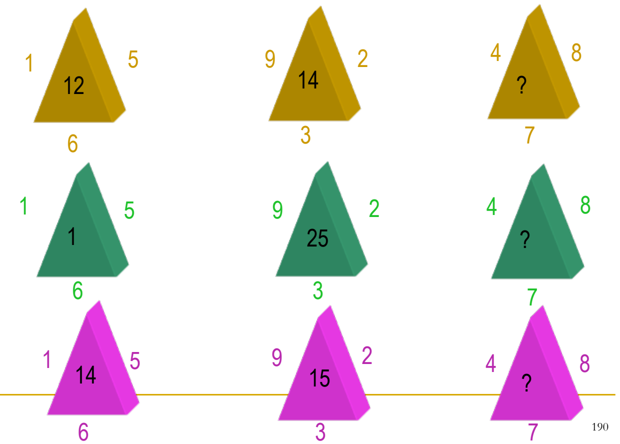
Architecture as Configurational Knowledge

- To summarise, architectural knowledge
  - Deals with process, organisation as well as "product"
  - Recognises that the "whole" is greater than the sum of its parts
    - structure "carves out" space
  - That a design choice in one place will have unintended side effects elsewhere that have to be imagined
    - and dealt with in design
  - Deals with multiple kinds of "spaces"
    - Physical, logical, social

Architecture as Non-discursive Knowledge

- Architecture is knowledge "to-design-with" rather than knowledge "of" a design
- This kind of knowledge is inherently difficult to express ("non-discursive")
  - Creative, not analytical thought
- Is typically acquired socially
  - "learning-by-doing"
- Only becomes explicit when different sets of configurational rules are compared and contrasted
  - E.g., different "styles" of office building

Exercise: Non-discursive Knowledge



Explicit Architecture

- Architectural knowledge is, therefore, normatively
  - Configurational
  - Non-discursive
- It tends to be made explicit only when there is a need to contrast normative approaches
  - To distinguish Gothic and Romanesque cathedrals
- ...But it always exists, even in vernacular design
  - Where it is implicit: "the way things are done here"

Three Filters of Applied Architectural Knowledge

Function imposes restraints on the configuration of space. Hillier (1996) suggests that three 'filters' are applied between the 'field of possibility' and the architectural reality:

- Generic Function
  - What type of building is it?
- Cultural requirements for that type of building
  - What aspects are typical for this kind of building?
- Idiosyncrasies of structure and expression
  - What uniquely distinguishes this building from all others?

Building for Change

Usage centered approach to office building requires a process that embraces change over time
Changeability is also a feature of the product

> The crux both builders and architects face is coming to terms with time. In technical terms this means shifting from a professional industry based on the assumption that the relationship with the client is synchronic (that is, each transaction is separate and each comes at a unique moment in time) whereas we should be trying to devise professional professional and technical services which are diachronic (that is, continuing and developing through time).
> [Duffy+1998]

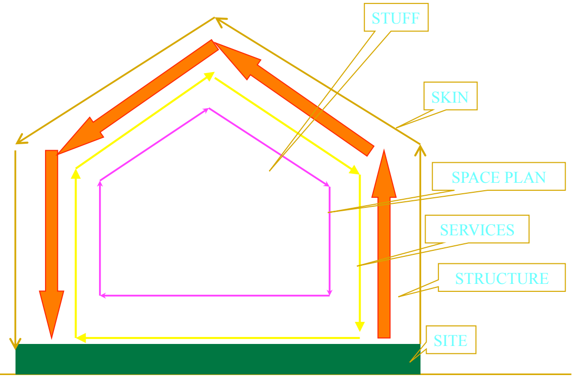
Designing in Slippage

- Flexible process requires building in slippage
- Recognising that change is both inevitable and necessary over time
- "Leeway" has to be designed in so that products do not become brittle when faced with change
  - We don't want to have to start again from scratch

Shearing Layers

- Shearing layers [Brand1994] underpin the process of new office design
  - Build in slippage between "layers" so that buildings don't tear themselves apart as usage changes
  - Layers constructed on the basis of different change rates
  - Changes therefore get localised to particular layers

Brand's Six Ss



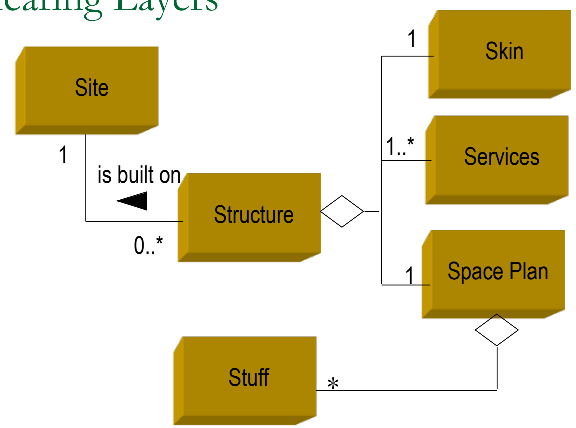
The Six S's of Shearing Layers (1)

- Site
  - Permanent or semi-permanent
  - Determined by geology, land-ownership etc.
- Structure
  - Typically sixty years for an office building
- Skin
  - Typically thirty year?

The Six S's of Shearing Layers (2)

- Services
  - Plumbing, wiring etc., changes every 7 years or so
- Space Plan
  - The division and sub-division of 'social spaces' tends to change every 5 years or so on average
- Stuff
  - Furniture, plantpots, other movables etc., can change daily

Shearing Layers



Lessons for Software Architecture

- 'Architectural Knowledge' is fundamental to successful, usable design in the new millenium
  - It can be regarded as design imagination
- It is by nature 'configurational' and often tacit
  - Especially in vernacular design
- It is knowledge that is socially acquired
  - "Culturally transmitted"
- It is both deontic and time-ordered
- It is not reducible to "high level structure"
  - Affects process and organisation too

Topic 4: Masterplans and Piecemeal Growth

The Current Debate on Software Architecture

- October 1999 special issue of IEEE Software exposed a debate
  - Edited by J.O. Coplien
  - Editorial entitled "Re-eveluating the Metaphor…"
  - Included text of Chris Alexander's speech to OOPSLA 1996 conference
- Identified two "camps"
  - Masterplan vs Piecemeal Growth
  - In the Masterplan camp: Carnegie Mellon's SEI
  - In the Piecemeal Growth camp: The Patterns Movement

Characteristics of the Masterplan "camp"

- Considers "Architecture" to be gross structure
  - Constrains, but is separate from, "lower levels of design"
- Utilises formal methods to present the semantics of architecture
- Emphasis is on design in the abstract
  - Drawings, models as "blueprints" to be completed before implementation
  - "Architecture is in the documentation"- Kazman
- Formal software engineering processes used to guide practical software building
  - E.g., Capability Maturity Model (CMM)
  - Architectural Tradeoff Analysis Method (ATAM)

Characteristics of Piecemeal Growth approach

- Rejection of abstract design
  - Cognitive complexity overcomes individual capacity to understand
- Stress on architecture existing at all levels of scale
  - Including "fine detail"
- Emphasis on an holistic, human-centred approach to design
  - Implies a crisis of traditional Computer Science
- Utilisation of "lightweight" processes
  - E.g., Scrum, DSDM, Xtreme Programming

Philosophical Differences

- Carnegie Mellon's SEI sees architecture as an extension of "software engineering"
  - Hence reliance on traditional Computer Science themes of formality, automation, process
- Holds to a "Logical Positivist" structure of knowledge

205

Logical Positivism

- Logical Positivism is a philosophy of science
  - Roots date back to Decartes
  - Dominated academia in the nineteenth, early twentieth century
  - Now in retreat in all disciplines *except* Computer Science
- Positivism underpins "The Scientific Method"
  - Scientist is neutral observer/recorder of scientific truths
  - Theories are established by reproducing observed phenomena in controlled laboratory experiments to test hypotheses

206

The Positivist Hierarchy of Knowledge

- Practical knowledge follows a linear, hierarchical model in this philosophy
  - First science, then engineering ("applied science"),then problem-solving

Form of Knowledge	Role performed
Science	Discovered by scientist
Engineering	Science applied by engineers
Problem Solving	Techniques applied by craftspeople

207

Critique of "Computer Science"

- The alternative philosophy is "holistic"
  - Allows for different forms of knowledge
  - Utilises science from many different disciplines
    - E.g., anthropology, sociology
- Regards computing as a design discipline
  - As opposed to a branch of mathematics
- N.B., this critique is broader than the debate in Software Architecture
  - E.g., Client-led Design, Human Computer Interaction, criticisms of Prof. Bruce Blum,Michael Jackson's *Problem Frames*

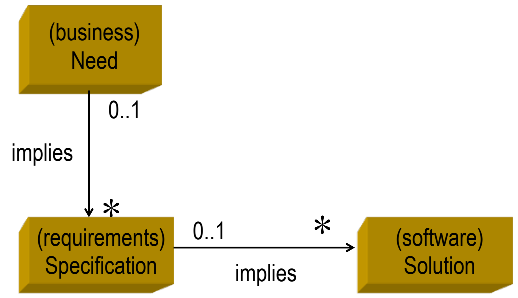
208

Software Development: Blum's Critique

- The essential aim of any software development is the construction of a solution(So) in software that meets a perceived need(N)
  - N → So
- Practically, most processes involve specifying (Sp) the solution (So) in advance of constructing it
  - N → So {N → Sp,Sp → So}

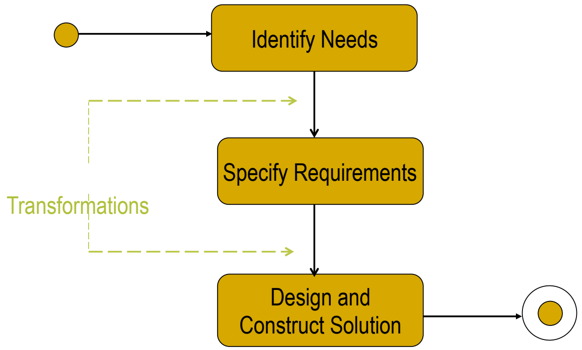
209

Needs Imply Solutions



210

The Process of Software Development



211

The Key Transformations

- Recall that the mapping of a Need (N) to a Software Solution (So) involves two transformations
  - N → So {N → Sp,Sp → So}
- Computer Science has developed formal means to aid in the production of a solution that meets a requirement
  - Sp → So
  - "The program in the computer" (Bruce I. Blum)
- We are historically weak in ensuring that the specification meets a need in the first place!
  - N → Sp
  - "The program in the world" (Bruce I. Blum)

212

Software Development Processes

- Most software development processes are
  - Solution-oriented
    - The focus is on the 'program-in-the-computer' as an end-product
  - Specification-driven
    - The process is initiated by a functional specification and tested against it
- Deadlines tend to squeeze out design
  - Since many solutions can meet a specification, effort spent on choosing the best solution is not on the critical path of the project
  - N.B.All other design disciplines recognise that "problem setting" (creating the spec.) is also part of design

213

Characteristics of a New Paradigm

- 'Use value' carries a higher premium than 'provable correctness'
  - The ultimate test of the 'program in the computer' is its usefulness as a 'program in the world'
- An increased attention to Non-functional Requirements
  - Operational (performance, robustness etc.)
  - Developmental (reuse, ease-of-maintenance etc.)
- The importance of 'Design'
  - 'form-fitting' to meet the above is non-trivial

214

Alexander and Piecemeal Growth

- Christopher Alexander's view can be found in his pattern, Gradual Stiffening
- He contrasts the work of the master carpenter...
  - Seems to work smoothly, effortlessly to construct a quality product
  - But actually builds in slippage and makes corrections through each "iteration"
- ... with the novice apprentice
  - "panic stricken by detail"
  - Needs a complete blueprint before making first step
    - For FEAR of irrecoverable failure

215

Alexander: Gradual Stiffening

"The fundamental philosophy behind the use of pattern languages is that buildings should be uniquely adaptable to individual needs and sites; and that plans of buildings should be rather loose and fluid, in order to accommodate these subtleties…"

"… Recognise that you are not assembling a building from components like an erector set, but that you are instead weaving a structure which starts out globally complete, but flimsy; then gradually making it stiffer but still rather flimsy; and only finally making it completely stiff and strong…."

216

What is a Design Pattern (1)?

"There are millions of particular solutions to any given problem; but it may be possible to find some one property which will be common to all these solutions. That is what a pattern tries to do"

What is a Design Pattern(2)?

" Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over without ever doing it the same way twice"

What is a Design Pattern (2)?

"...towns and buildings will not be able to come alive unless they are made by all of the people in society and unless these people share a common pattern language within which to make these buildings, and unless the pattern language is alive itself."

"...we present one possible pattern language....the elements of this language are so-called patterns...."

"Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over without ever doing it the same way twice"

Patterns vs. Pattern Languages

For Alexander

- Patterns do not exist outside of a wider "pattern language"
  - In which the use of one pattern sets the context for the use of others
- The pattern language is shareable amongst all "stakeholders" in a development

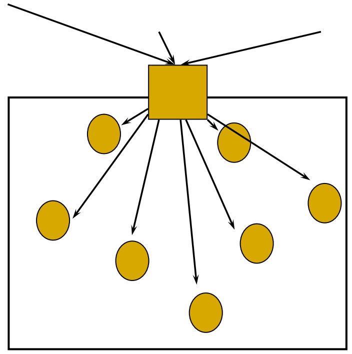
ADAPTOR represents (part of) an attempt to see whether pattern languages exist for software development

Pattern Languages as Process

- A Pattern Language is a system of patterns
  - But recall that patterns are abstract solutions
  - They cannot be composed together in advance to produce a solution
- Rather each pattern when applied changes the context of the solution
  - And creates a new context for the next pattern to be applied
- Solution is therefore "emerges" from the process itself
- N.B. This corresponds to modern "reflective design" theory
  - E.g., D. Schön "The Reflective Practitioner"

221

Example of Two Organisational Patterns



Sometimes development teams especially in a migration pilot project are distracted by "noise"

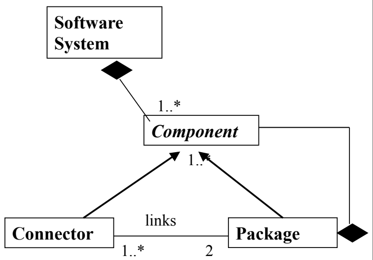
So - use "Firewall" to insulate the group from unwarranted distractions

Then use a "Gatekeeper" to filter incoming information and present it

222

System Composite

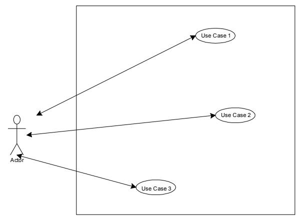
- A variation of the *Composite* pattern [Gamma 1995] allows us to describe any system as recursively composed of subsystems
- We can utilise high-level system-wide techniques recursively



223

Developing an Object Model of the Legacy System

- Using *Scenarios Define the Problem* [Coplien 1995] we employ Use Case analysis to develop descriptions of system and sub-system functionality
- (Perhaps using *Form Follows Function* to partition)



224

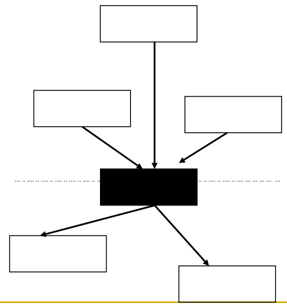
Separating Concerns

- The 'Shamrock' pattern identifies 3 kinds of packages
  - The Concept Model(s)
  - The Interaction Domain(s)
    - E.g., GUI's, protocols etc.,
  - The Infrastructure Domain(s)
    - Concurrency, Persistence etc.
- 'Time-Ordered Coupling'
  - Reflects Brand's Shearing Layers
    - See Appendix A

225

Encapsulating and "Wrapping" Subsystems

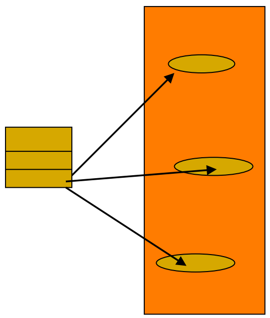
- To each subsystem add a Façade object which will present itself as the (sub) system interface to all of its clients. At this stage, the system will be a set of interdependent facades behind which sits legacy code



226

Develop Semantic Wrappers

- For the parts of the system which will be wrapped, develop objects that reflect the key abstractions
  - N.B. This requires building an Object analysis model of the legacy system
- The implementation of these objects calls existing legacy code...until you want to change that, too! (Semantic Wrapper)



227

Topic 5: Architecture Description Language (ADL)

ARCHITECTURE DESCRIPTION LANGUAGES

- Architecture is key to reducing development costs
  - development focus shifts to coarse grained elements
- Architectural models with well defined/understood semantics are needed
- Architecture Description Languages have been proposed as a possible answer
- Several prototype ADLs have been developed

ACME	MetaH
Aesop	Rapide
Aesop	SADL
C2	UniCon
Darwin	Weaves
LILEANNA	Wright

229

Architecture Definition Languages

- An architecture is generally considered to consist of components and the connectors (interactions) between them however we are inconsistent and informal in their use and therefore
  - Architectural designs are often poorly understood and not amenable to formal analysis or simulation.
    - (Remember.. How can I evaluate on Architecture over another?)
  - Architectural design decisions are based more on default than on solid engineering principles.
  - Architectural constraints assumed in the initial design are not enforced as the system evolves.
- Unfortunately there are few tools to help the architectural designers with their tasks.
- To address these problems, formal languages for representing and reasoning about software architecture have been developed.
- These languages, called architecture description languages (ADLs), seek to increase the *understandability* and *reusability* of architectural designs, and enable greater degrees of analysis.
- In contrast to Module Interconnection Languages (MILS), which only describe the structure of an implemented system, ADLs are used to *define and model (the) system architecture prior to system implementation.*

230

ADL Requirements

- Our Original Dilemma
  - When to pick our architecture over another ?
- Characteristics of Architectural Descriptions
  - Common Patterns of Software Organization
    - What do all these boxes and interconnecting lines really mean?
      - Data flow ? Data dependencies?, Control ? , Functional dependencies ? Functional Sequences ?. States & Modes ?
      - therefore we really do need a more precise way in which to capture and describe an architecture
  - Examples of Common Components and Interconnections:
  - Examples of Interactions between these components
  - Critical Elements of a Design Language
  - The Language Problem for Software Architecture

231

ADL Requirements

- Examples of Common Components and Interconnections
  - Computation, Memory, Server, Controller, Link (Interfaces), (list others)
- Examples of Interactions between these components
  - Procedure Call, Data Flow, Message Passing, Shared Data, .. (list Others)
- Note that components and interactions are evident across all the architecture styles and their variants.
  - The good thing .. A common set of primitives (Abstract concepts in an earlier lecture).
- Critical Elements of a Design Language
- The Language Problem for Software Architecture

232

ADL Requirements

- Critical Elements of a Design Language
  - Components: Primitive elements and their values
    - (give examples)
  - Operators: Functions that combine Components
    - (give examples)
  - Abstraction: Naming rules for components and operators
    - (give examples)
  - Closure: Rules that determine which abstractions can be added to the classes of components and operators
    - (give examples)
  - Specification: Association of semantics with syntactic forms.
    - (give examples)

ADL Requirements

- The Language Problem for Software Architecture
  - Note: SWA deals with the overall allocation of functions to components, with data and interface connectivity and overall system balance (task allocation, file allocation, dead-lock recovery, fault-tolerance, etc.…)
    - Do conventional programming languages support this ?
    - Does UML support this ?
    - Does "Z" support this ?
    - So where do we go from here ?
  - So we need a way to allow us to combine the components, operations, interfaces etc into an ARCHITECTURE.
    - So then why not just use Ada, and CORBA, … etc.?
    - So where do programming languages fit in the scheme of things ?

ADL Requirements

- An ADL therefore must:
  - Support the description of components and their interactions
    - Why ?
  - Handle large-scale, high-level designs
    - Why?
  - Support Translation of Design to a Realization
    - Why ?
  - Support user-defined or application Specific Abstractions
    - Why?
  - Support the Disciplined selection of architectural styles.
    - Why ?

ADL Requirements

- Composition:
  - *"It should be possible to describe a system as a composition of independent components and connections"*
    - This allows us to combine independent elements into larger systems (this is really critical in Network centric independent systems that demonstrate new emergent capabilities when combined together)
    - An ADL therefore:
      - Must allow the hierarchical decomposition of and assembly of a system.
      - Final Decomposed elements must be independent (stand-alone) pieces in their own right.
      - Must be able to separate architectural design approach from realization approach.
      - Note: the ADL closure rule must allow us to view entities of an architectural description as primitive at one level and as composite structures at a lower level of decomposition.
    - Why is this important ?

ADL Requirements

- Abstraction:
  - *"It should be possible to describe the components and their interactions within the software architecture in a way that clearly and explicitly describes their abstract roles in a system"*
    - This property will allow us to describe explicitly the kind of architectural elements used and the relationships between the elements
    - Note the contrast with high level programming languages vs ADL.
    - For example:
      - It should be possible to describe an architecture without having to relay on implicit coding conventions or unstated assumptions about the intended realization.
        - (Remember how benign ACME looks)
      - Note: It should be able to indicate explicitly that components are related via a Client-Server relationship (regardless of how they might be implemented) NOT implicitly by looking at lower level IDL or procedure calls.
      - For example I could implement a C-S relationship in "C". But you would not know that we have C-S relationship unless you went digging through the low level code.)
      - We want to get away from the code and IDL level. (those are implementation realization, not abstract roles. Ie. We want to be at the Client module and a Server Module level.
        - Service based Architecture ?

ADL Requirements

- Reusability:
  - *"It should be possible to reuse components, connectors, and architectural patterns in different architectural descriptions, even if they were developed outside the context of the system"*
    - This property will allow us to describe families of architectures as an open-ended collection of architectural elements, together with constraints on the structure and semantics.
      - These Architectural patterns require further instantiation of substructure and indefinite replication of relations.
        - See the GOF book.
      - Note that programming languages permit reuse of individual components, FEW make it possible to describe generic patterns of components and connectors.
        - Programming languages provide module constructs only (Ada) few allow us to talk about collections of modules or structural patterns.
        - For example a pipeline architecture uses pipes & filters AND also constrains the topology to be a linear sequence (but we cannot describe the topology).

ADL Requirements

- Configuration:
  - *"Architectural Descriptions should localize the description of system structure independently of the elements being structured. They should also support dynamic reconfiguration."*
    - This property allows us to understand and change the architectural structure of a system without having to examine each of the systems individual components.
    - Therefore an ADL should separate the description of composite structures from the elements in these compositions so that we can reason about the composition as a whole.
    - See the comment on dynamic architectures in your text.

ADL Requirements

- Heterogeneity:
  - *"It should be possible to combine multiple, heterogeneous architectural descriptions"*
    - This property will allow us to:
      - Combine single combined different architectural patterns into a single system.
      - Combine components written in different languages
        - (marshaling).
        - Module connection systems that support interaction between distinct address spaces often provide this capability (see book examples)

ADL Requirements

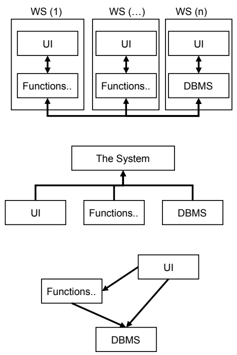
- Analysis:
  - *"It should be possible to perform rich and varied analyses of architectural descriptions"*
    - Note that current module connection languages (Ada) provide only weak support for analysis. (only type checking at component boundaries)
    - Note how JINI has started to address the issue of event broadcast.
    - Need to associate an specification with an architecture as they become relevant to a particular components, connections, and patterns.

Problems with Existing Languages

- Informal Diagrams
- Programming Language Modularization Facilities
- Module Interconnect Languages
- Support for Alternative Kinds of Interaction
- Specialized Notation for Certain Architectural Styles

Informal Diagrams

- Informal diagrams are used to express many ideas:
  - the boxes can represent anything from components to functions,
  - the interconnections are equally vague at identifying the interaction they were meant to convey.
    - Data flow? , Control Flow? , Event Handling? Inheritance ? , etc.
  - Note that although they <u>intuitively convey</u> the architecture, they are limited in the use for analysis



Issues with Programming Language Modularization Facilities

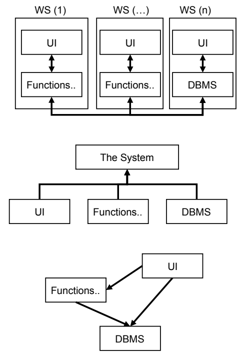
- This approach uses a programming language (PL) modularization facilities to convey the description of the architecture.
- These languages are based on the notion that modules define an interface that declares:
  - Exports: The services the module provides
  - Imports: the Services on which it relies
- Issues:
  - Composition:
    - PL provide poor support for the independent composition of architectural elements
    - Inter-module connection is determined by name matching. Good for PL but poor for AD.
    - Naming Exports and Imports forces interconnection structure to be embedded in the module definitions.
      - Consequently, modules can rarely be REUSED in another system for which they were not designed.

Issues with Programming Language Modularisation Facilities

- Abstraction:
  - PL represent module interfaces as a collection of independent procedures, data with types and possibly constraints.
  - The result is that the High Level Architecture has to be described in these low level implementation primitives of the PL.
  - Usually the interconnection is also limited to data sharing or procedure calls.
    - So how do we capture the other types of interconnections such as pipes, message passing, etc.
    - Simplicity of the ability to describe an interconnection therefore has both positive and negative effects.
      - For programming, we know the types,
      - However we do not have the freedom to describe the system interactions not can we describe the architectural components (services)
      - Forces the designer to think in only the terms of the PL primitive constructs
      - Limits reusability as one set of interconnections may not be valid for another architecture
      - Limits the level of abstraction that can be used to describe interconnections.

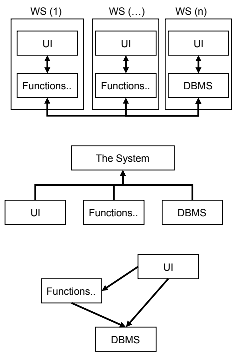
Issues with Programming Language Modularisation Facilities

- Consequence of Abstraction:
  - A MAJOR part of the design, (the interconnection) between modules at the ARCHITECURE level is therefore buried in procedure calls and shared data access, is distributed across the modules, and difficult to change because of module inter-dependencies.
- Reuse:
  - Modules explicitly declare their exports and imports, they do NOT declare the export and import REQUIREMENTS.
  - Note that module definitions ONLY supports the reuse of the module. There is NO support for reuse of the patterns of composition.



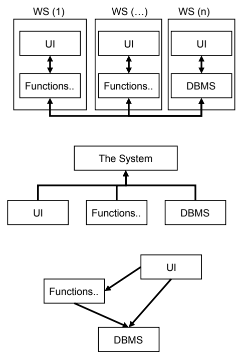
Issues with Programming Language Modularization Facilities

- Configuration:
  - The requirements of modules to define EXPORTS and IMPORTS leads to a condition where the connectivity of the architecture is distributed throughout the module definitions.
  - Makefiles are the only single place where Connectivity dependencies are visible.
  - Note that we only get a notion of the dependency NOT the nature of the dependency or design intention.
  - Also the declarations of EXPORTS and IMPORTS is STATIC. There is no notion of dynamic reconfiguration.



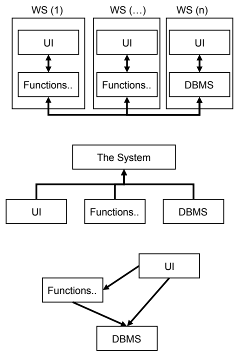
Issues with Programming Language Modularization Facilities

- Heterogeneity:
  - Modules written in different languages cannot (generally) be combined or require the use of special purpose middleware
    - (Actually the approach DEC took to their language interface easily allowed Fortran to call PL/I to call C … etc. (on the same machine).
  - Due to the limited number of abstract primitives we can only describe the interactions in the low level primitives available.
  - So we do NOT have a way in which to express architectural paradigms, let alone combined them.



Issues with Programming Language Modularization Facilities

- Analysis:
  - Given a set of modules – What is the Architecture ?
  - The current approach of name matching (modules and EXPORTS and IMPORTS) makes it difficult to check for consistency of interconnection.
  - Name matching DOES NOT assure proper use !!



SADL: Structural Architecture Description Language

- http://www.sdl.sri.com/programs/dsa/sadl-main.html
- http://www.sdl.sri.com/programs/dsa/README.html
- SADL (Stanford): A Language for Specifying Software Architecture Hierarchies
  - Sadl is programming language independent, intended for both abstract and concrete modeling of system architectures.
  - The Sadl language provides a precise textual notation for describing software architectures while retaining the intuitive box-and-arrow model.
  - (Sadl) makes a clean distinction between several kinds of architectural objects (e.g., components and connectors) and make explicit their intended uses.
  - The Sadl language provides facilities for specifying architectures and for also specifying well-formed ness constraints on particular classes of architectures.
    - For example, it is possible to specify not only the kinds of components and connections in, client-server and blackboard systems, but also the intended configurations of the components and connections.
    - Sadl can be used to describe the architecture of a specific system or a family of systems.

SADL: Structural Architecture Description Language

- A vertical hierarchy serves to bridge the gap between abstractions in architectures and the more primitive structural concepts in conventional programming languages.
- Each level in a vertical hierarchy typically is described using a different vocabulary, reflecting a change in representation.
  - For example, a pipe-and-filter architecture would be described using a different vocabulary than an event-based architecture.
- A horizontal hierarchy is analogous to ``bubble decomposition'' in dataflow modeling, where the same vocabulary is used to describe every level in the decomposition.

SADL: Structural Architecture Description Language

- The Sadl language is intended to be used in describing both vertical and horizontal hierarchy and in relating different levels of representation by means of mappings.
- They also make it possible to reason about the relationship among architectures in a hierarchy and the relation between the hierarchy and its implementation.
- For example, we can determine whether the architectural objects in one architecture are present in the other even if there is a change in representation.
- We also can show that, if a particular communication path is not allowed in one architecture, it is not allowed in others in the hierarchy.
- An architecture hierarchy may describe a specific system or a family of systems.

SADL: Structural Architecture Description Language

- Sadl can be used to represent the following architectural elements.
- 1. Architecture. An architecture is a, possibly parameterized, collection of the following items.
- (a) Component. A component represents a locus of computation or a data store.
  - The various types of components include a module, process, procedure, or variable.
  - A component has a name, a type (a subtype of type COMPONENT), and an interface, the ports of the component.
  - A port is a logical point of interaction between a component and its environment.
  - A port has a name, a type, and is designated for input or output.

253

SADL: Structural Architecture Description Language

- (b) Connector.
  - A connector is a typed object (a subtype of type CONNECTOR) relating ports.
  - Every connector is required to accept values of a given type on one end and to produce output values of the same type on the other.
- (c) Configuration.
  - A configuration constrains the wiring of components and connectors into an architecture.
  - A configuration can contain two kinds of elements.
    - Connections. A connection associates type-compatible connectors and ports.
    - Constraints. Constraints are used to relate named objects or to place semantic restrictions on how they can be related in an architecture.

254

SADL: Structural Architecture Description Language

- Mapping.
  - A mapping is a relation that defines a syntactical interpretation from the language of an abstract architecture to the language of a concrete architecture.
- Architectural style.
  - A style consists of a vocabulary of design elements, well formed-ness constraints that determine how they can be used, any semantic constraints needed for refinement, and a logical definition of the semantics of the connectors associated with the style.
  - A constraint is declarative and might say, for example, that clients initiate communication with servers, but not vice versa.
  - A given architecture may be homogeneous, involving one style, or heterogeneous, involving multiple styles.

255

SADL: Structural Architecture Description Language

- Refinement pattern.
  - A refinement pattern consists of two architecture schemas, an association of the objects in the two schemas, and possibly constraints on one or both schemas.
  - An instance of a pattern is formed by matching schema variables against the appropriate portions of Sadl specifications.
- Components, interfaces, connectors, and constraints:
  - Are treated as first-class objects --- i.e., they are named and typed objects that can appear as parameters.
  - They can be refined into (decomposed, aggregated, or eliminated) objects in more concrete architectures.

256

ADL ROLES

- Provide *models*, *notations*, and *tools* to describe components and their interactions
- Support for large scale, high level designs
- Support for *principled selection* and *application of architectural paradigms*
- Support for abstractions
  - user defined
  - application specific
- Support for implementing designs
  - systematic
  - possibly automated
- Close interplay between language and environment
  - language enables precise specifications
  - environment makes them (re)usable

WHAT DOES AN ADL DESCRIPTION LOOK LIKE? (1)

- A Rapide Component

type Application **is interface**
extern action Request(p : params);
public action Results(p : params);
behavior
(?M **in** String) Receive(?M) => Results(?M);;
end Application;

WHAT DOES AN ADL DESCRIPTION LOOK LIKE? (2)

- A Wright connector

connector Pipe = **role** W = write → W ∩ close → √ **role** R = **let** Exit = close → √ **in let** DoR = (read → R read-eof → Exit) **in** DoR ∩ Exit	**glue** = **let** ROnly = R.read → ROnly R.read-eof → R.close → √ R.close → √ **in let** WOnly = W.write → WOnly W.close → √ **in** W.write → **glue** R.read → **glue** W.close → ROnly Reader.close → WriteOnly

WHAT DOES AN ADL DESCRIPTION LOOK LIKE? (2)

- An ACME architecture

System simple_cs = {
Component client = {Port send-request}
Component server = {Port receive-request}
Connector rpc = {Roles {caller, callee}}
Attachments : {
client.send-request to rpc.caller;
server.receive-request to rpc.callee
}
}

ADL DEFINITION

- ADL Definition
  - An ADL is a language that provides features for modelling a software system's conceptual architecture
- *Essential* features: *explicit* specification of
  - components
    - interfaces
  - connectors
  - configurations
- *Desirable* features
  - specific aspects of components, connectors, and configurations
  - tool support

261

DIFFERENTIATING ADLS

- Approaches to modeling configurations
  - implicit configuration
  - in ine configuration
  - explicit configuration
- Approaches to associating architecture with implementation
  - implementation constraining
  - implementation independent

262

ADL COMPONENTS

- Definition
  - A *component* is a unit of computation or a data store or a unit of structure
  - Components are loci of computation and state
- All ADLs support component modeling
- Differing terminology
  - component
  - interface
  - process

263

COMPONENT CLASSIFICATION CATEGORIES

- Interfaces
  - model both required and provided services
- Types
  - enable reuse and multiple instances of the same functionality
- Semantics
  - facilitate analyses, constraint enforcement, and mapping of architectures across levels of refinement
- Constraints
  - ensure adherence to intended component uses, usage boundaries, and intra-component dependencies
- Evolution
  - components as design elements evolve
  - supported through subtyping and refinement

264

ADL CONNECTORS

- Definition
  - A *connector* is an architectural building block used to model interactions among components and rules that govern those interactions
- All ADLs support connector modeling
  - several ADLs do not model connectors as first-class entities
  - all ADLs support at least syntactic interconnection
- Differing terminology
  - connector
  - connection
  - binding

CONNECTOR CLASSIFICATION CATEGORIES

- Interfaces
  - ensure proper connectivity and communication of components
- Types
  - abstract away and reuse complex interaction protocols
- Semantics
  - analyse component interactions, enforce constraints, and ensure consistent refinements
- Constraints
  - ensure adherence to intended interaction protocols, usage boundaries, and intra-connector dependencies
- Evolution
  - maximize reuse by modifying or refining existing connectors

ADL CONFIGURATIONS

- Definition
  - An *architectural configuration* or *topology* is a connected graph of components and connectors that describes architectural structure
- ADLs must model configurations explicitly by definition
- Configurations help ensure architectural properties
  - proper connectivity
  - concurrent and distributed properties
  - adherence to design heuristics and style rules

CONFIGURATION CLASSIFICATION CATEGORIES (1)

- Understandability
  - enables communication among stakeholders
  - system structure should be clear from configuration alone
- Compositionality
  - system modeling and representation at different levels of detail
- Heterogeneity
  - development of large systems with pre-existing elements of varying characteristics
- Constraints
  - depict dependencies among components and connectors

CONFIGURATION CLASSIFICATION CATEGORIES (2)

- Refinement and Traceability
  - bridge the gap between high level models and code
- Scalability
  - supports modelling of systems that may grow in size
- Evolution
  - evolution of a single system or a system family
- Dynamism
  - enables runtime modification of long running systems
- Non Functional Properties
  - enable simulation, analysis, constraints, processor specification, and project management

269

TOOL SUPPORT CLASSIFICATION CATEGORIES

- Active Specification
  - support architect by reducing cognitive load
  - proactive vs reactive
- Multiple Views
  - support for different stakeholders
- Analysis
  - upstream evaluation of large, distributed, concurrent systems
- Refinement
  - increase confidence in correctness and consistency of refinement
- Code Generation
  - ultimate goal of architecture modeling activity
  - manual approaches result in inconsistencies and lack of traceability
- Dynamism
  - enable changes to architectures during execution

270

Architectural Description Languages

- Architectural Description Languages can be evaluated or described by listing the following:
  - System-oriented attributes
  - Language-oriented attributes
  - Process-oriented attributes

271

ADLs: System-Oriented Attributes

- How suitable is the ADL for representing a particular type of application?
- How well does the ADL allow descriptions of architectural style?
- What broad classes of system can have their architectures represented in the ADL?
  - E.g., hard real-time, distributed, embedded etc.,

272

ADL's: Language-Oriented Attributes

- Syntax and semantics formally defined?
- Does the ADL define completeness for an architecture
- Does the ADL support the ability to add new types of components, connectors
- How easily is the software architecture description modified?
- How saleable are its descriptions etc.,

273

ADL's: Process-Oriented Descriptions

- Is there a textual editor or tool for manipulating ADL text?
- Is there a graphical editor?
- Can the tool import information from other descriptions into the architecture?
- Does the ADL support incremental refinement?
- Does the ADL support comparison between two architectures?

274

Topic 6: Architectural Styles

Architectural Styles

- Shaw and Garlan present a number of architectural styles, identified by asking:
  - What is the design vocabulary ?
    - types of connectors and components
  - What are the allowable structural patterns?
  - What is the underlying computational model?
  - What are the essential invariants of the style?
  - What are some common examples of its use?
  - What are the advantages/disadvantages of use?
  - What are the common specialisations?

276

...But Style Has Proven to Help

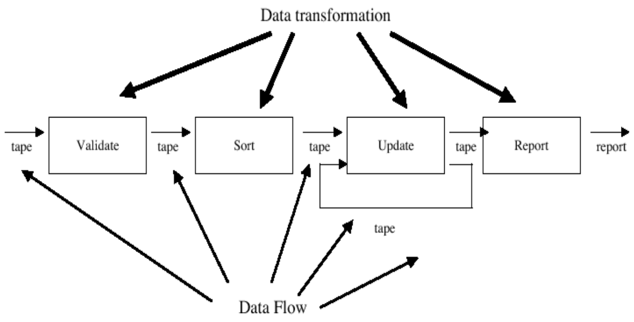
- Architectural styles restrict the way in which components can be connected
  - Prescribe patterns of interaction
  - Promote fundamental principles
    - » Rigor, separation of concerns, anticipation of change, generality, incrementality
    - » Low coupling
    - » High cohesion
- Architectural styles are based on success stories
  - Almost all compilers are build as "pipe-and-filter"
  - Almost all network protocols are build as "layers"

Common Architectural Idioms

- Data flow systems
  - (1) Batch sequential & (2) pipe-and-filter
- (3) Data and/or service-centric systems: the Client-Server style
- The (pre-1994) WWW
  - Database servers
- (4) Hierarchical systems
  - Main program and subroutines;
- (5) Data abstraction/OO systems
- (6) Peer-to-Peer
- (7) Layered systems
- (8) Interpreters
- (9) Implicit invocation (event-based)
- (10) Three-level architectures

Note: not all of these are of equal value, current use, or intellectual depth

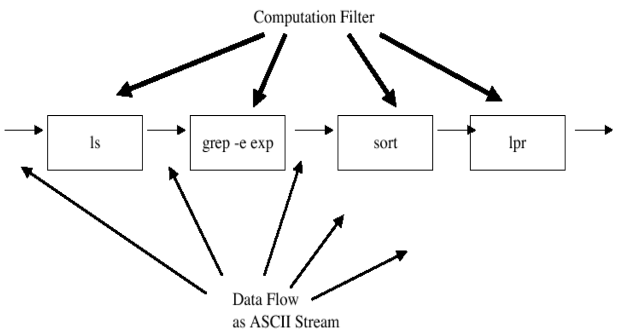
Style 1: Batch Sequential



Batch Sequential

- Components
  - components are independent programs
  - each component runs to completion before next step starts
- Connections
  - Data transmitted as a whole between components
- Topology
  - Connectors define data flow graph
- Typical application: classical data processing

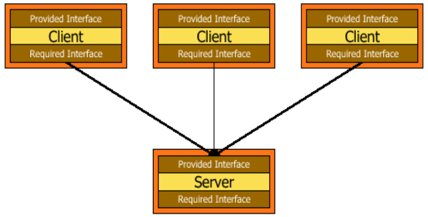
Style 2: Pipe and filter



Pipe and filter

- Components
  - Like batch sequential, but components (filters) incrementally transform some amount of the data at their inputs to data at outputs
  - Little local context used in processing input stream
  - No state preserved between instantiations
- Connections
  - Pipes move data from a filter output to a filter input
  - Data is a stream of ASCII characters
- Topology
  - Connectors define data flow graph
- Pipes and filters run (non-deterministically) until no more computation possible
- Typical applications: many Unix applications

Style 3: Client-Server



Connections are remote procedure calls or remote method invocations

Client-Server Systems

- Components
  - 2 distinguished kinds
    - Clients: towards the user; little persistent state; active (request services)
    - Servers: "in the back office"; maintains persistent state and offers services; passive
- Connectors
  - Remote procedure calls or network protocols
- Topology
  - Clients surround the server

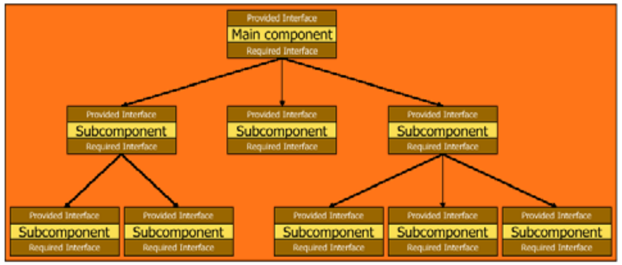
The pre-1994 WWW as a Client-Server Architecture

- Browsers are clients
- Web servers maintain state
- Connections by HTTP/1.0 protocol

Database Centered Systems

- Components
  - Central data repository
  - Schema (how the data is organized) designed for application
  - Independent operators
    - » Operations on database implemented independently, one per transaction type
    - » interact with database by queries and updates
- Connections
  - Transaction stream drives operation
  - Operations selected on basis of transaction type
  - May be direct access to data; may be encapsulated

Style 4: Hierarchy: Main Program and Subroutines

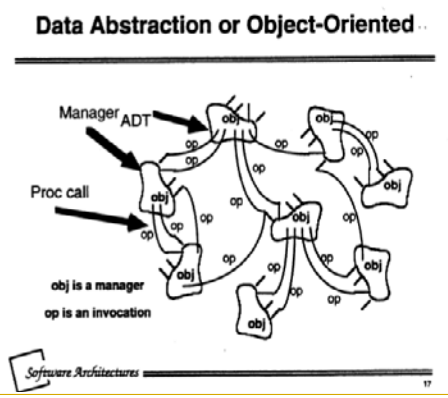


Connections are function or method calls

Main Program and Subroutines

- Components
  - Computational elements as provided by programming language
  - Typically single thread
- Connections
  - Call/return as provided by programming language
  - Shared memory
- Topology
  - Hierarchical decomposition as provided by language
  - Interaction topologies can vary arbitrarily

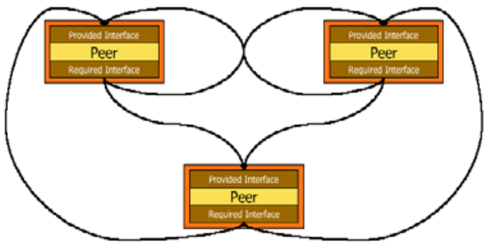
Style 5: Data Abstraction/OO Systems



Data Abstraction/OO Systems

- Components
  - Components maintain encapsulated state, with public interface
  - Typically single threaded, though not logical
- Connections
  - Procedure calls ("method invocations") between components
  - Various degrees of polymorphism and dynamic binding
  - Shared memory a common assumption
- Topology
  - Components may share data and interface functions through inheritance hierarchies
  - Interaction topologies can vary arbitrarily

Style 6: Peer-to-Peer

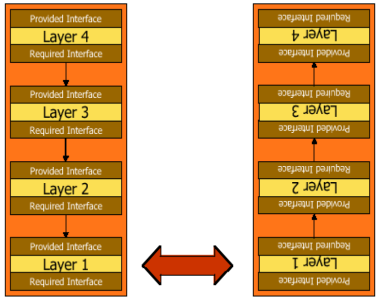


Connections are remote procedure calls or remote method invocations

Peer-to-Peer Architectures

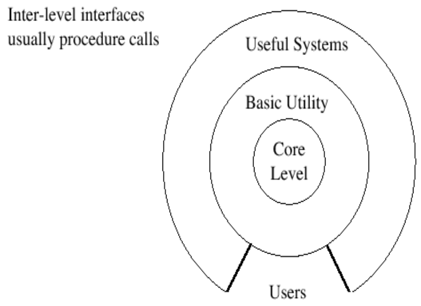
- Components
  - Autonomous
  - Act as both clients and servers
- Connectors
  - Asynchronous and synchronous message passing ("remote procedure calls")
  - By protocols atop TCP/IP
  - No shared memory (except as an optimization when the configuration allows)
- Topology
  - Interaction topologies can vary arbitrarily and dynamically

Style 7: Layered Systems, Take 1



Connections are function or method calls + "something in between"

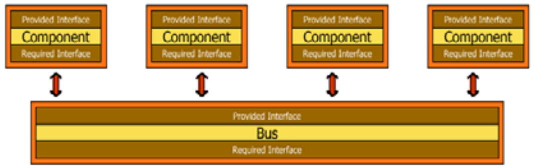
Layered Systems, Take 2



Layered Systems

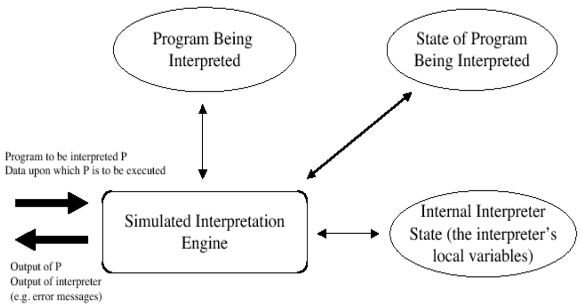
- Components
  - Each layer provides a set of services
- Connections
  - Typically procedure calls
  - A layer typically hides the interfaces of all layers below, but others use "translucent" layers
- Topology
  - Nested
- Typical applications: support for portability, systems with many variations ("core features" v. extended capabilities)

Style 8: Implicit Invocation



Connections are events on the software bus

Style 9: Interpreters



297

Interpreters

- Components
  - Execution engine simulated in software (with its internal data)
  - Program being interpreted
  - State of program being interpreted
- Connections
  - program being interpreted determines sequence of actions by interpreter
  - shared memory
- Topology
- Typical applications: end-user customization; dynamically changing set of capabilities (e.g. HotJava)

298

Style 10: "Three Level Architectures"

- User interface
- Application Logic
- Database (server)

299

Choosing the Right Style

- Ask questions on whether a certain style makes sense
  - The Internet as a blackboard
    - » Does that scale?
  - Stock exchange as a layers
    - » How to deal with the continuous change?
  - Math as hierarchy
    - » How to properly call different modules for different functions?
- Draw a picture of the major entities
- Look for the natural paradigm
- Look for what "feels right"

300

Software architecture styles

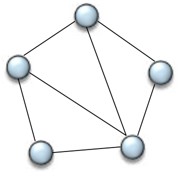
Aim similar to Design Patterns work: classify styles of software architecture

Characterizations are more abstract; no attempt to represent them directly as code

Software architecture styles

An architectural style is defined by

- Type of basic architectural components (e.g. classes, filters, databases, layers)
- Type of connectors (e.g. calls, pipes, inheritance, event broadcast)



Architecture styles

Overall system organization:

- Hierarchical
- Client-server
- Cloud-based
- Peer-to-peer

Individual program structuring:

- Control-based
  - Call-and-return (Subroutine-based)
  - Coroutine-based
- Dataflow:
  - Pipes and filters
  - Blackboard
  - Event-driven
- Object-oriented

Hierarchical

Each layer provides **services to the layer above** it and acts as a client of the layer below

Each layer collects services at a particular level of **abstraction**

A layer depends only on lower layers

- Has no knowledge of higher layers

Example

- Communication protocols
- Operating systems

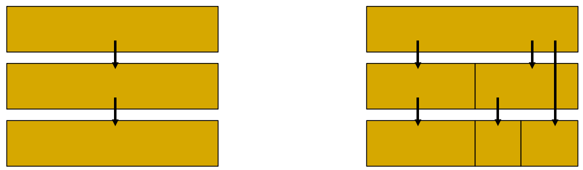
Hierarchical

Components

- Group of subtasks which implement an abstraction at some layer in the hierarchy

Connectors

- Protocols that define how the layers interact



Hierarchical: examples

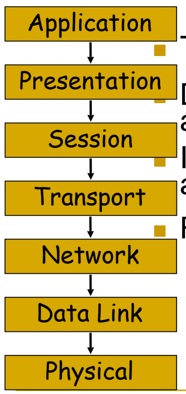
THE operating system (Dijkstra) The OSI Networking Model

- Each level supports communication at a level of abstraction
- Protocol specifies behavior at each level of abstraction
- Each layer deals with specific level of communication and uses services of the next lower level

Layers can be exchanged

- Example: Token Ring for Ethernet on Data Link Layer

OSI model layers



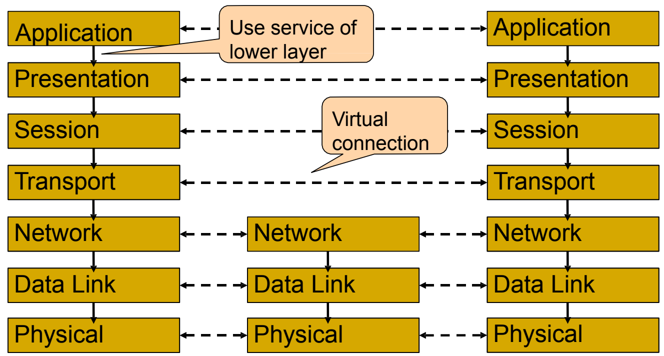
- The system you are designing
- Data transformation services, such as byte swapping and encryption
- Initializes a connection, including authentication
- Reliably transmits messages

Transmits & routes data within network

Sends & receives frames without error

Sends and receives bits over a channel

Hierarchical style example



Hierarchical: discussion

Strengths:

- Separation into levels of abstraction; helps partition complex problems
- Low coupling: each layer is (in principle) permitted to interact only with layer immediately above and under
- Extendibility: changes can be limited to one layer
- Reusability: implementation of a layer can be reused

Weaknesses:

- Performance overhead from going through layers
- Strict discipline often bypassed in practice

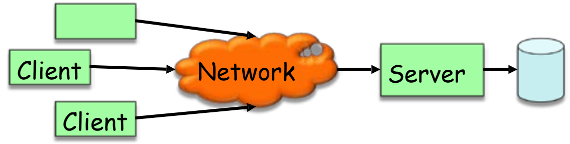
309

Client-server

Components

- Subsystems, designed as independent processes
- Each server provides specific services, e.g. printing, database access
- Clients use these services Connectors
- Data streams, typically over a communication network

Client



310

Client -server example: databases

Clients: user applications

- Customized user interface
- Front-end processing of data
- Initiation of server remote procedure calls
- Access to database server across the network

Server: DBMS, provides:

- Centralized data management
- Data integrity and database consistency
- Data security
- Concurrent access
- Centralized processing

311

Client-server variants

Thick / fat client

- Does as much processing as possible
- Passes only data required for communications and archival storage to the server
- Advantage: less network bandwidth, fewer server requirements

Thin client

- Has little or no application logic
- Depends primarily on server for processing
- Advantage: lower IT admin costs, easier to secure, lower hardware costs.

312

Client-server: discussion

Strengths:

- Makes effective use of networked systems
- May allow for cheaper hardware
- Easy to add new servers or upgrade existing servers
- Availability (redundancy) may be straightforward

Weaknesses:

- Data interchange can be hampered by different data layouts
- Communication may be expensive
- Data integrity functionality must be implemented for each server
- Single point of failure

313

Client-server variant: cloud computing

The server is no longer on a company's network, but hosted on the Internet, typically by a providing company

Example: cloud services by Google, Amazon, Microsoft

Advantages:

- Scalability
- Many issues such as security, availability, reliability are handled centrally

Disadvantages:

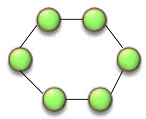
- Loss of control
- Dependency on Internet

314

Peer-to-peer

Similar to client-server style, but **each component is both client and server**

Pure peer-to-peer style



- No central server, no central router

Hybrid peer-to-peer style

- Central server keeps information on peers and responds to requests for that information

Examples

- File sharing applications, e.g., Napster
- Communication and collaboration, e.g., Skype

315

Peer-to-peer: discussion

Strengths:

- Efficiency: all clients provide resources
- Scalability: system capacity grows with number of clients
- Robustness
  - Data is replicated over peers
  - No single point of failure (in pure peer-to-peer style)

Weaknesses:

- Architectural complexity
- Resources are distributed and not always available
- More demanding of peers (compared to client-server)
- New technology not fully understood

316

Call-and-return

Components: Objects
Connectors: Messages (routine invocations) Key aspects

- Object preserves integrity of representation (encapsulation)
- Representation is hidden from client objects

Variations

- Objects as concurrent tasks

Call-and-return

Strengths:

- Change implementation without affecting clients
- Can break problems into interacting agents
- Can distribute across multiple machines or networks

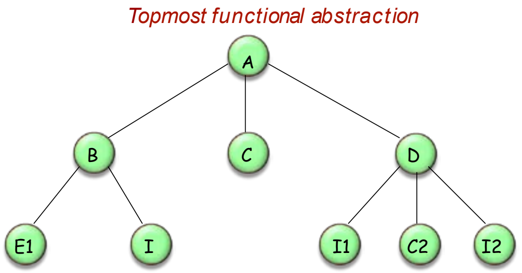
Weaknesses:

- Objects must know their interaction partners; when partner changes, clients must change
- Side effects: if *A* uses *B* and *C* uses *B*, then *C*s effects on *B* can be unexpected to *A*

Subroutines

Similar to hierarchical structuring at the program level

Functional decomposition



Subroutines

Advantages:

- Clear, well-understood decomposition
- Based on analysis of system's function
- Supports top-down development

Disadvantages:

- Tends to focus on just one function
- Downplays the role of data
- Strict master-slave relationship; subroutine loses context each time it terminates
- Adapted to the design of individual functional pieces, not entire system

Dataflow systems

Availability of data controls the computation

The structure is determined by the orderly motion of data from component to component

Variations:

- Control: push versus pull
- Degree of concurrency
- Topology

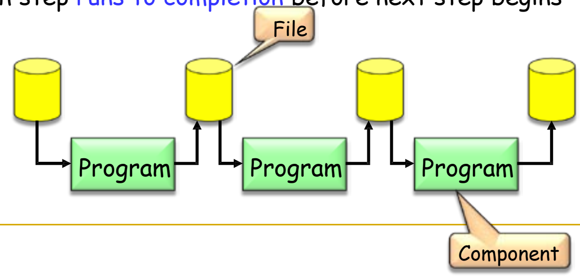
Dataflow: batch-sequential

Frequent architecture in scientific computing and business data processing

Components are independent programs

Connectors are media, typically files

Each step runs to completion before next step begins



Batch-sequential

History: mainframes and magnetic tape

Business data processing

- Discrete transactions of predetermined type and occurring at periodic intervals
- Creation of periodic reports based on periodic data updates

Examples

- Payroll computations
- Tax reports

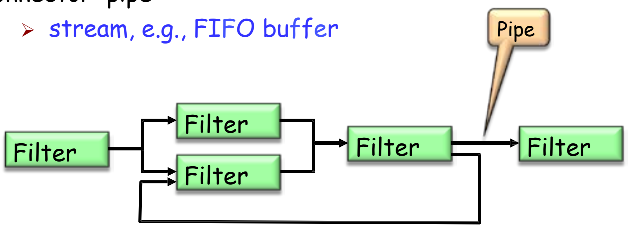
Dataflow: pipe-and-filter

Component: filter

- Reads input stream (or streams)
- Locally transforms data
- Produces output stream (s)

Connector: pipe

- stream, e.g., FIFO buffer



Pipe-and-filter

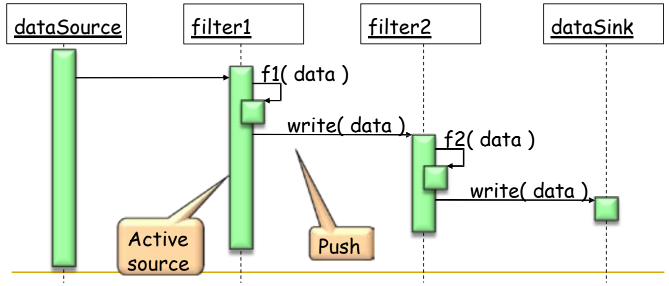
Data processed incrementally as it arrives Output can begin before input fully consumed

Filters must be independent: no shared state
Filters don’ t know upstream or downstream filters

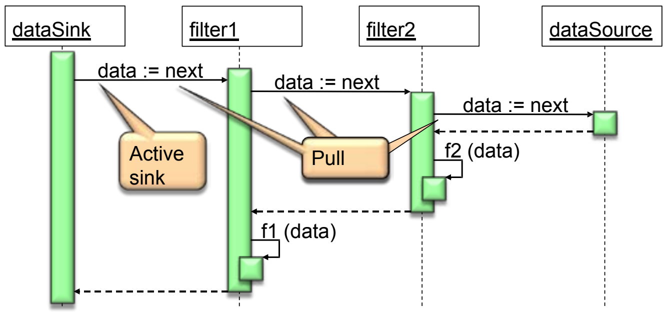
Examples
- lex/yacc-based compiler (scan, parse, generate…)
- Unix pipes
- Image / signal processing

Push pipeline with active source

Source of each pipe pushes data downstream
Example with Unix pipes:
grep p1 * | grep p2 | wc | tee my_file



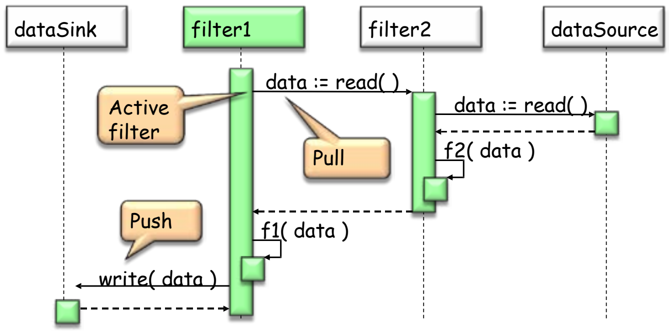
Pull pipeline with active sink



- Sink of each pipe pulls data from upstream
- Example: Compiler: t := lexer.next_token

Combining push and pull

Synchronization required:



Pipe-and-filter: discussion

Strengths:

- Reuse: any two filters can be connected if they agree on data format
- Ease of maintenance: filters can be added or replaced
- Potential for parallelism: filters implemented as separate tasks, consuming and producing data incrementally

Weaknesses:

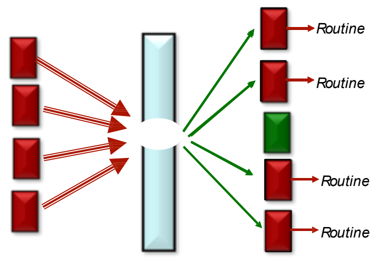
- Sharing global data expensive or limiting
- Scheme is highly dependent on order of filters
- Can be difficult to design incremental filters
- Not appropriate for interactive applications
- Error handling difficult: what if an intermediate filter crashes?
- Data type must be greatest common denominator, e.g. ASCII

Dataflow: event-based (publish-subscribe)

A component may:

- Announce events
- Register a callback for events of other components

Connectors are the bindings between event announcements and routine calls (callbacks)



Event-based style: properties

Publishers of events do not know which components (subscribers) will be affected by those events

Components cannot make assumptions about ordering of processing, or what processing will occur as a result of their events

Examples

- Programming environment tool integration
- User interfaces (Model-View-Controller)
- Syntax-directed editors to support incremental semantic checking

Event-based style: example

Integrating tools in a shared environment

Editor announces it has finished editing a module

- Compiler registers for such announcements and automatically re-compiles module
- Editor shows syntax errors reported by compiler

Debugger announces it has reached a breakpoint

- Editor registers for such announcements and automatically scrolls to relevant source line

Event-based: discussion

Strengths:

- Strong support for reuse: plug in new components by registering it for events
- Maintenance: add and replace components with minimum effect on other components in the system

Weaknesses:

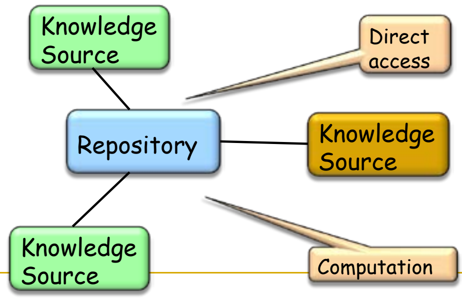
- Loss of control:
  - What components will respond to an event?
  - In which order will components be invoked?
  - Are invoked components finished?
- Correctness hard to ensure: depends on context and order of invocation

333

Data-centered (repository)

Components

- Central data store component represents state
- Independent components operate on data store



334

Data-Centered: discussion

Strengths:

- Efficient way to share large amounts of data
- Data integrity localized to repository module

Weaknesses:

- Subsystems must agree (i.e., compromise) on a repository data model
- Schema evolution is difficult and expensive
- Distribution can be a problem

335

Blackboard architecture

Interactions among knowledge sources **solely through repository**

Knowledge sources make changes to the shared data that lead incrementally to solution

Control is driven entirely by the state of the blackboard

Example

- Repository: modern compilers act on shared data: symbol table, abstract syntax tree
- Blackboard: signal and speech processing

336

Interpreters

Architecture is based on a **virtual machine** produced in software

Special kind of a **layered architecture** where a layer is implemented as a true language interpreter

Components

- "Program" being executed and its data
- Interpretation engine and its state

Example: Java Virtual Machine

- Java code translated to platform independent bytecode
- JVM is platform specific and interprets the bytecode

Object-oriented

Based on analyzing the types of objects in the system and deriving the architecture from them

Compendium of techniques meant to enhance extendibility and reusability: contracts, genericity, inheritance, polymorphism, dynamic binding…

Thanks to broad notion of what an "object" is (e.g. a command, an event producer, an interpreter…), allows many of the previously discussed styles

Conclusion: assessing architectures

General style can be discussed ahead of time

Know pros and cons

Architectural styles → Patterns → Components

Architectural Style

- An architectural style is a description of <u>component types</u> and their topology.
- It also includes a description of the pattern of <u>data and control interaction</u> among the components and an informal description of the benefits and drawbacks of using that style.
- Architectural styles define classes of designs along with their associated known properties.
- They offer experience-based evidence of how each class has been used historically, along with qualitative reasoning to explain why each class has its specific properties. (patterns)

Architecture Styles

- Looking for a Uniform Description of an Architecture
  - Which kinds of Components and connectors are used in the style
    - Examples: programs, objects, processes, filters
    - The allowable kinds of components and connectors are primary discriminants among the styles, however the following four items also contribute to defining the particular style
  - How is control shared, allocated m and transferred among the components
    - Topology : ? Linear (non-branching) , ? A cyclical, ? Hierarchical, ? Star , ? Arbitrary, ? Static or Dynamic
    - Synchronicity: ? Lockstep (sequential or parallel depending on the threads of control) , ? Synchronous, ? Asynchronous
  - How is data communicated through the system
    - Data Flow Topology as above for control
    - Continuity: Continuous Flow: fresh data available at all times, Sporadic Flow, High-Volume vs Low Volume (see USB)
      - (? How much network band-with is / can be dedicated to data synchronization)
    - Mode: Describes how data is made available throughout the system
      - Object style: data is passed from component to component
      - Shared data style: data is made available in a place accessible to all
        - Copy out- Copy In mode, vs. broadcast or multicast,
  - How do Data and control interact
    - (data flow & topology vs. control flow & topology)
      - Pipe & Filter (data & control pass together) vs. Client-Server control flows into the servers and data flows in to the clients.
  - What type of reasoning (analysis) is compatible with the style
    - Asynchronously operating components (Non-deterministic) vs. fixed sequence of atomic steps,

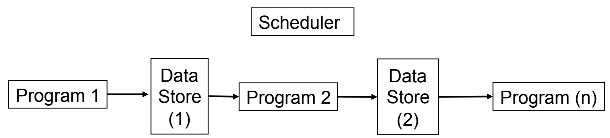
A View of Architecture Styles

- Data Flow
  - (1) Batch Sequential (traditional systems) & Pipeline Systems
  - (2) Pipes & Filters (Linked Stream transformers)
- Call & return
  - (3) Main Program & Subroutine
  - (4) OO Systems
  - (5) Hierarchical Layers
- Independent Components
  - (6) Communicating Processes
  - (7) Event Systems
- Virtual Machines
  - (8) Interpreters
  - (9) Rule-Base systems
- Data-Centered Systems (Repository)
  - (10) Databases
  - (11) Hypertext Systems
  - (12) Blackboards
  - (13) Distributed Processing Styles
  - (14) Process Control Styl

Batch Sequential

- http://www-2.cs.cmu.edu/afs/cs/project/tinker-arch/www/html/1998/Lectures/06.dataflow/quick_index.html
- General Constructs:
  - Processing Steps are independent programs
  - Each steps runs to completion before the next program starts
  - Data is transmitted in complete data sets between programs
  - Historically used in Data processing
  - Needs a scheduler to submit the jobs (jcl)
- Advantages:
  - Allows the designer to understand the system in terms of business process steps.
  - Easy to maintain (supposedly) and add new or replace programs. However experience has shown that program and data stores are really tied to the business process.
- Disadvantages:
  - Not good at interactive applications
  - Limited Support for concurrent execution as each program needs ALL the data before it starts.
  - Need to get ALL the data through the system before we can really see results.
  - Not responsive to changes, No Event Handling, No Fault Tolerance, Many many problems if tapes are run out of sequence.

Suggested Batch Sequential Style



Components are the Programs and Data stores.
Connectors are one way pipes that transfer bulk data sets.

Pipe & Filter (Pipeline)Architecture Styles

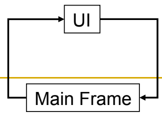
- General Constructs:
  - Pipes move data between filters.
  - The Filters must be independent entities and do NOT share state with other filters.
  - Filters Do NOT know their sources and sinks of data.
  - The correctness of the output of a pipe & filter network should not depend on the order in which the filters preformed their processing.
  - Examples: Image Processing, Unix pipe shell programs, Compilers
- Advantages:
  - Allow the designer to to understand the system in terms of composition of filters.
  - Support Re-Use
  - Easy to maintain and add new or replace old filters.
  - Permit specialized analyses: (Throughput & deadlock)
  - Each filter can be implemented as a separate task and executed in parallel with other filters.
- Disadvantages:
  - Not good at interactive applications, incremental display updates
  - May need to maintain connections between separate yet related streams.
    - Different filters types may therefore require a common representation (packing & unpacking costs)
    - Each event handled from font to back.

One way Data Flow Through a Network of Filters

- Filters can be interconnected in different ways
  - As long as the input assumptions and output behaviors are the same, one filter process or a network of filters can be replaced by a different implementation of a filter process or network that accomplish the same tasks.
    - The output of a filter process is a function of its input
    - This specification relates the value of messages sent on output channels to the values of messages received on input channels.
    - The actions a filter takes in response to receiving input must ensure this relation every time the filter sends output.
    - Requires us to understand and specify our communication and underlying data assumptions.
    - The output produced by one filter meet the input assumptions of another.

Call & Return Style: Mainframe

- General Constructs:
  - All intelligence is within the central host computer.
  - Users interact with the host through a terminal that captures keystrokes and sends that information to the host.
- Advantages:
  - Mainframe software architectures are not tied to a hardware platform.
  - User interaction performed with workstations.
- Disadvantages:
  - A limitation of mainframe software architectures is that they do not easily support graphical user interfaces or access to multiple databases from geographically dispersed sites.
  - Mainframes have found a use as a server in distributed client/server architectures

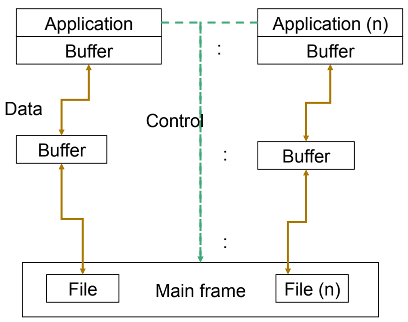


Note the connectors may be two-way as contrasted with the dataflow style as we have control from UI to Mainframe and Data from the Mainframe to UI

Call & Return Style: File Sharing

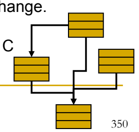
- General Constructs:
  - The original PC networks were based on file sharing architectures, where the server downloads files from the shared location to the desktop environment.
  - The requested user job is then run (including logic and data) in the desktop environment.
- Advantages:
  - File sharing architectures work if shared usage is low, update contention is low, and the volume of data to be transferred is low.
- Disadvantages:
  - In the 1990s, PC LAN (local area network) computing changed because the capacity of the file sharing was strained as the number of online user grew (it can only satisfy about 12 users simultaneously) and graphical user interfaces (GUIs) became popular (making mainframe and terminal displays appear out of date).
- Addendum:
  - PCs are now being used in client/server architectures.

Suggested File Sharing Architecture



Data Abstraction & OO Architecture

- General Constructs:
  - Data representations and their associated operations encapsulated in an abstract data type.
  - The *components* are the objects and connectors operate through procedure calls (methods).
  - Objects maintain the integrity of a resource and the representation is hidden from others.
- Advantages:
  - Server is able to change an implementation without affecting the client.
  - Grouping of methods with objects allows for more modular design and therefore decomposes the problems into a series of collections of interacting agents.
- Disadvantages:
  - For one object to interact with another, the client object must know how to interact with the server object and therefore must know the identity of the server.
  - If the server changes its interface ALL interacting clients must also change.
    - Services declared as PUBLIC and IMPORTED into the CLIENT
  - Also need to consider side affects, A uses B , B uses C and we mod C



Hierarchical Layered Architecture Styles

- Layered Systems
  - General Constructs
    - A layered system is organized hierarchically with each layer providing service to the layer above it and serving as a client to the layer below.
    - In some systems inner layers are hidden from all except the adjacent outer layer.
    - Connectors are defined by the protocols that determine how layers will interact.
    - Constraints include limiting interactions to adjacent layers. The best known example of this style appears in layered communication protocols OSI-ISO (Open Systems Interconnection - International Standards Organization) communication system.
    - Lower levels describe hardware connections and higher levels describe application.

GUI
Application Layer
DBMS
OS Layer

Hierarchical Layered Architecture Styles

- Issues:
  - No one agrees exactly on what a 'layer' is.
    - For example, some layering schemes have very little relationship to the runtime.
    - Others confuse layers with a 'tiered' runtime architecture where the various layers are not only split by build-time packaging, but by running in different processes and possibly different nodes at run time.
- Clearly separate the idea of 'tiers' from 'layers'.
  - A tier is a structure for runtime component interaction that implies nothing about the construction of the build time.
    - For example, a 3 tier system may be composed of a web browser, web server/asp pages, and database.
    - Note that each tier runs in a different process and possibly many different system nodes.

Hierarchical Layered Architecture Styles

- Layers is a pattern for 'application architectures'.
  - Layers are normally thought of as a build-time structuring technique for building an application or service that will execute in a single process.
- There are many variations on the layers pattern.
  - Four-layer architecture (analysis)
  - Layered and sectioned architecture
    - (http://c2.com/ppr/envy/#well_encapsulated_code)
  - Layered architecture
  - Layers (from POSA1)
  - Patterns for generating a layered architecture
  - Relational database access layer
    - (http://www.objectarchitects.de/ObjectArchitects/orpatterns/index.htm?Performance/performance.htm)
- Secure access layer (analysis)

GUI
Application Layer
DBMS
OS Layer

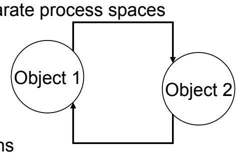
Hierarchical Layered Architecture Styles

- Layered Systems
  - Advantages:
    - Layered systems support designs based on increasing levels of abstraction.
    - Complex problems may be partitioned into a series of steps.
    - Enhancement is supported through limiting the number of other layers with which communication occurs.
  - Disadvantages:
    - Disadvantages include the difficulty in structuring some systems into a layers.
    - Performance considerations may not be well served by layered systems especially when high level functions require close coupling to low level implementations.
    - It may be difficult to find the right level of abstraction especially if existing systems cross several layers.

Communicating Processes

- Each component has it's own thread of execution.
- The Provider/Observer Role Pattern
- Push and Pull Interaction Patterns
- Opaque Interaction Patterns
  - Synchronous Opaque Interaction Patterns
  - Asynchronous Opaque Interaction Patterns
- Monitorable Interaction Patterns
  - Pull-Monitorable Interaction Patterns
  - Push-Monitorable Interaction Patterns
- Abortable Interaction Patterns
  - Abortable Async Opaque Interaction Patterns
  - Abortable Pull-Monitorable Interaction Patterns
  - Abortable Push-Monitorable Interactions Patterns
- Handshaking Patterns
- Combining Patterns

The *components* are the objects and connectors operate through message passing. Components are often in separate process spaces



Event Based Architecture Styles

- Event-Based Implicit Invocation (Look at Jini)
  - General Constructs:
    - A component announces (broadcasts) one or more events.
    - System Components register interest in an event by associating a procedure with it.
    - The system invokes all events which have registered with it.
    - Event announcement ``implicitly'' causes the invocation of procedures in other models.
    - This style originates in constraint satisfaction (Planning), daemons, and packet-switched networks.
      - Used in Planning Domains
    - Architectural components are modules whose interface provides both a collection of procedures and a set of events.
    - Procedures may be called normally or be registered with events in the system.
    - Implicit invocation systems are used in:
      - programming environments to integrate tools
      - database management systems to ensure consistency constraints
      - user interfaces to separate data from representation

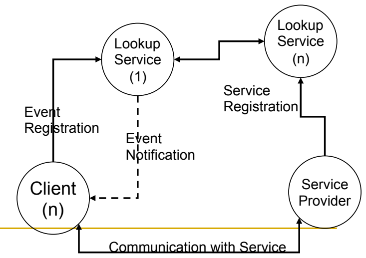
Event Based Architecture Styles

- Event-Based Implicit Invocation
  - Advantages:
    - Allows any component to register for events
    - Eases system evolution by allowing components to be replaced without affecting the interfaces of other components in the system.
  - Disadvantages:
    - Components relinquish control over the computation performed by the system.
    - A component cannot assume that other components will respond to its requests
    - A component does not know in which order events will be processed.
    - In systems with a shared repository of data the performance and accuracy of the resource manager can become critical.
    - Reasoning about correctness can be difficult because the meaning of a procedure that announces events will depend on the context in which it was invoked.

357

Event Based Architecture Styles

- The first paper predates (1996) the JINI Architecture (1999) and proposes three services
  - Event Specification (by services)
    - Event Classification
  - Event Registration (by clients at Services)
    - Event Templates
    - Event Binding to identify UNIQUE service "that printer" ,not "this printer"
    - Event Timestamps
  - Event Notification (by network)



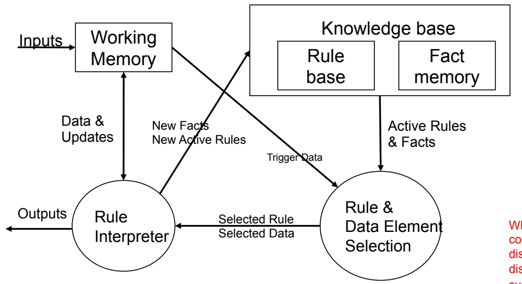
358

Interpreter Architecture Styles

- Interpreters
  - Interpreters create a virtual machine in software. There are generally four components to an interpreter,
    - the program being interpreted,
    - the state of the program,
    - the state of the interpreter,
    - and the interpreter itself.
  - This style is used to narrow the gap between the computing engine in hardware and the semantics of a program.
  - Programming languages that provide a virtual *language* machine include:
    - Pascal, Java, BASIC.

359

Rule Based Style



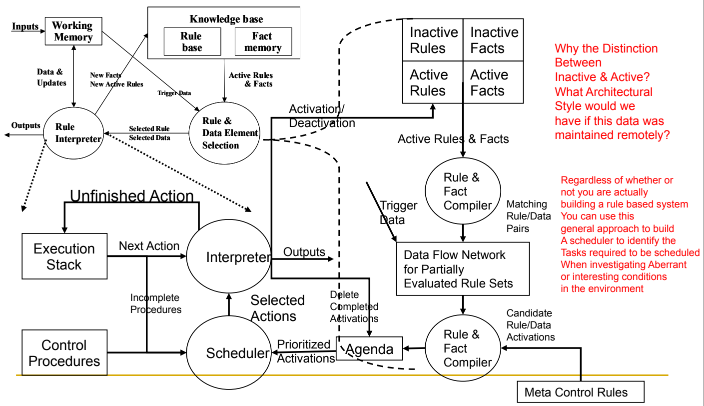
Select and and Execute Rules based on the Current State and Derived Facts

What are the different considerations for a local vs distributed homogeneous vs distributed heterogeneous system ?

The Example in the book shows how the Architecture Elements For a Rule-based Systems are realized using a number of Architecture Types.

360

Rule-Based Style



Data Repository Architecture Styles

- Repositories
  - General Constructs:
    - Repository style systems have two distinct components:
      - A central data structure which represents the current state, and
      - A collection of independent components which operate on the data-store.
    - Two methods of control exist for these systems.
      - If input transactions select the processes to execute then a traditional database can be used as a repository. (Client-Server)
      - If the state of the data-store is the main trigger for selecting processes then the repository can be a blackboard.

Client-Server: Continued

- Client/server architecture.
  - C/S architecture emerged due to file sharing architectures limitations
    - This approach introduced a database server to replace the file server.
    - Using a relational database management system (DBMS), user queries could be answered directly.
    - The C/S architecture reduced network traffic by providing a query response rather than total file transfer.
    - C/S improves multi-user updating through a GUI front end to a shared database.
    - C/S architectures, use (RPCs) or standard query language (SQL) statements to communicate between the client and server.

Repository: Client-Server Cont.

- The term client/server was first used in the 1980s in reference to personal computers (PCs) on a network.
- The client/server model started gaining acceptance in the late 1980s.
- The client/server software architecture is a versatile, message-based and modular infrastructure that is intended to improve usability, flexibility, interoperability, and scalability as compared to centralized, mainframe, time sharing computing
- A client is defined as a requester of services and a server is defined as the provider of services.
- A single machine can be both a client and a server depending on the software configuration.

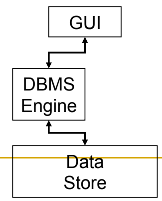
Client Server Architecture Types

- Two Tiered
  - Three components distributed in two tiers:
    - User System Interface
    - Processing Management process development, process enactment, process monitoring, and process resource services)
    - Database Management (such as data and file services)
- Three Tiered
  - Three tier with transaction processing monitor technology
  - Three tier with message server.
  - Three tier with an application server.
  - Three tier with an ORB architecture.
  - Distributed/collaborative enterprise architecture.

365

Two Tiered Client-Server Architectures

- General
  - The user system interface is usually located in the user's desktop environment in two tier client/server architectures.
  - The database management services are usually in a server that is a more powerful machine that services many clients.
  - Processing management is split between the user system interface environment and the database management server environment.
  - The database management server provides stored procedures and triggers.
  - Software vendors provide tools to simplify development of applications for the two tier client/server architecture.



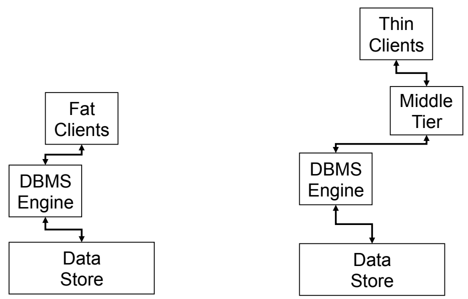
366

Two Tiered Client-Server Architectures

- Advantages:
  - Good solution for distributed computing when work groups are defined as a dozen to 100 people interacting on a LAN simultaneously.
- Disadvantages:
  - Server performance deteriorates as number of clients increases as a result of maintaining a connection with each client
    - (even when no work is being done)
  - Vendor proprietary database implementations restricts flexibility and choice of DBMS for applications.
  - Current implementations provide limited flexibility in repartitioning program functionality from one server to another without manually regenerating procedural code.

367

Two Tiered VS Three Tiered C/S



368

Three Tiered Client-Server Architectures

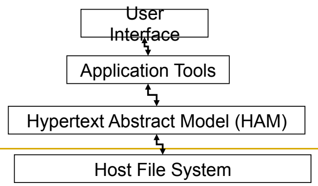
- Proposed to overcome two tier architecture limitations
- A middle tier added between the UI client environment and the DBMS.
  - There are a variety of ways of implementing this middle tier, such as transaction processing monitors, message servers, or application servers.
  - Middle tier performs queuing, application execution, and DB staging.
    - For example, if the middle tier provides queuing, the client can deliver its request to the middle layer and disengage because the middle tier will access the data and return the answer to the client.
  - Middle layer adds scheduling and prioritization for work in progress.
  - The three tier client/server architecture improves performance for groups with a large number of users (in the thousands) and improves flexibility when compared to the two tier approach.
  - Flexibility in partitioning can be a simple as "dragging and dropping" application code modules onto different computers in some three tier architectures.
- Difficult Development environments

Hypertext Style

- Three hypertext architectures variants:
  - Linear, hierarchical, and relational/hierarchical
- Reference Models
  - The HAM or Hypertext Abstract Machine, as described by Campbell and Goodman.
  - The Trellis model, a reference model by Stotts and Furuta.
  - The Dexter model, a reference model by Halasz and Schwartz, written in the specification language Z.
  - The Formal Model by B. Lange, a reference model written in the specification language VDM.
  - The Tower Model, a more general object-oriented model by De Bra, Houben and Kornatzky.

HAM Reference Model : Hypertext Style

- HAM is a transaction-based server for a hypertext storage system.
- The server is designed to handle multiple users in a networked environment.
- The storage system consists of a collection of contexts, nodes, links, and attributes that make up a hypertext graph."
- HAM sits in between the file system and the user interface. Campbell and Goodman envisioned the graphical representation given below:
- HAM is a lower level machine, tied closely to the storage (file) system, while having a looser connection to the applications and user interfaces.
- HAM is only part of this architecture and not the whole system.



Trellis Reference Model : Hypertext Style

- Richard Furuta and P. David Stotts developed a hypertext system, based on Petri Nets, called the Trellis System.
  - From the Trellis model they deduced a meta-model, which they called the Trellis hypertext reference model, abbreviated as r-model.
  - The r-model is separated into five logical levels, as shown in the figure below. Within each level one finds one or more representations of part or all of the hypertext.
  - In contrast to the HAM (and the other reference models) the levels represent levels of abstraction, not components of the system.
  - The levels may be grouped into three categories: abstract, concrete and visible.

Abstract Component Level
Abstract Hypertext Level
Concrete Context Level
Concrete Hypertext Level
Visible Hypertext Level

Dexter Reference Model : Hypertext Style

- The goal of the model is to provide a principled basis for comparing systems as well as for developing interchange and interoperability standards.
- The focus of the Dexter model is on the storage layer, which models the basic node/link network structure that is the essence of hypertext.
- The storage layer describes a "database" that is composed of a hierarchy of data-containing "components" (normally called "nodes") which are interconnected by relational "links".
- The storage layer focuses on the mechanisms by which the components and links are "glued together" to form hypertext networks.
- The components are treated in this layer as generic containers of data.
- The model is divided in three layers, with glue in between, as shown in the figure below:

Runtime layer
Storage layer
Within Component layer

A Formal Model of Hypertext

- The main motivation for the definition of this formal model is the lack of means to interchange and communicate between existing hypertext systems.
- Hypertext research is driven mostly by user interface and implementation considerations.
- Very few attempts have been made to provide a formal basis for the research field.
- David Lange chose the Vienna Development Method (VDM) [BJ82,Jones-86] because it supports the top-down development of software systems specified in a notion suitable for formal verification.
- Like the Dexter model Lange's model emphasizes the data structure of hyper-documents. Therefore Lange calls it a data-model of hypertext.
- This data-model defines nodes, links, network structures, etc.
- The model goes further than the Dexter model in looking inside the nodes of a hyper-document to find slots,buttons and fields.
- The basic data-model is then extended with features to become an object-oriented model for hypertext.

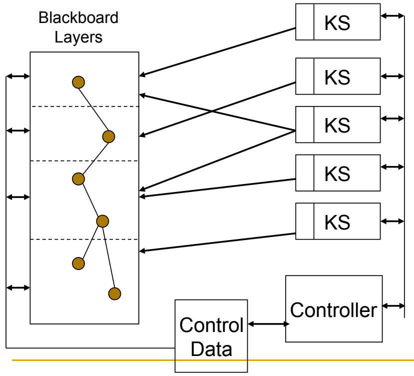
Tower Reference Model: Hypertext Style

- Background:
  - Trellis model describes the "abstract component level" in a way that makes it sound like there would be no need for containers containing containers (or in more common terminology: composites containing composites).
  - The Dexter model allows undirected (and bidirectional) links, but only between two nodes (called components).
    - Links between more than two nodes are allowed but must be directed (they must have at least one "destination" or "TO" endpoint).
    - Another restriction in the Dexter model is that, while the model allows composites within composites, the hierarchy of composites must be acyclic, thus forbidding so called "Escher effects".
- The Tower model contains basic structural elements, nodes, links and anchors, tower objects and city objects.
  - The tower objects are used to model different descriptions of an object, somewhat like the layers in the Dexter model.
  - Type, storage structure, presentation, etc. are all levels of the tower object.
  - Cities represent sets of views onto (tower) objects.
  - The model allows every kind of object to be a virtual object (i.e. the result of a function or algorithm).
    - Operators for defining virtual structures are Apply-to-All, Filter, Enumeration and Abstraction (or grouping).

Blackboard Style

- http://www.cs.virginia.edu/~acc2a/techie/notes/blkbrds.htm
- BB Metaphor:
  - A group of specialists work cooperatively to solve a problem, using a blackboard as the workplace for developing the solution.
  - The problem and initial data are written on the blackboard.
  - The specialists watch the blackboard, and when a specialist finds sufficient information on the board to make a contribution, he records his contribution on the blackboard.

BB Architecture Overview



KS consist of a Condition (Trigger) Section and the Body

Essentially what happens is:

An event has occurred that has resulted in the BB state changing.

If am registered to Accept events on that level of the BB and if the event satisfies my curiosity and any constraints (Trigger conditions), then I will apply the KS body to evaluate the Event and perform the requested operation

Repository Architecture Styles: Blackboard

- A *blackboard* model usually has three components:
  - General Constructs
    - The knowledge source: independent pieces of application specific knowledge. Interaction between knowledge sources takes place only through the blackboard.
    - The blackboard data structure: state data, organized into an application-dependent hierarchy. Knowledge sources make changes to the blackboard that lead incrementally to a solution to the problem.
    - Control: driven by the state of the blackboard. Knowledge sources respond opportunistically when changes in the blackboard make them applicable.
  - General Operation
    - Invocation of a knowledge source is dependent upon the state of the blackboard.
    - Control can be implemented in the knowledge source, the blackboard, externally, or a combination of these.
    - Blackboard systems have traditionally been used for applications requiring requiring complex interpretations of signal processing.
    - Programming environments can be considered as having a shared repository of programs and program fragments.

Repository Architecture Styles: Blackboard

- Independence of Expertise
  - Each *knowledge source* is a specialist at solving certain aspects of the problem.
  - No KS requires other KSs in making its contribution.
  - Once it finds the information it needs on the blackboard, it can proceed without any assistance from other KSs.
  - KSs can be added, removed, and changed without affecting other KSs.
- Diversity in Problem Solving Techniques
  - Internal KS representation and inference machinery is hidden from view.
- Flexible Representation of Blackboard Information
  - The blackboard model does not place any prior restrictions on what information can be placed on the blackboard.
  - One blackboard application might require consistency, another might allow incompatible alternatives.
- Common Interaction Language
  - There must be a common understanding of the representation of the information on the blackboard, understood by all KSs.
  - There's a tradeoff between representational expressiveness of a specialized representation shared by only a few KSs and a representation understood by all.

Repository Architecture Styles: Blackboard

- Positioning Metrics
  - When the blackboard gets full, we must still have a way for the KSs to immediately see the information important to them.
  - Often we have multiple or subdivided blackboards, or information is sorted alphabetically or by reference.
  - Efficient retrieval is also important.
- Event Based Activation
  - KSs are triggered in response to events (they don't actively watch the blackboard).
  - The board knows what kind of event each KS is looking for, and considers it for activation whenever that kind of event occurs.
- Need for Control
  - A *control component* separate from the individual KSs is responsible for managing the course of problem solving.
  - The control component doesn't share the specialties of the KS's, but looks at each KSs evaluation of its own contribution to decide which one gets to go.

Repository Architecture Styles: Blackboard

- The Blackboard Model of Problem Solving
  - Incremental Solution Generation
    - Blackboard systems are effective when there are many steps towards the solution and many potential paths involving these steps.
    - It works opportunistically, exploring the paths most effective in solving the particular problem and can outperform a solver that uses a predetermined approach
  - Knowledge Sources
    - Each KS is separate and independent of all other KSs.
    - Each KS does not need to know of their expertise or even existence.
    - KSs must understand the state of the problem-solving process and the representation of relevant information on the blackboard.
    - Each KS knows its *triggering conditions* -- the conditions under which it can contribute.
    - KSs are not active, but *KS activations* -- combinations of KS knowledge and a specific triggering condition -- are the active entities competing for executing instances. KSs are static repositories of knowledge.
    - Ks activations are the active processes.

381

Repository Architecture Styles: Blackboard

- The Blackboard
  - The *blackboard* is a global structure available to all KSs.
  - It is a community memory of raw input data, partial solutions, alternatives, final solutions, and control information. It is a communication medium and buffer.
  - It is a KS trigger mechanism.
- Control Component
  - An explicit control mechanism directs the problem solving process by allowing KSs to respond opportunistically to changes on the blackboard.
  - On the basis of the state of the blackboard and the set of triggered KSs, the control mechanism chooses a course of action.
  - At each step to the solution, the system can execute any triggered KS, or choose a different focus of attention, on the basis of the state of the solution.

382

Uses of the Blackboard Style

- It has been used for
  - sensory interpretation,
  - design and layout,
  - process control,
  - planning and scheduling,
  - computer vision,
  - case based reasoning,
  - knowledge based simulation,
  - knowledge based instruction,
  - command and control,
  - symbolic learning, and
  - data fusion..

Why Use the Blackboard Problem Solving Approach

–When many diverse, specialized knowledge representations are needed.

–When an integration framework for heterogeneous problem solving representations and expertise is needed

–When the development of an application involves numerous developers.

–When uncertain knowledge or limited data inhibits absolute determination of a solution, the incremental approach of the blackboard system will still allow progress to be made.

–When multilevel reasoning or flexible, dynamic control of problem-solving activities is required in an application.

383

Repository Architecture Styles: Blackboard

- Advantages:
  - Provides an explicit forum for the discussion of data access, distribution, synchronization
  - Provides an explicit forum for the discussion of Task Allocation Policies
  - Provides an explicit for the discussion of control and task sequencing and prioritization
  - Provides an explicit forum for the discussion of Load Redistribution.
- Disadvantages:
  - Blackboard systems to not seem to scale down to simple problems, but are only worth using for complex applications

384

Repository Architecture Styles: Blackboard

- Conclusion:
  - I have found architecture style to be able to subsume the styles others to a great extent.
    - Even for Real-Time Video processing !!
  - I usually start with this view and then relax the architecture to accommodate the functional and performance requirements with attributes of the other styles

A View of Distributed Architecture Styles

Distributed Processing is classified into nine styles from the viewpoint of the location of data and the processing type between client and server.

Data is classified as Centralized or Distributed

Processing as either synchronous or asynchronous

Transaction Type

Atomic, Consistency, Isolation, Durability

Query Type

A reply from the server is synchronized with a request from the client

For Asynchronous processing:

A Notification type indicates that the server process is not synchronized with a client request

A View of Distributed Architecture Styles Cont:

- Transaction Types
  - Centralized: Single DB, Single Server
  - Distributed: Multiple DBs on Multiple Servers with Synchronous processing between Servers.
  - Asynchronous: Multiple DB on Multiple Servers with Asynchronous processing between Servers.
- Query Types
  - Centralized: Query and Reply Processing
  - Distributed: Simultaneous access to to multiple data bases and support query intensive immediate processing
  - Asynchronous: Suited to asynchronous sharing of data (partial DB downloads)
- Notification Types
  - Centralized: Automation of simple workflow, shipping memos, etc.
  - Distributed: Distributed transaction and data processing from mobile clients
  - Asynchronous: Supports loose integration of independent multiple applications or systems.

Process Interaction in Distributed Programs Cont.

- Asynchronous Message Passing
  - Channel has unlimited capacity
  - Send & receive do not block
  - Different communication channels are used for different kinds of messages.
- Synchronous Message Passing
  - Channel has fixed capacity
  - Sending process waits for receiving process ready to receive, hence synchronized
- Buffered Message Passing
  - Channel has fixed capacity
  - Send is delayed only when the channel is full
- Generative Communication
  - Send & Receive processes share a single communication channel called tuple space.
  - Associative naming distinguishes message types in the tuple space
- Remote Procedure Call & Rendezvous
  - Calling process delays until the request is serviced and results returned.

PIPD: Requests & Replies between clients & Servers

- Server vs. monitors
  - A server is active, whereas a monitor is passive
  - Clients communicate with a server by sending and receiving messages,
    whereas clients call monitor procedures.
- A monitor is a synchronization mechanism that encapsulates permanent variables that record the state of some resource and exports a set of procedures that are called to access the resource.
  - The procedures execute with mutual exclusion; they use condition variables for internal synchronization.

A View of Distributed Processing Styles Cont.

Architectural Styles for Transaction Types
  Centralized vs. Distributed vs. Asynchronous Transaction Messages
Architectural Styles for Query Types
  Centralized vs. Distributed vs. Asynchronous Query Messages
Architectural Styles for Notification Types
  Centralized vs. Distributed vs. Asynchronous Notification Messages

Location of Data / Processing Types between C/S	Processing Type Between Servers / Msg. Type	Centralized	Distributed: Synchronous	Distributed: Asynchronous
Synchronous Processing	Transaction Type (ACID)	Centralized Transactions	Distributed Transactions	Asynchronous Transactions
	Query Type	Centralized Query	Distributed Query	Asynchronous Query
Asynchronous Processing	Notification Type	Centralized Notification	Distributed Notification	Asynchronous Notification

Process Interaction in Distributed Programs (PIDP)

- Cooperating Message Passing Processes:
  - One way Data Flow Through a Network of Filters
  - Request & Replies between clients & servers
  - Heartbeat Interaction between neighboring processes
  - Probes & Echoes in Graphs
  - Broadcasts between processes in complete graphs
  - Token passing along edges in a graph
  - Coordination between centralized server processes
  - Replicated workers sharing a bag of tasks

Distributed Processes Architecture Styles

- Other familiar architectures
  - Distributed processes –
    - have developed a number of common organizations for multi-process systems.
    - Some are defined by their topology (e.g. ring, star)
    - Others are characterized in terms of the kind of inter-process protocols that are used (e.g. heartbeat algorithms).
    - A common form of distributed system architecture is *client-server*.
      - A server provides services to the clients.
      - The server does not usually know the number or identity of the clients which will access it.
      - The clients know the identity of the server (or can find it out through another name-server) and access it through a remote procedure call.
  - Main program/subroutine organizations: The primary organization of many systems mirrors the programming language in which the system is written.
  - Domain Specific Software Architectures (DSSA)
  - State-transition systems: A common organization for many reactive systems. Define in terms of a set of states and a set of named transitions

Process Control Architecture Styles

- Process Control
  - Process Control Paradigms
    - Usually associated with real-time control of physical processes. The system maintains specified properties of the output process near a reference value
      - Open Loop Systems: If the process is completely defined, repeatable, and the process runs without surveillance
        - Space Heater
      - Closed Loop Systems: Output is used to control the inputs to maintain a monitored value
        - Speed Control, etc. Feed back and Feed Forward controller.
  - General Constructs:
    - Computational Elements
    - Data Elements
    - Control Loop Paradigm
  - Concerns
    - We need to worry about the physical control laws (s-domain) versus the time sampled control laws (Z-Domain) and the introduction of poles and zeroes into the transfer function.

393

Heterogeneous Architecture Styles

- Heterogeneous Architectures
  - Most systems involve the combination of several styles.
  - Components of a hierarchical system may have an internal structure developed using a different method.
  - Connectors may also be decomposed into other systems (e.g. pipes can be implemented internally as FIFO queues).
  - A single component may also use a mixture of architectural connectors.
    - An example of this is Unix pipes-and-filter system in which the file system acts as the repository, receives control through initialization switches, and interacts with other components through pipes.

394

Topic 7: Architectural Patterns

Pattern-Oriented Software Architecture

- Frank Buschmann, Regine Muenier, Hans Rohnert, Peter Sommerlad, Michael Stal.1996.*Patterns of Software Architecture*
- Presented three categories of patterns
  - Architectural Patterns
  - Design Patterns
  - Idoms
- Have been confused with Architectural Styles
  - To see difference we need to look at origins of Software Patterns

396

Origins of Patterns

- There are a number of primary sources for the emergence of Software Patterns
  - PhD work on frameworks by Eric Gamma
  - Contract specification by Richard Helm
  - Smalltalk frameworks by Brian Foote and Ralph Johnson
  - Software Architecture handbook by Bruce Anderson
- But the patterns form originates in the built environment with the work of Christopher Alexander

A Brief History of Software Patterns

- 1989: Alexander's ideas introduced by Kent Beck, Ward Cunningham
- 1991-4: OOPSLA workshops on Software Architecture
  - Gamma, Helm, Johnson and Vlissides meet
- 1993: Hillside group formed
- 1995: *Design Patterns* book published
- 1996: Alexander's keynote at OOPSLA

Christopher Alexander

- Born in Vienna, educated in Britain and the US
- A leader of "post-modern" architecture
  - Driven by observation that most of what humanity has built since WWII has been dehumanising rubbish
  - Believes architecture impacts directly on our behaviour and well being
    - "Tall buildings make people mad"
  - Professional architects have failed humanity and the environment
- Views are controversial even amongst architects

Christopher Alexander (2)

- **Work is represented in an 11-volume series of books (8 currently in print)**
  - **The Timeless Way of Building**
  - **A Pattern Language**
  - **The Oregon Experiment**
  - **The Linz Café**
  - **The Production of Houses**
  - **A New Theory of Urban Design**
  - **A Foreshadowing of 21st Century Art**
  - **The Mary Rose Museum**
  - **The Nature of Order**
  - **Sketches of a New Architecture**
  - **Battle: The story of a Historic Clash Between World System A and World system B**

Christopher Alexander (3)

- Also presented a critique of modern design in *Notes on the Synthesis of Form (1964)*
  - Identified cognitive complexity and the alienation of builders from users as the root of failure of modern design
  - Later proposed "pattern languages" as a way of recovering lost ability to design useful things

401

Homeostatic Structure

- Alexander's criticism of modern design is rooted in the belief that we have lost the ability to create 'homeostatic (self-adjusting) structures'
- In homeostatic structure the failures of form are 'one-offs'
- The culture/tradition that supports them changes more slowly
  - Strongly resists all changes other than those provoked by failure
  - Is embodied in a culturally acquired "pattern language"
- Each failure is easily accommodated, equilibrium between form and context is dynamically re-established after each 'failure'

402

Biological Forms

- An individual tree is an homeostatic structure
- It responds, genetically, to the local availability of sunlight and rain
  - Through numbers of leaves, branches
- To competition
  - Through achieving maximum height
- To wind
  - By bending its shape
- Each individual tree's shape fits its individual context

403

Reasons for Modern Design Crisis

- Design tradition has been decisively weakened
  - In face of new requirements, new materials
- Feedback loop no longer immediate
  - Professionalisation of architecture separates designers from users
- The rising control of the designer
  - Has an unachievable responsibility
  - Cognitive burden too large

404

A Pattern Language

- Alexander’s book: “A Pattern Language” presents 253 patterns for the built environment
  - Written in a standard, narrative form supported by hand-drawn sketches
  - Includes patterns to build alcoves, rooms, houses, towns, cities and even global society
- Together the patterns form a network
  - A “pattern language”

Example of an Alexandrian pattern

- “Waist High Shelf”
  - Proposes that every domestic home needs a “waist-high shelf”
  - A convenient place to deposit office keys, car keys, mobile phone etc.
    - Everything you don’t need at home, but do need for work
    - Can be implemented in a number of ways
      - Shelf; kitchen worktop; particular stair on stairway
  - Is an abstract *solution* to a general, recurring *problem* in a particular *context*

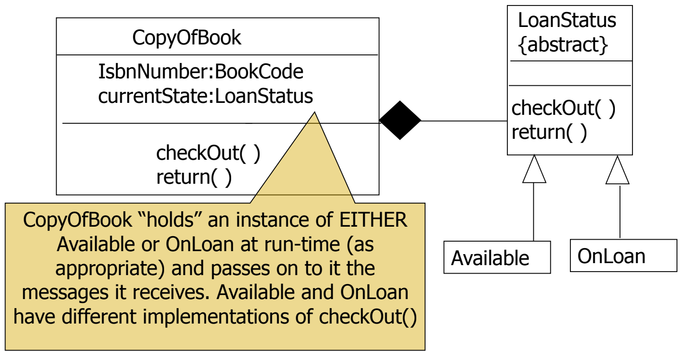
Design Patterns

- *Design Patterns* are elements of reusable software
- They provide the abstract core of solutions to problems that are seen over and over again

Example of a Design Pattern (Simplified)

- Example Design Pattern: State
- Use when
  - Behaviour depends on current state or mode
  - When otherwise a large switch statement or long if statement would need to be used
    - These are difficult to maintain
- Solution
  - Abstract state-specific behaviour into a shallow inheritance hierarchy; instantiate the appropriate state object as needed at run-time

State Pattern Applied



409

The "Gamma Patterns"

- The patterns in the *Design Patterns* book are sometimes called "Gamma patterns"
  - After the lead author, Erich Gamma
    - Also called GoF or Gang-of-Four patterns
- They are a catalogue of 23 patterns
  - NOT a pattern *language*
  - Each pattern is written in a standard template form
  - Classified into Structural, Behavioural and Creational patterns
  - Links shown via a Pattern Map

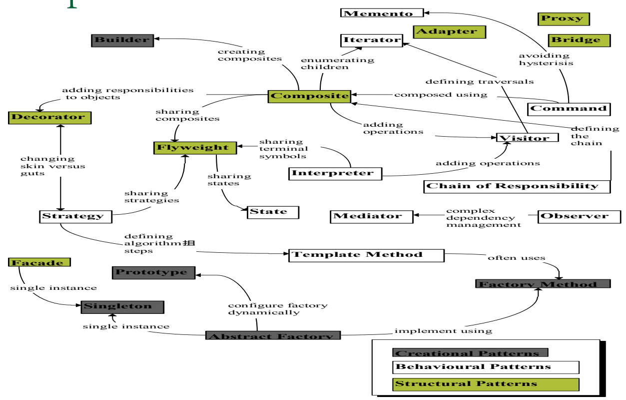
410

The Gamma Pattern Template

- Intent
- A.K.A.
- Motivation
- Applicability
- Structure
- Participants
- Collaborations
- Consequences
- Implementation
- Sample Code
- Known Uses
- Related Patterns

411

Map of the Gamma Patterns



412

Shared Values of People Documenting Software Patterns

- Success is more important than novelty
  - good patterns are discovered not invented
- Emphasis on writing down and communicating "best practice" in a clear way
  - most patterns use a standard format - a ***patterns template*** which combines literary and technical qualities
- "Qualitative validation of knowledge"
  - effort is to describe concrete solutions to real problems, not to theorise

Shared Values of People Documenting Patterns (contin.)

- Good patterns arise from practical experience
  - Every experienced developer has patterns that we would like them to share. We do this in ***Pattern Writers' workshops***
- Respect and recognition for the human dimension of software development
  - Full recognition that design is a creative, human activity
  - Full respect for previous gains and conquests

Characteristics of Software Design Patterns (e.g. Gamma et al)*

- Problem, not solution-centred
- Focus on "non-functional" aspects
- Discovered, not invented
- Complement, do not replace existing techniques
- Proven record in capturing, communicating "best practice" design expertise

*Gamma E., Helm R., Johnson R., Vlissides J. 1994.***Design Patterns-Elements of Reusable Object-Oriented Software.*** Addison-Wesley

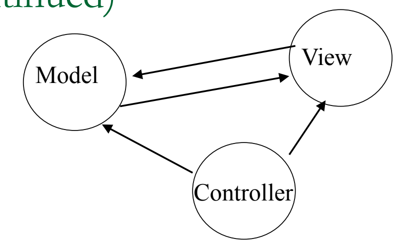
Architectural Patterns

- "An architectural pattern expresses a fundamental organising structural organisation schema for software systems. It provides a set of predefined subsystems, specifies their responsibilities, and includes rules and guidelines for organising the relationships between them"

Architectural Patterns

- Buschmann et al., present a catalogue that includes 8 architectural patterns in 4 categories
  - "From Mud to Structure"
    - Layers, Pipes and Filters, Blackboard
  - Distributed Systems
    - Broker
  - Interactive Systems
    - Model View Controller, Presentation-Abstraction-Controller
  - Adaptable Systems
    - Microkernel, Reflection

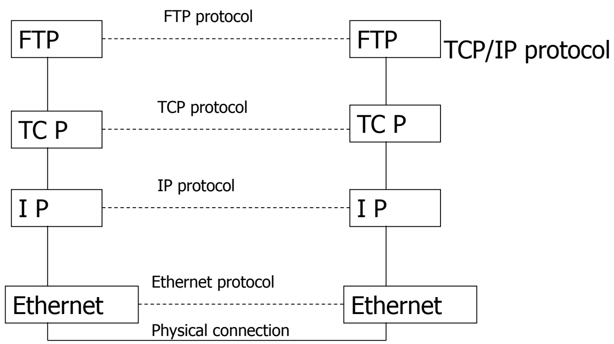
The Model-View-Controller Pattern (continued)



•M-V-C originated with Smalltalk-80
- Informs the entire architecture of modern Smalltalk environments

•Microsoft's Document-View architecture is an instance of M-V-C
•Model = Document, View = View
-So where is the Controller? (answer: it is MS Windows!)

Layers Pattern: Example



Layers Pattern

- Context
  - large system needing decomposition
- Problem
  - How to structure systems that contain a mix of high and low-level functionality
- Solution
  - Conceptually layer the system, from level 0 upwards

Layers Pattern: Structure

Class LayerJ	
Responsibility Provides services to Layer J+1 Delegates subtasks to Layer J-1	Collaborator Layer J-1

Layers Pattern: Consequences

- Benefits
  - Reuse of Layers
  - Support for standardisation
  - Localisation of dependencies
  - Exchangeability
- Liabilities
  - Cascades of Changing Behaviour
  - Lower Efficiency
  - Unnecessary work
  - Difficulty of getting 'granularity' right

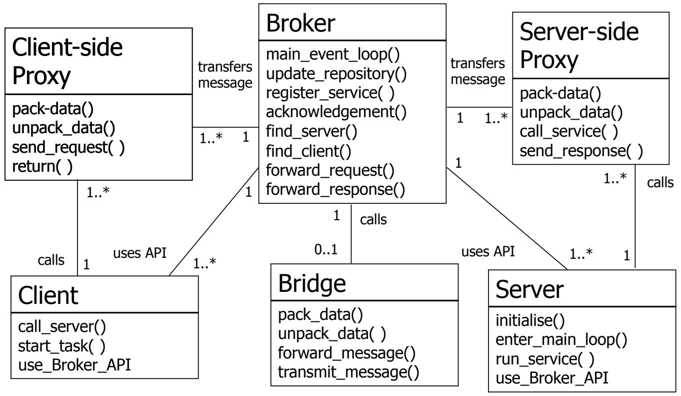
Layers Pattern: Variants

- Relaxed Layer System
  - A.k.a. 'open' layered system
  - Layer can talk to any layer below it
    - In 'closed' layer systems can talk only to the layer immediately below
  - Gains in performance, flexibility
    - Loses maintainability
- Layering through Inheritance
  - Abstract classes at Layer 0 etc.,

Broker Pattern

- Context
  - Distributed, possibly heterogeneous system of independent co-operating "components"
- Problem
  - How to partition functionality to deliver a set of decoupled, interoperating components
- Solution
  - Introduce a Broker component to decouple clients and servers

Broker Pattern:Structure



Broker Pattern: Variants

- Direct Communication Broker System
  - Clients communicate directly with servers, broker identifies the communication channel
- Message Passing Broker System
  - Servers use type of message to determine action
- Trader System
  - Client-side servers provide *service* ids rather than *server* ids
- Adapter Broker System
- Callback Broker System
  - Reactive, event-driven model; makes no distinction between clients and servers

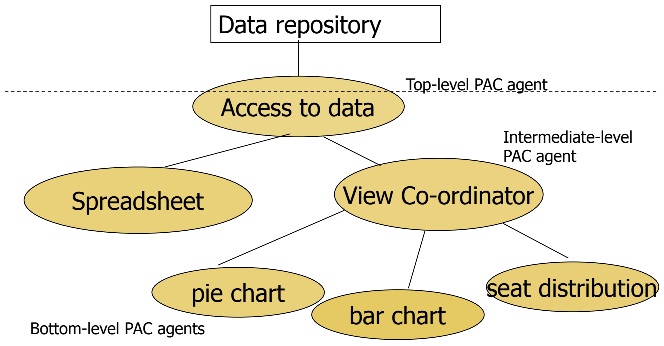
Broker Pattern: Consequences

- Benefits
  - Location transparency
  - Changeability/Extensibility of components
  - Portability
  - Interoperability between Broker Systems
  - Reusability
  - Testing and Debugging
- Liabilities
  - Restricted efficiency
  - Lower fault tolerance
  - Testing and Debugging

Presentation-Abstraction-Control Pattern

- Context
  - Interactive systems with the help of agents
- Problem
  - Partitioning of interactive systems horizontally and vertically
- Solution
  - Structure the solution as a tree-like hierarchy of PAC agents

Presentation-Abstraction-Control Pattern: Structure



Presentation-Abstraction-Control Pattern: Variants

- PAC agents as active objects
- PAC agents as processes

Presentation-Abstraction-Control Pattern: Consequences

- Benefits
  - Separation of Concerns
  - Support for Change/Extension
  - Support for multi-tasking
- Liabilities
  - Increased system complexity
  - Complex control components
  - Efficiency
  - Restricted applicability

Summary

- Patterns
  - Open, informal, abstract solutions to a general, recurring problem in a particular context
  - Capture "best practice" experience of design
  - Can be organised as Catalogues or as Pattern Languages
  - Differ from ABAS which are closed, formal composable abstractions
- Architectural Patterns are a sub-category of patterns
  - To do with "gross structure"
  - But arguably ALL patterns are architectural

Topic 8: Domain-Specific Software Architecture

Domain-Specific Software Architecture (DSSA)

- “The relationships between functions in programs for a software domain
- This is also known as a reference model, functional partitioning, meta-model, logical model, . . .
- Why do we want this?
  - 1. To build better: tools, specification languages, domain-specific reusable components, application frameworks, product families.
  - 2. To understand better. Software problems are very complex. A DSSA is ready-made, reusable domain analysis, problem decomposition.

434

Example DSSA: Architecture (the real kind)

- The problem: “obtain an artificial environment for some human activity”.
- Specific instances of this problem: a house, an apartment, a store, a warehouse, a jail, a paint factory.
- User: wants the building
- Constructor: builds the building
- Requirements: two-car garage, roof will last 30 years, doorways have 5 meter clearance, ...
- Specifications: plans/blueprints

435

Architecture (the real kind) - 2

- These instances of buildings have much in common and can reuse analyses:

  • Functional requirements: how many people should this building hold? how should it be heated/cooled? how should the parts be connected?

  • Non-functional requirements: how easy is it to add an addition? how secure should it be? how many years should it last?

436

Architecture (the real kind) - 3

- These instances can also reuse designs/ components:
  - • house plans
  - • trusses
  - • door-knobs
- How does the reuse of designs/components affect the quality of the resulting building?

The DSSA Solution

- Three kinds of knowledge:
  - • how to decompose the composite problem into component problems
  - • how to solve each of the component problems
  - • how to compose the individual solutions of the component problems into a solution of the composite problem
- This knowledge is not trivial. The ease of the solution to an instance of a class depends on the existence of a DSSA for that class.

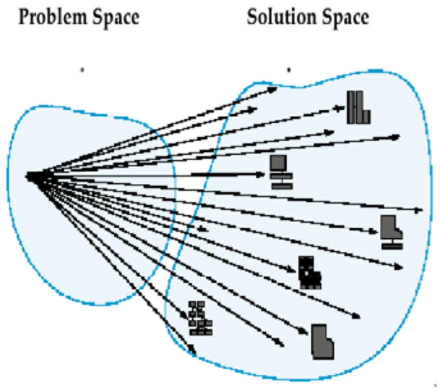
Solving Problems with DSSAs

- Decompose the problem into the language of the DSSA: the atomic actions (install wiring, generate an attribute evaluation module, create a cache manager, ...).
- Different atoms will be required for different end-products.
- The DSSA does not specify an inventory that all products must use.
- Some atoms may exist as off-the-shelf components.
- Some may need to be tailor-made.
- A DSSA focuses requirements/design decisions; highlights changes from canonical solutions.

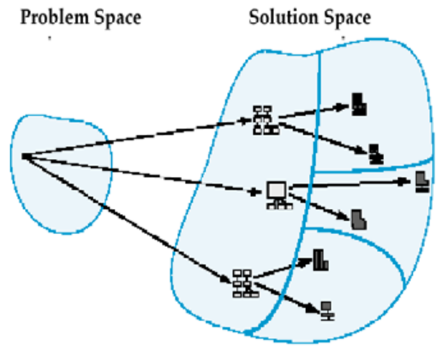
WHY DOMAIN-SPECIFIC?

- Development in specific domains can be optimised
  - maximise method and reuse
  - minimise intuition
- Reuse in specific domains is most realistic
  - reuse in general has proved too difficult to achieve up till now
  - focus on particular classes of applications with similar characteristics
- Criteria for successful reuse [Biggerstaff]
  - well-understood domain
  - slowly changing
  - has intercomponent standards
  - provides economies of scale
  - fits existing infrastructure

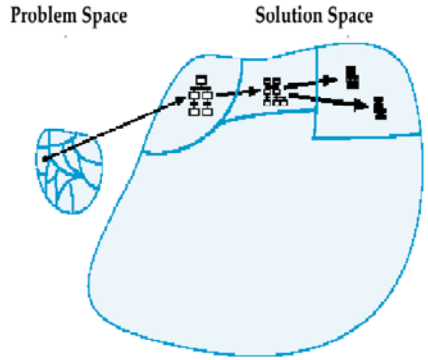
SOFTWARE DEVELOPMENT



ARCHITECTURE-BASED SOFTWARE DEVELOPMENT



DSSA-BASED SOFTWARE DEVELOPMENT



WHAT IS DSSA?

- DSSA is an assemblage of software components
  - specialised for a particular type of task (domain)
  - generalised for effective use across that domain
  - composed in a standardised structure (topology) effective for building successful applications
    - *Rick Hayes-Roth, 1994*
- DSSA is comprised of
  - a domain model
  - reference requirements
  - a reference architecture (expressed in an ADL)
  - its supporting infrastructure/environment, and
  - a process/methodology to instantiate/refine and evaluate it
    - *Will Tracz, 1995*

WHAT IS A DOMAIN MODEL?

- "All models are wrong; some are useful."
  - *unknown, SEI*
- A domain model is a representation of knowledge about a domain
  - functions being performed
  - data, information, and entities flowing among the functions
- It deals with the problem space
- Fundamental objective:
  - standardise problem domain terminology and semantics
  - terminology + semantics = ontology
  - provide basis for standardised descriptions of problems to be solved in the domain

445

DOMAIN ANALYSIS

- Domain model is a product of domain analysis
- Domain analysis entails
  - identifying, capturing and organising objects and operations
  - their description using a standardised vocabulary
  - in a class of similar systems
  - in a particular problem domain
  - to make them reusable when creating new systems
- "Domain analysis is like several blind men describing an elephant"

446

ELEMENTS OF A DOMAIN MODEL (1)

- Customer needs statement
  - identifies functional requirements
  - high level
  - informal
  - ambiguous
  - incomplete
- Scenarios
  - functional requirements
  - data flow
  - control flow
  - elicited from the customer
- Domain dictionary
  - definitions of terms used in the domain model
  - updated over time

447

ELEMENTS OF A DOMAIN MODEL (2)

- Context (block) diagram
  - high-level data flow between the major components in the system
- Entity/relationship diagrams
  - aggregation ("a-part-of") and generalisation ("is-a") relationships
- Data flow models
- State transition models

Object model
  - first phase of component interface design

448

WHAT ARE REFERENCE REQUIREMENTS?

- Requirements that apply to the entire domain
- Reference requirements contain
  - *defining* characteristics of the problem space
    - functional requirements
  - *limiting* characteristics (constraints) in the solution space
    - non-functional requirements (e.g., security, performance)
    - design requirements (e.g., architectural style, UI style)
    - implementation requirements (e.g., platform, language)

449

WHAT IS A REFERENCE ARCHITECTURE?

- A standardised, generic architecture describing all systems in a domain
  - focuses on fundamental abstractions of the domain that expose a hierarchical, compositional nature
  - *a set* of reference architectures may be needed to satisfy all reference requirements
- Based on the constraints in reference requirements
- Syntax and semantics of constituent high level components are specified
- It is reusable, extendable, and configurable
- A reference architecture is *instantiated* to create a design architecture for a specific application system

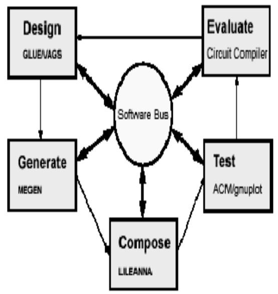
450

ELEMENTS OF A REFERENCE ARCHITECTURE

- Reference architecture model
  - topology based on the architectural style
- Configuration decision tree
  - to instantiate a system from the reference architecture
- Architecture schema or design record
  - information about components and design alternatives
  - domain and implementation specific
- Reference architecture (component) dependency diagram
- Component interface descriptions
- Constraints
  - parameter value ranges and relationships
- Rationale

451

ADAGE DEVELOPMENT ENVIRONMENT ARCHITECTURE



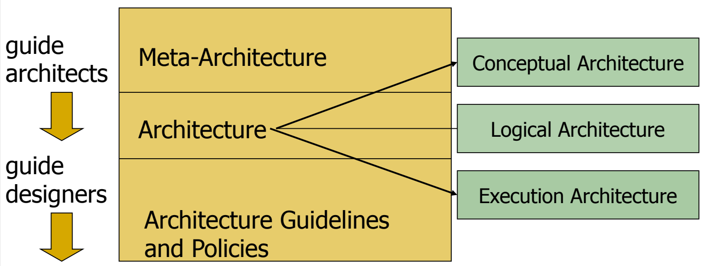
452

Topic 9: Discipline of Software Architecture

Software Architecture

- Dan Bredemeyer outlines Software Architecture in the following way:
  - *Architecture*: Addressing "What is SW Architecture?; architecture views, patterns, styles, component specs., interfaces etc.,
  - *Architecting*: How do I create, recapture or migrate an architecture?: modelling, documenting, assessing.
  - *Architects*: What are the roles, responsibilities and skills of architects?
  - *Architecture Strategy*: Why do architecture? And When?: competitive differentiation
  - *Architectural context*: Where in the organisation is architecture done?

Bredemeyer's Software Architecture Model (1)



Bredemeyer's Software Architecture Model (2)

Meta-Architecture

•Architectural vision, principles, styles, key concepts and mechanisms

•*Focus: high-level decisions that will strongly influence the structure of the system; rules certain structural choices out, and guides selection choices and tradeoffs among others*

Architecture

•Structures and relationships, static and dynamic views, assumptions and rationale

•*Focus: decomposition and allocation of responsibility, interface design, assignment to processes and threads*

Architecture Guidelines and Policies

•Use model and guidelines: policies, mechanisms and design patterns; frameworks, infrastructure and standards

•*Focus: guide engineers in creating designs that maintain the integrity of the architecture*

Bredemeyer's Software Architecture Model (3)

Conceptual Architecture

•Architecture diagram, CRC-R cards*Focus: identification of components and allocation of responsibilities to components*

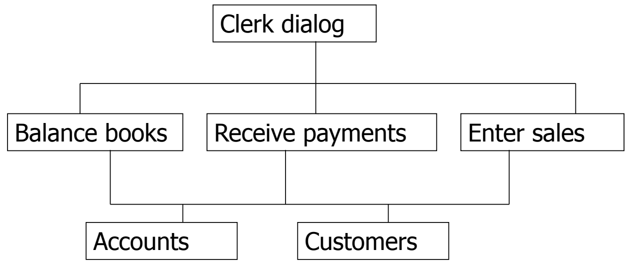
Logical Architecture

•Updated Architecture diagram (showing interfaces), *Focus: design of component interactions, connection mechanisms and protocols; interface design and specification; providing contextual information for component users*

Execution Architecture

•Process view (shown on Collaboration Diagrams)*Focus: assignment of the runtime component instances to processes, threads and address spaces; how they communicate and co-ordinate; how physical resources are allocated to them*

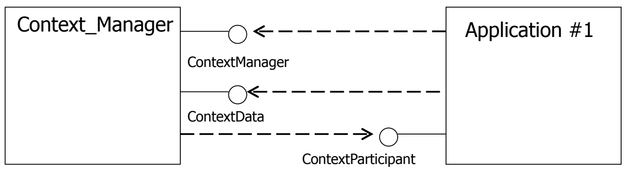
Conceptual Architecture



•Abstract, system-wide view

•Basis for communication

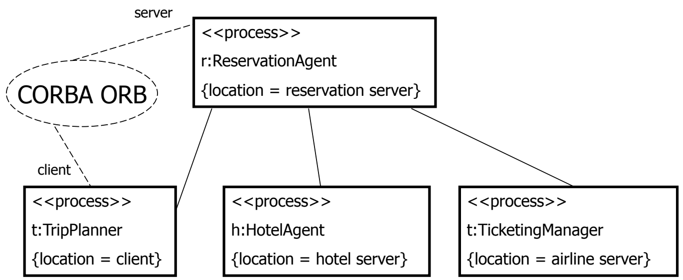
Logical Architecture



- "Blueprint": precise, unambiguous, actionable

•Basis for supplier/client contract

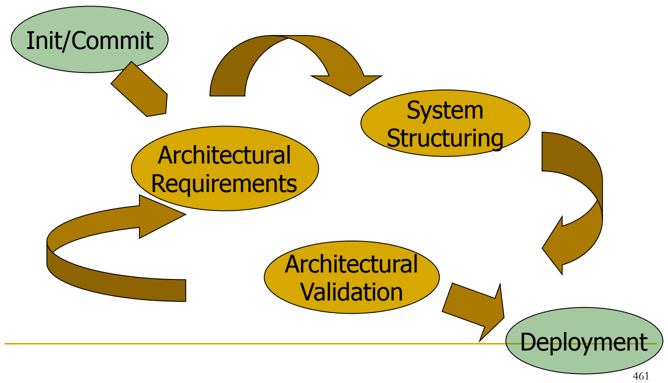
Execution Architecture



•Configuration of components at run-time

•Basis for early system tuning

Process Overview



How to Create a Good Architecture

- Elicit stakeholder goals
  - Ensure architecture supports what is valued and scopes out what is not
    - Bredemeyer creates Context Maps, Stakeholder Profiles for documentation
  - Drives creation of principles, selection of architectural styles, and selection/creation of mechanisms in "meta-architecture phase"
  - Functionality goals are starting points for Use Cases
  - Quality goals are starting points for non-functional requirements specification

Functionality Goals

- Functional requirements capture the intended behaviour of the system
- Use Cases capture *who* (actor) does *what* (interaction) with the system for what *purpose* (goal) without dealing with system internals
- A Use Case defines a goal-oriented set of interactions between external actors and the system under consideration

Quality Goals

- Qualities are properties or characteristics of the system that its stakeholders care about and hence will affect their degree of satisfaction with the system.
- Explicit, documented qualities are needed to:
  - Define architectures so that they achieve required qualities
  - Make tradeoffs
  - Form a basis for comparing alternatives
  - Enhancing communication within architecture team and between architects and stakeholders
  - Evaluate architectures to ensure compliance

Init/Commit Summary

- **Gain Management Sponsorship**
  - **Purpose**: Ensure management support
  - **Activities**: Create/communicate the architectural vision showing how it contributes to long-term success
  - **Checks**: Resources? Full-time architect-team members? Management championship?
- **Build the Architecture team**
  - **Purpose**: ensure a cohesive and productive team
  - **Activities:** Use arch. vision to build team alignment; assess team capabilities and needs; establish team operating model
  - **Checks:** strong accepted leader? Collaborative, creative team? Effective decision-making?

Architectural Requirements Summary

- Capture Context, Goals and Scope
  - Ensure architecture is aligned to business strategy and directions, and anticipates market and technology changes
- Capture Functional Requirements
  - Document, communicate and validate the intended behaviour of the system
- Capture Non-Functional Requirements

System Structuring Summary

- Create the Meta-Architecture
  - Make strategic architectural choices to guide the architecting effort
- Create the Conceptual Architecture
  - Create conceptual models to communicate the architecture to managers, project managers, developers and users
- Create the Logical Architecture
  - Create detailed architectural specs. In a way that is directly actionable

Validation Summary

- Validate the Architecture
  - Assess the architecture to validate that it meets the requirements and identify issues and areas for improvement early
  - Construct prototypes or proof-of-concept demonstrators or skeletal architecture to validate communication and control mechanisms
  - Conduct reviews
  - Conduct architectural assessments (e.g., SAAM Action Guide)

Deployment Summary

- Build Understanding
  - Help all developers understand the architecture, its rationale, and how to use it.
  - Help managers understand its implications for organisational success, work assignments etc.,
- Ensure Compliance
  - Ensure designs and implementations adhere to the architecture and do not cause "architectural drift"
- Evolve the Architecture
  - Ensure the architecture remains current

Evolutionary Technical Process

	Pass1 From Business Strategy to Architectural Strategy	Pass 2 From Strategy to Concept	Pass 3 From Concept to Specification	Pass 4 From Specification to Execution	Pass 5 From Execution to Deployment
Architectural Requirements	Context Goals Scope	Use cases Qualities	Refine Use Cases	Concurrency	Developer needs
System Structuring	Meta-architecture	Conceptual Architecture	Logical Architecture	Execution Architecture	Architectural Guidelines
Architectural Validation	Reasoned arguments	Impact Analysis	Estimates	Prototypes	

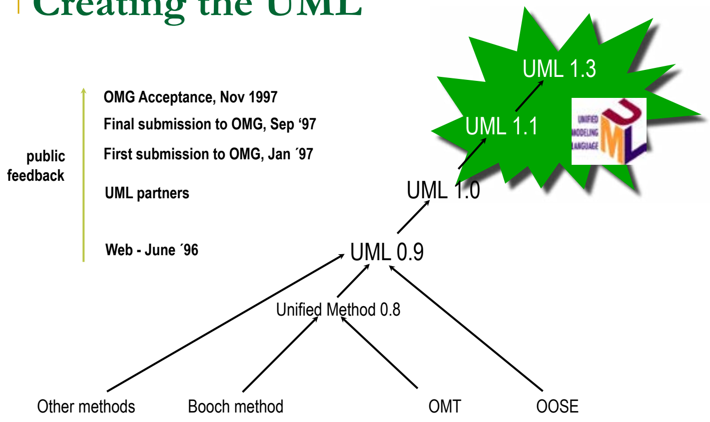
Topic 10: Software Architecture and the UML

"Adapted from "Software Architecture and the UML" by Grady Booch

The Value of the UML

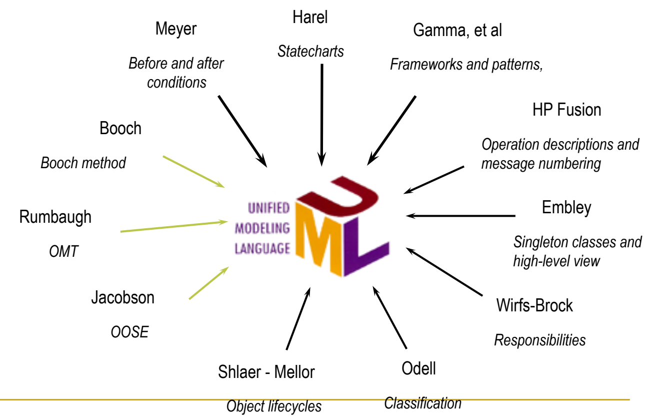
- Is an open standard
- Supports the entire software development lifecycle
- Supports diverse applications areas
- Is based on experience and needs of the user community
- Supported by many tools

Creating the UML



473

Contributions to the UML



474

Overview of the UML

- The UML is a language for
  - visualising
  - specifying
  - constructing
  - documenting

  the artifacts of a software-intensive system

475

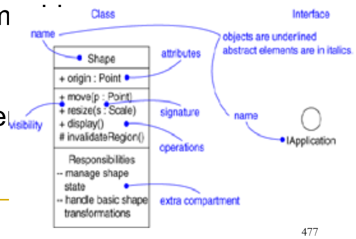
Building Blocks of the UML

- Things
- Relationships
- Diagrams

476

Things in the UML

- Structural things
  - mostly static parts of a model
    - class, interface, collaboration, use case, active class, component, node
- Behavioral things
  - dynamic parts of UML models
    - interaction, state m
- Grouping things
  - organisational parts of UML
    - package, subsyste
- Other things
  - explanatory parts of UML
    - note



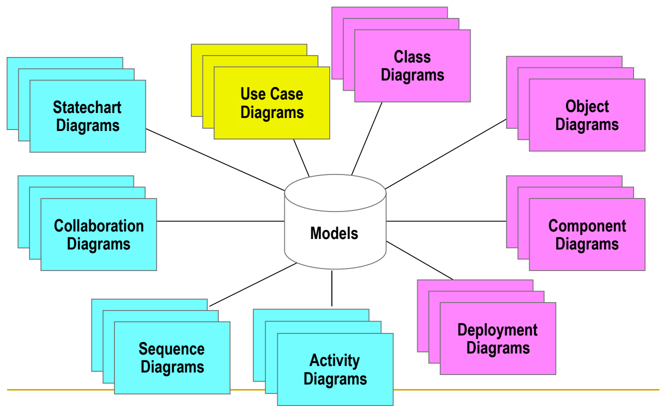
Relationships

- Dependency
  - a semantic relationship between two things in which a change to one thing (the independent thing) may affect the semantics of the other thing (the dependent thing)
- Association
  - a structural relationship that describes a set of links

Relationships

- Generalisation
  - a specialisation/generalisation relationship in which the child shares the structure and the behavior of the parent
- Realisation
  - a realisation is a relationship in which one classifier, such as an interface or a use case, specifies a "contract" that another classifier, such as a class or a collaboration, guarantees to carry out

Diagrams in UML



Requirements

- Structure
  - e.g., system hierarchies, interconnections
- Behavior
  - e.g., function-based behaviors, state-based behaviors
- Properties
  - e.g., parametric models, time variable attributes
- Requirements
  - e.g., requirements hierarchies, traceability
- Verification
  - e.g., test cases, verification results

Evaluation Criteria

- Ease of use
- Unambiguous
- Precise
- Complete
- Scalable
- Adaptable to different domains
- Capable of complete model interchange
- Evolvable
- Process and method independent
- Compliant with UML 2.0 metamodel
- Verifiable

Design Goals

- Satisfy UML for SE RFP requirements
  - 6.5 Mandatory req'ts, 6.6 Optional req'ts
- Reuse UML 2.0 to the extent practical
  - select subset of UML 2.0 reusable for SE apps
  - parsimoniously add new constructs and diagrams needed for SE
- Incrementally grow the language
  - prevent scope and schedule creep
  - take advantage of SE user feedback as language evolves via minor and major revisions

UML 2.0 Support for SE

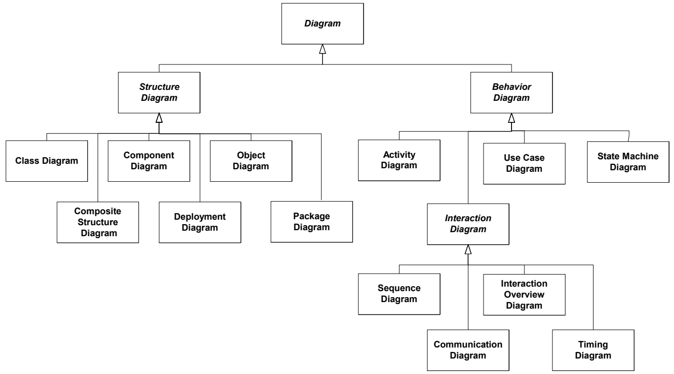
- Allows for more flexible System, Subsystem and Component representations
- Structural decomposition
  - e.g., Classes, Components, Subsystems
- System and component interconnections
  - via Parts, Ports, Connectors
- Behavior decomposition
  - e.g., Sequences, Activities, State Machines
- Enhancements to Activity diagrams
  - e.g., data and control flow constructs, activity partitions/ swim lanes

UML 2.0 Support for SE (cont.)

- Enhancements to Interaction diagrams
  - e.g., alternative sequences, reference sequences, interaction overview, timing diagrams
- Support for information flows between components
- Improved Profile and extension mechanisms
- Support for complete model interchange, including diagrams
- Compliance points and levels for standardizing tool compliance
- Does not preclude continuous time varying properties
  - especially important for SE applications

485

UML 2.0 Diagram Taxonomy



486

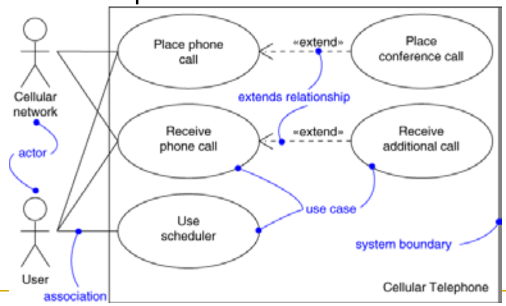
Diagrams

- A diagram is a view into a model
  - Presented from the aspect of a particular stakeholder
  - Provides a partial representation of the system
  - Is semantically consistent with other views
- In the UML, there are nine standard diagrams
  - Static views: use case, class, object, component, deployment
  - Dynamic views: sequence, collaboration, statechart, activity

487

Use Case Diagram

- Shows a set of use cases and actors and their relationships



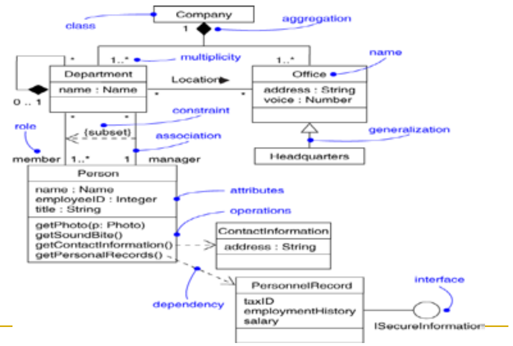
488

Use Case Diagram

- Captures system functionality as seen by users
- Built in early stages of development
- Purpose
  - Specify the context of a system
  - Capture the requirements of a system
  - Validate a system's architecture
  - Drive implementation and generate test cases
- Developed by analysts and domain experts

Class Diagram

- Shows a set of classes, interfaces, and collaborations and their relationships

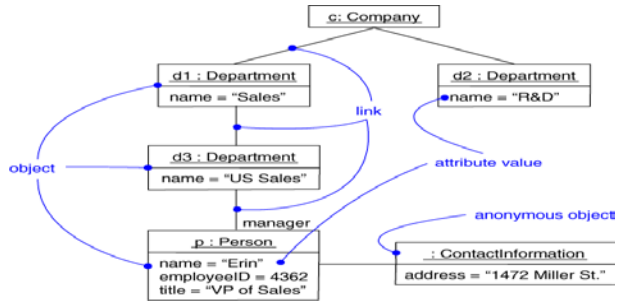


Class Diagram

- Captures the vocabulary of a system
- Addresses the static design view of a system
- Built and refined throughout development
- Purpose
  - Name and model concepts in the system
  - Specify collaborations
  - Specify logical database schemas
- Developed by analysts, designers, and implementers

Object Diagram

- Shows a set of objects and their relationships

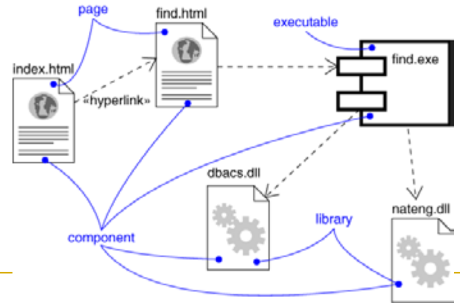


Object Diagram

- Represents static snapshots of instances of the things found in class diagrams
- Addresses the static design view or static process view of a system
- Built during analysis and design
- Purpose
  - Illustrate data/object structures
  - Specify snapshots
- Developed by analysts, designers, and implementers

Component Diagram

- Shows the organisations and dependencies among a set of components

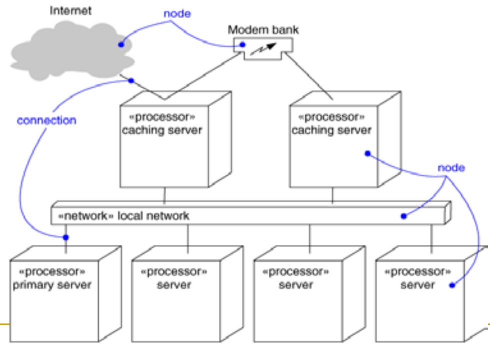


Component Diagram

- Addresses the static implementation view of a system
- Built as part of architectural specification
- Purpose
  - Organise source code
  - Construct an executable release
  - Specify a physical database
- Developed by architects and programmers

Deployment Diagram

- Shows the configuration of run-time processing nodes and the components that live on them

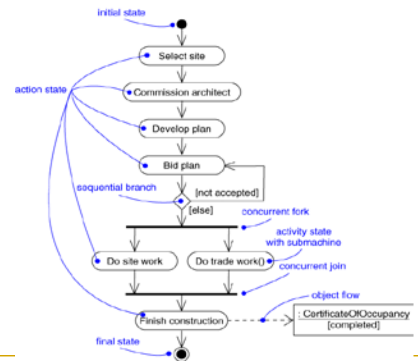


Deployment Diagram

- Captures the topology of a system's hardware
- Built as part of architectural specification
- Purpose
  - Specify the distribution of components
  - Identify performance bottlenecks
- Developed by architects, networking engineers, and system engineers

Activity Diagram

- Shows the flow from activity to activity within a system

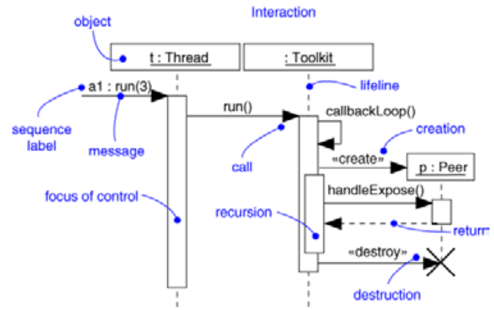


Activity Diagram

- Captures dynamic behavior (activity-oriented)
- A special kind of statechart diagram
- Purpose
  - Model the function of a system
  - Model the flow of control among objects
  - Model business workflows
  - Model operations

Sequence Diagram

- Emphasises the time-ordering of messages



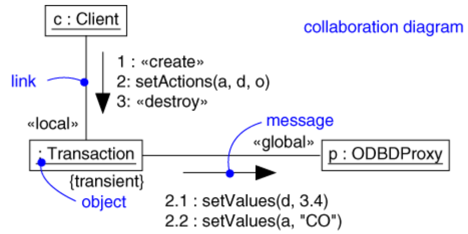
Sequence Diagram

- Captures dynamic behavior (time-oriented)
- A kind of interaction diagram
- Purpose
  - Model flow of control
  - Illustrate typical scenarios

501

Collaboration Diagram

- Emphasises the structural organisation of the objects that send and receive messages



502

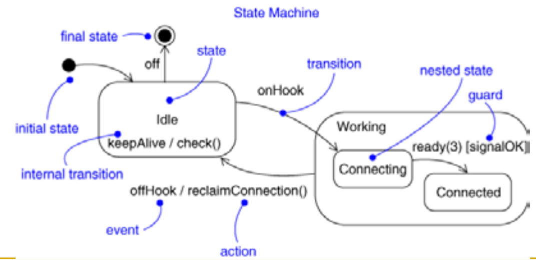
Collaboration Diagram

- Captures dynamic behavior (message-oriented)
- A kind of interaction diagram
- Purpose
  - Model flow of control
  - Illustrate coordination of object structure and control

503

Statechart Diagram

- Shows a state machine, consisting of states, transitions, events, and activities

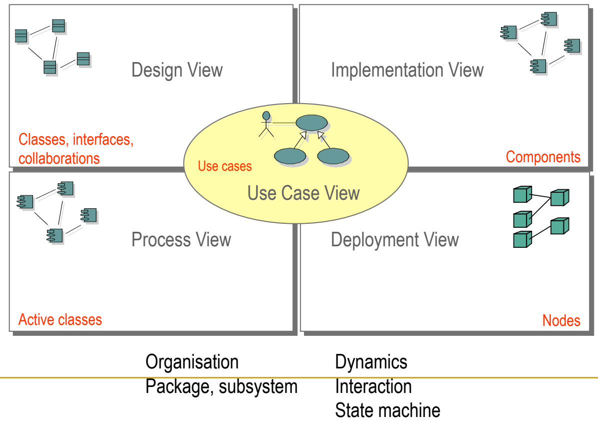


504

Statechart Diagram

- Captures dynamic behavior (event-oriented)
- Purpose
  - Model object lifecycle
  - Model reactive objects (user interfaces, devices, etc.)

Architecture and the UML



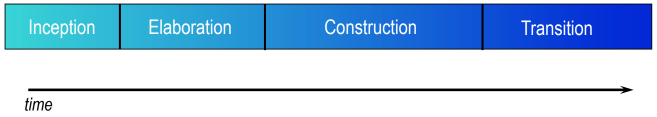
UML Software Development Life Cycle

- Use-case driven
  - use cases are used as a primary artifact for establishing the desired behavior of the system, for verifying and validating the system's architecture, for testing, and for communicating among the stakeholders of the project
- Architecture-centric
  - a system's architecture is used as a primary artifact for conceptualising, constructing, managing, and evolving the system under development

UML Software Development Life Cycle

- Iterative
  - one that involves managing a stream of executable releases
- Incremental
  - one that involves the continuous integration of the system's architecture to produce these releases

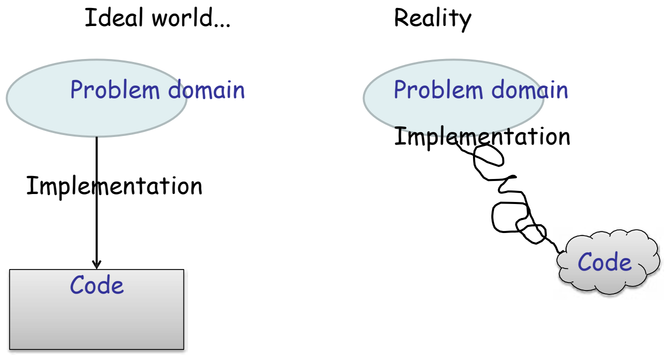
Lifecycle Phases



- Inception — Define the scope of the project and develop business case
- Elaboration — Plan project, specify features, and baseline the architecture
- Construction — Build the product
- Transition — Transition the product to its users

509

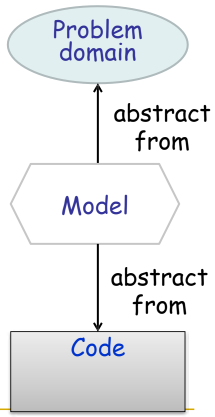
Why do we need models?



510

Why do we need models?

- Models are abstractions of „the real thing“
- They hide complexity by looking at a problem from a certain perspective
  - Focus on relevant parts
  - Ignoring irrelevant details
  - What is relevant depends on the model
- Example: to model the main components of a car, we do not need internal details of the engine.



511

Why model software?

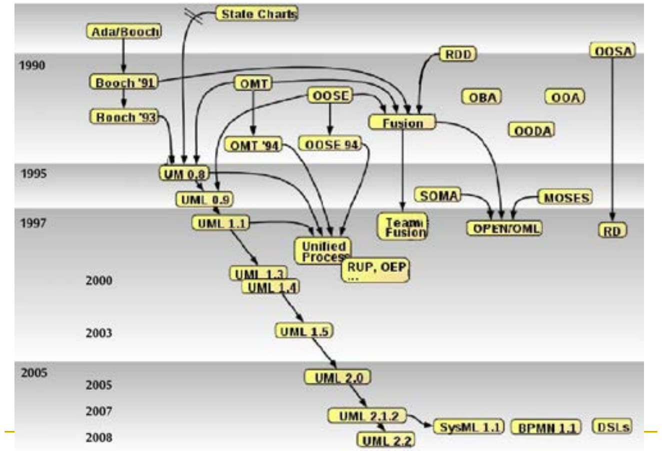
- Why is code itself not a good model?
- Software is getting increasingly more complex
  - Windows XP: ~40 millions lines of code
  - A single programmer **cannot manage** this amount of code in its entirety
- Code is **not easily understandable** by developers who did not write it
- We need **simpler representations** for complex systems
- Modeling is a means for **dealing with complexity**

512

UML - Unified Modeling Language

- Unified Modeling Language (UML)
  - General purpose modeling language (for [OO software] systems)
  - Today 's de-facto standard in Industry
- Sine '97,UML is defined/evolved by the Object Management Group (OMG)
  - Founded 1989 by IBM, Apple, Sun, …
  - Microsoft joined 2008
  - Today more than 800 members

Why "Unified" Modeling Language?



What is UML?

UML is a standardized language for specifying, visualizing, constructing and documenting (software) systems

- Specification: the language is supposed to be simple enough to be understood by the clients
- Visualization: models can be represented graphically
- Construction: the language is supposed to be precise enough to make code generation possible
- Documentation: the language is supposed to be widespread enough to make your models understandable by other developers

What is UML?

- UML defines
  - Entities of models and their (possible) relations
  - Different graphical notations to visualize structure and behavior
- A model in UML consist of
  - Diagrams
  - Documentation which complements the diagrams

What UML is *not*!

- Programming language
  - this would bound the language to a specific computing architecture
  - however code generation is encouraged
- Software development process
  - Choose your own process, (e.g. Waterfall-model, V-model, …)
  - Use UML to model & document
- CASE tool specification
  - however tools do exist: Sun, IBM Rose, Microsoft Visio, Borland Together etc.

Diagrams in UML

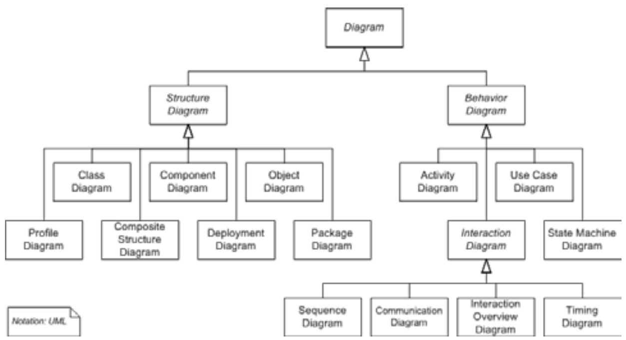
- UML currently defines 14 types of diagrams
  - 7 types of Structure Diagrams
  - 7 types of Behavior Diagrams
- Different diagrams provide different levels of abstraction
  - High-level structure vs. low-level structure

Example: *components* vs. *objects*

  - High-level behavior vs. low-level behavior

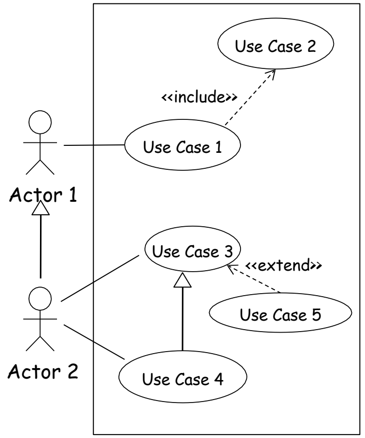
Example: *use-case* vs. *feature-call sequence*

Diagrams in UML



Elements of Use Case diagrams

- Entities:
  - actors
  - use cases
- Relations:
  - association between an actor and a use case
  - generalization between actors
  - generalization between use cases
  - dependencies between use cases
- Comments:
  - system boundaries



Use Case specification

- Each Use Case shown in a diagram should be accompanied by a textual specification
- The specification should follow the scheme:
  - Use Case name
  - Actors
  - Entry Condition
  - Normal behavior
  - Exceptions
  - Exit Condition
  - Special Requirements (e.g. non-functional requirements)

521

Use Case specification

- Example for „ Pay Food " Use Case

Name: Pay Food

Actors: Client, Teller

Entry Condition: Client has food and wants to pay it

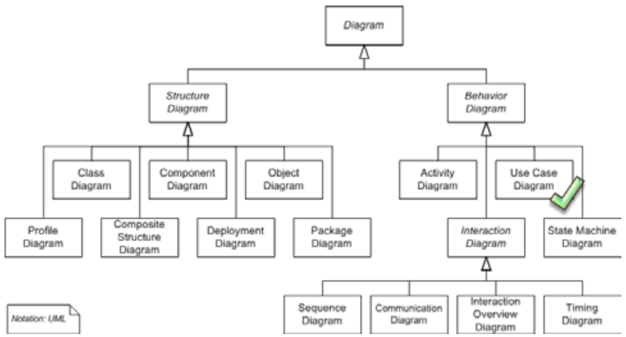
Normal behavior: Teller types in food; Total amount is shown on display; Client puts card into reading device; Amount gets withdrawn; If not enough money on card, then an error message is shown; Return card to client

Exceptions: If card is not readable, then show error message and return card; If power failure while card in reading device, wait until power is back and return card – payment needs to be redone

Exit Condition: Client has paid the food and gets the card back

522

Diagrams in UML



523

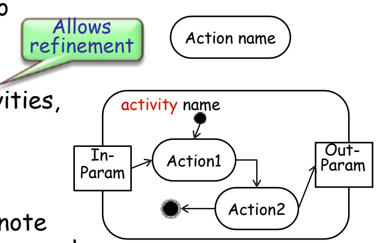
Activity diagrams

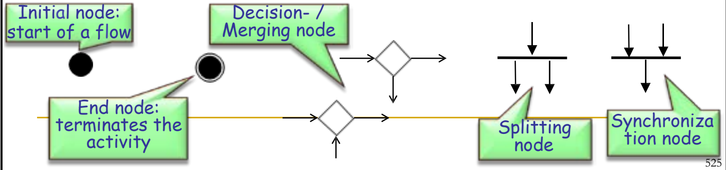
- Activity diagrams are used to model (work)flows
- They are used visualize complex behavior, e.g.
  - Business process
  - Algorithms (though less common)
- Tokens are used to determine the flow, similar to Petri-nets
- A common usage: detailed modeling of Use Cases

524

Elements of Activity diagrams

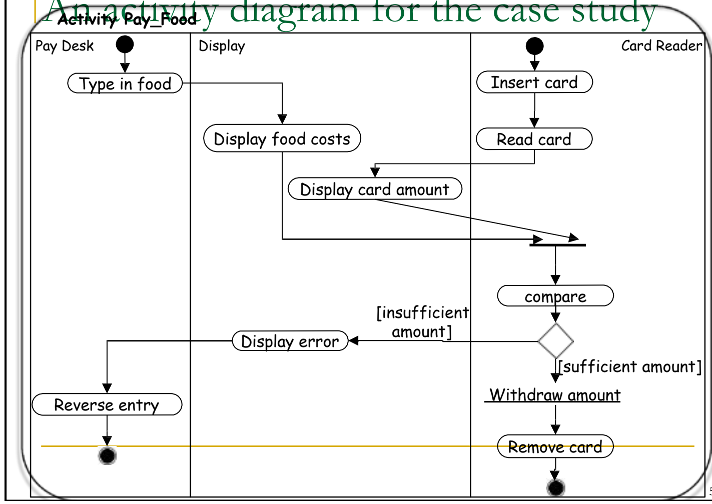
- Action: atomic element, no further splitting possible
- Activity: can contain activities, actions, control nodes
- Control nodes: used to denote control struture in the flowgraph





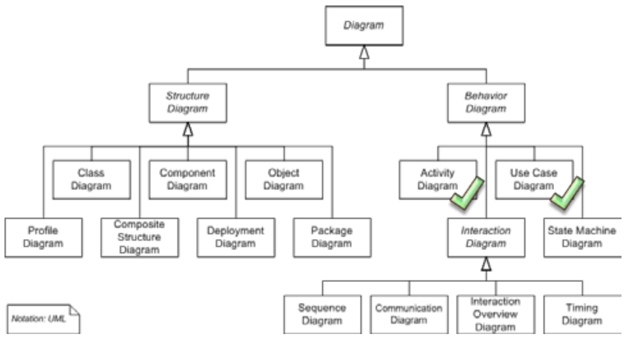
525

An activity diagram for the case study



526

Diagrams in UML



527

UML Class diagrams

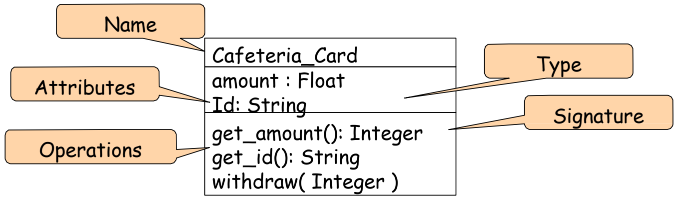
- Keep in mind:
  - Use Cases represent an external view of the system "s behavior
  - Classes represent inner structure of the system

→ No correlation between use cases and classes

- Class diagrams are used at different levels of abstraction with different levels of details
  - Early phase: identifying classes and their relations in the problem domain (high-level, no implementation details)
  - Implementation phase: high level of detail (attributes, visibility, …), all classes relevant to implement the system

528

Classes



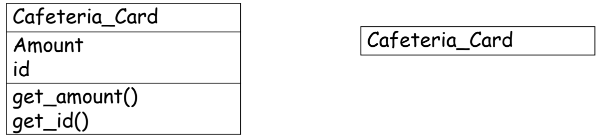
A class encapsulates **state** (attributes) and **behavior** (operations)

- Each attribute has a type
- Each operation has a signature

The class name is the only mandatory information

More on classes

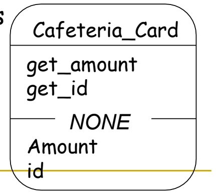
Valid UML class diagrams



Corresponding BON diagram

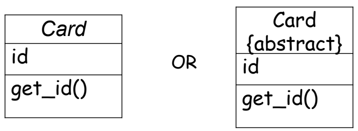
- No distinction between attributes and operations

(uniform access principle)

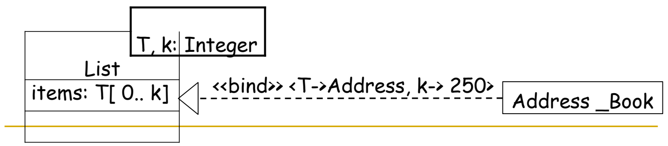


More on classes

- Abstract classes have a italicized class name or {abstract} property (also applicable to operations)

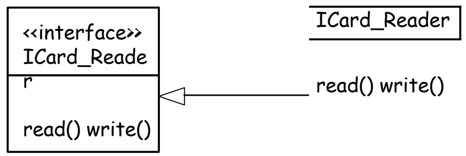


- Parameterized classes



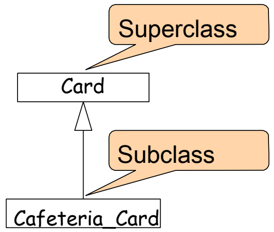
Interface classes

- Interface classes have a keyword <<interface>>
- Interfaces have no attributes
- Classes implement an interface using an implementation relation



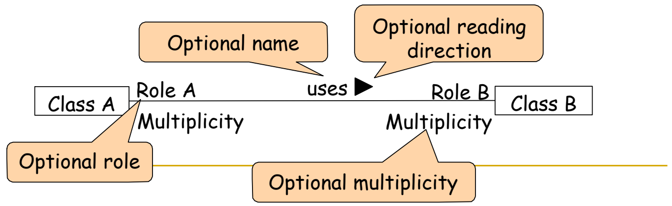
Generalization and specialization

- Generalization expresses a **kind-of** ("is-a") **relationship**
- Generalization is implemented by **inheritance**
  - The child classes inherit the attributes and operations of the parent class
- Generalization simplifies the model by **eliminating redundancy**



Associations

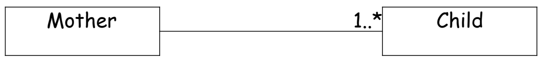
- A line between two classes denotes an association
- An association is a type of relation between classes
- Objects of the classes can communicate using the association, e.g.
  - Class A has an attribute of type B
  - Class A creates instances of B
  - Class A receives a message with argument of type B



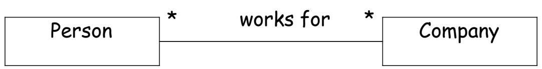
Association multiplicity

- Multiplicity denotes how many objects of the class take part in the relation
- 1-to-1

- 1-to-many

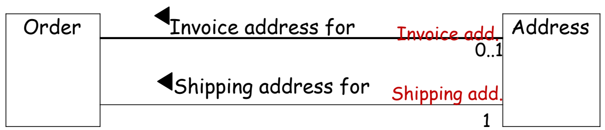


- many-to-many

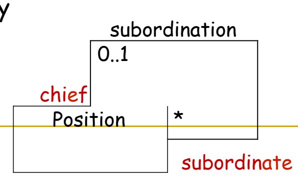


Association roles

- Different instances of an class can be differentiate using roles
- Example: Invoice and shipping address are both addresses



- Example: Position hierarchy



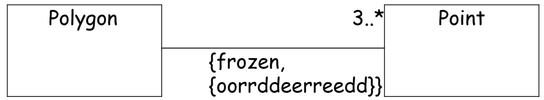
Special associations

- Aggregation – "part-of" relation between objects
  - Component can be part of multiple aggregates
  - Component can be created and destroyed independently of the aggregate

- Composition – strong aggregation
  - A component can only be part of a single aggregate
  - Exists only together with the aggregate

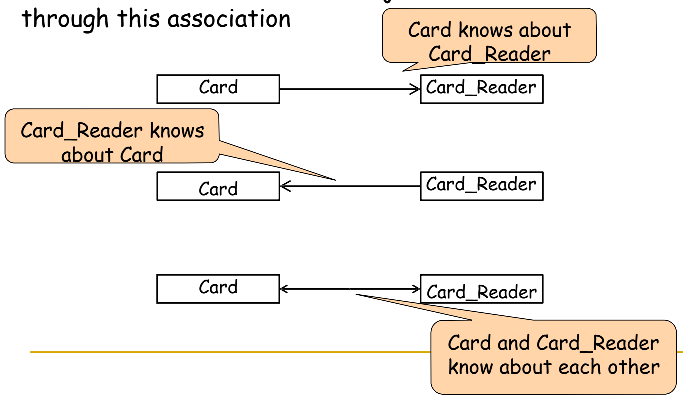
More on associations

- Ordering of an end – whether the objects at this end are ordered
- Changeability of an end – whether the set of objects at this end can be changed after creation

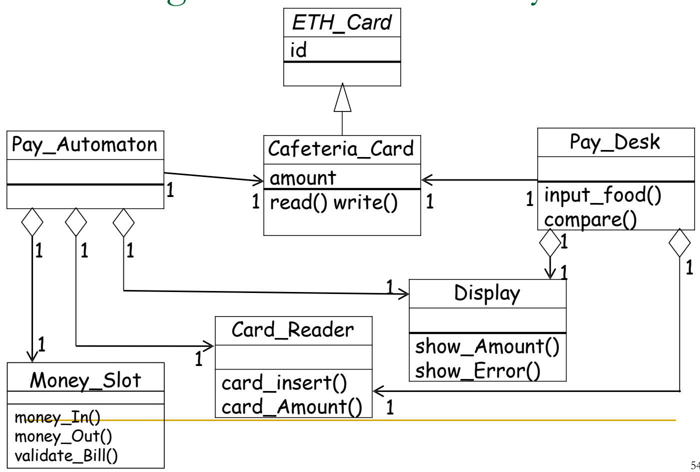


Navigability of association

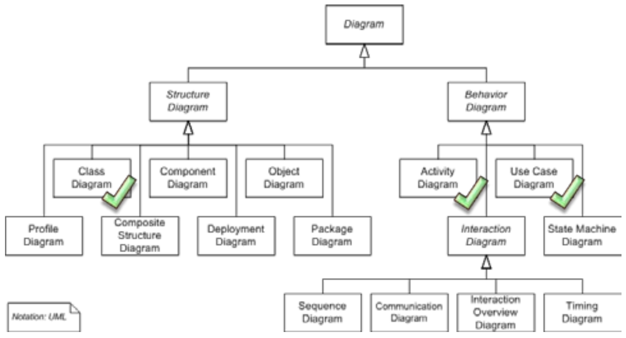
- Associations can be directed
- Direction denotes whether objects can be accessed through this association



Class diagram for the case study

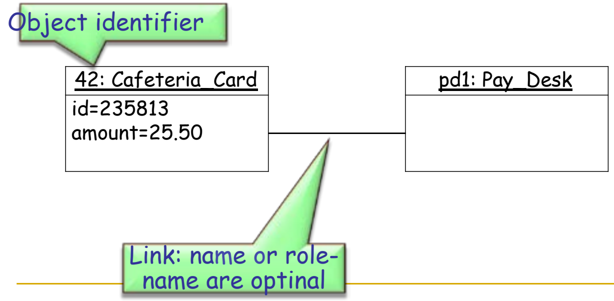


Diagrams in UML

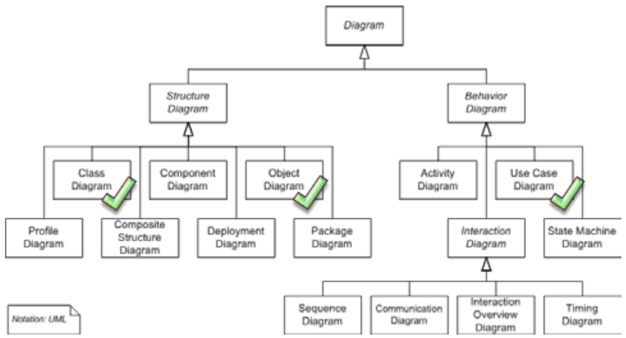


UML Object diagrams

- An Object diagram is used to denote a snapshot of the system at runtime
- It shows the existing objects, their attribute values and relations at that particular point of time



Diagrams in UML



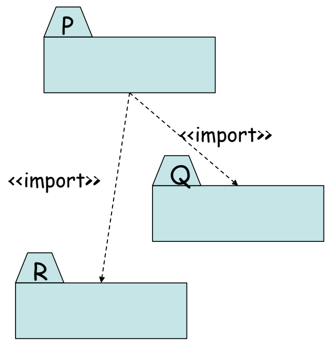
UML packages

A package is a UML mechanism for **organizing elements** into groups

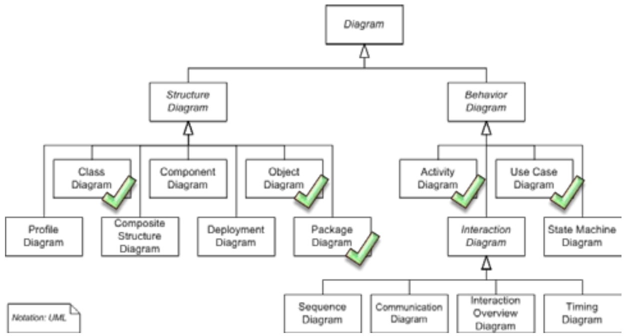
- Usually not an application domain concept
- Increase readability of UML models

Decompose complex systems into subsystems

- Each subsystem is modeled as a package

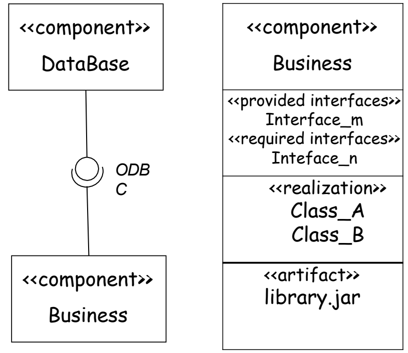


Diagrams in UML

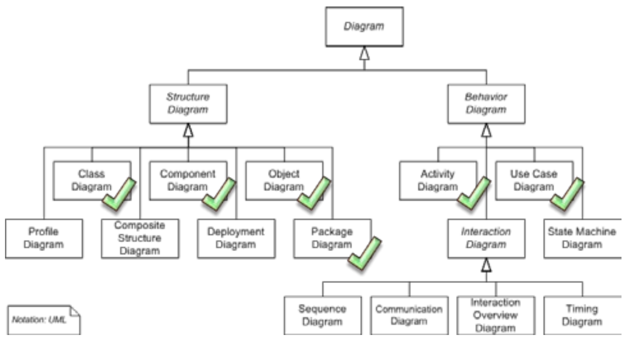


Component diagrams

- Entities:
  - components
    - programs
    - documents
    - files
    - libraries
    - DB tables
  - interfaces
  - classes
  - objects
- Relations:
  - dependency
  - association (composition)
  - implementation



Diagrams in UML

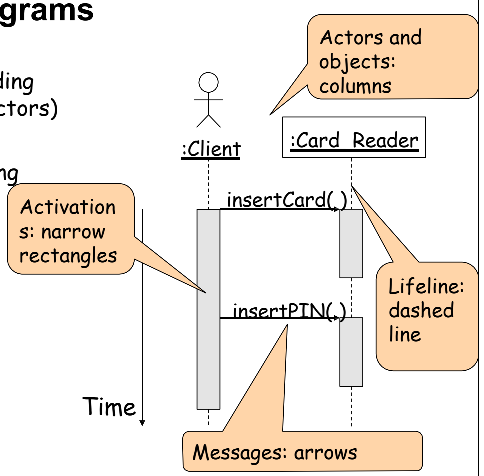


Overview

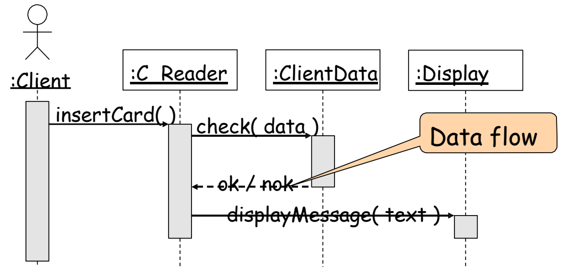
- We will now look at two more diagrams which are used to model the behavior of a system.
- Sequence diagrams: used to describe the interaction of objects and show their "communication protocol"
- State diagrams: focus on the state of an object (or system) an how it changes due to events

Sequence diagrams

- Entities:
  - objects (including instances of actors)
- Relations:
  - message passing
- Sugar:
  - lifelines
  - activations
  - creations
  - destructions
  - frames



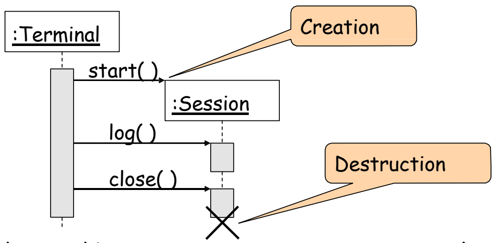
Nested messages



The source of an arrow indicates the activation which sent the message

An activation is as long as all nested activations

Creation and destruction



Creation is denoted by a message arrow pointing to the object

In garbage collection environments, destruction can be used to denote the end of the useful life of an object

From Use Cases to Sequence diagrams

Sequence diagrams are **derived from flows of events** of use cases

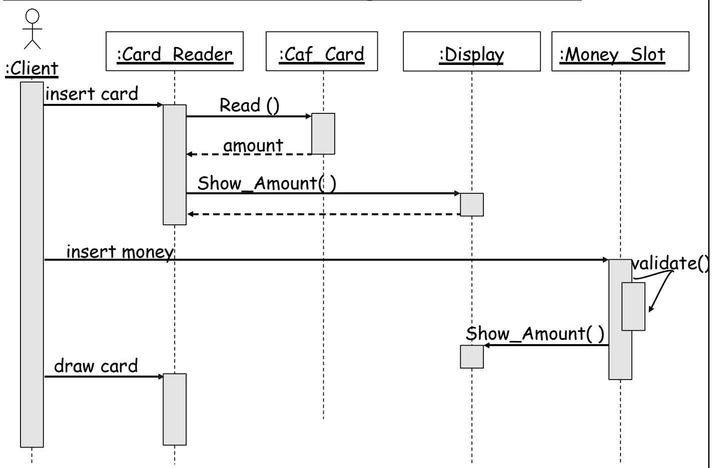
An event always has a **sender** and a **receiver**

- Find the objects for each event

Relation to object identification

- Objects/classes have already been identified during object modeling
- Additional objects are identified as a result of dynamic modeling

Example Sequence diagram



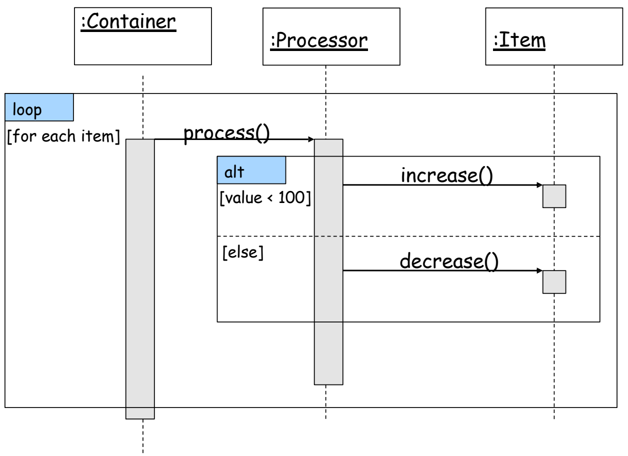
553

Example Sequence diagram

- The diagram shows only the successful case
- Exceptional case could go either on another diagram or could be incorporated to this one
- Sequence diagrams show main scenario and "interesting" cases
  - interesting: exceptional or important variant behavior
- Need not draw diagram for every possible case
- would lead to too many diagrams

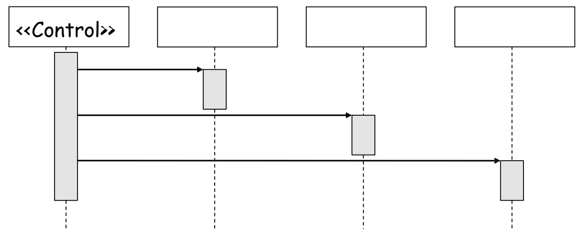
554

Interaction frames



555

Fork structure

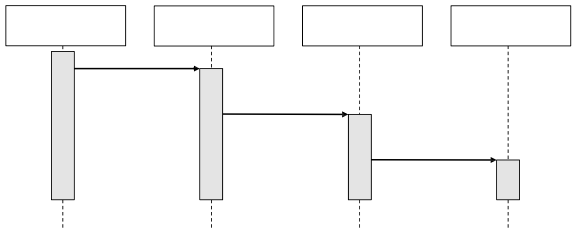


The **dynamic behavior is placed in a single object**, usually a control object

It knows all the other objects and often uses them for direct queries and commands

556

Stair structure



The **dynamic behavior is distributed**

- Each object delegates some responsibility to other objects
- Each object knows only a few of the other objects and knows which objects can help with a specific behavior

Fork or stair?

Object-oriented supporters claim that the stair structure is better

- The more the responsibility is spread out, the better

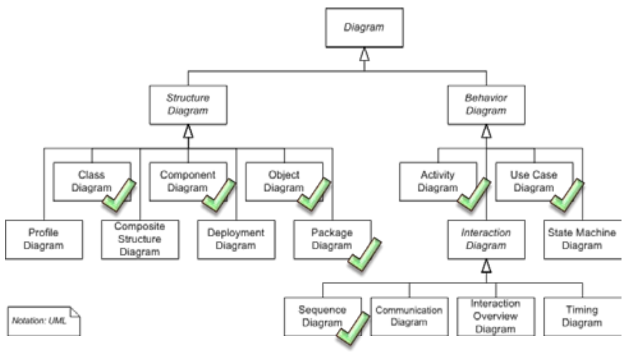
Choose the **stair** (decentralized control) if

- The operations have a **strong connection**
- The operations will **always** be performed in the **same order**

Choose the **fork** (centralized control) if

- The operations can **change order**
- **New operations** are expected to be added as a result of new requirements

Diagrams in UML

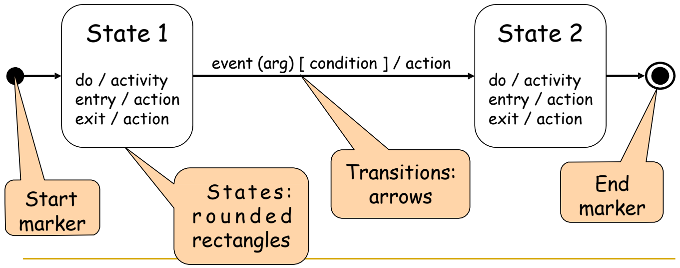


State Machine Diagrams

- UML State Machine Diagrams are a powerful notation to model finite automata
- It shows the states which an object or a (sub)system – depending on the level of abstraction – can have at runtime
- It also shows the events which trigger a change of state

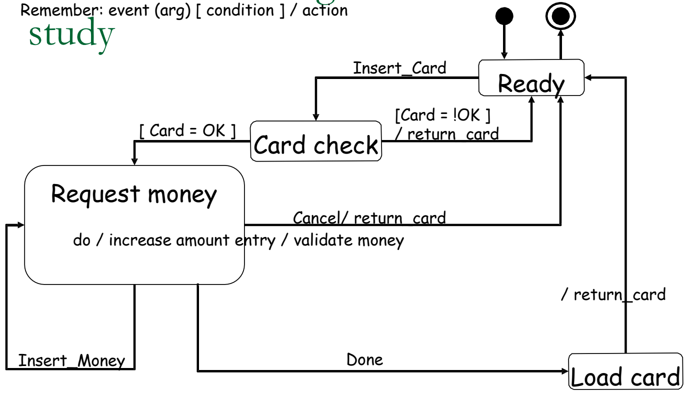
State Machine diagrams

- Entities:
  - states: name, activity, entry/exit action
- Relations:
  - transitions between states: event, condition, action



State Machine diagram for the case study

Remember: event (arg) [condition] / action



Composite/nested State Machine diagrams

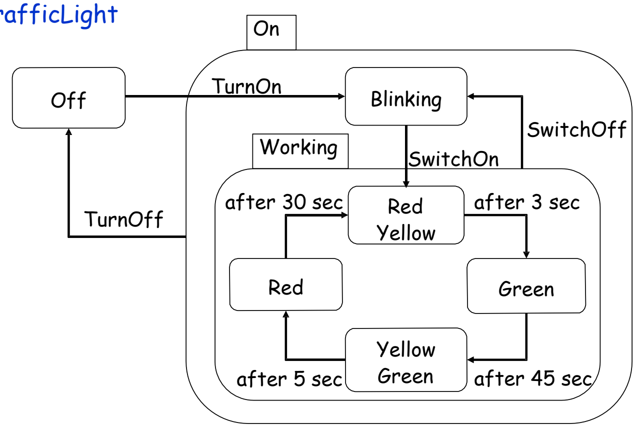
Activities in states can be **composite items** that denote other state diagrams

Sets of substates in a nested state diagram can be denoted with a superstate

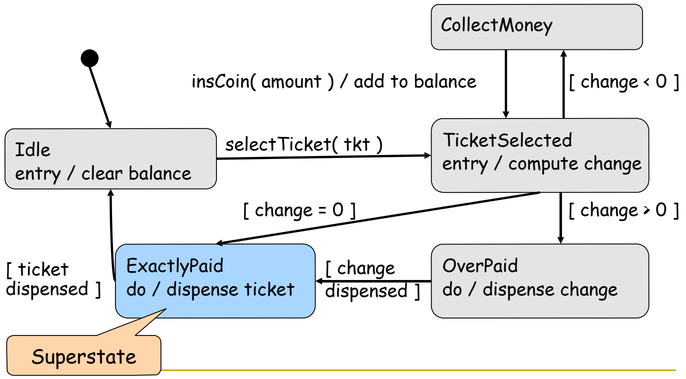
- Avoid spaghetti models
- Reduce the number of lines in a state diagram

State diagrams: example composite state

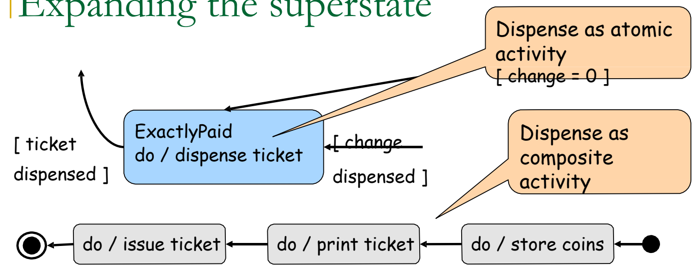
TrafficLight



Example: superstate



Expanding the superstate



Transitions from other states to the superstate **enter the first substate** of the superstate

Transitions to other states from a superstate are **inherited by all the substates** (state inheritance)

State diagram vs. Sequence diagram

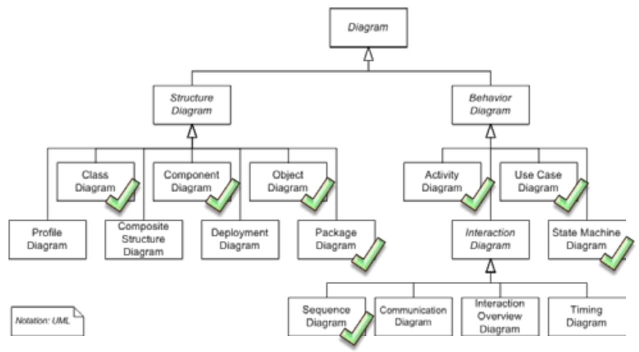
State diagrams help to identify

- Changes to an individual object over time

Sequence diagrams help to identify

- The temporal relationship between objects
- Sequence of operations as a response to one or more events

Diagrams in UML



Practical tips

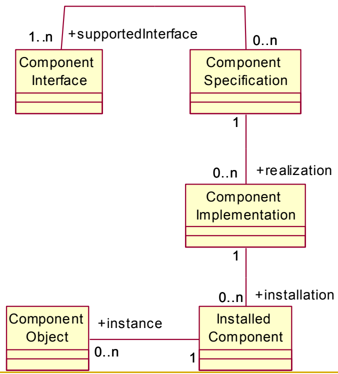
- Create component diagrams only for large, distributed systems
- Create state diagrams only for classes with complex, interesting behavior (usually classes representing entities from the problem domain or performing control)
- Create activity diagrams for complex algorithms and business processes (not for every operation)
- Create sequence diagrams for nontrivial collaborations and protocols (not for every scenario)
- Don “t put too much information on a diagram
- Choose the level of abstraction and maintain it

Topic 11: Architecture and Component-Based Development

Architectural Mismatch

- The biggest single problem for Component-Based Development is “architectural mismatch”
- A component created for one context won’t work in another
- Recent work by John Daniels and John Cheeseman about specifying components addresses this problem

Component Concepts



Component Concepts

- Component Interface
  - How to use the component
- Component Specification
  - How to build the component
- Component Implementation
  - Software that meets a component specification
- Installed Component
  - Executable
- Component Object
  - Instantiation of a component

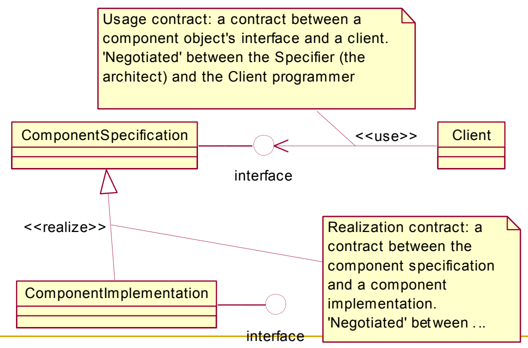
573

Problems with Interfaces

- The operational interface of a component is a list of operations and their signatures
- But this tells you how to USE a component...
  - E.g., what legal messages to send
- ..but not how it will behave
  - Programming syntax can't tell us, we need semantic interfaces
- Therefore we need to separate out the notion of a *Component Interface* from a *Component Specification*

574

Two distinct contracts



575

Interfaces v Component Specs.

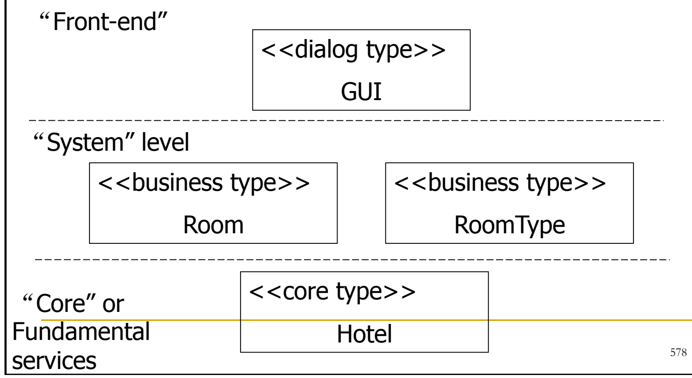
- Component Interface
  - Represents usage contract
  - Provides a list of operations
  - Defines underlying logical information model specific to that interface
  - Specifies how operations affect or rely on the information model
  - Describes local effects only
- Component Specification
  - Represents the realisation contract
  - Provides a list of supported interfaces
  - Defines the un-time unit
  - Defines the relationships between the information models of different interfaces
  - Specifies how operations should be implemented in terms of usage of other interfaces

576

Two Levels of Component Interfaces

- Cheeseman and Daniels propose two basic levels of components/interfaces...
- System Interfaces
  - Derived from use cases
    - One system interface supporting one UI dialog type
      - E.g., 'Make Reservation' dialog type which uses 'IMakeReservation' System Interface
      - Use Case steps -> individual operations
- Core Business Interfaces
    - Represent fundamental services that support system use cases
      - Requires identifying <<core types>>

An Architecture for CBD



How to Design a First-Cut Architecture for CBD

- Create a "Business Type" Diagram
  - A UML class diagram representing the key concepts and the associations between them
- Separate the Core Types from other "business types"
- Add an Interface Type for each Core Type identified
- Create a GUI "dialog type" by mapping steps in Use Cases to operations on interfaces
  - Requires a standard template to be used with Use Cases

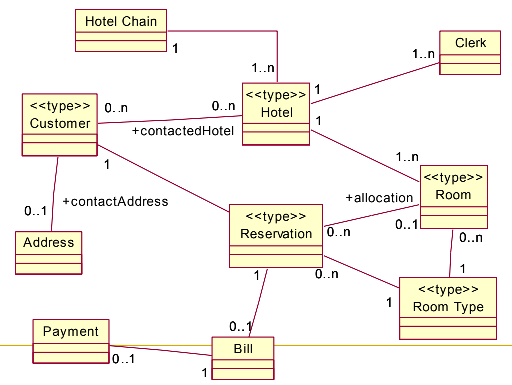
Identifying Core Types

- Core Business Types represent the primary business information that the system must manage
- Each core type will correspond directly to a Core Business Interface
- A core type has
  - A business identifier (a name or unique number), usually independent of other identifiers
  - Has independent existence (no mandatory associations), except to a categorising type
    - NB a mandatory association has a "1" at the opposite end

Worked Example: Hotel Reservation

- We want to provide some automated support for managing hotel reservations using CBD.

Concept Model



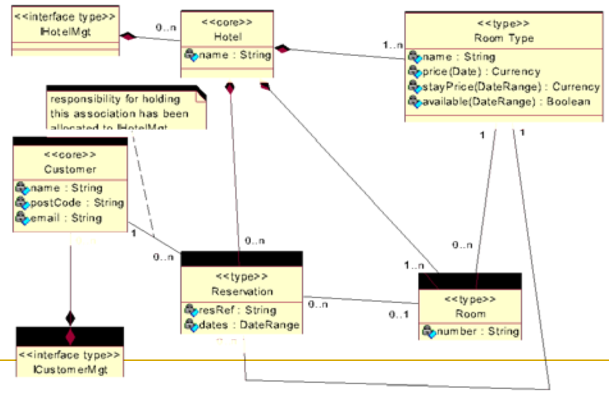
Analysis of the Concept Model

- The following types are excluded as being redundant, out of scope etc., or as attributes of other types
  - Hotel Chain, Clerk, Bill, Payment, Address
- The following have mandatory associations and will become system-level <<business types>>
  - Room, RoomType, Reservation
- The following are left as <<core types>>
  - Customer, Hotel
    - They have meaningful identifiers (names) in the world and no mandatory associations

Adding <<Interface Types>>

- For each <<core type>> add an <<interface type>>
  - ICustomerMgt and IHotelMgt
- Remaining <<business types>> are represented in the class diagram as "contained by" the <<core types>>
- Design decisions have to made about which <<interface type>> owns any associations between <<business types>>

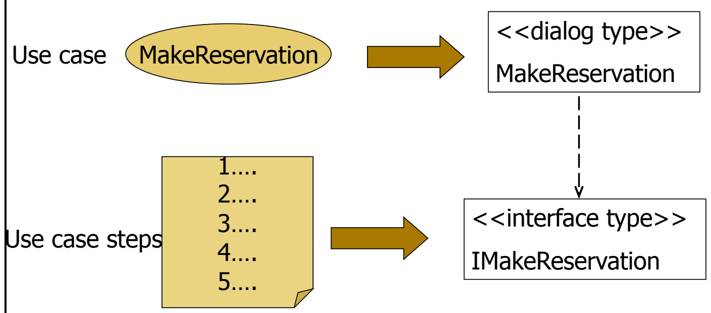
Identifying Business Interfaces



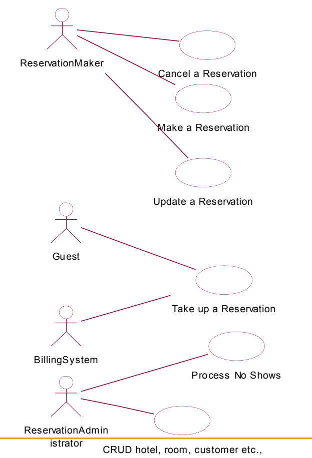
Creating System-Level Interfaces

- The <<interface types>> in the previous step represent core business services
  - They are not user interfaces
- System-level interfaces are needed for the <<business types>>
  - These are created by examining the use cases they participate in

Mapping Use Cases to Interfaces



Use Case Diagram



Sample Use Case (1)

Name	Make a Reservation
Initiating Actor	Reservation Maker
Goal	Reserve a Room at Hotel

Happy Case Scenario:

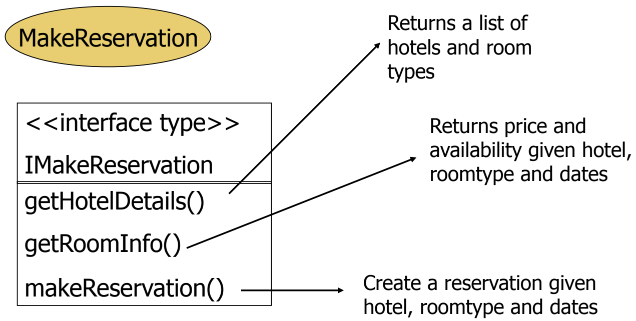
1. ReservationMaker asks to make a reservation
2. ReservationMaker selects hotel, dates and room type
3. System provides availability and price
4. ReservationMaker agrees to proceed
5. ReservationMaker provides name and postcode
6. ReservationMaker provides contact e-mail address
7. System makes reservation and gives it a tag
8. System reveals tag to ReservationMaker
9. System creates and sends confirmation by e-mail

Sample Use Case (2)

Extensions

3. Room Not Available
    - a) System offers alternative dates and room types
    - b) ReservationMaker selects from alternatives
6. Customer already on file
    - a) Resume 7

Use Case Step Operations

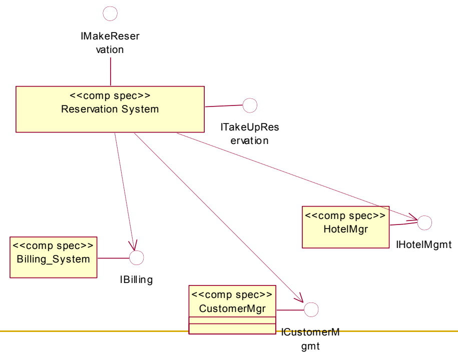


The "MakeReservation" use case becomes the "IMakeReservation" interface

Component Specifications

- Deciding which components, and which interfaces they support, are fundamental architectural decisions
- Starting assumption: one component spec. per business interface...
- ...and a single system component spec.for all use cases...(Reservation System)
    - IMakeReservation would be folded in with other use cases
    - Major alternative option is one per use case
- ...and one each for any existing legacy systems that need to be "wrapped"

Initial Component Architecture for a Hotel System



Summary

- An architectural approach to CBD requires:
  - Separating Component Interfaces from Component Specifications
  - Creating Interfaces (usage contracts) for each core business function
  - ...and System-level Interfaces for use case functionality
  - Providing Component Specifications to support the Component Interfaces
  - Dialog types are used to provide GUIs for the application

Topic 12: Software Architecture Evaluation

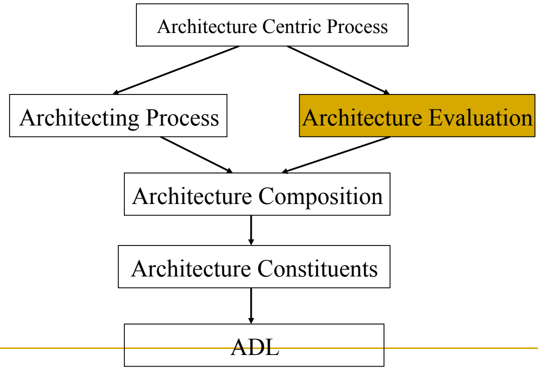
Outline

- Architecture Evaluation Introduction
- Evaluation Methods
- Component Based Architecture Evaluation
- Beyond Components
- Conclusion

Introduction

- What is architecture Evaluation
  - Architecture Assessment/Evaluation: Assign a specific value to software architecture suitability
  - Architecture Review: Doesn't assign a specific as a measure of suitability
  - Architecture Analysis: Techniques used to perform architecture reviews/assessment/evaluation
- Being used in two context
  - A validation step for an architecture being developed
  - A step in the acquisition of software system

Architecting Landscape



Benefits

- Financial
- Increased Understanding and Documentation of System
- Detection of Problems with Existing Architecture
- Prediction of final product quality/Risk Management
- Clarification and Prioritisation of Requirements
- Organisational Learning

Evaluation Preconditions

- Understanding of Evaluation Context
- The Right People
- Organisational Expectations and Support
- Evaluation Preparation
- Architecture Representation

Evaluation Activities

- Recording and Prioritising
- Evaluating
  - Cost
  - Functionality
  - Performance
  - Modifiability
  - …
- Reviewing Requirements
- Reviewing Issues
- Reporting Issues

Evaluation Output

- Ranked Issues
- Report
- Scenario Set
- Preliminary System Predictions
- Enhanced Documentation

Evaluation Approaches

Review Method	Generality	Level of Detail	Phase	What is Evaluated	Example
Questionnaire	General	Coarse	Early	Artifact Process	SREM
Checklist	Domain-specific	Varies	Middle	Artifact Process	AT&T
Scenarios	System-Specific	Medium	Middle	Artifact	SAAM breeds, ATAM
Metric	General or domain-specific	Fine	Middle	Artifact	Adapted Traditional Metrics
Prototype, Simulation, Experiment	Domain-specific	Varies	Early	Artifacts	

Scenario Based Analysis

- **SAAM** (Scenario-based architecture analysis Method)
  - **SAAMCS** (SAAM Complexity of Changes)
  - **ESAAMI** (Integrating SAAM in Domain Centric and Reused Based Development Process)
  - **SAAMER** (SAAM Evolution and Reusability)
- **ATAM** (Architecture Tradeoff Analysis Method)
- **SBAR** (Scenario-Based Architecture Reengineering)
- **ALPSM** (Architecture Level Prediction of Software Maintenance)

Metrics for Quality Attribute

- Traditional information hiding and modulisaiton (cohesion/coupling), complexity metrics
- Object-Oriented Metrics
- Architecture Metrics adapted from OO metrics
  - Depth of Inheritance Tree (DIT)
  - Message Passing Coupling (MPC)
  - Data Abstraction Coupling (DAC)
  - Lack of Cohesion in Methods (LCOM)
  - NOM, NOC, RFC, WMC

Architecture Quality Metrics

- Service Utilising Metrics for component framework and product line
- Evolution Metric
  - Evolution Cost Metrics (Add/Remove/Modify cost)
  - Architecture Preservation Factor
  - Architecture Preservation Core
- SAEM (Software Architecture Evaluation Model)

Component Based Architecture Evaluation

- Component and frameworks have *certified properties*
  - *Some properties of components are imposed by underlying framework*
  - *Some interaction between components and their topologies are imposed by underlying framework*
- The certified properties provide the basis for *predicting the properties of systems* built from components

Topic 13: Software Architecture and OO Development

Structure and Space in Object-Oriented Software

'Space' in Software

- Software has no physicality
  - Michael Jackson says in order to create virtual machines we just create them
  - Fred Brooks Jr. says software is "pure thought stuff"
  - Source code is just a set of instructions that translates into machine instructions
    - N.B. strictly, therefore, source code is a specification of an executable program
- But in architecture (of the built environment) space is a *logical* as well as a physical concept

609

Object-Oriented Software Construction

- Objects and Classes are behavioural abstractions
  - We separate objects in Class A from those in Class B on the basis of their different behaviour
- But in the machine an object instance is a data abstraction
  - A pointer or reference is returned to the object's data variables ONLY
    - I.e. each object instance has its OWN copy of the data, but no individual operations – these belong to the class as a whole
- Objects are therefore *logical* abstractions
  - -don't really exist at machine-level

610

Responsibility-Driven Design (1)

- Designing object system involves
  - Identifying behavioural abstractions (Object Types)
  - Assigning them responsibilities
  - Mapping Object Types to Object Classes
    - Enforcing encapsulation and information-hiding
    - Creating Interfaces so that client objects know how to request executable behaviour from server objects
  - Turning responsibilities into operations
    - and the methods that implement them
  - Turning collaborating classes into data members to hold Object IDs

611

Responsibility-Driven Design (2)

- A well-established technique for Responsibility-Driven Design is CRC cards
- 6" x 4" index cards
  - Divided into 3 fields (Class, Responsibility, Collaborators)
  - One for each candidate Object
- Used to role-play scenarios to see if responsibilities have been distributed properly
  - Cheap and fun way to validate the dynamic behaviour of object systems prior to coding!

612

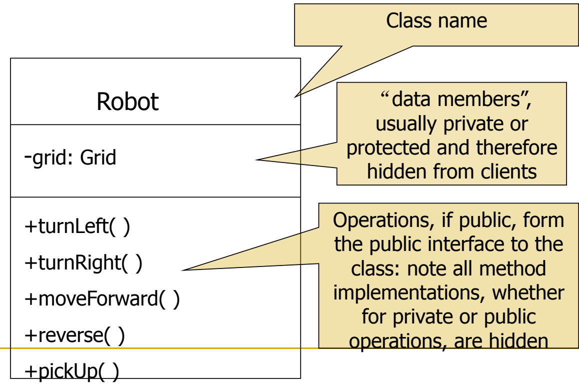
CRC (Class, Responsibility, Collaboration) Cards

Class	
Responsibilities	Collaborators

The CASE tool is a 6" x 4" index card!

Each person "role-plays" a class (i.e. a CRC card) to explore distribution of responsibilities

Encapsulation and Information Hiding



Encapsulation

- Encapsulation is the hiding of all design decisions that the client doesn't need to know about. Typically this includes:
  - Data structures
  - Collaborating classes and objects
  - Methods
  - Private Operations etc.,
- An object is therefore a sort of protected virtual space
  - Like a "neighbourhood" in the previous Topic

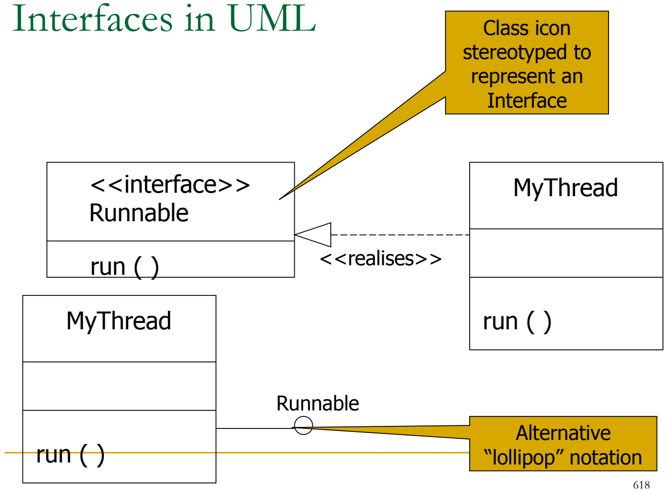
Interfaces (1)

- Ideally we would like classes and objects to be completely decoupled from each other
  - But references (objectID's) are needed otherwise programs won't work
  - Collaboration requires some objects to know how to call other objects and request their behaviour
- Therefore Interfaces are needed
  - Compare with gateways and access paths in "Neighborhood Boundary" pattern
  - In Java a special Interface construct is provided
  - In C++ an Abstract Class can be used as a protocol class to the same effect

Interfaces (2)

- Interfaces should be designed to be stable
  - Operation names and parameters of abstract behaviours
- Implementation can therefore vary without the Client object needing to know
  - Different methods, even different (collaborating) objects can handle the request for executable behaviour
  - Client only needs guarantee that the behaviour will be performed correctly in response to the request (message)
- N.B. A class can support 1 or many interfaces

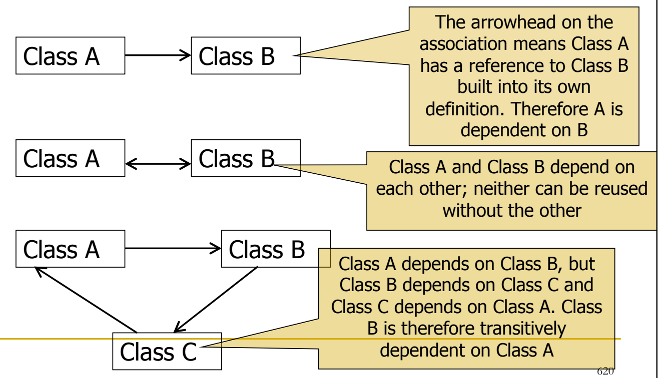
Interfaces in UML



The Significance of Interfaces for Architecture

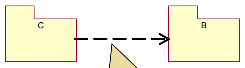
- An interface can be thought of as an access path or gateway to an encapsulated "space"
  - Space can be an object instance, a class, a package or any logical computational component
- We need to narrow interface bandwidth between spaces
  - An interface also implies a dependency
    - If Class A holds a reference to Class B it is dependent upon it (e.g., may not be compilable without it)
    - If the Interface is physically separate from the class that realises it and the reference is to the Interface the client is only dependent on the Interface (recommended)
- Interfaces can form a "scaffolding" for the system

Cyclical Dependency



The UML Package Construct

- UML has a construct higher than a class
  - The package
- Represents a "namespace" for collecting UML elements
  - Often a collection of logical classes
  - E.g/. Collaboration, component
- Same dependency rules apply



This arrow means "DependsOn". If the package collects classes it means at least one class in package C depends on at least one class in package B

Dependency Heuristics

- **Minimise all dependencies**
- **As far as possible make dependencies unilateral**
  - i.e. "one-way"
  - In the previous diagram (top) Class B is reusable and, provided its interface doesn't change, can be reimplemented without disturbance to Class A
- **Avoid cyclical dependencies**
  - They seem unilateral but have transitive dependencies
- **Only use bilateral or cyclical dependencies if classes are designed to work together in ALL circumstances**
  - In which case package them together as collaborations or components

Different Kinds of Dependencies

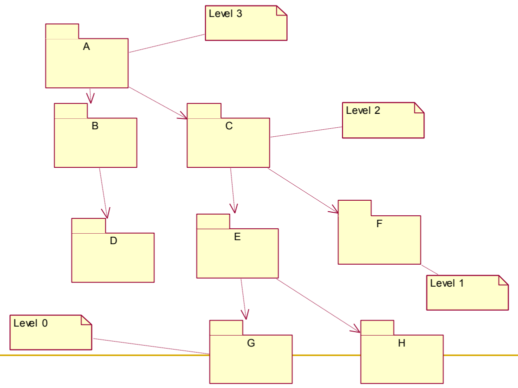
- B is directly dependent on A
  - if B is a superclass of A
  - if B holds (as a data member) a reference or pointer to A
  - If B refers to A in a parameter in one or more of its operations
    - In which cases it "uses" A
  - If B refers to A in the implementation of any of its methods
- B is indirectly, or transitively dependent on A
  - If any other class that B is dependent on is dependent on A

l

Levelisation

- Levelisation is desirable
  - Assuming a "bottom" level 0 each component in a dependency hierarchy can be assigned a unique level number
  - Implies a structure which is a Directed Acyclic Graph (DAG)
- In "good" OO System Structures
  - Abstract, stable (unchanging) classes tend to be at or near level 0
  - Concrete, volatile (often changing) classes tend to be at the highest levels
    - We want to minimise the number of classes dependent on changeable classes

A Levelised OO System



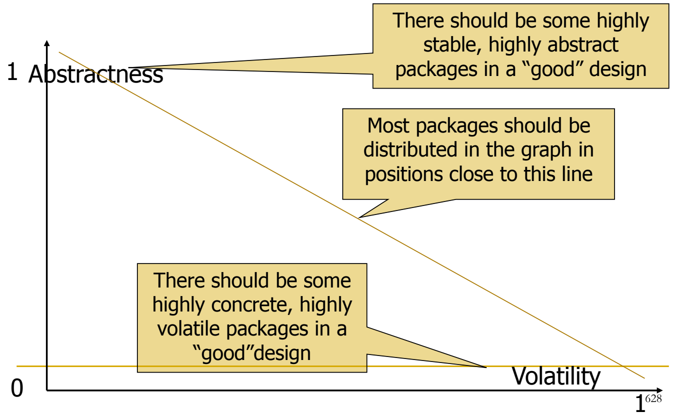
Analysing Structure in OO Systems (1)

- Simple measurements can be used to measure the structural quality of OO systems
- Abstractness
  - the total number of abstract classes (and interface types in Java) divided by the total number of classes and types
- Stability/Volatility
  - The number of efferent couplings (Ce) divided by the number of efferent couplings plus the number of afferent couplings (Ce +Ca)
    - Ce is the number of classes inside a package that directly depend on classes outside the package
    - Ca is the number of classes outside the package that depend on classes inside the package

Analysing Structure in OO Systems (2)

- Relational Cohesion
  - Relational cohesion (H) is the total number of Relationships (R) plus 1, divided by the total number of internal relationships (N)
    - H=R+1/N
- All these measures can be derived from plotting the relationship between classes on a spreadsheet
- The values for Abstractness and Stability can be plotted on a Stability Graph

Stability Graph



Summary

- Objects are logical structures to which responsibilities are allocated in design
- Objects can therefore be thought of architectural spaces
  - As can Classes, Components and Packages
- We can apply the lessons of "Neighbourhood Boundary"
  - Use encapsulation, Interfaces to minimise dependencies
- We can measure the quality of a structure with simple dependency metrics

Topic 14: Software Architecture Models

Also Software Architecture

- Architectural Design
  - process for identifying the subsystems that make up a system
  - defines framework for sub-system control and communication
- Software Architecture
  - description of the system output by architectural design

Architectural Design Process

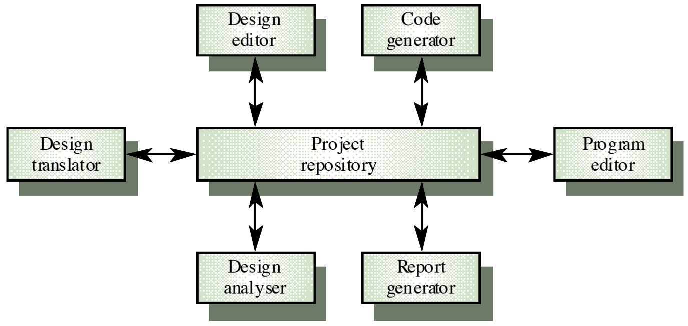
- System structuring
  - system decomposed into several subsystems
  - subsystem communication is established
- Control modeling
  - model of control relationships among system components is established
- Modular decomposition
  - identified subsystems decomposed into modules

Architectural Models

- Static structural model
  - shows major system components
- Dynamic process model
  - shows process structure of the system
- Interface model
  - defines subsystem interfaces
- Relationships model
  - data flow or control flow diagrams

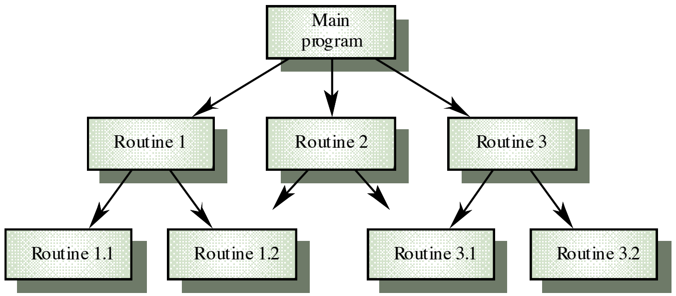
633

CASE Repository Model



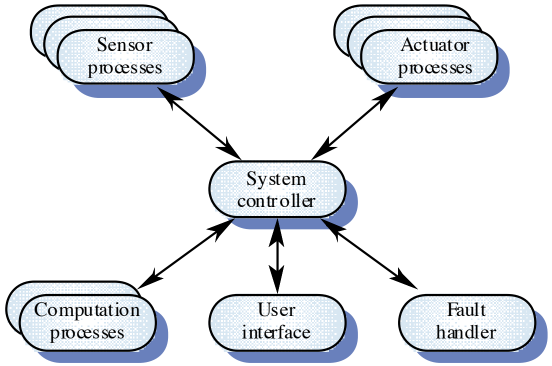
634

Call-Return Model



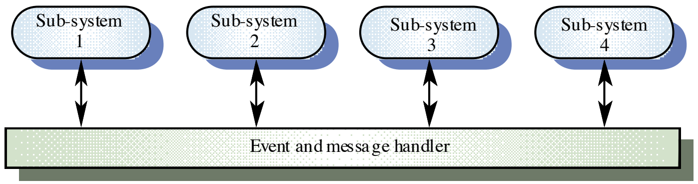
635

Real-Time System Control Model

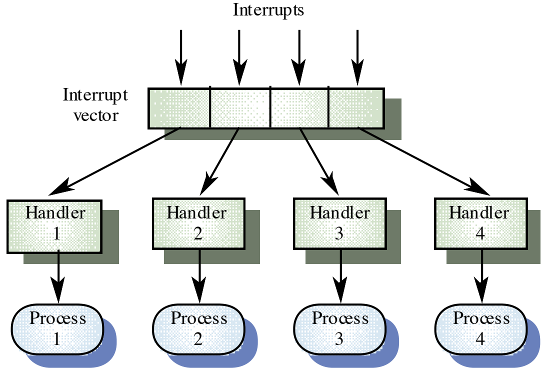


636

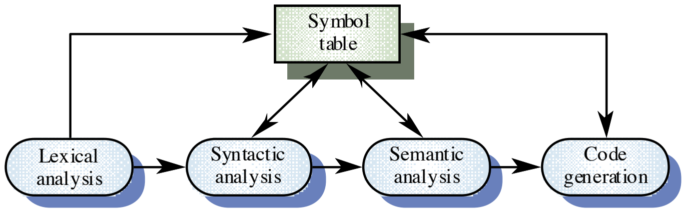
Selective Broadcasting Model



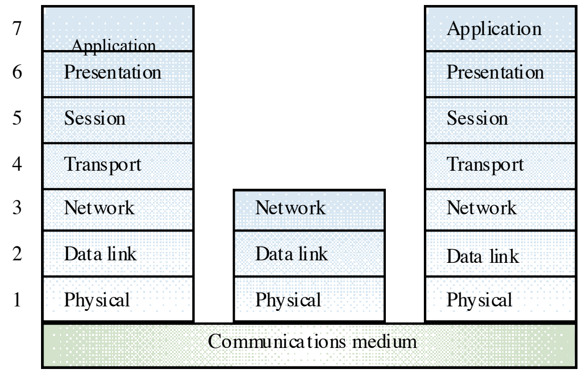
Interrupt-Driven Control Model



Compiler Model



OSI Reference Model



Distributed Systems

- Most large computer systems are implemented as distributed systems
- Information is also distributed over several computers rather than being confined to a single machine
- Distributed software engineering has become very important

Distributed Systems Architectures

- Client/Server
  - offer distributed services which may be called by clients
  - servers providing services are treated differently than clients using the services
- Distributed Object
  - no distinctions made between clients and servers
  - any system object may provide and use services from any other system object

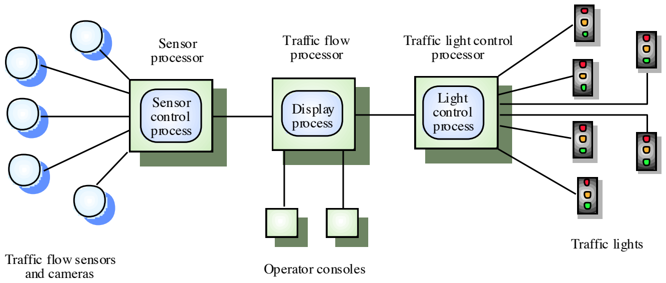
Middleware

- Software that manages and supports the different components of a distributes system
- Sits in the middle of the system to broker service requests among components
- Usually off-the-shelf products rather than custom
- Representative architectures
  - CORBA (ORB)
  - COM (Microsoft)
  - JavaBeans (Sun)

Multiprocessor Architecture

- Simplest distributed system model
- System composed of multiple processes that may execute on different processors
- Model used in many large real-time systems
- Distribution of processes to processors may be preordered or may be under control of a dispatcher

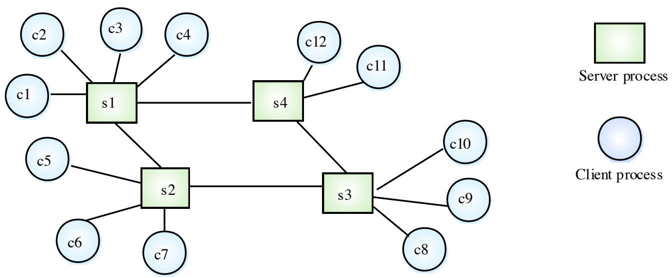
Multiprocessor Traffic Control System



Client/Server Architectures

- Application is modeled as a set of services that are provided by servers and a set of clients that use these services
- Clients know the servers but the servers do not need to know all the clients
- Clients and servers are logical processes (not always physical machines)
- The mapping of processes to processors is not always 1:1

Client/Server System



Representative Client/Server Systems Part 1

- File servers
  - client requests selected records from a file
  - server transmits records to client over the network
- Database servers
  - client sends SQL requests to server
  - server processes the request
  - server returns the results to the client over the network

Representative Client/Server Systems part 2

- Transaction servers
  - client sends requests that invokes remote procedures on the server side
  - server executes procedures invoked and returns the results to the client
- Groupware servers
  - server provides set of applications that enable communication among clients using text, images, bulletin boards, video, etc.

Client/Server Software Components

- User interaction/presentation subsystem
- Application subsystem
  - implements requirements defined by the application within the context of the operating environment
  - components may reside on either client or server side
- Database management subsystem
- Middleware
  - all software components that exist on both the client and the server to allow exchange of information

Representative Client/Server Configurations - part 1

- Distributed presentation
  - database and application logic remain on the server
  - client software reformats server data into GUI format
- Remote presentation
  - similar to distributed presentation
  - primary database and application logic remain on the server
  - data sent by the server is used by the client to prepare the user presentation

Representative Client/Server Configurations - part 2

- Distributed logic
  - client is assigned all user presentation tasks associated with data entry and formulating server queries
  - server is assigned data management tasks and updates information based on user actions
- Remote data management
  - applications on server side create new data sources
  - applications on client side process the new data returned by the server

Representative Client/Server Configurations - part 3

- Distributed databases
  - data is spread across multiple clients and servers
  - requires clients to support data management as well as application and GUI components
- Fat server
  - most software functions for C/S system are allocated to the server
- Thin clients
  - network computer approach relegating all application processing to a fat server

653

Thin Client Model

- Used when legacy systems are migrated to client server architectures
  - the legacy system may act as a server in its own right
  - the GUI may be implemented on a client
- It chief disadvantage is that it places a heavy processing load on both the server and the network

654

Fat Client Model

- More processing is delegated to the client as the application processing is locally extended
- Suitable for new client/server systems when the client system capabilities are known in advance
- More complex than thin client model with respect to management issues
- New versions of each application need to installed on every client

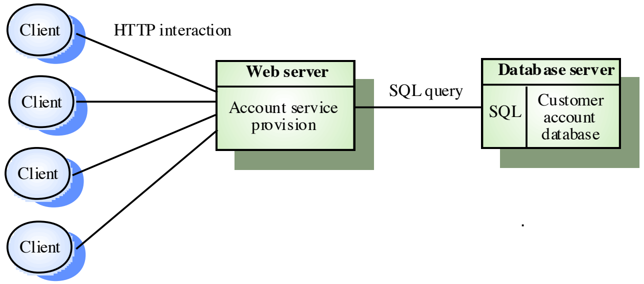
655

Three-tier Architecture

- Each application architecture layers (presentation, application, database) may run on separate processors
- Allows for better performance than a thin-client approach
- Simpler to manage than fat client approach
- Highly scalable (as demands increase add more servers)

656

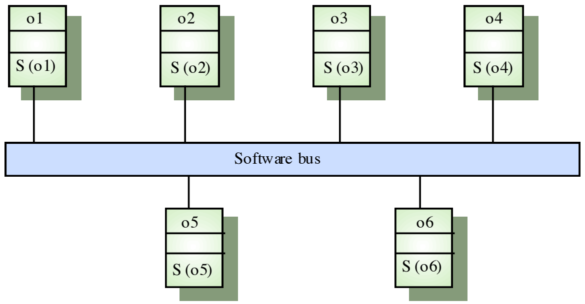
Three-Tier Architecture from Sommerville



Distributed Object Architectures

- No distinctions made between client objects and server objects
- Each distributable entity is an object that both provides and consumes services
- Object communication is though an object request broker (middleware or software bus)
- More complex to design than client/server systems

Distributed Object Architecture



Distributed Object Architecture Advantages

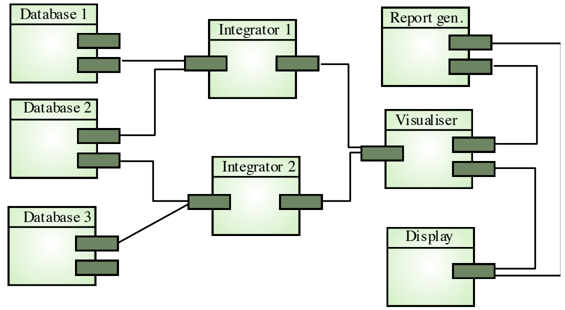
- Allows system designer to delay decisions on where and how services should be provided
- Very open architecture that allows new resources to be added as required
- System is flexible and scalable
- Dynamic reconfiguration is possible by allowing objects to migrate across the network as required

Uses of Distributed Object Architectures

- As a logical model that allows you to structure and organise the system
  - think about how to provide application functionality solely in terms of services and combinations of services
- As a flexible approach to the implementation of client/server systems
  - the logical model of the system is client/server with both clients and servers realised as distributed object communicating through a software bus

661

Data Mining System Example



662

CORBA

- International standard for an Object Request Broker (e.g. middleware) to manage distributed object communication
- Largely platform and language independent (by having versions for several OO environments)
- DCOM is Microsoft's alternative approach (but it is highly platform dependent)

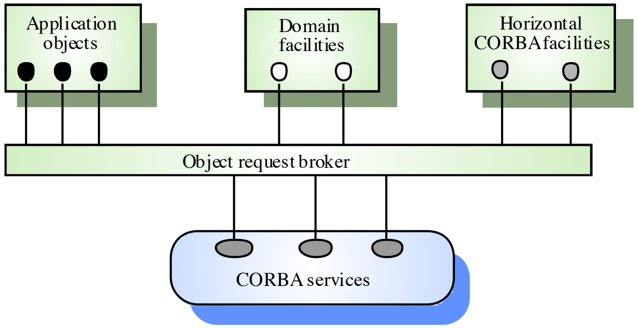
663

CORBA Services

- Naming and trading services
  - allow object to discover and refer to other objects on the network
- Notification services
  - allow objects to notify each other when events have occurred
- Transaction services
  - support atomic transactions and rollback on failure

664

CORBA Application Structure



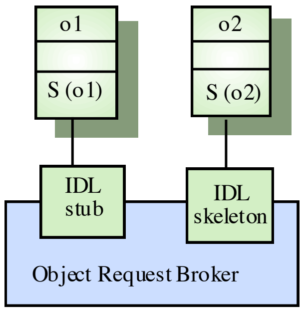
CORBA Standards

- Provides a model for application objects
  - COBRA objects encapsulate a state with a well-defined, language neutral interface
- A single object request broker that manages requests for object services
- A set of object services that may be of use to many distributed applications
- A set of common components built on top of these services

Object Request Broker (ORB)

- Handles object communications
- Knows about all system objects and their interfaces
- Using the ORB a client object binds an IDL (interface definition language) stub and defines the interface of the server object
- Calling the stub results in calls to the ORB which calls the required object through a published IDL skeleton that links the interface to the service implementation

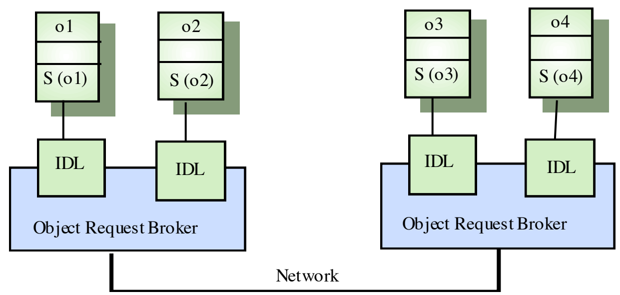
ORB-based Communication



Inter-ORB Communications

- ORB's are not usually separate programs
- ORB's are usually linked object libnraries
- ORB's handle communication between objects executing on the same machine
- Each computer in a distributed system may have its own ORB
- Inter-ORB communication is used to handle distributed object calls

Inter-ORB Communications



Topic 15: The OMG's Model Driven Architecture (MDA)

Why the MDA?

- The OMG specified the Object Management Architecture (OMA) in 1990
- Specified the CORBA standards
- Since 1997 has specified a number of non-CORBA standards
  - Unified Modeling Language (UML)
  - XML/XMI (XML Metadata Interchange)
  - Common Warehouse Model (CWM)
  - MetaObject Facility (MOF)
- Rapid expansion of new technologies raises issues of integration

Aims of the MDA

- MDA aims to
  - Support evolving standards in diverse application domains
  - Embrace new technologies and changes in existing technologies
  - Address the complete lifecycle of
    - Designing Applications
    - Deploying Applications
    - Integrating Applications
    - Managing Applications
  - ...while using open data standards

673

Basic Principles

- "MDA separates the fundamental logic from behind a specification from the detail of the particular middleware that implements it"
- The 'Architecture' assures:
  - Portability
  - Cross-platform interoperability
  - Platform independence
  - Domain specificity
  - Productivity

674

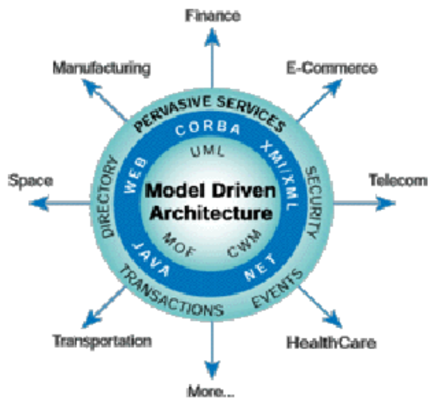
"Bottom-line Benefits"

- Benefits claimed include
  - Reduced cost throughout the application life-cycle
  - Reduced development time for applications
  - Improved application quality
  - Increased return on technology investments
  - Rapid inclusion of emerging new technologies

675

MDA

- Provides an abstract of the middleware environment with the bringing standardisa application design
- Leverages existing C standards
  - UML,MOF, CWM
- Platform-independer descriptions can be deployed in many technologies
  - CORBA, J2EE,.NE
- Includes already spe pervasive services
- Enables creation of s domain models



676

MDA Models

- MDA defines a model as
  - Something that "must be paired with a definition of a modelling language using semantics provided by the MOF"
- Three categories of models
  - Computationally Independent (CIM)
    - E.g., business type model (problem space)
  - Platform Independent (PIM)
    - E.g., system type model (specification space)
  - Platform Specific (PSM)
    - Target technology specific (solution space)

677

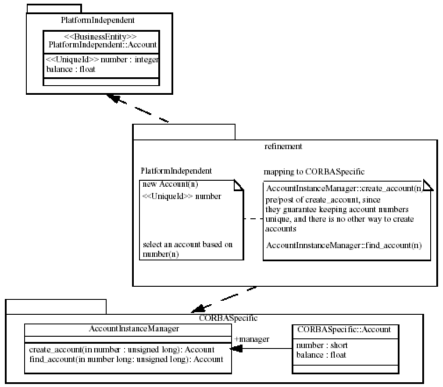
Model Relationships

- Models are related by
  - Abstraction
  - Refinement
  - Viewpoint
    - i.e., the mappings can be vertical (abstraction-refinement) or horizontal

678

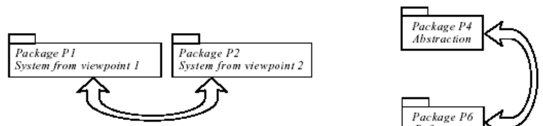
Example of Vertical Mapping

- The PIM version Account specifie object class Acco without saying ar about its implem
- A CORBA-specif implementation v conforms to this specification wou PSM *refinement abstraction*



679

Vertical/Horizontal Mapping



(a) Viewpoint Correspondence

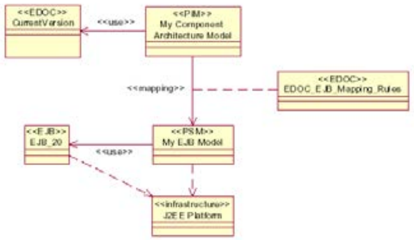
(b) Refinement

- Viewpoints are projections (i.e., they are not necessarily disjoint). Therefore the same abstractions may appear in different viewpoints, but perhaps with different properties. Because they are at the same level of abstraction this is referred to as horizontal mapping.

680

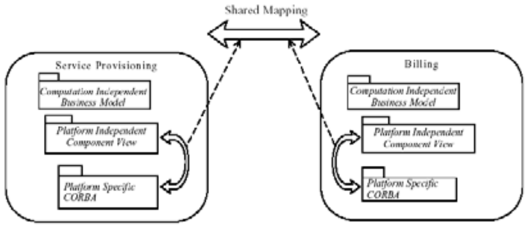
Using UML to Model Relationships

- The MDA proposes modelling the mapping rules themselves as "first class" citizens. The EDOC profile is a UML profile that has already been established. Others will follow



In practice, the MDA is realised as a *UML Profile*: a meta-model of the middleware environment

Automation of Mapping?



- An issue that arises – and is critical to the discussion on Software Architecture – is whether mappings between models can be abstracted and automated and reused in tools, or whether the various models are more important as reference points for creative design

Developing Using MDA (1)

- Development of the application in MDA starts with a "first-level" PIM
  - With an aspiration to "automatically generate all or most of the running application"
  - This base PIM expresses only business functionality and behaviour for client and server
    - Built by business and modelling experts
    - Appears, nevertheless, to be a system's view of business information (presumably a model of the business itself is expressed in a CIM)
- A second-level PIM adds some aspects of technology
  - E.g., activation patterns(*session*, *entity*), persistence, transactionality, security levelling etc.

Developing in MDA (2)

- MDA application tools will contain representations of the Domain Facilities and Pervasive Services
  - Any facility modelled in UML can be imported
  - The PIM will model links to these services
- The PIM is stored in the MOF and input to the mapping step for producing the PSM
  - UML Profiles give UML the ability to model both PIMs and PSMs
    - Profiles are standard extensions for a particular domain consisting of UML stereotypes and tagged values

Developing in MDA (3)

- The MDA definition document specifies 4 ways of moving from a PIM to a PSM
  1. Transformation by hand, working with each application on a separate, individual basis
  2. Transformations by hand using established patterns to convert from the PIM to a particular PSM
  3. The established patterns define an algorithm which is selected in the MDA tool, producing skeleton code which is finished by hand
  4. The tool, applying the algorithm, generates the entire PSM
- Generate the application from the PSM

685

Issues for Software Architecture (1)

- From the "Masterplan" view of Software Architecture
  - MDA extends the practical possibilities of "blueprinting"
  - MDA has the aim to fully automate software development
    - From UML PIMs to running application and back
      - NB it is accepted that *currently* any changes in code would have to be handcrafted into the UML first
  - Rigorous modelling of mapping rules provides scope for extending use of formal methods
  - Focus is on transformation methods 3 and 4 in the list provided in the MDA Definition Document

686

Issues for Software Architecture (2)

- From the Piecemeal Growth viewpoint:
  - The classification of models validates the significance of a Problem Space
  - The idea of standard mappings and patterns implies a base of shared knowledge
  - MDA claims to support iterative development
  - Focus is on transformation methods 1 and 2 in the list provided in the MDA Definition Document

687

Issues for Software Architecture (3)

- The MDA will provide a new framework in which the arguments of both camps can still be put forward
- The commercial interest in MDA tools will focus on abstraction/retrieval
  - i.e., vertical mapping
  - Fairly well understood notions of *retrieval*, existence of OCL etc., will boost this
  - Relatively easy to automate
- Research interest may focus on horizontal mapping
  - Developing PIMs from multiple, overlapping viewpoints
  - Possibly not automatable at all

688

Summary

- The OMG's MDA provides a new framework for the ongoing debate on Software Architecture
- Key elements are the 3 classes of models...
  - CIMs, PIMs, and PSMs
- ...and the mappings between them
  - Vertical (abstraction/refinement)
  - Horizontal (viewpoint)
- ... and existing OMG standards
  - UML, CWM, XML/XMI, MOF
  - Pervasive Services, Domain Facilities
- The PIM->PSM transformation methods are where the future focus of 'masterplan' v 'piecemeal growth' will lie

689

Topic 16: Software Architecture and Process

Architecture, Organisation and Process

- Architecture strongly influences Organisation and Process
  - E.g., Conway's Law says "organisation follows architecture, or architecture follows organisation"
- Waterfall Software Development Life Cycle
  - Implies structured methods
    - Top-down design, step-wise development
  - Together with an hierarchical organisational structure
    - Business Analyst->System Analyst->Project Leader -> Analyst/Programmer->Programmer
- Debate on architecture is also a debate on process

691

'Heavywight' Process :ATAM

- Carnegie Mellon University's SEI now promotes the Architecture Tradeoff Analysis Method (ATAM)
  - Successor to SAAM
  - Utilises ABAS (see "Architectural Styles" Topic)
- Purpose
  - "...is to assess the consequences of architectural decisions in the light of quality attribute requirements"
- Aim
  - To assess architectural specifications **before** resources are committed to development

692

Underlying Concepts of ATAM

- ATAM focuses on quality attribute requirements
  - What are the stimuli to which the architecture must respond?
  - What is the measurable and observable manifestation of a quality attribute by which its achievement is judged?
  - What are the key architectural decisions that impact achieving the attribute requirement?

The Steps of ATAM(1)

Presentation

1. Present the ATAM
   To assembled stakeholders
2. Present business drivers
   By Project Manager
3. Present architecture
   By Architect

Investigation and Analysis

4. Identify architectural approaches
5. Generate quality attribute utility trees
   See discussion on ABAS'
6. Analyse architectural approaches

The Steps of ATAM (2)

Testing

7. Brainstorm and prioritise scenarios
8. Analyse architectural approaches

Reporting

9. Present results

Other Heavyweight Processes

- Capability Maturity Model
- SSADM
- Prince
- MASCOT
- HOOD
  - General characteristic
    - Process maintained by Management
    - Imposed by QA staff etc.,

An 'agile' process: The SCRUM Approach

- The SCRUM Software Development Process
  - For small (10 members or less) development teams
    - Compare with Surgical Team
  - Utilises rugby-metaphor
    - In a scrum 8 players, each with specific roles, co-operate tightly together to gain forward movement while controlling the ball

How SCRUM Works

- Initial Planning Phase
  - Chief architect identified, architecture developed
  - SCRUM teams chosen
    - Can change architecture in discussion with Chief Architect
    - Each team headed by a Scrum Master
  - Functionality delivered in *Sprints*
    - Typically 1-4 weeks
    - Timeboxed development controlled by short, *daily* meetings
      - Deadlines are ALWAYS met, even if functionality dropped
    - All identified tasks captured in *Backlog*
  - Product development completed by a *Closure Phase*

Benefits of SCRUM

- SCRUM is comparable to other lightweight processes
  - Dynamic System Development Method, Xtreme Programming etc.,
- Focuses effort of the developers on backlog items
- Communicates priorities constantly to all developers
  - Changing them "on the fly" if necessary
- Addresses risk dynamically
  - In the construction process itself

SCRUM Meetings

- Each of the daily SCRUM meetings answers the following 3 questions:
  1. What have you completed, relative to the backlog, since the last Scrum meeting?
  2. What obstacles got in the way of completing your work?
  3. What specific things do you plan to accomplish, relative to the backlog, between now and the next Scrum meeting?

Other Agile Processes

- Xtreme Programming
  - Pair programming etc.,
- Dynamic Systems Development Method
- Pattern Languages
  - General characteristic
    - Designed and maintained by development staff

Architecture and the RUP

- The most well-known software development process is the Rational Unified Process (RUP)
  - Proprietory "process framework" of Rational Inc.
  - Likely to be the basis of the OMG's standardisation of process
- The RUP claims to be:
  - Use-case driven
  - Architecture-centric
  - Object-oriented

Multiviewed Software Architecture

- Rational's Phillipe Krutchen says that Software Architecture deals with issues of
  - Abstraction
  - Composition
  - Decomposition
  - Style
  - Aesthetics
- Proposes a multiview model of Software Architecture
  - '4+1' views
  - Underpins the RUP

The '4+1' Views Model

- The original views were:
  - Logical View
    - Object model of the system
  - Process View
    - Concurrency and synchronisation issues
  - Development View
    - Static organisation of the system in its development environment
  - Physical View (now called Component View or Implementation View in RUP)
    - Mapping of software to hardware
  - Scenario-based View (the 'plus one': Use Case View in the RUP)
    - Usage scenarios

Characteristics of the 4+1 Views Model

- Krutchen applies Perry and Wolf's equation to each model separately
  - Software Architecture = {elements, form, rationale}
- Each view captured in a blueprint
  - Appropriate notation
  - May include attached architectural style
    - As per Garlan and Wolf
- Scenarios (use cases) used to drive an iterative, incremental approach to architecture

Krutchen's Process

- Small number of scenarios selected for an iteration
  - Based on relative risk, criticality
- "Strawman" architecture established
  - cf. UP's "small, skinny system"
- Scenarios scripted to drive major abstractions
  - Classes, subsystems, collaborations, processes etc.,
- Architectural elements mapped on to four blueprints
- Architecture then tested, measured, analysed, adjusted
- Documentation for each view includes *Architectural Blueprint* and *Architectural Style Guide*

Evaluation of '4+1' Views Model and the RUP

- Extends the notion of architecture beyond mere structure
  - Includes rationale, aesthetics etc.,
- Places Software Architecture on the critical path
- BUT Software Architecture is discovered IN the project
  - CBD, Software Productline architectures, enterprise architectures etc., require conformity to pre-existing architectures
- Unclear whether RUP is 'heavyweight' or 'agile'
  - See O'Callaghan v Jacobson in *ApplicationDevelopment Advisor*

Summary

- Processes cannot be divorced from Software Architecture
- Processes can be categorised as
  - Heavyweight
  - Agile (or 'lightweight')
- The RUP is based on Krutchen's 4+1 Views Model of Software Architecture
- There is a debate as to whether it is heavyweight or agile

Topic 17: Software Architecture and Reengineering

Legacy Systems

Definition:

- *Any information system that significantly resists evolution*
- *to meet new and changing business requirements*

Characteristics

- Large
- Geriatric
- Outdated languages
- Outdated databases
- Isolated

Software Volume

- Capers Jones software size estimate:
  - 700,000,000,000 lines of code
  - (7 * 10^9 *function points*)
  - (1 fp ~ 110 *lines of code*)
- Total number of programmers:
  - 10,000,000
  - 40% new dev. 45% *enhancements*, 15% *repair*
  - (2020: 30%, 55%, 15%)

Reverse Architecting: Motivation

- Architecture description lost or outdated
- Obtain advantages of expl. arch.:
  - Stakeholder communication
  - Explicit design decisions
  - Transferable abstraction
- Architecture conformance checking
- Quality attribute analysis

Software Architecture

Structure(s) of a system which

- *comprise the software components*
- *the externally visible properties of those systems*
- *and the relationships among them*

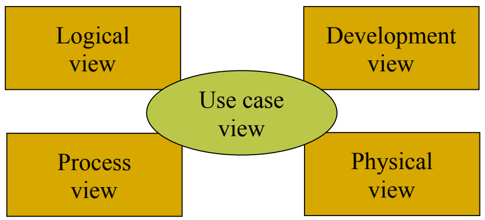
713

Architectural Structures

- Module structure
- Data model structure
- Process structure
- Call structure
- Type structure
- GUI flow
- ...

714

The 4 + 1 View Model



Extract & compare!

715

Reverse Engineering

- The process of analysing a subject system with two goals in mind:
  - to identify the system's components and their interrelationships; and,
  - to create representations of the system in another form or at a higher level of abstraction.

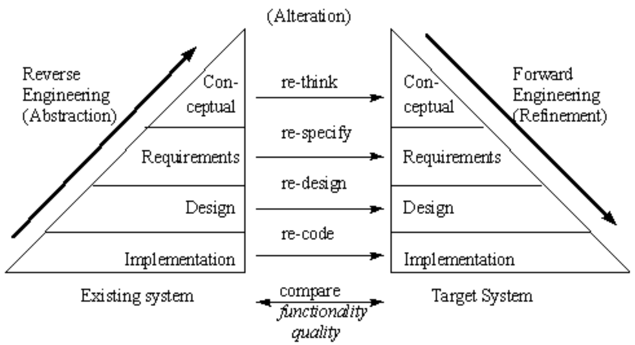
716

Reengineering

- The examination and alteration of a subject system
- to reconstitute it in a new form
- and the subsequent implementation of that new form

Beyond analysis -- actually improve.

Reengineering



Program Understanding

- the task of building mental models of an underlying software system
- at various abstraction levels, ranging from
  - models of the code itself to
  - ones of the underlying application domain,
- for software maintenance, evolution, and reengineering purposes

Cognitive Processes

- Building a mental model
- Top down / bottom up / opportunistic
- Generate and validate hypotheses
- Chunking: create higher structures from chunks of low-level information
- Cross referencing: understand relationships

Supporting Program Understanding

- Architects build up *mental models*:
  - various *abstractions* of software system
  - *hierarchies* for varying levels of detail
  - *graph-like structures* for dependencies
- How can we support this process?
  - infer number of *predefined abstractions*
  - *enrich* system's source code with abstractions
  - let architect *explore* result

Topic 18: Service-Oriented Architecture (SOA)

contents

- *Software Architecture and SOA*
- *Service-oriented architecture (SOA) definition*
- *Service-oriented modeling framework (SOMF)*
- *Security in SOA*
- *The Cloud and SOA*

Software Architecture and SOA

- Service-oriented architecture is a special kind of software architecture that has several unique characteristics.

Service-oriented architecture (SOA) definition

A service-oriented architecture is essentially a collection of services. These services communicate with each other. The communication can involve either simple data passing or it could involve two or more services coordinating some activity. Some means of connecting services to each other is needed.

725

Services ,Web Services and SOA

- *Web Services* refers to the technologies that allow for making connections.
- ***Services*** are what you connect together using Web Services. A service is the endpoint of a connection. Also, a service has some type of underlying computer system that supports the connection offered.
- The combination of services - internal and external to an organization - make up a *service-oriented architecture*.

726

SOA Characteristics

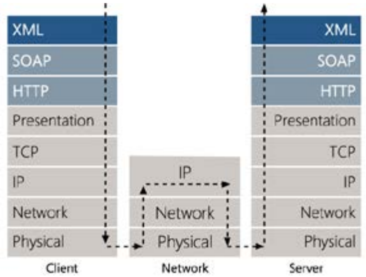
- *discoverable and dynamically bound.*
- *Self-contained and modular.*
- *interoperability.*
- *loosely coupled.*
- *network-addressable interface.*
- *coarse-grained interfaces.*
- *location-transparent.*
- *composable.*
- *self-healing.*

727

SOA Basics

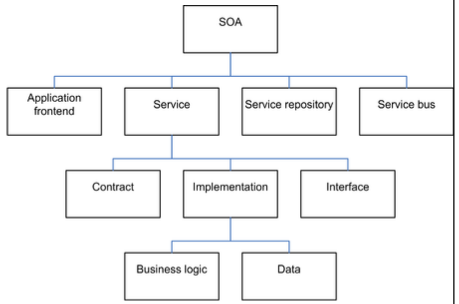
- In most cases, an SOA will be implemented using a variety of standards, the most common being:
  - HTTP
  - WSDL (Web Services Definition Language)
  - SOAP (Simple Object Access Protocol),
  - XML (eXtensible Markup Language)
- These latter three standards work together to deliver messages between services much in the same way as the post office.

728

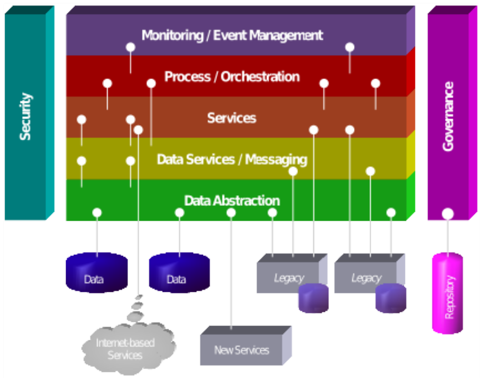


- WSDL Entry in an address book
- HTTP Postal carrier (transportation)
- SOAP Envelope (encapsulation)
- XML Letter (message)

SOA Elements



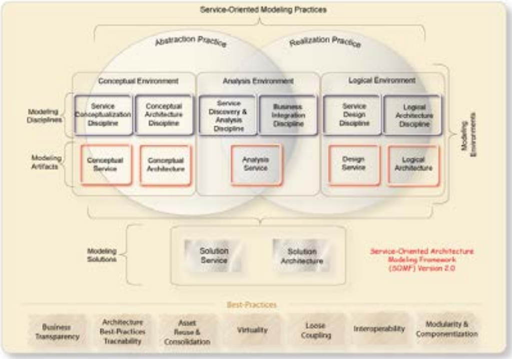
SOA meta-model



Service-oriented modeling framework (SOMF)

- SOMF offers a modeling language and a work structure or "map" depicting the various components that contribute to a successful service-oriented modeling approach.
- The model enables practitioners to craft a project plan and to identify the milestones of a service-oriented initiative.
- SOMF also provides a common modeling notation to address alignment between business and IT organizations.

Service-Oriented Modeling Framework (SOMF) Version 2.0



SOMF addresses the following principles:

- business traceability
- architectural best-practices traceability
- technological traceability
- SOA value proposition
- software assets reuse
- SOA integration strategies
- technological abstraction and generalization
- architectural components abstraction

Security in SOA

"The revolutionary idea that defines the boundary between modern times and the past is the mastery of risk: the notion that the future is more than a whim of the Gods and that men and women are not passive before nature. Until human beings discovered a way across that boundary, the future was a mirror of the past or the murky domain of oracles and soothsayers who held monopoly over knowledge of anticipated events..."

- Peter Bernstein, "Against the Gods"

Security Services

	Object-Orientation	Component-Based Design	Service-Orientation with Web Services
Paradigm	abstract models (objects) used to bundle data and methods	technology-specific implementation model for distributed programming	services designed as autonomous and standardized programs with an emphasis on reuse
Security Implications	vendor or implementation provides security mechanisms, authentication, authorization, audit, for object processing runtime (example: Java authentication and authorization services in JDK)	component model provides implementation specific security models (example: container security in EJB)	interoperable industry security standards deal with message security, identity federation, and service security (example: WS-Security)

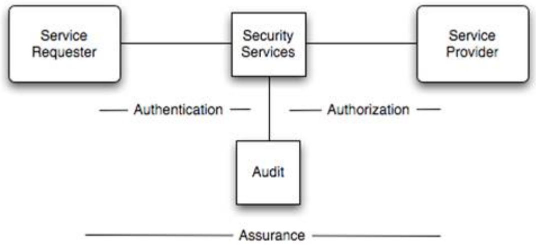
Different distributed programming paradigms introduce different security considerations

Security Services

- The primary security functions required by most systems are:
  - *authentication*
  - *authorization*
  - *auditing*
  - *assurance*

737

Security Services



How security services can be positioned as intermediaries between service requester and provider

738

Security-centric Service Models(1)

- **Authentication** is concerned with validating the authenticity of the request and binding the results to a principle. This is frequently a system-level service because it deals with the processing of system policies (such as password policies) and implementing complex protocols (like Kerberos). This warrants a separate service because authentication logic is generally not valuable (or reusable) when intertwined with other application logic.

739

Security-centric Service Models(2)

- **Authorization,** on some level, is always enforced locally, close to the thing being protected. In SOA, this thing is the service provider. While coarse-grained authorization can be implemented at a global level, finer grained authorization requires mapping to the service and its operations. From a design perspective, authorization should be viewed at both system and service levels (the latter always being enforced locally).

740

Security-centric Service Models(3)

- **Audit services** provide detection and response features that serve to answers questions around what digital subjects performed what actions to what objects.
- **Assurance services** essentially exists as a set of system processes that increase the assessor's confidence in a given system.

741

two major problems:

- ***encrypting and decrypting*** (it's nice and all, but we still need to move that data around and use it)
- ***access control***

742

Problem 1: *encrypting and decrypting*

- SOA movement has produced useful standards like WS-Security that help solve the first problem (moving data around).
- WS-Security SOAP headers facilitate encrypting data in the message and because the data is packaged in XML, other service providers can decrypt the message.
- Additionally, WS-Security allows for multiple security token types, so if your enterprise is using Active Directory, LDAP, and digital certificates, you can still mesh security requests together in a consistent manner.

743

Problem 2: *access control*

- Access control is comprised of：
  - ***authentication*** (who made this request?)
  - ***authorization*** (what is this request allowed to do?)
  - ***auditing*** (what security decisions were made for what requests?).

744

SOA security architects : how to do

- Map out a security architecture that looks at the system from an end to end perspective, and focuses on your assets (it's the car, not the garage).
- For each service requester and service provider - and anything in the middle like proxies - understand the current state of access control by analyzing authentication, authorization, and auditing (secure access to the car).
- Determine what policy enforcement and policy decision points exist today and which can be strengthened in the future (fortify the car to the best of your ability).

745

SOA security : Conclusion

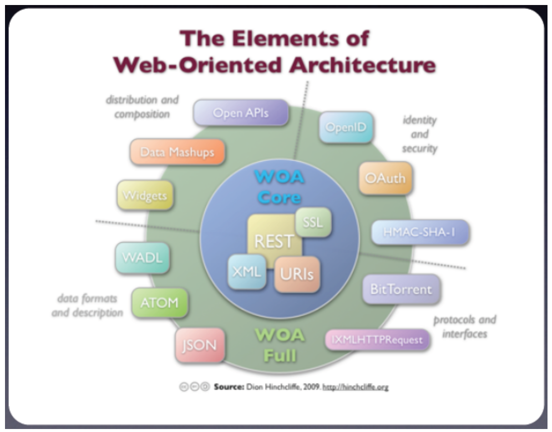
- There is no perfect security solution, there is only the management of security risk that relies on judgment and prioritization, driven by assets and values.
- Security is contextual and has a form factor that must adhere to that which the supporting mechanisms can protect.
- Risk is increasingly engendered in data and effective security mechanisms adhere to data to provide the necessary level of protection.
- When SOA security standards are properly leveraged, the potential is there to create entirely new and robust service-oriented security architectures.

746

Web Oriented Architecture

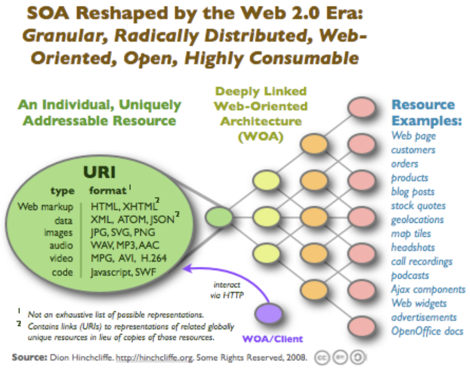
- **Web Oriented Architecture (WOA)** is a style of software architecture that extends service-oriented architecture (SOA) to web based applications, and is sometimes considered to be a light-weight version of SOA.
- **WOA** is also aimed at maximizing the browser and server interactions by use of technologies such as REST and POX.

747



748

WOA: An organic service fabric



What is WOA? The Basic Tenets(1)

- Information in a WOA is represented in the form of *resources* on the network and are accessed and manipulated via the protocol specified in the URI, typically HTTP.
- Every resource on the network can located via a globally unique address known as a Universal Resource Identifier or URI complying with RFC 3986.
- Resources are manipulated by HTTP verbs (GET, PUT, POST, DELETE) using a technique known as Representational State Transfer or REST.
- Manipulation of network resources is performed solely by *components* on the network (essentially browsers and other Web servers).

What is WOA? The Basic Tenets(2)

- Access to resources must be layered and not require more than local knowledge of the network.
- It is the responsibility of the components to understand the representations and valid state transitions of the resources they manipulate.
- The service contract of WOA resources is implicit; it's the representation that is received.
- WOA resources contain embedded URIs that build a larger network of granular representative state (i.e. order resources contain URLs to inventory resources).
- WOA embodies Thomas Erl's essential Principles of SOA, though in often unexpected ways (such as having a contract, albeit implicit).

Topic 19: Security and Trust for Software Architecture

contents

- *What is Security*
- *Design Principles for Computer Security*
- *Security Architecture Blueprint*
- *Security Architecture Lifecycle*
- *Architectural Access Control Models*
- *Architecture and Trust Management*

753

Security

"The protection afforded to an automated information system in order to attain the applicable objectives of preserving the **integrity**, **availability** and **confidentiality** of information system resources (includes hardware, software, firmware, information/ data, and telecommunications)."

——National Institute of Standards and Technology

754

Confidentiality, Integrity, and Availability

- Confidentiality
  - Preserving the **confidentiality** of information means preventing unauthorized parties from accessing the information or perhaps even being aware of the existence of the information. I.e., secrecy.
- Integrity
  - Maintaining the **integrity** of information means that only authorized parties can manipulate the information and do so only in authorized ways.
- Availability
  - Resources are **available** if they are accessible by authorized parties on all appropriate occasions.

755

Design Principles for Computer Security

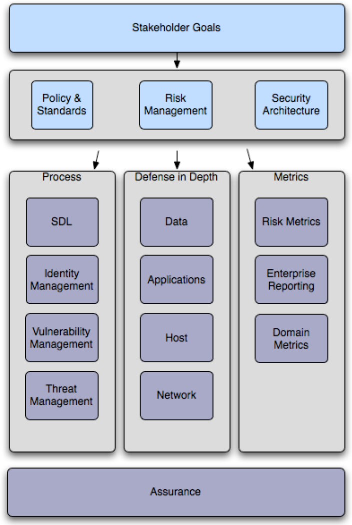
- **Least Privilege**: give each component only the privileges it requires
- **Fail-safe Defaults**: deny access if explicit permission is absent
- **Economy of Mechanism**: adopt simple security mechanisms
- **Complete Mediation**: ensure every access is permitted
- **Design**: do not rely on secrecy for security

756

Design Principles for Computer Security

- **Separation of Privilege**: introduce multiple parties to avoid exploitation of privileges
- **Least Common Mechanism**: limit critical resource sharing to only a few mechanisms
- **Psychological Acceptability**: make security mechanisms usable
- **Defense in Depth**: have multiple layers of countermeasures

Security Architecture Blueprint



Stakeholders:

- Anyone with a material stake in the systems development and operations, including business users, customers, legal team, and so on.
- The stakeholder's business and risk goals drive the overall security architecture.

Risk Management:

- Risk is comprised of assets, threats, vulnerabilities, and countermeasures.
- The risk management process implements risk assessment to ensure the enterprise's risk exposure is in line with risk tolerance goals.

Security policy and standards:

- organizational policies and standards that govern the system's design, deployment, and run time.
- The security policy describes both what is allowed as well as not allowed in the system.
- Security standards should be prescriptive guidance for people building and operating systems, and should be backed by reusable services wherever practical.

761

Security architecture:

- unifying framework and reusable services that implement policy, standards, and risk management decisions.
- The security architecture is a strategic framework that allows the development and operations staff to align efforts, in addition the security architecture can drive platform improvements which are not possible to make at a project level.

762

Security processes:

- Security functions as a collaborative design partner in the software development lifecycle (SDL), from requirements, architecture, design, coding, deployment, and withdrawal from service.
- Security adds value to the software development lifecycle through prescriptive and proscriptive guidance and expertise in building secure software.
- Security can play a role in all phases of the SDL, each additional security process improvement must fit with the overall SDL approach in the enterprise, which vary widely.

763

Security processes :
Example roadmap for adding security to the SDL

SDL Roadmap

	This Quarter	Next Quarter	12 Month Goal
WHAT?	• Align SDL with Threat Management process goals	• Align SDL with vulnerability management process goals	• Align SDL with identity management process goals
HOW?	• Introduce threat modeling into SDL design phase • Train development staff on threat modeling • Develop threat-centric attack patterns	• Integrate static analysis tooling for automated detection of some security bugs • Evaluate static analysis tools • Define phased implementation	• Develop identity abstraction layer to access identity attributes through reusable components
Who?	• Security staff bootstrap initial threat modeling process and training • Long term process owner - development staff	• Security staff conduct evaluation • Software development staff owns implementation and operations	• Software architects and security collaborate to design and develop identity abstraction layer • Software development builds test cases and implementation

764

Security processes :
Example roadmap for adding security to the SDL(1)

- The diagram shows an example approach for iterating through a number of security artifacts and evolving the SDL over time
- The goal is to identify reusable services that, over time, can speed development of reliable software
- for example: building reusable attack patterns that are implemented across a particular set of threats like a set of web attack patterns that can be used for security design in any enterprise web application

765

Security processes :
Example roadmap for adding security to the SDL(2)

- **Identity management** deals with the creation, communication, recognition, and usage of identity in the enterprise.
- **Threat management**: deals with the threats to systems such as virus, Trojans, worms, malicious hackers, force majeure, and intentional and unintentional system misuse by insiders or outsiders.
- **Vulnerability management:** the set of processes and technologies for discovering, reporting, and mitigating known vulnerabilities.

766

Defense in depth:

- Defense in depth is predicated on the notion that every security control is vulnerable somehow, but that if one component fails another control at a separate layer still provides security services to mitigate the damage
- Each level of the defense in depth stack has its own unique security capabilities and constraints. The core security services - authentication, authorization, and auditing apply at all levels of the defense in depth stack

767

Defense in depth （1）

- **Network security:** design and operations for security mechanisms for the network.
- **Host security:** is concerned with access control on the servers and workstations.
- **Application security:** deals with two main concerns: 1) protecting the code and services running on the system; 2) delivering reusable application security services.
- **Data security:** deals with securing access to data and its use, this is a primary concern for the security architecture and works in concert with other domains.

768

Metrics:

- Security metrics are a basis for assessing the security posture and trends of the systems.
- The goal of security metrics is objective measurement that enables decision support regarding risk management for the business without requiring the business to be information security experts to make informed choices.
- Audit, assurance services, and risk assessment use security metrics for ongoing objective analysis.

769

Metrics（1）

- **Risk metrics**: measure the overall assets, and their attendant countermeasures, threats, and vulnerabilities.
- **Enterprise reporting**: enterprise view of security and risk. Enterprise reports show the states and rates of security, they can show which areas deserve additional focus and where the security services are increasing or decreasing the overall risk exposure.
- **Domain specific metrics**: domain specific instrumentation of metrics, for example vulnerabilities not remediated, provide granular view of security in a system.

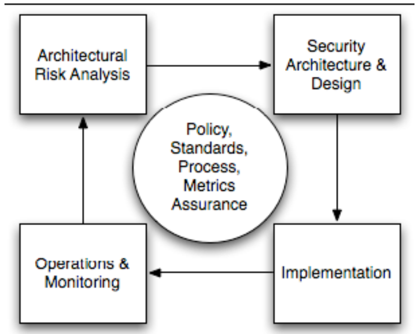
770

Assurance:

- Assurance is the set of activities that create higher confidence in the system’s ability tocarry out its design goals even in the face of malicious abuse.
- These activities are performed by, or on behalf of, an enterprise as tests of the security practices. Activities include penetration testing, code auditing and analysis, and security specific hardware and software controls.
- The security processes, defense in depth technologies, and metrics are all built on sets of assumptions; assurance activities challenge these assumptions, and *especially the implementations*.

771

Security Architecture Lifecycle



772

Security Architecture Lifecycle（1）

- **Architecture Risk Assessment**: assesses the business impact to critical business assets, the probability and impact of security threats and vulnerabilities.
- **Security Architecture and Design**: architecture and design of security services that enable business risk exposure targets to be met.
- **Implementation**: security processes and services implemented, operational, and managed.
- **Operations and Monitoring**: Ongoing processes, such as vulnerability management and threat management, that monitor and manage the operational state as well as the breadth and depth of systems security.

773

Architectural Access Control Models

- Decide whether access to a protected resource should be granted or denied
- Discretionary access control
  - Based on the identity of the requestor, the resource, and whether the requestor has permission to access
- Mandatory access control
  - Policy based

774

Role of Trust Management

- Each entity (peer) must protect itself against these threats
- Trust Management can serve as a potential countermeasure
  - Trust relationships between peers help establish confidence
- Two types of decentralized trust management systems
  - Credential and policy-based
  - Reputation-based

775

Architecture and Trust Management

- Decentralized trust management has received a lot of attention from researchers [Grandison and Sloman, 2000]
  - Primary focus has been on developing new models
- **But how does one build a trust-enabled decentralized application?**
  - **How do I pick a trust model for a given application?**
  - **And, how do I incorporate the trust model within each entity?**

776

Approach

- Select a suitable reputation-based trust model for a given application
- Describe this trust model precisely
- Incorporate the model within the structure (architecture) of an entity
  - Software architectural style for trust management (PACE)
- Result – entity architecture consisting of
  - components that encapsulate the trust model
  - additional trust technologies to counter threats

Key Insights

- **Trust**
  - Cannot be isolated to one component
  - Is a dominant concern in decentralized applications and should be considered early on during application development
  - Having an explicit architecture is one way to consistently address the cross-cutting concern of trust
- **Architectural styles**
  - Provide a foundation to reason about specific goals
  - Facilitate reuse of design knowledge
  - Allow known benefits to be leveraged and induce desirable properties

Design Guidelines: Approach

- Identify threats of decentralization
- Use the threats to identify guiding principles that help defend against the threats
- Incorporate these principles within an architectural style focused on decentralized trust management

Topic 20 :Web 2.0 and Software Architecture

contents

What is Web 2.0?

"Design Patterns and Business Models for the Next Generation of Software"

- Tim O'Reilly, 2005

What is Web 2.0?

"The central principle behind the success of the giants born in the Web 1.0 era who have survived to lead the Web 2.0 era appears to be this, that they have embraced the power of the web to harness collective intelligence"

- Tim O'Reilly, 2006

What is Web 2.0?

"The Semantic Web is not a separate Web but an extension of the current one, in which information is given well-defined meaning, better enabling computers and people to work in cooperation"

- Tim Berners-Lee, 2001

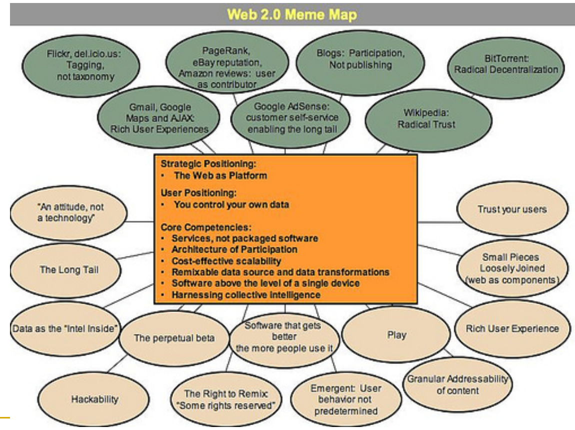
What is Web 2.0?

Both are Ecosystems

Semantic Web: interaction between machines

Social Web: conversations between people

The Web As Platform



History: From Web 1.0 to 2.0

- The term "Web 2.0" was coined in 1999 by Darcy DiNucci. In her article, "Fragmented Future"
- The term did not resurface until 2003
- In 2004, the term began its rise in popularity when O'Reilly Media and MediaLive hosted the first Web 2.0 conference
- O'Reilly's Web 2.0 conferences have been held every year since 2004, attracting entrepreneurs, large companies, and technology reporters
- Since that time, Web 2.0 has found a place in the lexicon; the Global Language Monitor recently declared it to be the one-millionth English word

Web 1.0 and Web 2.0 : What's the difference

"Web 1.0 was about connecting computers and making technology more efficient for computers. Web 2.0 is about connecting people and making technology efficient for people."

--Dan Zambonini

What's the difference?

- Web 1.0
  - HTML Web pages you read like a book. Static web pages, use of search engines, and surfing.
  - Web applications, information served to users, user interaction with online information
- Web 2.0
  - the Web was actually the platform that allowed people to get things done.
  - Internet based services such as social networking, communication tools. Sites that generally encourage collaboration and information sharing among users.

789

Web 1.0 vs. Web 2.0

Web 1.0	Web 2.0
DoubleClick	Google AdSense
Ofoto	Flickr
Akamai	BitTorrent
mp3.com	Napster
Britannica Online	Wikipedia
personal websites	blogging
evite	upcoming.org and EVDB
domain name speculation	search engine optimization
page views	cost per click
screen scraping	web services
publishing	participation
content managment systems	wikis
directories (taxonomy)	tagging ("folksonomy")
stickiness	syndication

790

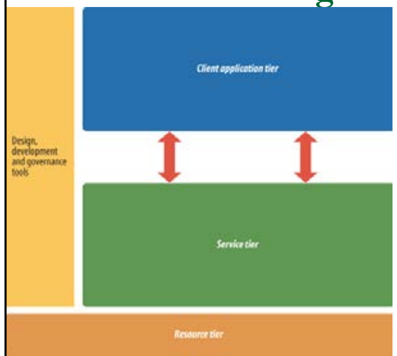
Web 1.0 and Web 2.0
Netscape & Google

- Netscape framed "the web as platform" in terms of the old software paradigm: their flagship product was the web browser, a desktop application, and their strategy was to use their dominance in the browser market to establish a market for high-priced server products. In short, Netscape focused on creating software, updating it on occasion, and distributing it to the end users.

- Google, a company which did not at the time focus on producing software, such as a browser, but instead focused on providing a service based on data. The data being the links Web page authors make between sites. Google exploits this user-generated content to offer Web search based on reputation through its "page rank" algorithm. Unlike software, which undergoes scheduled releases, such services are constantly updated, a process called "the perpetual beta".

791

Basic Web 2.0 Reference Architecture diagram



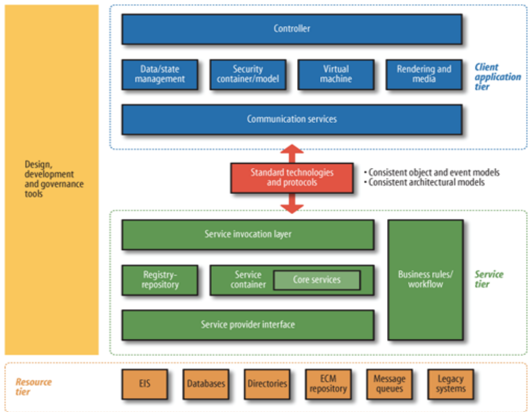
792

Basic Web 2.0 Reference Architecture components（1）

- **Resource tier**
  - capabilities or backend systems that can support services that will be consumed over the Internet
  - data or processing needed for creating a rich user experience
  - typically includes files; databases; enterprise resource planning (ERP) and customer relationship management (CRM) systems; directories; and other common applications
- **Service tier**
  - connects to the resource tier and packages as a service, giving the service provider control over what goes in and out
  - Within enterprises, the classic examples of this functionality are J2EE application servers deploying SOAP or EJB endpoints

Basic Web 2.0 Reference Architecture components（2）

- **Connectivity**
  - means of reaching a service
  - must be visible to and reachable by the service consumer
  - Connectivity is largely handled using standards and protocols such as XML over HTTP
- **Client tier**
  - helps users to consume services and displays graphical views of service calls to users
  - Examples of client-side implementations include web browsers, Adobe Flash Player, Microsoft Silverlight, Acrobat, iTunes



Detailed reference architecture for Web 2.0 application architects and developers

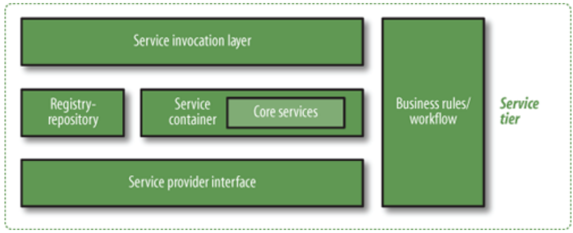
Basic Web 2.0 Reference Architecture ——The Resource Tier

Detail view of the resource tier

Basic Web 2.0 Reference Architecture
——The Resource Tier （1）

- **EIS：** Enterprise Information System (EIS) is an abstract moniker for a component common in most IT systems.
- **Databases：** Databases are typically used to persist data in a centralized repository designed in compliance with a relational model.
- **Directories：** Directories are lookup mechanisms that persist and maintain records containing information about users.
- **ECM repository：** Enterprise content management (ECM) repositories are specialized types of EIS and database systems.
- **Message queues：** Message queues are ordered lists of messages for inter-component communications within many enterprises.
- **Legacy systems：** The last component is a catchall generally used to denote anything that has existed through one or more IT revolutions.

Basic Web 2.0 Reference Architecture
——The Service Tier

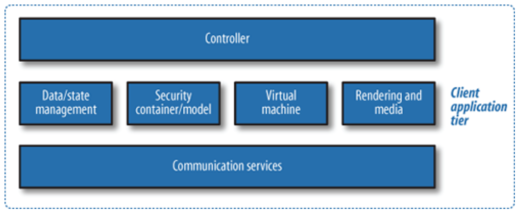


Detail view of the service tier

Basic Web 2.0 Reference Architecture
——The Service Tier（1）

- **Service invocation layer：** The service invocation layer is where listeners are plugged in to capture events that may trigger services to perform certain actions
- **Service container：** Once a service is invoked, a container instance is spawned to carry out the service invocation request until it is either successfully concluded or hits a fatal error.
- **Business rules and workflow：** All service invocation requests are subject to internal workflow constraints and business rules.
- **Registry/repository：** A registry is a central component that keeps track of services, perhaps even in multiple versions.
- **Service provider interface (SPI)：** Since the service tier makes existing capabilities available to be consumed as services, an SPI is required to connect to the resource tier.

Basic Web 2.0 Reference Architecture
——The Client Application Tier



Detail view of the client application tier

Basic Web 2.0 Reference Architecture
——The Client Application Tier（1）

- **Controller：** The controller contains the master logic that runs all aspects of the client tier
- **Data/state management：** Any data used or mutated by the client tier may need to be held in multiple states to allow rollback to a previous state or for other auditing purposes.
- **Security container/model：** A security model expresses how components are constrained to prevent malicious code from performing harmful actions on the client tier.
- **Virtual machines：** Virtual machines (VMs) are plug-ins that can emulate a specific runtime environment for various client-side technologies.
- **Rendering and media：** Management of the media and rendering processes is required to present a graphical interface to users (assuming they are humans).
- **Communications：** With every client-tier application, communication services are required.

Architectural Models That Span Tiers

- **The SOA and MVC architectural models are key pillars of Web 2.0.**
- **The services tier and the client application tier must be built using similar design principles so that they can provide a platform for interaction.**
- **Resource tier and client application tier designers tend to abide by the core tenets and axioms of the Reference Model for SOA and apply application design principles such as MVC.**
- **The MVC paradigm encourages design of applications in such a way that data sets can be repurposed for multiple views or targets on the edge, as it separates the core data from other bytes concerned with logic or views.**

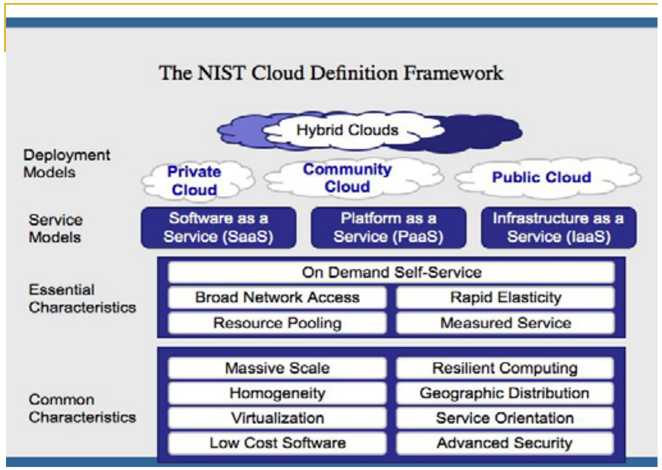
- **Model-View-Controller (MVC)**
- **Service-Oriented Architecture (SOA)**

Topic 21: Cloud Computing and Software Architecture

Definition of Cloud Computing

"***Cloud computing*** *is a model for enabling convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction. This cloud model promotes availability and is composed of five essential* ***characteristics,*** *three* ***service models****, and four* ***deployment models****."*

——National Institute of Standards and Technology, Information Technology Laboratory



Definition of Cloud Computing
——Essential Characteristics(1)

- ***On-demand self-service***
  A consumer can unilaterally provision computing capabilities, such as server time and network storage, as needed automatically without requiring human interaction with each service's provider.
- ***Broad network access***
  Capabilities are available over the network and accessed through standard mechanisms that promote use by heterogeneous thin or thick client platforms (e.g., mobile phones, laptops, and PDAs).
- ***Resource pooling***
  The provider's computing resources are pooled to serve multiple consumers using a multi-tenant model, with different physical and virtual resources dynamically assigned and reassigned according to consumer demand. There is a sense of location independence in that the customer generally has no control or knowledge over the exact location of the provided resources but may be able to specify location at a higher level of abstraction (e.g., country, state, or datacenter).

Definition of Cloud Computing
——Essential Characteristics(1)

- ***Rapid elasticity***.

  Capabilities can be rapidly and elastically provisioned, in some cases automatically, to quickly scale out and rapidly released to quickly scale in. To the consumer, the capabilities available for provisioning often appear to be unlimited and can be purchased in any quantity at any time.

- ***Measured Service***
  Cloud systems automatically control and optimize resource use by leveraging a metering capability at some level of abstraction appropriate to the type of service (e.g., storage, processing, bandwidth, and active user accounts). Resource usage can be monitored, controlled, and reported providing transparency for both the provider and consumer of the utilized service.

Definition of Cloud Computing
—— Service Models

- *Cloud Software as a Service (SaaS).*
  - The capability provided to the consumer is to use the provider's applications running on a cloud infrastructure.
  - applications are accessible from various client devices through a thin client interface
  - The consumer does not manage or control the underlying cloud infrastructure
- *Cloud Platform as a Service (PaaS).*
  - The capability provided to the consumer is to deploy onto the cloud infrastructure consumer-created or acquired applications created using programming languages and tools supported by the provider.
  - The consumer does not manage or control the underlying cloud infrastructure
- *Cloud Infrastructure as a Service (IaaS).*
  - The capability provided to the consumer is to provision processing, storage, networks, and other fundamental computing resources where the consumer is able to deploy and run arbitrary software, which can include operating systems and applications.
  - The consumer does not manage or control the underlying cloud infrastructure but has control over operating systems, storage, deployed applications, and possibly limited control of select networking components (e.g., host firewalls).

Definition of Cloud Computing —— Deployment Models

- ***Private cloud***
  The cloud infrastructure is operated solely for an organization. It may be managed by the organization or a third party and may exist on premise or off premise.
- ***Community cloud***
  The cloud infrastructure is shared by several organizations and supports a specific community that has shared concerns (e.g., mission, security requirements, policy, and compliance considerations). It may be managed by the organizations or a third party and may exist on premise or off premise.
- ***Public cloud***
  The cloud infrastructure is made available to the general public or a large industry group and is owned by an organization selling cloud services.
- ***Hybrid cloud***
  The cloud infrastructure is a composition of two or more clouds (private, community, or public) that remain unique entities but are bound together by standardized or proprietary technology that enables data and application portability (e.g., cloud bursting for load-balancing between clouds).

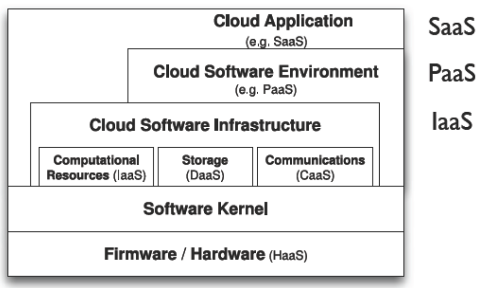
809

Motivation

- Pay-as-you-go (utility computing)
  - No initial investments
- Reduced operational costs
- Scalability
  - Exploit variable load and bursts
  - Scalable services provided
- Availability (replication, avail. zones)

810

Cloud ontology



Toward a Unified Ontology of Cloud Computing. Youseff, Lamia, University of California, Santa Barbara.

811

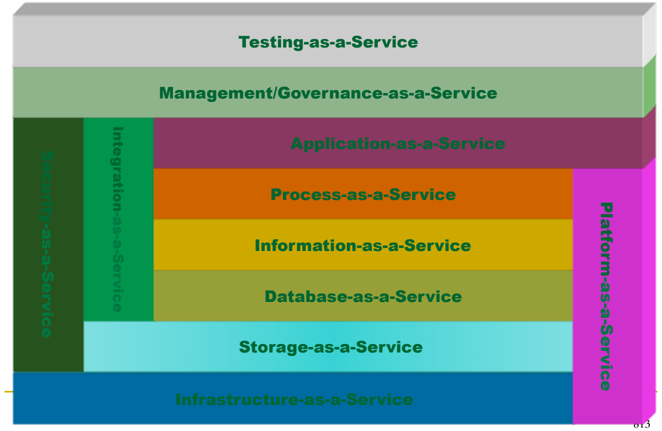
NIST's Foundational Elements of Cloud Computing

- Primary Technologies
  - Virtualization
  - Grid technology
  - Service Oriented Architectures
  - Distributed Computing
  - Broadband Networks
  - Browser as a platform
  - Free and Open Source Software
- Additional Technologies
  - Autonomic Systems
  - Web 2.0
  - Web application frameworks
  - Service Level Agreements

©2005-9 Avtec Group

812

Organizing the Clouds

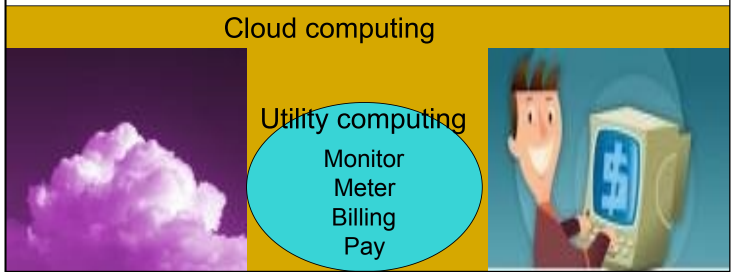


Cloud Computing's Brother Buzzwords

- *Utility computing*
- *Distributed computing*
- *Grid computing*

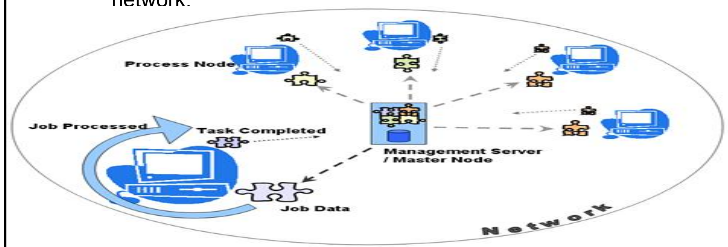
Comparison of Utility Computing and Cloud Computing

- Utility computing is a business model, it is a type of price model to deliver application infrastructure resource.



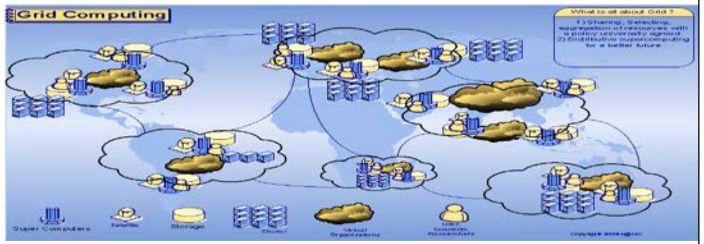
Comparison of Distributed Computing and Cloud Computing

- Distributed computing deals with hardware and software systems containing more than one processing element or storage element, concurrent processes, or multiple programs, running under a loosely or tightly controlled regime.
- In distributed computing， a program is split up into parts that run simultaneously on multiple computers communicating over a network.

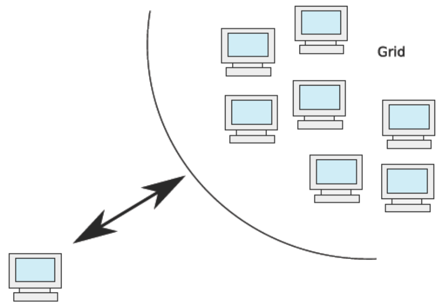


Comparison of Grid computing and Cloud Computing

- Often found in scientific environments.
- Motivation: high performance, improving resource utilization.
- Aims to create illusion of a simple, yet powerful computer out of a large number of heterogeneous systems.
- Jobs are submitted and distributed on nodes in the grid.



Grid computing



Grid vs. cloud computing

Area	Grid	Cloud
Motivation	Performance, capacity	Flexibility and scalability
Infrastructure	Owned by participants	Provided by third party
Business model	Share costs	Pay-as-you-go
Virtualization	In some cases	Prevalent
Typical applications	Research, batch jobs	On-demand infrastructure, web applications
Advantages	Mature technology	Low entry barrier, flexible
Disadvantages	Initial investment, less flexibility	Open issues, third-party dependence

Use cases

- Scientific experiments
- Web services with bursts or variable load
  - "Black friday"
  - Slashdot-effect
- Temporary infrastructure
  - Test environments
  - Projects

Amazon EC2

- Infrastructure as a Service provider, and current market leader.
- Data centers in USA and Europe
  - Different regions and availability zones
- Uses Xen hypervisor
- Users provision instances in classes, with different CPU, memory and I/O performance.

821

Amazon EC2

- Users provision instances with an Amazon Machine Image (AMI), packaged virtual machines.
- Instances ready in 10-20 seconds.
- Amazon provides a range of AMIs
- Users can upload and share custom AMIs, preconfigured for different roles.
- Supports Windows, OpenSolaris and Linux

822

Amazon EC2

- Pricing based on instance hours
- + bandwidth charges
- + service charges (S3, SQS etc.)

- SLA w/ some availability guarantees

823

Google AppEngine

- Platform as a Service
- Target: Web applications
- Provides custom Python runtime environment, with a specialized version of the Django framework.
- Integrated with Google data store (Bigtable), and other "Internet-scale" infrastucture.

824

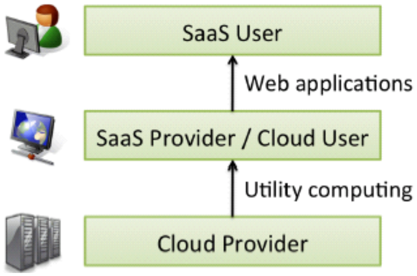
Eucalyptus

- Open source cloud implementation from University of California, Santa Barbara.
- Makes it possible to host a private cloud on your hardware.
- Compatible with EC2 interfaces.
- Intended for research and experimentation.
- Limited support for services like S3.

Windows Azure

- Platform as a Service (in pre-release)
- "Cloud OS"
- .NET libraries for managed code like C#
- Web and worker roles (w/queues)
- Topology described in metadata
- Live upgrades (w/upgrade zones)

Users and Providers of Cloud Computing



Why cloud computing now?

- Reasons for emergence of Cloud Computing:
  - **Construction and operation of large-scale datacenters.**
  - **Additional technology trends.**
  - **New business models.**

New Application Opportunities

- Mobile interactive applications.
- Parallel batch processing.
- The rise of analytics.
- Extension of compute-intensive desktop applications.

829

Obstacles and Opportunities

Obstacle	Opportunity
Availability of Service	Use Multiple Cloud Providers to provide Business Continuity;
Data Lock-In	Standardize APIs;
Data Confidentiality	Deploy Encryption, VLANs, and Firewalls;
Data Transfer Bottlenecks	FedExing Disks; Data Backup/Archival; Lower WAN Router Costs
Scalable Storage	Invent Scalable Store
Bugs in Large-Scale Distributed Systems	Invent Debugger that relies on Distributed VMs
Scaling Quickly	Invent Auto-Scaler that relies on Machine Learning; Snapshots to encourage Cloud Computing Conservationism
Reputation Fate Sharing	Offer reputation-guarding services like those for email
Software Licensing	Pay-for-use licenses; Bulk use sales

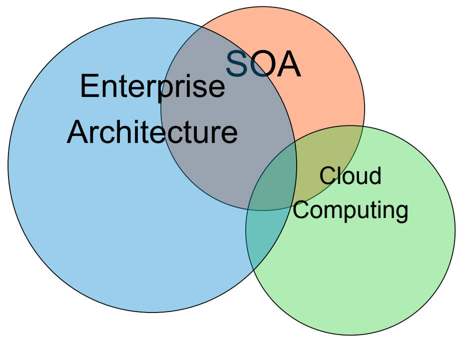
830

Some Observations

- **The long dreamed vision of computing as a utility is finally emerging. The elasticity of a utility matches the need of businesses providing services directly to customers over the Internet.**
- **From the cloud provider's view, the construction of large datacenters at low cost uncovered the possibility of selling resources on a pay-as-you-go model below the costs of medium-sized datacenters.**
- **From the cloud user's view, it would be as startling for a new software startup to build its own datacenter. Also many other organizations take advantage of the elasticity of Cloud Computing such as newspapers like Washington Post, movie companies like Pixar.**

831

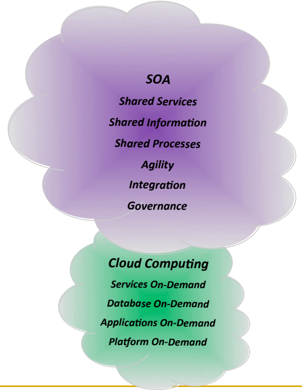
Relationships：SOA and Cloud Computing



832

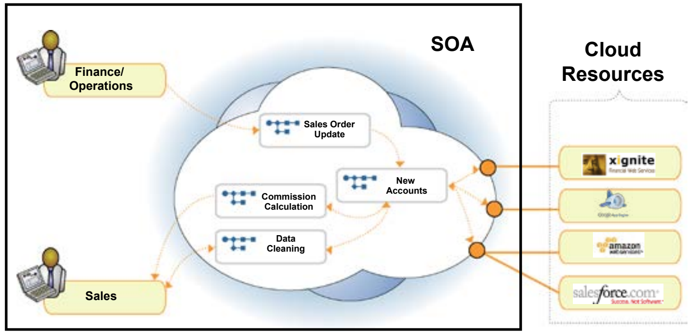
SOA and Cloud Computing

- One can consider cloud computing the extension of SOA out to cloud-delivered resources, such as storage-as-a-service, data-as-a-service, platform-as-a-service -- you get the idea.
- The trick is to determine which services, information, and processes are good candidates to reside in the clouds, as well as which cloud services should be abstracted within the existing or emerging SOA.



833

The Basic Idea



834

IT is Skeptical

- Enterprise IT is understandably skittish about cloud computing.
- However, many of the cloud computing resources out there will actually provide better service than on-premise.
- Security and performance are still issues.

835

However, Not So Fast

- Not all computing resources should exist in the clouds.
- Cloud computing is not always cost effective.
- Do your homework before making the move.

836

When Cloud Computing may be a Fit

- When the processes, applications, and data are largely independent.
- When the points of integration are well defined.
- When a lower level of security will work just fine.
- When the core internal enterprise architecture is healthy.
- When the Web is the desired platform.
- When cost is an issue.
- When the applications are new.

837

When Cloud Computing may not a Fit

- When the processes, applications, and data are largely coupled.
- When the points of integration are not well defined.
- When a high level of security is required.
- When the core internal enterprise architecture needs work.
- When the application requires a native interface.
- When cost is an issue.
- When the application is legacy.

838

Start with the Architecture

Understand:

- Business drivers
- Information under management
- Existing services under management
- Core business processes

839

Getting Ready

- So, how do you prepare yourself? I have a few suggestions:
  - **First, accept the notion that it's okay to leverage services that are hosted on the Internet as part of your SOA.** Normal security management needs to apply, of course.
  - Second, **create a strategy for the consumption and management of cloud services**, including how you'll deal with semantic management, security, transactions, etc.
  - Finally, **create a proof of concept now.** This does a few things including getting you through the initial learning process and providing proof points as to the feasibility of leveraging cloud computing resources.

840

Stepping to the Clouds

1. Access the business.
2. Access the culture.
3. Access the value.
4. Understand your data.
5. Understand your services.
6. Understand your processes.
7. Understand the cloud resources.
8. Identify candidate data.
9. Identify candidate services.
10. Identify candidate processes.
11. Create a governance strategy.
12. Create a security strategy.
13. Bind candidate services to data and processes.
14. Relocate services, processes, and information.
15. Implement security.
16. Implement governance.
17. Implement operations.

841

Topic 22: Software Architecture and Concurrency

Why is concurrency so important?

Traditionally, specialized area of interest to a few experts:

- Operating systems
- Networking
- Databases

Multicore and the Internet make it relevant to every programmer!

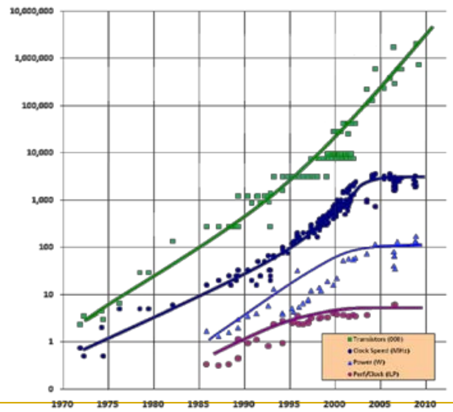
843

What they say about concurrency

- Intel Corporation: *Multi-core processing is taking the industry on a fast-moving and exciting ride into profoundly new territory. The defining paradigm in computing performance has shifted inexorably from raw clock speed to parallel operations and energy efficiency.*
- Rick Rashid, head of Microsoft Research: *Multicore processors represent one of the largest technology transitions in the computing industry today, with deep implications for how we develop software.*
- Bill Gates: *"Multicore: This is the one which will have the biggest impact on us. We have never had a problem to solve like this. A breakthrough is needed in how applications are done on multicore devices.*

844

Evolution of hardware (source: Intel)

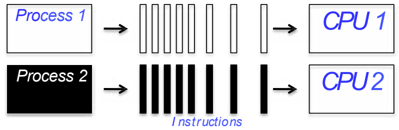


845

Multiprocessing

•Until a few years ago: systems with one processing unit were standard

•Today: most end-user systems have multiple processing units in the form of multi-core processors



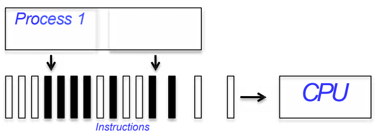
•*Multiprocessing*: the use of more than one processing unit in a system

•Execution of processes is said to be *parallel*, as they are running at the same time

Multitasking & multithreading

Even on systems with a single processing unit programs may appear to run in parallel:

- Multitasking*
- Multithreading (within a process, see in a few slides)



Multi-tasked execution of processes is said to be *interleaved*, as all are in progress, but only one is running at a time. (Closely related concept: **coroutines**.)

*This is common terminology, but "multiprocessing" was also used previously as a synonym for "multitasking"

Processes

•A (sequential) *program* is a set of instructions

•A *process* is an instance of a program that is being executed

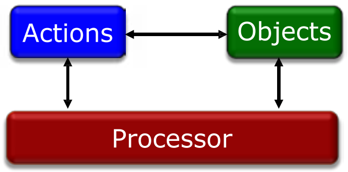
Concurrency

- Both multiprocessing and multithreading are examples of concurrent computation
- The execution of processes or threads is said to be *concurrent* if it is either parallel or interleaved

Computation

To perform a computation is

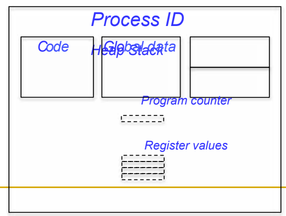
- To apply certain actions
- To certain objects
- Using certain processors



10

Operating system processes

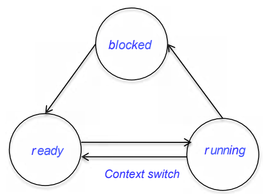
- How are processes implemented in an operating system?
- Structure of a typical process:
  - *Process identifier:* unique ID of a process.
  - *Process state:* current activity of a process.
  - *Process context:* program counter, register values
  - *Memory:* program text, global data, stack, and heap.



The scheduler

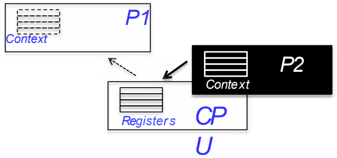
A system program called the *scheduler* controls which processes are running; it sets the process states:

- *Running*: instructions are being executed.
- *Blocked:* currently waiting for an event.
- *Ready:* ready to be executed, but has not been assigned a processor yet.



The context switch

- The swapping of processes on a processing unit by the scheduler is called a *context switch*

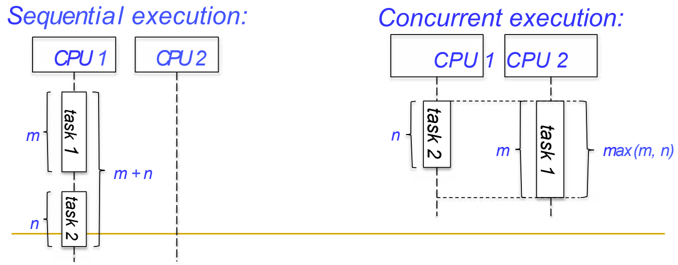


- Scheduler actions when switching processes P1 and P2:
  - P1.set_state (ready)
  - Save register values as P1's context in memory
  - Use context of P2 to set register values
  - P2.set_state (running)

Concurrency within programs

- We also want to use concurrency within programs

```
compute
do
t1.do_task1
t2.do_task2
end
```



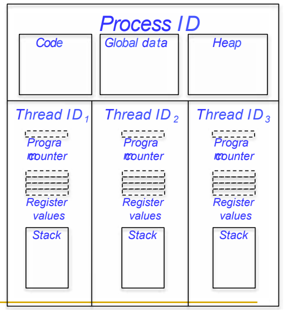
Threads ("lightweight processes")

Make programs concurrent by associating them with threads

A *thread* is a part of an operating system process

Private to each thread:
- Thread identifier
- Thread state
- Thread context
- Memory: only stack

Shared with other threads:
- Program text
- Global data
- Heap



Processes vs threads

Process:
- Has its own (virtual) memory space (in O-O programming, its own objects)
- Sharing of data (objects) with another process:
  - Is explicit (good for reliability, security, readability)
  - Is heavy (bad for ease of programming)
- Switching to another process: expensive (needs to back up one full context and restore another

Thread:
- Shares memory with other threads
- Sharing of data is straightforward
  - Simple go program (good)
  - Risks of confusion and errors: data races (bad)
- Switching to another thread: cheap

Concurrent programs in Java

Associating a computation with a thread:

```
class Thread1 extends Thread {
    public void run() {
      // implement task1 here
    }
}
class Thread2 extends Thread {
    public void run() {
      // implement task2 here
    }
}
```

```
void compute() {
Thread1 t1 = new Thread1();
Thread2 t2 = new Thread2();
t1.start();
t2.start();
}
```

•Write a class that inherits from the **class Thread** (or implements the **interface Runnable**)

• Implement the method run()

Joining threads

Often the final results of thread executions need to be combined:

```
return t1.getResult() + t2.getResult();
```

To wait for both threads to be finished, we *join* them:

```
t1.start();
t2.start();
t1.join();
t2.join();
return t1.getResult() + t2.getResult();
```

The join() method, invoked on a thread *t*, causes the caller to wait until *t* is finished

Race conditions (1)

Consider a counter class:

```
class Counter {
private int value = 0;

public int getValue() {
return value;
}

public void setValue(int someValue)
{ value = someValue;
}

public void increment() {
value++;
}
}
```

Assume two threads:

Thread 1:

```
x.setValue(0);
x.increment();
int i = x.getValue();
```

Thread 2:

```
x.setValue(2);
```

Race conditions (2)

• Because of the interleaving of threads, various results can be obtained:

x.setValue(2) x.setValue(0) x.increment() int i = x.getValue()	x.setValue(0) x.setValue(2) x.increment() int i = x.getValue()	x.setValue(0) x.increment() x.setValue(2) int i = x.getValue()	x.setValue(0) x.increment() int i = x.getValue() x.setValue(2)
i == 1 x.value == ?	i == 3 x.value == ?	i == 2 x.value == ?	i == 1 x.value == ?

Such dependence of the result on nondeterministic interleaving is a ***race condition*** (or ***data race***)

Such errors can stay hidden for a long time and are difficult to find by testing

Race conditions (2)

• Because of the interleaving of threads, various results can be obtained:

x.setValue(2) x.setValue(0) x.increment() int i = x.getValue()	x.setValue(0) x.setValue(2) x.increment() int i = x.getValue()	x.setValue(0) x.increment() x.setValue(2) int i = x.getValue()	x.setValue(0) x.increment() int i = x.getValue() x.setValue(2)
i == 1 x.value == **1**	i == 3 x.value == **3**	i == 2 x.value == **2**	i == 1 x.value == **2**

Such dependence of the result on nondeterministic interleaving is a ***race condition***(or ***data race***)

Such errors can stay hidden for a long time and are difficult to find by testing

8
a

Synchronization

To avoid data races, threads (or processes) must *synchronize* with each other, i.e. communicate to agree on the appropriate sequence of actions

How to communicate:

➢By reading and writing to shared sections of memory (*shared memory synchronization*)
In the example, threads should agree that at any one time at most one of them can access the resource

➢By explicit exchange of information (*message passing synchronization*)

Mutual exclusion

Mutual exclusion (or "mutex") is a form of synchronization that avoids the simultaneous use of a shared resource

- To identify the program parts that need attention, we introduce the notion of a *critical section* : a part of a program that accesses a shared resource, and should normally be executed by at most one thread at a time

Mutual exclusion in Java

•Each object in Java has a mutex lock (can be held only by one thread at a time!) that can be acquired and released within **synchronized** blocks:

•Object lock = **new** Object();

```
synchronized (lock) {
// critical section
}
```

•The following are equivalent:

```
synchronized type m(args) {        type m(args) {
                                      synchronized (this) {
   // body                               // body
                                      }
}                                  }
```

Example: mutual exclusion

To avoid data races in the example, we enclose instructions to be executed atomically in synchronized blocks protected with the same lock objects

```
synchronized (lock)
   { x.setValue(0);
   x.increment();
int i = x.getValue();
}
```

```
synchronized (lock) {
   x.setValue(2);
}
```

30

The producer-consumer problem

Consider two types of looping processes:

- *Producer:* At each loop iteration, produces a data item for consumption by a consumer
- *Consumer:* At each loop iteration, consumes a data item produced by a producer

Producers and consumers communicate via a shared buffer (a generalized notion of bounded queue)

Producers append data items to the back of the queue and consumers remove data items from the front

Condition synchronization

The producer-consumer problem requires that processes access the buffer properly:

- Consumers must wait if the buffer is empty
- Producers must wait if the buffer is full

Condition synchronization is a form of synchronization where processes are delayed until a condition holds

In producer-consumer we use two forms of synchronization:

- Mutual exclusion: to prevent races on the buffer
- Condition synchronization: to prevent improper access to the buffer

Condition synchronization in Java (2)

•The following methods can be called on a synchronized object (i.e. only within a synchronized block, on the lock object):

- **wait()**: block the current thread and release the lock until some thread does a **notify()** or **notifyAll()**
- **notify()**: resume one blocked thread (chosen nondeterministically), set its state to "ready"
- **notifyAll()**: resume all blocked threads

•No guarantee that the notification mechanism is fair

Producer-Consumer problem: Consumer code

```
public void consume() throws InterruptedException {
int value;
synchronized (buffer) {
while (buffer.size() == 0) { buffer.wait();
}
value = buffer.get();
}
}
```

Consumer blocks if buffer.size() == 0 is true (waiting for a notify() from the producer)

Producer-Consumer problem: Producer code

```
public void produce() {
int value = random.produceValue();
synchronized (buffer)
{ buffer.put(value); buffer.notify();
}
}
```

Producer notifies consumer that the condition buffer.size() == 0 is no longer true

The problem of deadlock

The ability to hold resources exclusively is central to providing process synchronization for resource access

Unfortunately, it brings about other problems!

A *deadlock* is the situation where a group of processes blocks forever because each of the processes is waiting for resources which are held by another process in the group

Deadlock example in Java

Consider the class

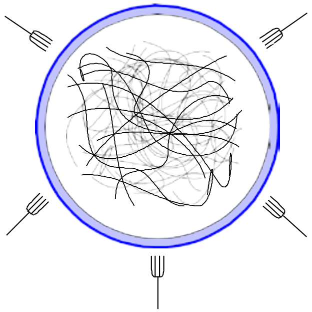
```
public class C extends Thread {
private Object a;
private Object b;

public C(Object x, Object y) { a =
x;
b = y;
}
public void run() {
synchronized (a) {
synchronized (b) {
        ...
}
}
}}
```

... and this code being executed:

```
C t1 = new C(a1, b1); C
t2  =  new  C(b1,  a1);
t1.start();
t2.start();
```

Dining philosophers



Are deadlock & data races of the same kind?

No

Two kinds of concurrency issues (Lamport):

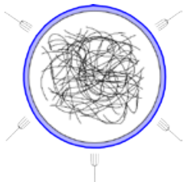
- Safety: no bad thing will happen
- Liveness: some good thing will happen

Dining philosophers

```
class PHILOSOPHER inherit
PROCESS
            rename
setup as getup
redefine step end

feature {BUTLER}
step
           do
                  think ;   eat (left, right)
           end

      eat (l, r : separate FORK)
                -- Eat, having grabbed l and r.
           do ... end
end
```



Producer-Consumer problem: Producer code

```
put (b: separate BUFFER [T]; v: T)
   require
      not b.is_full
   local
      value: INTEGER
   do
      b.put (v)
   end
```

•Very easy to provide a solution for bounded buffers

•No need for notification, the SCOOP scheduler ensures that preconditions are automatically reevaluated at a later time

Contracts

```
put (buf : separate QUEUE [INTEGER ] ; v : INTEGER)
        -- Store v into buffer.
    require
        not buf.is_full
        v > 0
    do
        buf.put (v)
    ensure
        not buf.is_empty
    end

...
put (my_buffer, 10 )
```

Precondition becomes wait condition

For more

Several concurrency courses in the ETH curriculum, including our (Bertrand Meyer, Sebastian Nanz) "Concepts of Concurrent Computation" (Spring semester)

Good textbooks:

Kramer Herlihy

Topic 23: Visualising Software Architectures

Objectives

- Concepts
  - What is visualization?
  - Differences between modeling and visualization
  - What kinds of visualizations do we use?
  - Visualizations and views
  - How can we characterize and evaluate visualizations?
- Examples
  - Concrete examples of a diverse array of visualizations
- Constructing visualizations
  - Guidelines for constructing new visualizations
  - Pitfalls to avoid when constructing new visualizations
  - Coordinating visualizations

Objectives

- Concepts
  - What is visualization?
  - Differences between modeling and visualization
  - What kinds of visualizations do we use?
  - Visualizations and views
  - How can we characterize and evaluate visualizations?
- Examples
  - Concrete examples of a diverse array of visualizations
- Constructing visualizations
  - Guidelines for constructing new visualizations
  - Pitfalls to avoid when constructing new visualizations
  - Coordinating visualizations

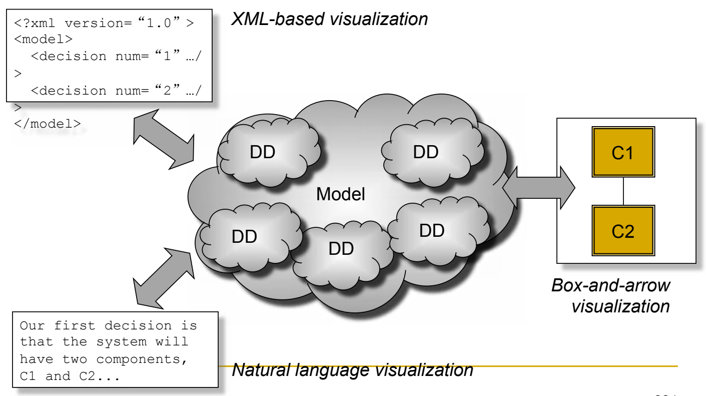
What is Architectural Visualization?

- Recall that we have characterized architecture as *the set of principal design decisions* made about a system
- Recall also that models are artifacts that capture some or all of the design decisions that comprise an architecture
- An architectural **visualization** defines how architectural models are depicted, and how stakeholders interact with those depictions
  - Two key aspects here:
    - **Depiction** is a picture or other visual representation of design decisions
    - **Interaction** mechanisms allow stakeholders to interact with design decisions in terms of the depiction

Models vs. Visualizations

- It is easy to confuse models and visualizations because they are very closely related
- In the previous lectures, we have not drawn out this distinction, but now we make it explicit
- A **model** is just abstract information – a set of design decisions
- **Visualizations** give those design decisions form: they let us **depict** those design decisions and **interact** with them in different ways
  - Because of the interaction aspect, visualizations are often active – they are both pictures AND tools

Models vs. Visualizations



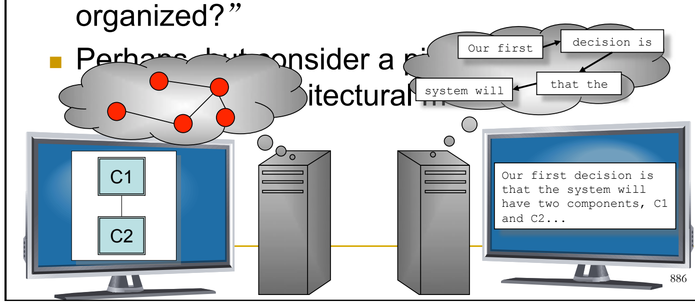
Canonical Visualizations

- Each modeling notation is associated with one or more canonical visualizations
  - This makes it easy to think of a notation and a visualization as the same thing, even though they are not
- Some notations are canonically textual
  - Natural language, XML-based ADLs
- …or graphical
  - PowerPoint-style
- …or a little of both
  - UML
- …or have multiple canonical visualizations
  - Darwin

885

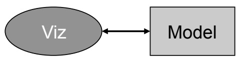
Another Way to Think About It

- We may ask "isn't the canonical visualization the same as the notation since that is how the information is fundamentally organized?"
- Perha[...]onsider a [...]itectural

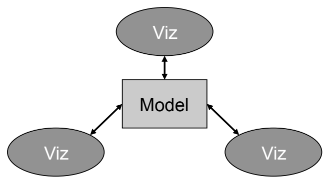


886

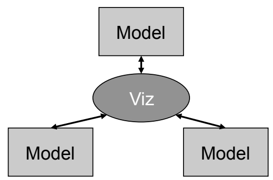
Different Relationships



One (canonical) visualization (common)



Many visualizations for one model (common)



One visualization bringing together many models (uncommon)

887

Kinds of Visualizations: Textual Visualizations

- Depict architectures through ordinary text files
  - Generally conform to some syntactic format, like programs conform to a language
  - May be natural language, in which case the format is defined by the spelling and grammar rules of the language
  - Decorative options
    - Fonts, colors, bold/italics
    - Tables, bulleted lists/outlines

888

Textual Visualizations

```
<instance:xArch xsi:type=" instance:XArch" >
  <types:archStructure xsi:type=" types:ArchStructure"
                      types:id=" ClientArch" >
    <types:description xsi:type=" instance:Description" >
      Client Architecture
    </types:description>
    <types:component xsi:type=" types:Component"
                    types:id=" WebBrowser" >
      <types:description xsi:type=" instance:Description" >
        Web Browser
      </types:description>
      <types:interface xsi:type=" types:Interface"
                      types:id=" WebBrowserInterface" >
        <types:description xsi:type=" instance:Description" >
          Web Browser Interface
        </types:description>
        <types:direction xsi:type=" instance:Direction" >
          inout
        </types:direction>
      </types:interface>
    </types:component>
  </types:archStructure>
</instance:xArch>
```

XML visualization

Textual Visualizations (cont’ d)

```
<instance:xArch xsi:type=" instance:XArch" >
  <types:archStructure xsi:type=" types:ArchStructure"
                      types:id=" ClientArch" >
    <types:description xsi:type=" instance:Description" >
      Client Architecture
    </types:description>
    <types:component xsi:type=" types:Component"
                    types:id=" WebBrowser" >
      <types:description xsi:type=" instance:Description" >
        Web Browser
      </types:description>
      <types:interface xsi:type=" types:In
                      types:id=" WebBrows
        <types:description xsi:type=" inst
          Web Browser Interface
        </types:description>
        <types:direction xsi:type=" instan
          inout
        </types:direction>
      </types:interface>
    </types:component>
  </types:archStructure>
</instance:xArch>
```

XML visualization

```
xArch{
  archStructure{
    id = “ClientArch”
    description = “Client Architecture”
    component{
      id = “WebBrowser”
      description = “Web Browser”
      interface{
        id = “WebBrowserInterface”
        description = “Web Browser Interface”
        direction = “inout”
      }
    }
  }
}
```

Compact visualization

Textual Visualizations: Interaction

- Generally through an ordinary text editor or word processor
- Some advanced mechanisms available
  - Syntax highlighting
  - Static checking
  - Autocomplete
  - Structural folding

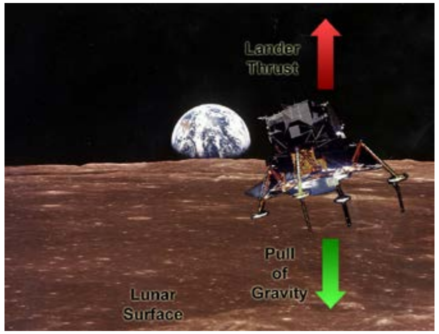
Textual Visualizations

- Advantages
  - Depict entire architecture in a single file
  - Good for linear or hierarchical structures
  - Hundreds of available editors
  - Substantial tool support if syntax is rigorous (e.g., defined in something like BNF)
- Disadvantages
  - Can be overwhelming
  - Bad for graphlike organizations of information
  - Difficult to reorganize information meaningfully
  - Learning curve for syntax/semantics

Kinds of Visualizations: Graphical Visualizations

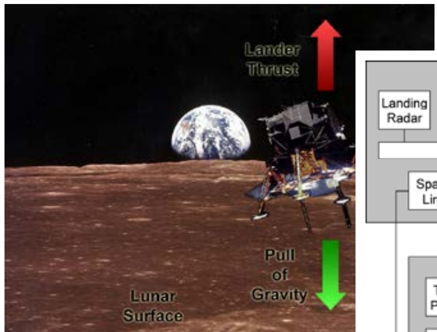
- Depict architectures (primarily) as graphical symbols
  - Boxes, shapes, pictures, clip-art
  - Lines, arrows, other connectors
  - Photographic images
  - Regions, shading
  - 2D or 3D
- Generally conform to a symbolic syntax
  - But may also be ‘free-form’ and stylistic

Graphical Visualizations

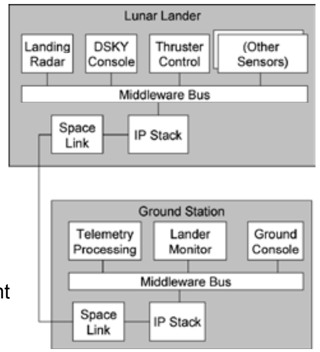


Abstract, stylized visualization

Graphical Visualizations



Abstract, stylized visualization



More rigorous deployment visualization

Graphical Visualizations: Interaction

- Generally graphical editors with point-and-click interfaces
  - Employ metaphors like scrolling, zooming, ‘drill-down’
- Editors have varying levels of awareness for different target notations
  - For example, you can develop UML models in PowerPoint (or Photoshop), but the tools won’t help much
- More exotic editors and interaction mechanisms exist in research
  - 3D editors
  - “Sketching-based” editors

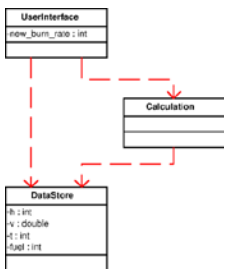
Graphical Visualizations

- Advantages
  - Symbols, colors, and visual decorations more easily parsed by humans than structured text
  - Handle non-hierarchical relationships well
  - Diverse spatial interaction metaphors (scrolling, zooming) allow intuitive navigation
- Disadvantages
  - Cost of building and maintaining tool support
    - Difficult to incorporate new semantics into existing tools
  - Do not scale as well as text to very large models

Hybrid Visualizations

- Many visualizations are text-only
- Few graphical notations are purely symbolic
  - Text labels, at a minimum
  - Annotations are generally textual as well
- Some notations incorporate substantial parts that are mostly graphical alongside substantial parts that are mostly or wholly textual

Hybrid Visualizations (cont’d)



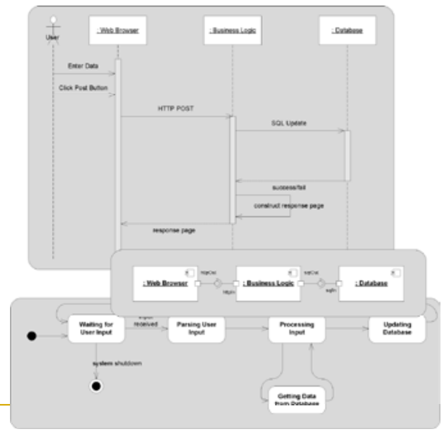
context UserInterface
inv: new_burn_rate >= 0

Primarily graphical UML class diagram

Architectural constraints expressed in OCL

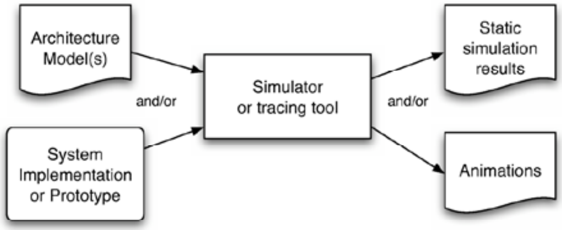
Views, Viewpoints, & Visualizations

- Recall that a view is a subset of the design decisions in an architecture
- And a viewpoint is the perspective from which a view is taken (i.e., the filter that selects the subset)
- Visualizations are associated with viewpoints



Effect Visualizations

- Not all visualizations used in architecture-centric development depict design decisions directly
- Some depict the results or effects of design decisions
  - We call these 'effect visualizations'
- May be textual, graphical, hybrid, etc.



901

Evaluating Visualizations

- Scope and Purpose
  - What is the visualization for? What can it visualize?
- Basic Type
  - Textual? Graphical? Hybrid? Effect?
- Depiction
  - What depiction mechanisms and metaphors are primarily employed by the visualization?
- Interaction
  - What interaction mechanisms and metaphors are primarily employed by the visualization?

902

Evaluating Visualizations (cont' d)

- Fidelity
  - How well/completely does the visualization reflect the information in the underlying model?
  - Consistency should be a minimum requirement, but details are often left out
- Consistency
  - How well does the visualization use similar representations for similar concepts?
- Comprehensibility
  - How easy is it for stakeholders to understand and use a visualization
    - Note: this is a function of both the visualization and the stakeholders

903

Evaluating Visualizations (cont' d)

- Dynamism
  - How well does the visualization support models that change over time (dynamic models)?
- View Coordination
  - How well the visualization is connected to and kept consistent with other visualizations
- Aesthetics
  - How pleasing is the visualization (look and feel) to its users?
    - A very subjective judgment
- Extensibility
  - How easy is it to add new capabilities to a visualization?

904

Objectives

- Concepts
  - What is visualization?
  - Differences between modeling and visualization
  - What kinds of visualizations do we use?
  - Visualizations and views
  - How can we characterize and evaluate visualizations?
- Examples
  - Concrete examples of a diverse array of visualizations
- Constructing visualizations
  - Guidelines for constructing new visualizations
  - Pitfalls to avoid when constructing new visualizations
  - Coordinating visualizations

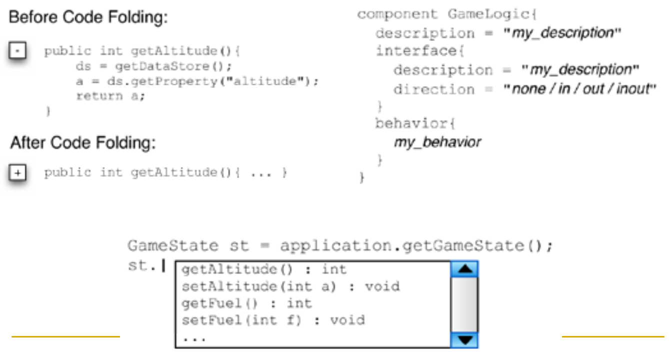
Text Visualizations

- Text visualizations are generally provided through text editors
- Examples:
  - Simple: Windows Notepad, SimpleText, pico, joe
  - For experts: vi, emacs
  - With underlying language support: Eclipse, UltraEdit, many HTML editors
  - Free-form text documents: Microsoft Word, other word processors

Text Visualizations (cont' d)

- Advantages
  - Provide a uniform way of working with many different underlying notations
  - Wide range of editors available to suit any need
  - Many incorporate advanced 'content assist' capabilities
  - Many text editors can be extended to handle new languages or integrate new tools easily
- Disadvantages
  - Increasing complexity as models get bigger
  - Do not handle graph structures and complex interrlationships well

Advanced Interaction Mechanisms



Text Visualizations: Evaluation

- Scope/Purpose
  - Visualizing design decisions or effects as (structured) text
- Basic Type
  - Textual
- Depiction
  - Ordered lines of characters possibly grouped into tokens
- Interaction
  - Basic: insert, delete, copy, paste
  - Advanced: coloring, code folding, etc.
- Fidelity
  - Generally canonical
- Consistency
  - Generally good; depends on underlying notation
- Comprehensibility
  - Drops with increasing complexity
- Dynamism
  - Rare, but depends on editor
- View coordination
  - Depends on editor
- Aesthetics
  - Varies; can be overwhelming or elegant and structured
- Extensibility
  - Many extensible editors

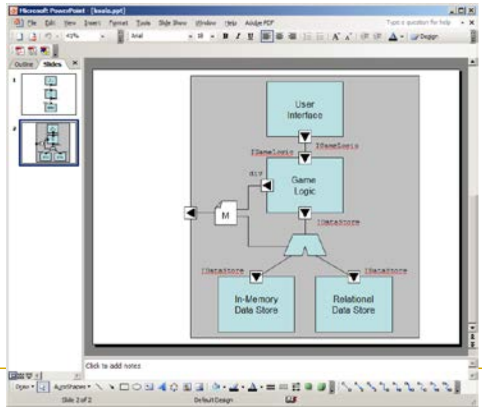
909

General Graphical Visualizations

- E.g., PowerPoint, OmniGraffle, etc.
- Provide point-and-click manipulation of graphical symbols, interconnections, and text blocks
- Advantages
  - Friendly UI can create nice-looking depictions
  - Nothing hidden; no information difference between model and depiction
- Disadvantages
  - No underlying semantics; difficult to add them
    - Visio is a partial exception
    - This means that interaction mechanisms can offer minimal support
  - Difficult to connect to other visualizations

910

General Graphical Example



911

General Graphical: Evaluation

- Scope/Purpose
  - Visualizing design decisions as symbolic pictures
- Basic Type
  - Graphical
- Depiction
  - (Possibly) interconnected symbols on a finite canvas
- Interaction
  - Point and click, drag-and-drop direct interactions with symbols, augmented by menus and dialogs
- Fidelity
  - Generally canonical
- Consistency
  - Manual
- Comprehensibility
  - Depends on skill of the modeler and use of consistent symbols/patterns
- Dynamism
  - Some animation capabilities
- View coordination
  - Difficult at best
- Aesthetics
  - Modeler's responsibility
- Extensibility
  - Adding new symbols is easy, adding semantics is harder

912

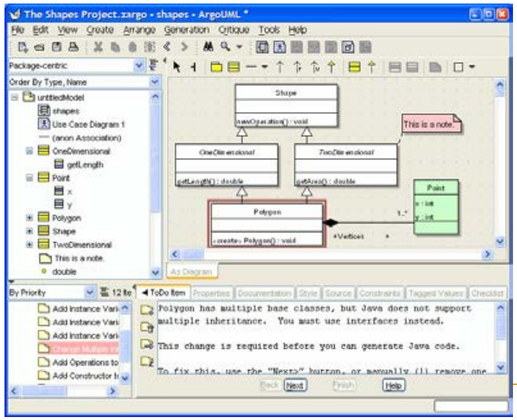
Objectives

- Concepts
  - What is visualization?
  - Differences between modeling and visualization
  - What kinds of visualizations do we use?
  - Visualizations and views
  - How can we characterize and evaluate visualizations?
- Examples
  - Concrete examples of a diverse array of visualizations
- Constructing visualizations
  - Guidelines for constructing new visualizations
  - Pitfalls to avoid when constructing new visualizations
  - Coordinating visualizations

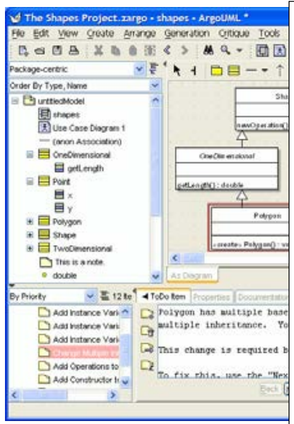
UML Visualizations

- Canonical graphical depictions + tool-specific interactions
- XMI: Textual depiction in XML + text-editor interactions
- Advantages
  - Canonical graphical depiction common across tools
  - Graphical visualizations have similar UI metaphors to PowerPoint-style editors, but with UML semantics
  - XMI projection provides textual alternative
- Disadvantages
  - No standard for interaction as there is for depiction
  - In some tools hard to tell where UML model ends and auxiliary models begin
  - Most UML visualizations are restricted to (slight variants) of the canonical UML depiction

UML Visualization



UML Visualization



```
<UML:Class xmi.id = '723'
           name = 'Data Store'
           visibility = 'public'
           isSpecification = 'false'
           isRoot = 'false'
           isLeaf = 'false'
           isAbstract = 'false'
           isActive = 'false'/>

<UML:Association xmi.id = '725'
                 name = ''
                 isSpecification = 'false'
                 isRoot = 'false'
                 isLeaf = 'false'
                 isAbstract = 'false'>
  <UML:Association.connection>
    <UML:AssociationEnd xmi.id = '726'
                        visibility = 'public'
                        isSpecification = 'false'
                        isNavigable = 'true'
                        ordering = 'unordered'
                        aggregation = 'none'
                        targetScope = 'instance'
                        changeability = 'changeable'>
      <UML:AssociationEnd.multiplicity>
        <UML:Multiplicity xmi.id = '727'>
          <UML:Multiplicity.range>
            <UML:MultiplicityRange xmi.id = '728'
                                   lower = '1'
                                   upper = '1'/>
          ...
```

UML Visualizations: Evaluation

- Scope/Purpose
  - Visualization of UML models
- Basic Type
  - Graphical (diagrams), textual (XMI)
- Depiction
  - Diagrams in UML symbolic vocabulary/XML-formatted text
- Interaction
  - Depends on the editor; generally point-and-click for diagrams; text editor for XMI
- Fidelity
  - Diagrams are canonical, XMI elides layout info
- Consistency
  - Generally good across diagrams; small exceptions
- Comprehensibility
  - Ubiquity assists interpretations
- Dynamism
  - Rare
- View coordination
  - Some editors better than others
- Aesthetics
  - Simple symbols reduce complexity; uniform diagrams
- Extensibility
  - Profile support OK; major language extensions hard

Rapidé

- Rapidé models are generally written with a canonical textual visualization
  - Some graphical builders available as well
- Focus: Interesting *effect visualization* of simulation results
- Advantages
  - Provides an intuitive way to visualize the causal relationships between events
  - Automatically generated from Rapide specifications
- Disadvantages
  - Complex applications generate complex graphs
  - Difficult to identify why particular causal relationships exist
    - Simulation is not interactive

Rapidé Examples

```
type DataStore is interface
  action in  SetValues();
         out NotifyNewValues();
  behavior
  begin
         SetValues => NotifyNewValues();;
end DataStore;

type Calculation is interface
  action in  SetBurnRate();
         out DoSetValues();
  behavior
         action CalcNewState();
  begin
         SetBurnRate => CalcNewState(); D
end Calculation;

type Player is interface
  action out DoSetBurnRate();
         in  NotifyNewValues();
  behavior
         TurnsRemaining : var integer :=
         action UpdateStatusDisplay();
         action Done();
```

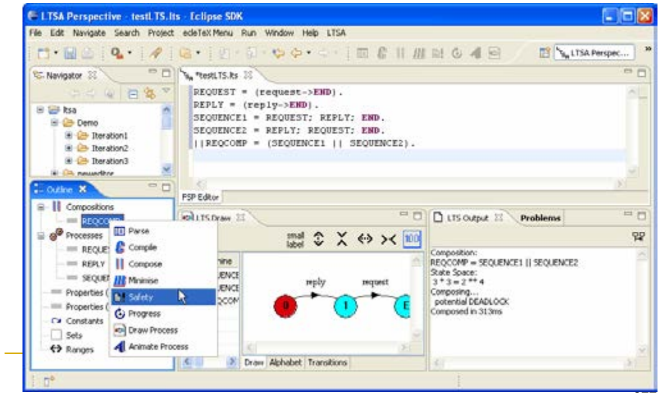
Rapidé Effect Visualization: Evaluation

- Scope/Purpose
  - Graphical event traces
- Basic Type
  - Graphical
- Depiction
  - Directed acyclic graph of events
- Interaction
  - No substantial interaction with generated event traces
- Fidelity
  - Each trace is an instance; different simulation runs may produce different traces in a non-deterministic system
- Consistency
  - Tiny symbol vocabulary ensures consistency
- Comprehensibility
  - Easy to see causal relationships but difficult to understand why they’re there
- Dynamism
  - No support
- View coordination
  - Event traces are generated automatically from architectural models
- Aesthetics
  - Simple unadorned directed acyclic graph of nodes and edges
- Extensibility
  - Tool set is effectively a ‘black box”

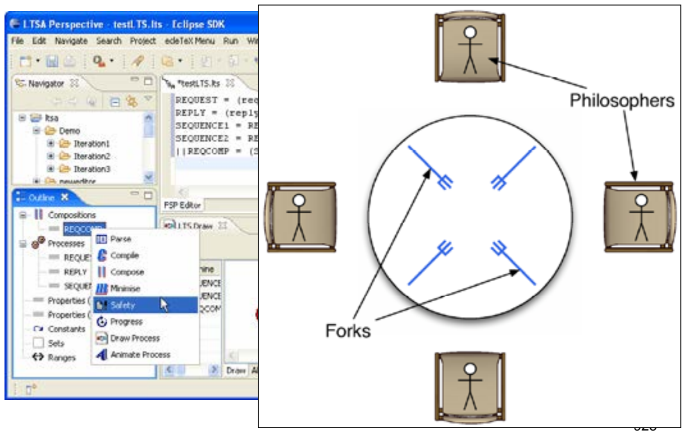
Labeled Transition State Analyzer (LTSA)

- A tool for analyzing and simultaneously visualizing concurrent systems' behavior using a modeling language called FSP
- Advantages
  - Provides multiple concurrent visualizations of concurrent behavior
  - Integrates both model and effect visualizations, textual and graphical depictions
  - Can develop domain-specific visualizations to understand abstract models
- Disadvantages
  - Behavior specification language has somewhat steep learning curve
  - Developing domain-specific graphical visualizations can be expensive

LTSA Examples



LTSA Examples



LTSA: Evaluation

- Scope/Purpose
  - Multiple coordinated visualizations of concurrent systems' behavior
- Basic Type
  - Textual, Graphical, Effect
- Depiction
  - Text & state machines for models, various effect viz.
- Interaction
  - FSP can be edited textually or graphically
- Fidelity
  - Graphical visualizations may elide some information
- Consistency
  - Limited vocabulary helps ensure consistency
- Comprehensibility
  - FSP has some learning curve; domain-specific effect visualizations are innovative
- Dynamism
  - Animation on state-transition diagrams and domain-specific visualizations
- View coordination
  - Views are coordinated automatically
- Aesthetics
  - State transition diagrams are traditional; domain-specific visualizations can enhance aesthetics
- Extensibility
  - New domain-specific effect visualizations as plug-ins

xADL Visualizations

- Coordinated set of textual, graphical, and effect visualizations for an extensible ADL
- Advantages
  - Provides an example of how to construct a wide variety of (often) coordinated or interrelated visualizations
  - Lets users move fluidly from one visualization to another
  - Guidance available for extending visualizations or adding new ones
- Disadvantages
  - Some learning curve to extend graphical editors
  - Adding or extending visualizations has to be done carefully so they play well with existing ones

xADL Visualization Examples

```
<types:component xsi:type="types:Component"
                 types:id="myComp">
  <types:description xsi:type="instance:Description">
    MyComponent
  </types:description>
  <types:interface xsi:type="types:Interface"
                   types:id="iface1">
    <types:description xsi:type="instance:Description">
      Interface1
    </types:description>
    <types:direction xsi:type="instance:Direction">
      inout
    </types:direction>
  </types:interface>
</types:component>
```

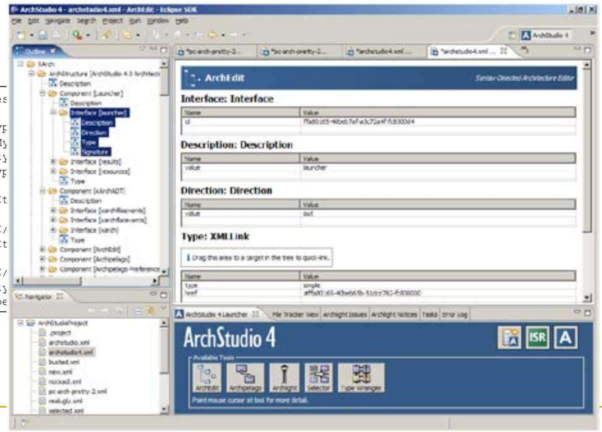

xADL Visualization Examples

```
<types:component xsi:type="types:Component"
                 types:id="myComp">
  <types:description xsi:type="instance:Description">
    MyComponent
  </types:description>
  <types:interface xsi:type="typ
                   types:id="ifac
    <types:description xsi:type=
      Interface1
    </types:description>
    <types:direction xsi:type="i
      inout
    </types:direction>
  </types:interface>
</types:component>
```

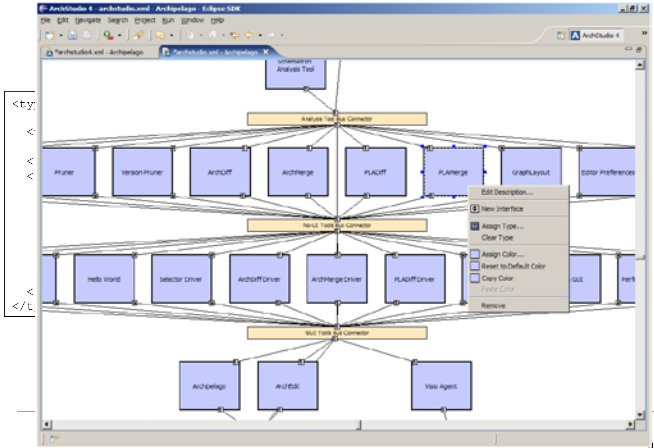
```
component{
  id = "myComp";
  description = "MyComponent";
  interface{
    id = "iface1";
    description = "Interface1";
    direction = "inout";
  }
}
```


xADL Visualization Examples

```
<types
  <typ
    My
  </ty
  <typ
    <t
    </
    <t
    </
  </ty
</type
```

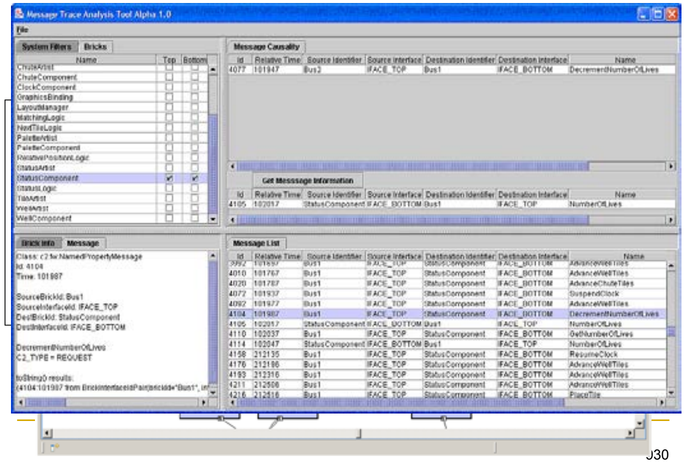


xADL Visualization Examples



929

xADL Visualization Examples



930

xADL Visualizations: Evaluation

- Scope/Purpose
  - Multiple coordinated visualizations of xADL models
- Basic Type
  - Textual, Graphical, Effect
- Depiction
  - XML, abbreviated XML, symbol graphs, hybrid effect (MTAT)
- Interaction
  - Visualizations emulate various editing paradigms
- Fidelity
  - Textual & ArchEdit complete; graphical leave detail out
- Consistency
  - Effort to follow conventions
- Comprehensibility
  - Varies; some easier than others
- Dynamism
  - Animation on state-transition diagrams and domain-specific visualizations
- View coordination
  - Many views coordinated 'live,' MTAT leverages some animation
- Aesthetics
  - Varies; Archipelago promotes aesthetic improvements by allowing fine customization
- Extensibility
  - Many extensibility mechanisms at different levels

931

Objectives

- Concepts
  - What is visualization?
  - Differences between modeling and visualization
  - What kinds of visualizations do we use?
  - Visualizations and views
  - How can we characterize and evaluate visualizations?
- Examples
  - Concrete examples of a diverse array of visualizations
- Constructing visualizations
  - Guidelines for constructing new visualizations
  - Pitfalls to avoid when constructing new visualizations
  - Coordinating visualizations

932

Constructing New Visualizations

- Developing a new visualization can be expensive both in initial development and maintenance
- Must answer many questions in advance
  - Can I achieve my goals by extending an existing visualization?
  - Can I translate into another notation and use a visualization already available there?
  - How will my visualization augment the existing set of visualizations for this notation?
  - How will my visualization coordinate with other visualizations?
  - (Plus all the evaluation categories we' ve been exploring)

New Visualizations: Guidelines

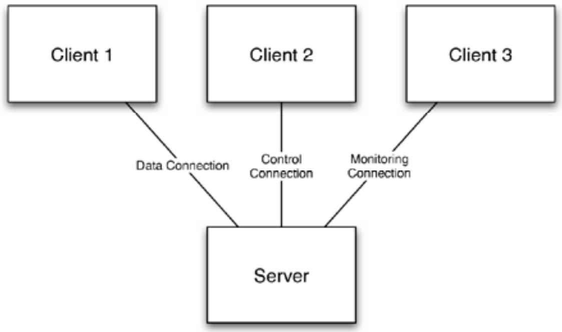
- Borrow elements from similar visualizations
  - Leverages existing stakeholder knowledge
  - Improves comprehensibility
- Be consistent among visualizations
  - Don' t conflict with existing visualizations without a good reason (e.g., developing a domain-specific visualization where the concepts and metaphors are completely different)
- Give meaning to each visual aspect of elements
  - Parsimony is more important than aesthetics
  - Corollary: avoid having non-explicit meaning encoded in visualizations

New Visualizations: Guidelines (cont' d)

- Document the meaning of visualizations
  - Visualizations are rarely self-explanatory
  - Focus on mapping between model and visualization
- Balance traditional and innovative interfaces
  - Stakeholders bring a lot of interaction experience to the table
  - But just because a mechanism is popular doesn' t mean it' s ideal

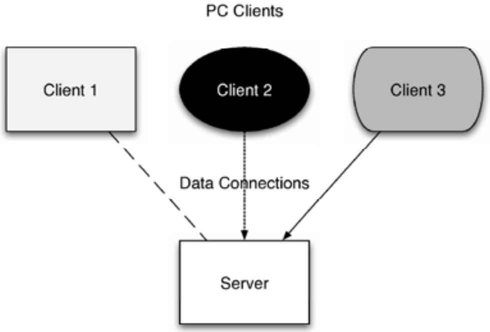
New Visualizations: Anti-Guidelines

- Same Symbol, Different Meaning



New Visualizations: Anti-Guidelines (cont'd)

- Differences without meaning

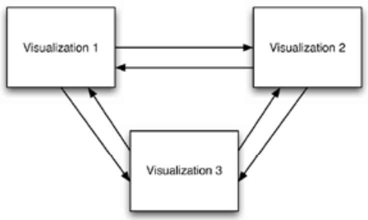


Coordinating Multiple Visualizations

- How do we keep multiple simultaneous visualizations of the same (part of the) architectural model consistent with each other and the model?
  - This is NOT the same as maintaining architectural consistency
  - If something is wrong with the model, this error would be reflected in the visualizations
- Can be made much easier by making simplifying assumptions, e.g.:
  - Only one visualization may operate at a time
  - Only one tool can operate on the model at a time
- But what if we can't simplify like this?

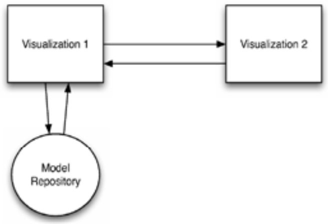
Strategy: Peer-to-Peer Coordination

- Each visualization communicates with each other visualization for updates
  - Has scaling problems
  - Works best for visualizations known *a priori*



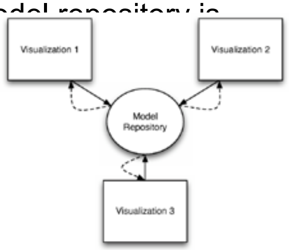
Strategy: Master-Slave

- One visualization is the master and others coordinate through it
- Works best when visualizations are subordinate
  - E.g., a "thumbnail" o main, zoomed-in visuali



Strategy: Pull-based

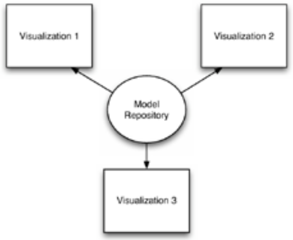
- Visualizations repeatedly poll a model repository for changes
- Potential consistency/staleness problems
- May be necessary if model repository is entirely passive
- May save computing pc



941

Strategy: Push-based

- Visualizations actively notified and update themselves whenever model changes for any reason
- Best for multiple simultaneous visualizations
- Hard to debug, must avo
subtle concurrency cond



Topic 24: Implementing Architectures

Objectives

- Concepts
  - Implementation as a mapping problem
  - Architecture implementation frameworks
  - Evaluating frameworks
  - Relationships between middleware, frameworks, component models
  - Building new frameworks
  - Concurrency and generative technologies
  - Ensuring architecture-to-implementation consistency
- Examples
  - Different frameworks for pipe-and-filter
  - Different frameworks for the C2 style
- Application
  - Implementing Lunar Lander in different frameworks

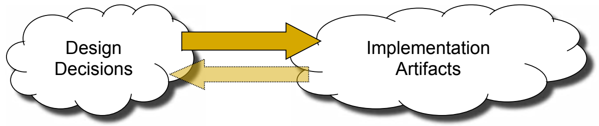
944

Objectives

- Concepts
  - Implementation as a mapping problem
  - Architecture implementation frameworks
  - Evaluating frameworks
  - Relationships between middleware, frameworks, component models
  - Building new frameworks
  - Concurrency and generative technologies
  - Ensuring architecture-to-implementation consistency
- Examples
  - Different frameworks for pipe-and-filter
  - Different frameworks for the C2 style
- Application
  - Implementing Lunar Lander in different frameworks

The Mapping Problem

- Implementation is the one phase of software engineering that is not optional
- Architecture-based development provides a unique twist on the classic problem
  - It becomes, in large measure, a *mapping* activity



- Maintaining mapping means ensuring that our architectural intent is reflected in our constructed systems

Common Element Mapping

- Components and Connectors
  - Partitions of application computation and communication functionality
  - Modules, packages, libraries, classes, explicit components/connectors in middleware
- Interfaces
  - Programming-language level interfaces (e.g., APIs/function or method signatures) are common
  - State machines or protocols are harder to map

Common Element Mapping (cont’d)

- Configurations
  - Interconnections, references, or dependencies between functional partitions
  - May be implicit in the implementation
  - May be externally specified through a MIL and enabled through middleware
  - May involve use of reflection
- Design rationale
  - Often does not appear directly in implementation
  - Retained in comments and other documentation

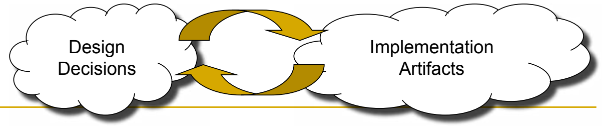
Common Element Mapping (cont' d)

- Dynamic Properties (e.g., behavior):
  - Usually translate to algorithms of some sort
  - Mapping strategy depends on how the behaviors are specified and what translations are available
  - Some behavioral specifications are more useful for generating analyses or testing plans
- Non-Functional Properties
  - Extremely difficult to do since non-functional properties are abstract and implementations are concrete
  - Achieved through a combination of human-centric strategies like inspections, reviews, focus groups, user studies, beta testing, and so on

949

One-Way vs. Round Trip Mapping

- Architectures inevitably change after implementation begins
  - For maintenance purposes
  - Because of time pressures
  - Because of new information
- Implementations can be a source of new information
  - We learn more about the feasibility of our designs when we implement
  - We also learn how to optimize them



950

One-Way vs. Round Trip Mapping (cont' d)

- Keeping the two in sync is a difficult technical and managerial problem
  - Places where strong mappings are not present are often the first to diverge
- One-way mappings are easier
  - Must be able to understand impact on implementation for an architectural design decision or change
- Two way mappings require more insight
  - Must understand how a change in the implementation impacts architecture-level design decisions

951

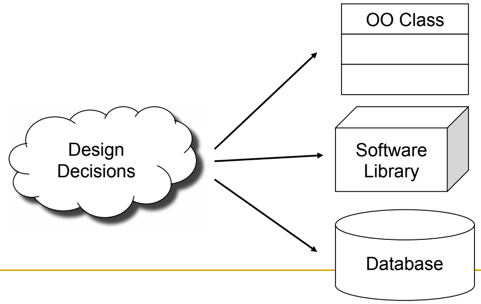
One-Way vs. Round Trip Mapping (cont' d)

- One strategy: limit changes
  - If all system changes must be done to the architecture first, only one-way mappings are needed
  - Works very well if many generative technologies in use
  - Often hard to control in practice; introduces process delays and limits implementer freedom
- Alternative: allow changes in either architecture or implementation
  - Requires round-trip mappings and maintenance strategies
  - Can be assisted (to a point) with automated tools

952

Architecture Implementation Frameworks

- Ideal approach: develop architecture based on a known style, select technologies that provide implementation support for each architectural element

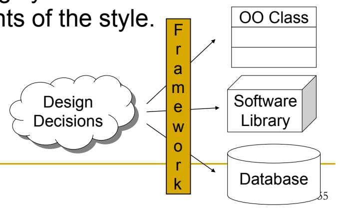


Architecture Implementation Frameworks

- This is rarely easy or trivial
  - Few programming languages have explicit support for architecture-level constructs
  - Support infrastructure (libraries, operating systems, etc.) also has its own sets of concepts, metaphors, and rules
- To mitigate these mismatches, we leverage an *architecture implementation framework*

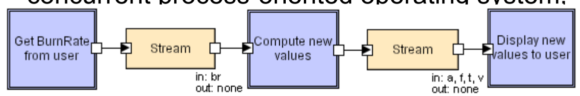
Architecture Implementation Frameworks

- **Definition:** An *architecture implementation framework* is a piece of software that acts as a bridge between a particular architectural style and a set of implementation technologies. It provides key elements of the architectural style *in code*, in a way that assists developers in implementing systems that conform to the prescriptions and constraints of the style.



Canonical Example

- The standard I/O ('stdio') framework in UNIX and other operating systems
  - Perhaps the most prevalent framework in use today
  - Style supported: pipe-and-filter
  - Implementation technologies supported: concurrent process-oriented operating system,



More on Frameworks

- Frameworks are meant to assist developers in following a style
  - But generally do not *constrain* developers from violating a style if they really want to
- Developing applications in a target style does not *require* a framework
  - But if you follow good software engineering practices, you' ll probably end up developing one anyway
- Frameworks are generally considered as underlying infrastructure or substrates from an architectural perspective
  - You won' t usually see the framework show up in an architectural model, e.g., as a component

957

Same Style, Different Frameworks

- For a given style, there is no one perfect architecture framework
  - Different target implementation technologies induce different frameworks
    - stdio vs. iostream vs. java.io
- Even in the same (style/target technology) groupings, different frameworks exist due to different qualitative properties of frameworks
  - java.io vs. java.nio
  - Various C2-style frameworks in Java

958

Evaluating Frameworks

- Can draw out some of the qualitative properties just mentioned
- Platform support
  - Target language, operating system, other technologies
- Fidelity
  - How much style-specific support is provided by the framework?
    - Many frameworks are more general than one target style or focus on a subset of the style rules
  - How much enforcement is provided?

959

Evaluating Frameworks (cont'd)

- Matching Assumptions
  - Styles impose constraints on the target architecture/application
  - Frameworks can induce constraints as well
    - E.g., startup order, communication patterns …
  - To what extent does the framework make too many (or too few) assumptions?
- Efficiency
  - Frameworks pervade target applications and can potentially get involved in any interaction
  - To what extent does the framework limit its slowdown and provide help to improve efficiency if possible (consider buffering in stdio)?

960

Evaluating Frameworks (cont'd)

- Other quality considerations
  - Nearly every other software quality can affect framework evaluation and selection
    - Size
    - Cost
    - Ease of use
    - Reliability
    - Robustness
    - Availability of source code
    - Portability
    - Long-term maintainability and support

Middleware and Component Models

- This may all sound similar to various kinds of middleware/component frameworks
  - CORBA, COM/DCOM, JavaBeans, .NET, Java Message Service (JMS), etc.
- They are closely related
  - Both provide developers with services not available in the underlying OS/language
  - CORBA provides well-defined interfaces, portability, remote procedure call…
  - JavaBeans provides a standardized packaging framework (the bean) with new kinds of introspection and binding

Middleware and Component Models (cont'd)

- Indeed, architecture implementation frameworks *are* forms of middleware
  - There's a subtle difference in how they emerge and develop
  - Middleware generally evolves based on a set of *services* that the developers want to have available
    - E.g., CORBA: Support for language heterogeneity, network transparency, portability
  - Frameworks generally evolve based on a particular *architectural style* that developers want to use
- Why is this important?

Middleware and Component Models (cont'd)

- By focusing on *services*, middleware developers often make other decisions that substantially impact architecture
- E.g., in supporting network transparency and language heterogeneity, CORBA uses RPC
  - But is RPC necessary for these services or is it just an enabling technique?
- In a very real way, middleware induces an architectural style
  - CORBA induces the 'distributed objects' style
  - JMS induces a distributed implicit invocation style
- Understanding these implications is essential for not having major problems when the tail wags the dog!

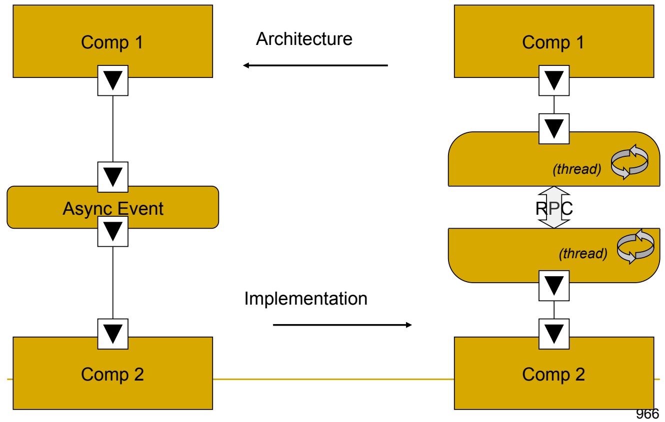
Resolving Mismatches

- A style is chosen first, but the middleware selected for implementation does not support (or contradicts) that style
- A middleware is chosen first (or independently) and has undue influence on the architectural style used
- Strategies
  - Change or adapt the style
  - Change the middleware selected
  - Develop glue code
  - Leverage parts of the middleware and ignore others
  - Hide the middleware in components/connectors

$\left.\begin{array}{l}\text{Develop glue code} \\ \text{Leverage parts of the middleware and ignore others}\end{array}\right\}$ Use the middleware as the basis for a framework

965

Hiding Middleware in Connectors



966

Building a New Framework

- Occasionally, you need a new framework
  - The architectural style in use is novel
  - The architectural style is not novel but it is being implemented on a platform for which no framework exists
  - The architectural style is not novel and frameworks exist for the target platform, but the existing frameworks are inadequate
- Good framework development is extremely difficult
  - Frameworks pervade nearly every aspect of your system
  - Making changes to frameworks often means changing the entire system
  - A task for experienced developers/architects

967

New Framework Guidelines

- Understand the target style first
  - Enumerate all the rules and constraints in concrete terms
  - Provide example design patterns and corner cases
- Limit the framework to the rules and constraints of the style
  - Do not let a particular target application's needs creep into the framework
  - "Rule of three" for applications

968

New Framework Guidelines (cont'd)

- Choose the framework scope
  - A framework does not necessarily have to implement all possible stylistic advantages (e.g., dynamism or distribution)
- Avoid over-engineering
  - Don't add capabilities simply because they are clever or "cool", especially if known target applications won't use them
  - These often add complexity and reduce performance

New Framework Guidelines (cont'd)

- Limit overhead for application developers
  - Every framework induces some overhead (classes must inherit from framework base classes, communication mechanisms limited)
  - Try to put as little overhead as possible on framework users
- Develop strategies and patterns for legacy systems and components
  - Almost every large application will need to include elements that were not built to work with a target framework
  - Develop strategies for incorporating and wrapping these

Concurrency

- Concurrency is one of the most difficult concerns to address in implementation
  - Introduction of subtle bugs: deadlock, race conditions…
  - Another topic on which there are entire books written
- Concurrency is often an architecture-level concern
  - Decisions can be made at the architectural level
  - Done carefully, much concurrency management can be embedded into the architecture framework
- Consider our earlier example, or how pipe-and-filter architectures are made concurrent without direct user involvement

Generative Technologies

- With a sufficiently detailed architectural model, various implementation artifacts can be generated
  - Entire system implementations
    - Requires extremely detailed models including behavioral specifications
    - More feasible in domain-specific contexts
  - Skeletons or interfaces
    - With detailed structure and interface specifications
  - Compositions (e.g., glue code)
    - With sufficient data about bindings between two elements

Maintaining Consistency

- Strategies for maintaining one-way or round-trip mappings
  - Create and maintain traceability links from architectural implementation elements
    - Explicit links in a database, in architectural models, in code comments can all help with consistency checking
  - Make the architectural model part of the implementation
    - When the model changes, the implementation adapts automatically
    - May involve "internal generation"
  - Generate some or all of the implementation from the architecture

Topic 25: Implementation Architectures (II)

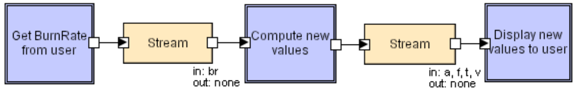
Objectives

- Concepts
  - Implementation as a mapping problem
  - Architecture implementation frameworks
  - Evaluating frameworks
  - Relationships between middleware, frameworks, component models
  - Building new frameworks
  - Concurrency and generative technologies
  - Ensuring architecture-to-implementation consistency
- Examples
  - Different frameworks for pipe-and-filter
  - Different frameworks for the C2 style
- Application
  - Implementing Lunar Lander in different frameworks

Objectives

- Concepts
  - Implementation as a mapping problem
  - Architecture implementation frameworks
  - Evaluating frameworks
  - Relationships between middleware, frameworks, component models
  - Building new frameworks
  - Concurrency and generative technologies
  - Ensuring architecture-to-implementation consistency
- Examples
  - Different frameworks for pipe-and-filter
  - Different frameworks for the C2 style
- Application
  - Implementing Lunar Lander in different frameworks

Recall Pipe-and-Filter



- Components ('filters') organized linearly, communicate through character-stream 'pipes,' which are the connectors
- Filters may run concurrently on partial data
- In general, all input comes in through the left and all output exits from the right

Framework #1: stdio

- Standard I/O framework used in C programming language
- Each process is a filter
  - Reads input from standard input (aka 'stdin')
  - Writes output to standard output (aka 'stdout')
    - Also a third, unbuffered output stream called standard error ('stderr') not considered here
  - Low and high level operations
    - `getchar(...)`, `putchar(...)` move one character at a time
    - `printf(...)` and `scanf(...)` move and format entire strings
  - Different implementations may vary in details (buffering strategy, etc.)

Evaluating stdio

- Platform support
  - Available with most, if not all, implementations of C programming language
  - Operates somewhat differently on OSes with no concurrency (e.g., MS-DOS)
- Fidelity
  - Good support for developing P&F applications, but no restriction that apps have to use this style
- Matching assumptions
  - Filters are processes and pipes are implicit. In-process P&F applications might require modifications
- Efficiency
  - Whether filters make maximal use of concurrency is partially up to filter implementations and partially up to the OS

Framework #2: java.io

- Standard I/O framework used in Java language
- Object-oriented
- Can be used for in-process or inter-process P&F applications
  - All stream classes derive from InputStream or OutputStream
  - Distinguished objects (System.in and System.out) for writing to process' standard streams
  - Additional capabilities (formatting, buffering) provided by creating composite streams (e.g., a Formatting-Buffered-InputStream)

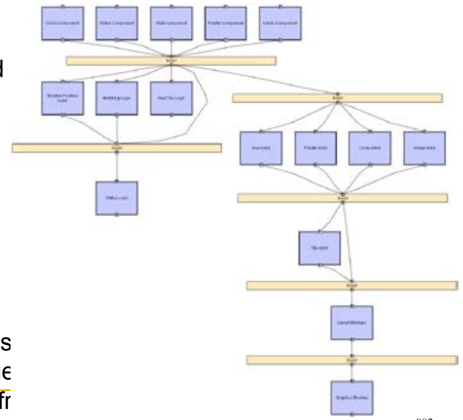
Evaluating java.io

- Platform support
  - Available with all Java implementations on many platforms
  - Platform-specific differences abstracted away
- Fidelity
  - Good support for developing P&F applications, but no restriction that apps have to use this style
- Matching assumptions
  - Easy to construct intra- and inter-process P&F applications
  - Concurrency can be an issue; many calls are blocking
- Efficiency
  - Users have fine-grained control over, e.g., buffering
  - Very high efficiency mechanisms (memory mapped I/O, channels) not available (but are in java.nio)

981

Recall the C2 Style

- Layered style with event-based communication over two-way broadcast buses
- Strict rules on concurrency, dependencies, and so on
- Many frameworks different language alternative Java fr



982

Framework #1: Lightweight C2 Framework

- 16 classes, 3000 lines of code
- Components & connectors extend abstract base classes
- Concurrency, queuing handled at individual comp/conn level
- Messages are request or notification objects

Evaluating Lightweight C2 Framework

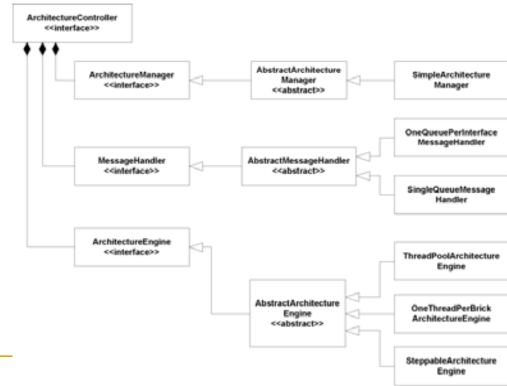
- Platform support
  - Available with all Java implementations on many platforms
- Fidelity
  - Assists developers with many aspects of C2 but does not enforce these constraints
  - Leaves threading and queuing policies up to individual elements
- Matching assumptions
  - Comp/conn main classes must inherit from distinguished base classes
  - All messages must be in dictionary form
- Efficiency
  - Lightweight framework; efficiency may depend on threading and queuing policy implemented by individual elements

984

Framework #2: Flexible C2 Framework

- 73 classes, 8500 lines of code
- Uses interfaces rather than base classes
- Threading policy for application is pluggable
- Message queuing policy is also pluggable

Framework #2: Flexible C2 Framework



Evaluating Flexible C2 Framework

- Platform support
  - Available with all Java implementations on many platforms
- Fidelity
  - Assists developers with many aspects of C2 but does not enforce these constraints
  - Provides several alternative application-wide threading and queuing policies
- Matching assumptions
  - Comp/conn main classes must implement distinguished interfaces
  - Messages can be any serializable object
- Efficiency
  - User can easily swap out and tune threading and queuing policies without disturbing remainder of application code

Objectives

- Concepts
  - Implementation as a mapping problem
  - Architecture implementation frameworks
  - Evaluating frameworks
  - Relationships between middleware, frameworks, component models
  - Building new frameworks
  - Concurrency and generative technologies
  - Ensuring architecture-to-implementation consistency
- Examples
  - Different frameworks for pipe-and-filter
  - Different frameworks for the C2 style
- Application
  - Implementing Lunar Lander in different frameworks

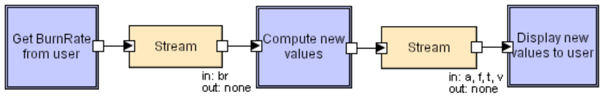
Implementing Pipe and Filter Lunar Lander

- Framework: java.io
- Implementing as a multi-process application
  - Each component (filter) will be a separate OS process
  - Operating system will provide the pipe connectors
- Going to use just the standard input and output streams
  - Ignoring standard error
- Ignoring good error handling practices and corner cases for simplicity

Implementing Pipe and Filter Lunar Lander

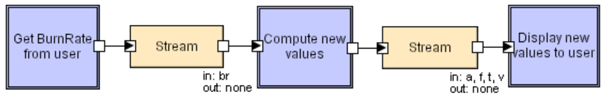
- A note on I/O:
  - Some messages sent from components are intended for output to the console (to be read by the user)
    - These messages must be passed all the way through the pipeline and output at the end
    - We will preface these with a ‘#’
  - Some messages are control messages meant to communicate state to a component down the pipeline
    - These messages are intercepted by a component and processed
    - We will preface these with a ‘%’

Implementing Pipe and Filter Lunar Lander



- First: GetBurnRate component
  - Loops; on each loop:
    - Prompt user for new burn rate
    - Read burn rate from the user on standard input
    - Send burn rate to next component
    - Quit if burn rate read < 0

Implementing Pipe and Filter Lunar Lander

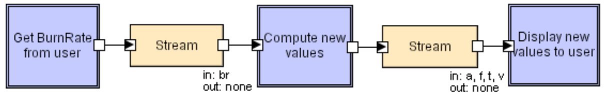


- Second: CalcNewValues Component
  - Read burn rate from standard input
  - Calculate new game state including game-over
  - Send new game state to next component
    - New game state is not sent in a formatted string; that’s the display component’s job

Implementing Pipe and Filter Lunar Lander

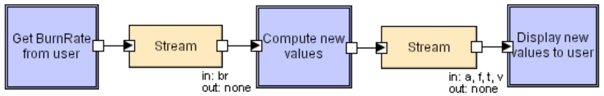
- Third: DisplayValues component
  - Read value updates from standard input
  - Format them for human reading and send them to standard output

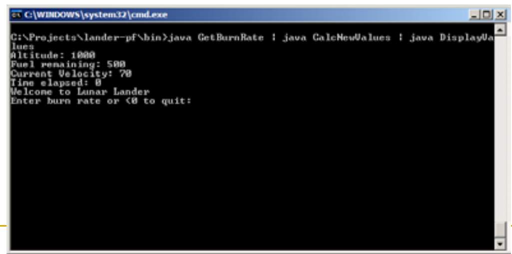
Implementing Pipe and Filter Lunar Lander



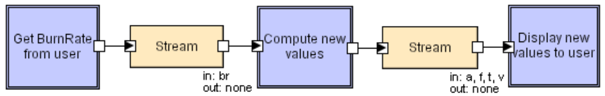
- Instantiating the application
  - java GetBurnRate | java CalcNewValues | java DisplayValues

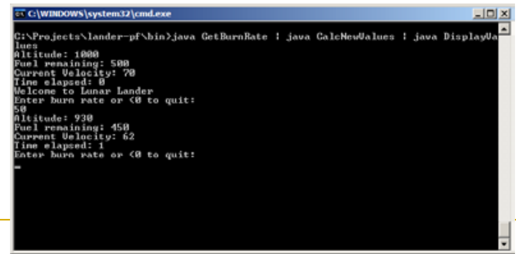
Implementing Pipe and Filter Lunar Lander



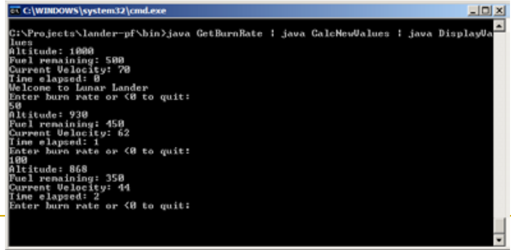


Implementing Pipe and Filter Lunar Lander

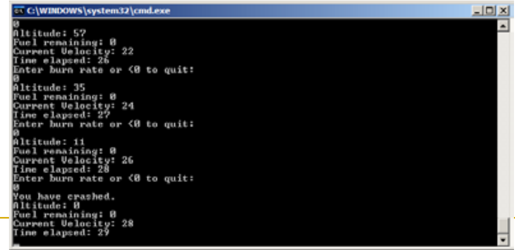




Implementing Pipe and Filter Lunar Lander



Implementing Pipe and Filter Lunar Lander

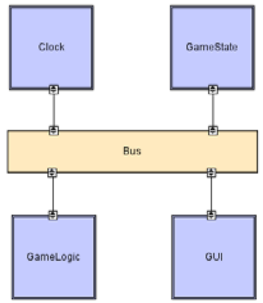


Takeaways

- java.io provides a number of useful facilities
  - Stream objects (System.in, System.out)
  - Buffering wrappers
- OS provides some of the facilities
  - Pipes
  - Concurrency support
    - Note that this version of the application would not work if it operated in batch-sequential mode
- We had other communication mechanisms available, but did not use them to conform to the P&F style
- We had to develop a new (albeit simple) protocol to get the correct behavior

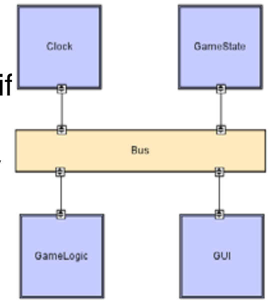
Implementing Lunar Lander in C2

- Framework: Lightweight C2 framework
- Each component has its own thread of control
- Components receive requests or notifications and respond with new ones
- Message routing follows C2 rules
- This is a real-time, clock-driven version of Lunar Lander



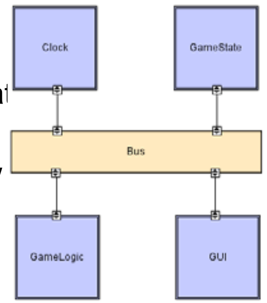
Implementing Lunar Lander in C2 (cont’d)

- First: Clock component
- Sends out a ‘tick’ notif periodically
- Does not respond to any messages



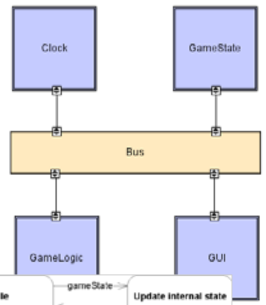
Implementing Lunar Lander in C2

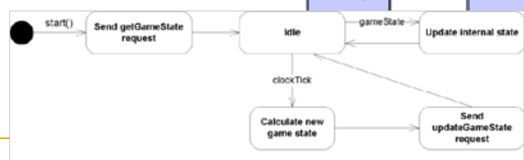
- Second: GameState Component
- Receives request to updat internal state
- Emits notifications of new game state on request or when state changes
- Does NOT compute new state
  - Just a data store



Implementing Lunar Lander in C2

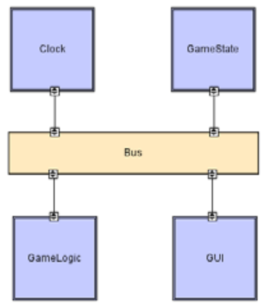
- Third: GameLogic Component
- Receives notifications of game state changes
- Receives clock ticks
  - On clock tick notification, calculates new state and sends request up





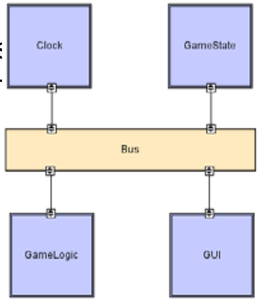
Implementing Lunar Lander in C2

- Fourth: GUI Component
- Reads burn rates from user and sends them up as requests
- Receives notifications of game state changes and formats them to console



Implementing Lunar Lander in C2

- Lastly, main program
- Instantiates and connects all elements of the system



Takeaways

- Here, the C2 framework provides most all of the scaffolding we need
  - Message routing and buffering
  - How to format a message
  - Threading for components
  - Startup and instantiation
- We provide the component behavior
  - Including a couple new threads of our own
- We still must work to obey the style guidelines
  - Not everything is optimal: state is duplicated in Game Logic, for example

Topic 26: Software Architecture: Being Creative

Yesterday – 1990's

- Box and lines – ad-hoc
- No analysis of consistency of specification
- No checking of architecture-implementation consistency
- Importance of architecture in industry
  - recognition of a shared repository of methods, techniques, patterns and idioms (engineering)
  - exploiting commonalities in specific domains to provide reusable frameworks for product families

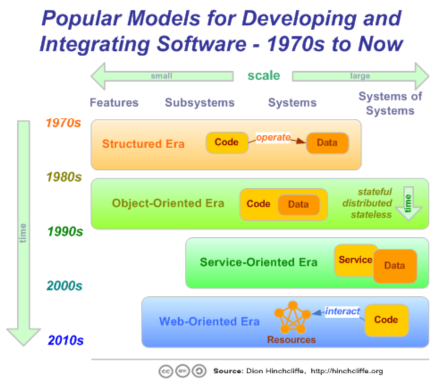
A Few Years Later

- Architecting A first class activity in software development life cycle
- Architecture Description Languages (ADLs)
- Product Lines and Standards
- Codification and Dissemination

Today

- Cloud
- Big Data
- Internet of Things
- …

A Short History of Software



The classical ways of describing architecture

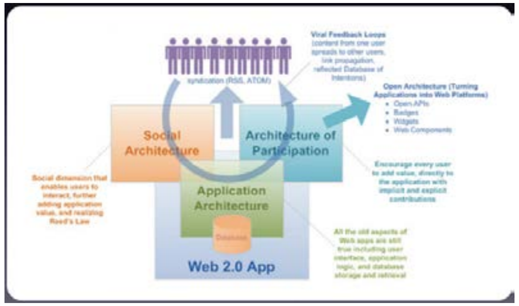
- Functional/logic view
- Code/module view
- Development/structural view
- Concurrency/process/thread view
- Physical/deployment view
- User action/feedback view
- Data view

Today's Software Architectures Are Also Extremely Sophisticated

- Highly distributed and federated
- have a social architecture
- Built from cutting edge ingredients
  Example: http://clickatell.com
- Have to scale globally
- Set with expectations that are very high for functionality and low for the cost to develop/own new solutions
- created with productivity-oriented design & development platforms
- Must co-exist with many other technologies, standards, and architectures

Integrating with 3rd party suppliers live on the Web as well as being a 3rd party supplier is the name of the game circa-2009

The application "stack" is richer now



Web-Oriented Architecture

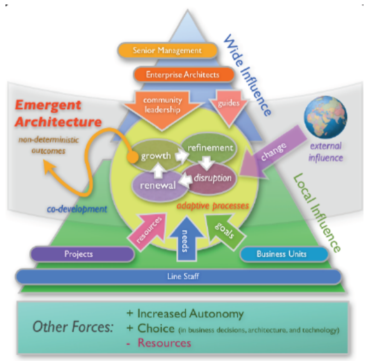
Recent technological innovations coming primarily from the online world

- Cloud computing
  - Utility/grid/Platform-as-a-service
- Non-relational databases
  - S3, CouchDB, GAE Datastore, Drizzle, etc.
- New "productivity-oriented" platforms
  - RIA: Flex/AIR, JavaFX
  - Stacks: Rails, CakePHP, Grails, GAE, iPhone, etc.
- Web-Oriented Architecture

Changes to the processes that create architecture

- Increasing move to *assembly and integration* over development of new code
- Perpetual Beta and "extreme" agile
- Community-based development and "commercial source"
  - Product Development 2.0

Emergent Architecture



Tenets of Emergent Architecture

- Community-driven architecture
- Autonomous stakeholders
- Adaptive processes
- Resource constraints
- Decentralized solutions
- Emergent outcomes

Benefits

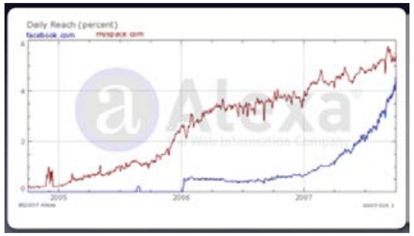
- Dynamic response and adaptation to change
- Architecture supported and driven widely by local users
- Less waste
- More access to opportunity
- Better fit to business needs

Motivations for Open Supply Chains

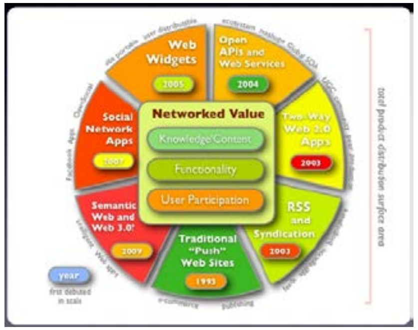
- Increase reach and head off competition
- Tap into innovation
- Grow external investment
- Cost-effectively scale business relationships

Going from 10s to thousands of integrated partners

Open Platform vs. Closed Platform



New Distribution Models



2.0 models are beginning to transform everything

- Product Development
- Marketing and Advertising
- Operations
- Customer Service

Challenges to Transitioning to New Architectural Modes

- Innovator's Dilemma
  - *"How do we disrupt ourselves before our competition does?"*
- Not-Invented Here
- Overly fearful of failure
- Deeply ingrained classical software culture
- Low level of 2.0 literacy

1025

Summary

- **Creativity should be regarded as a key to developing a software architecture.**
- **The challenge is how to reconcile objective precision with subjective ambiguity.**

Discussion Questions

- **Topic 1**
  - **What is software architecture, in your own words?**
  - **What do you think of Brooks' "Surgical Team"?**
  - **How did Fred Brooks Jr. describe the role of the architect in his "The Mythical Man-Month"?**
  - **What have you learnt from David Parnas, for software development?**
- **Topic 2**
  - **What is your explanation of ABC?**
  - **How do you plan to become a good software architect, referring to the Architectural Business Cycle?**
  - **What are the steps in the Software Architecture Analysis Method (SAAM)?**
- **Topic 3 - Software Architecture and the Built Environment**
  - **What does the software learn from built environment?**
  - **What are the six S's of shearing layers?**
  - **What are the Lessons for Software Architecture?**

1027

- **Topic 4**
  - **Compare and contrast the 'Masterplan' and 'Piecemeal Growth' views of Software Architecture.**
  - **Explain design pattern in your own words.**
  - **What are the relationships between pattern and pattern languages?**
- **Topic 5**
  - **What role does ADL play in software architecture?**
  - **Please give an definition to ADL.**
  - **What are the basic elements of an ADL?**
- **Topic 6**
  - **What is an 'architectural style' and what is an 'architectural pattern'?**
  - **What is the Blackboard Architecture Style?**
  - **What is an Attribute Based Architectural Style (ABAS)?**
- **Topic 7**
  - **According to Frank Buschmann et al.'s Patterns of Software Architecture, into which three levels that the patterns emerging during the software development can be divided?**
  - **Could you give an example of an architectural pattern?**
  - **Explain the following architectural patterns: MVC, Layers.**
- **Topic 8**
  - **What is the purpose of DSSA?**
  - **What is DSSA and what does DSSA consist of?**
  - **What are the general steps solving problems using DSSAs?**

1028

- Topic 9
  - What is Dan Bredemeyer's Software Architecture Model?
  - What is Bredemeyer's suggested architecting process, and its elemental steps?
  - How to ensure a good architecture be created?
- Topic 10
  - What is the building block of UML?
  - What is the typical architectural views(4+1 views) adopted by UML?
  - What are the characteristics of the UML software development life cycle?
- Topic 11
  - What's the biggest single problem for Component Based Development?
  - What's the suggested method to solve the problems with component interfaces?
  - what does an architectural approach to CBD require?
- Topic 12
  - What is software architecture evaluation, and what are the benefits?
  - Explain the preconditions, activities and outputs of architecture evaluation.
  - What are the problems with current evaluation approaches?

1029

- Topic 13
  - How could we understand that objects can be thought of architectural spaces?
  - What's the significance of interfaces for architecture?
  - What is a levelised system? How to recognise levelised structures?
- Topic 14
  - What's the purpose of the techniques such as Java RMI, CORBA, Microsoft Com/DCom etc.?Is there anything in common among them?
  - Describe the conception of middleware.
  - What are the functions of an ORB(Object Request Broker)?
- Topic 15
  - Explain the basic idea of MDA and its benefits.
  - What are the three types of models that MDA introduced?
  - Explain the process of development using MDA.

1030

- Topic 16
  - How to understand the relationships between architecture and process?
  - What are the underlying notions and steps of the Architectural Tradeoff Analysis Method (ATAM)?
  - What are the steps in the SCRUM process?
- Topic 17
  - Explain the conception of legacy systems and try to understand the challenges and chances they will bring to us.
  - Why reverse-architecting and the path to achieve it?
  - What's the idea of architecture exploration and what are the challenges we are facing in this step?
- Topic 18
  - What roles for architecture today?
  - How to understand that architecting is becoming a first-class activity in software development cycle?
  - What would be tomorrow's trends of software architecture?

1031

- Topic 19
- Topic 20
- Topic 21
- Topic 22
- Topic 23
- Topic 24
- Topic 25
- Topic 26

1032