BITS Pilani

Pilani|Dubai|Goa|Hyderabad

SS ZG653: Software Architecture

Introduction to the Course

Instructor: Prof. Santonu Sarkar

About the Instructor

- Professor at Dept of CSIS, BITS Pilani K.K.Birla Goa Campus
- Bachelors, Masters, PhD in Computer Science
- Nearly 20 years of experience in Applied Research, Product and Application Development, Consulting and Teaching
- Broad Area: Software Engineering
  - Software Architecture, Software Design, Software Modularity, Reengineering
  - Software Development Challenges in Cloud and Heterogenous Computing Platform

About the Course

- To study software architecture (we will simply call architecture in this context)
  - What is the abstraction of the software, and how to create, and how to represent
  - What are the relationships between various entities
  - How architectural principles are used during software system analysis and design.
- To study about the role of architecture patterns in software design
- To study about the applicability of design patterns in software design

Course Objective

- To have sound understanding of software architecture
  - and remove misconceptions
  - the current state of the discipline of Software Architecture
  - Know the way in which architecture can impact design
  - Know various architectural styles, views
  - Importance of nonfunctional requirements, or quality attributes of a system
- Apply the concept
  - Design new systems in principled ways, using well-understood architectural paradigms
  - Present concrete examples of actual system architectures that can serve as model for new designs
- Evaluate
  - Evaluate designs of existing software systems from an architecture perspective

Study Material

- Text Books
  - Len Bass et al, Software Architecture in Practice, Pearson, Third (or Second) Edition, ISBN 9789332502307
  - F. Buschmann et al, Pattern Oriented Software Architecture – Volume1, Wiley, 1996

Study Material contd…

- References
  - R. N. Taylor et al, Software Architecture: Foundations, Theory, and Practice, John Wiley & Sons, 2009
  - Mary Shaw & David Garlan, Software Architecture – Perspectives on an Emerging Discipline, PHI, 1996.
  - Stephen T. Albin, The Art of Software Architecture, Wiley Dreamtech, 2003.
  - Gamma, E. et. Al. Design Patterns: Elements of Reusable Object Oriented Software, Addison Wesley, 1995

Teaching and Evaluation

- Lectures: 16 + 2 Review
- Exams: 2
  - Midterm: 35%
  - Final: 50%
- Quizzes: 15%

- Midterm Exam
  - Closed Book and notes
- Final Exam
  - Open Book and notes
- The exam solutions/ answers are expected to be of Masters Level with crisp, to-the-point, concise, proper, neat and readable presentation

Detailed Schedule

Lecture#	Topics
1	Software Architecture and its Importance
2	Many perspectives of Software Architecture Introducing Quality Attributes
3-5	Role a few quality attributes in details – Architectural Tactics • Availability, Interoperability, Modifiability • Performance, Security, Testability
6-7	Object-oriented concepts and UML • Classes, Objects, Encapsulation, Polymorphism, Inheritance and their representation in UML • Class diagram, Sequence diagram, Class Responsibility and Collaboration (CRC) Cards
8	Styles and Patterns • Concept, Categories and descriptions • Architecture style - Layering
Review Session	
Syllabus for Mid-Semester Test (Closed Book) : Topics covered in S. No. 1 to 8	

Detailed Schedule contd…

Lecture#	Topics
10-12	Architecture Style: Blackboard style, Pipe and Filter style Distributed System Style: Broker Architecture Interactive System Style Model-View-Controller Adaptable System Microkernel, Reflection
13-17	Design Pattern • Components of a typical design pattern, and different categories of design patterns Behavioral Category • Iterator Pattern Behavioral Category • Observer, Strategy, Visitor, Command Structural Category • Adapter, Decorator, Composite, Proxy Creational Category • Factory Pattern, Factory Method, Singleton
Review Session	
Syllabus for Comprehensive Examination (Open Book): All topics given in the Plan.	

SS ZG653 (RL1.2): Software Architecture

A Brief History of Software Architecture

BITS Pilani

Pilani|Dubai|Goa|Hyderabad

Instructor: Prof. SantonuSarkar

Informally what is meant by (Software) Architecture

- Essentially a blueprint of a software system that helps **stakeholders** to understand how the system would be once it is implemented
- What's should be there in this blueprint?
  - A description at a higher level of abstraction than objects and lines of codes

  So that

  - Stakeholders <u>understand</u> and <u>reason about</u> without getting lost into a sea of details

Who are Stakeholders?

A complex software has multiple stakeholders who expect certain features of the software

Stakeholder	Area of Concern
Chief Technologist	• Does it adhere to organization standards ?
Database Designer	• What information to be stored, where, how, access mechanism??? • Information security issues?
Application Development team	• How do I implement a complex scenario? • How should I organize my code? • How do I plan for division of work?
Users/Customers	• Does it perform as per my requirement? • What about the cost/budget? • Scalability, performance and reliability of the system? • How easy it is to use? • Is it always available?
Infrastructure Manager	• Performance and scalability • Idea of system & network usage • Indication of hardware and software cost, scalability, deployment location • Safety and security consideration • Is it fault tolerant-crash recovery & backup
Release & Configuration Manager	• Build strategy • Code management, version control, code organization
System Maintainer	• How do I replace of a subsystem with minimal impact ? • How fast can I diagnosis of faults and failures and how quickly I can recover?

Why Architecture needs to be described?

Any Large Software Corporation

- Hundreds of concurrent projects being executed
  - 10-100 team size
- Projects capture requirements, there are architects, and large Development teams
- Architect start with requirements team & handover to Development teams

- Each stakeholder has his own interpretation of the systems
  - Sometimes no understanding at all
  - Architect is the middleman who co-ordinates with these stakeholders
- How will everyone be convinced that his expectations from the system will be satisfied?
- Even when the architect has created the solution blueprint, how does she handover the solution to the developers?
- How do the developers build and ensure critical aspects of the system?
- Misunderstanding leads to incorrect implementation
  - Leads to 10 times more effort to fix at a later stage

Software Architecture Definition

- No unique definition though similar…
  - (look at http://www.sei.cmu.edu/architecture/start/glossary/classicdefs.cfm)
- .. "**structure** or structures of the system, which comprise **software elements**, the **externally visible properties** of those elements, and the **relationships** among them" (Bass, Clements and Kazman, Software Architecture in Practice, 2nd edition)
- "description of elements from which systems are built, **interactions** among those elements, **patterns** that guide their **composition**, and **constraints** on these patterns. In general, a particular system is defined in terms of a collection of **components** and interactions among these components"
  Shaw and Garlan "*Software Architecture: Perspectives on an Emerging Disciplines*"
- "description of the **subsystems** and **components** of a software system and the **relationship** between them. Subsystems and components are typically specified in different **views** to show the relevant **functional** and **nonfunctional** properties of a software system"
  F. Buschmann et al, Pattern Oriented Software Architecture

Is this Architecture

What we understand

- The system has 5 elements
- They are interconnected
- One is on the top of another

Typically we describe architecture as a collection of diagrams like this

What's Ambiguous?

- Visible responsibilities
  - What do they do?
  - How does their function relate to the system
  - How have these elements been derived, is there any overlap?
- Are these processes, or programs
  - How do they interact when the software executes
  - Are they distributed?
- How are they deployed on a hardware
- What information does the system process?

What's Ambiguous?

- Significance of connections
  - Signify control or data, invoke each other, synchronization
  - Mechanism of communications
- Significance of layout
  - Does level shown signify anything
  - Was the type of drawing due to space constraint

What should Architecture description have?

- A structure describing
  - Modules
    - Services offered by each module
    - and their interactions- to achieve the functionality
  - Information/data modeling
  - Achieving quality attributes
  - Processes and tasks that execute the software
  - Deployment onto hardware
  - Development plan

What should Architecture description have?.....

- A behavioral description
  - describing how the structural elements execute "important" and "critical" scenarios
    - E.g. how does the system authenticates a mobile user
    - How does the system processes 1 TB of data in a day
    - How does it stream video uninterruptedly during peak load
  - These scenarios are mainly to implement various quality attributes

Architecture of Windows

https://http://blogs.msdn.com/b/hanybarakat/archive/2007/02/25/deeper-into-windows-architecture.aspx

Architecture of Android

http://www.techotopia.com/index.php/An_Overview_of_the_Android_Architecture

BITS Pilani

Pilani|Dubai|Goa|Hyderabad

SS ZG653 (RL 1.3): Software Architecture

Architecture Styles and Views

Instructor: Prof. SantonuSarkar

Jan 10, 2015

Architecture Styles

- Architecture style first proposed by Shaw and Garlan– synonymous to “architecture pattern”
  - A set of element types (what the element does- data store, compute linear regression function)
  - A set of interaction types (function call, publish-subscribe)
  - Topology indicating interactions and interaction types
  - Constraints
  - Also known as architectural pattern
- We shall cover some of these patterns in details

Views and Architectural Structure

- Since architecture serves as a vehicle for communication among stakeholders
  - And each stakeholder is interested about different aspects of the system
  - It is too complex to describe, understand and analyze the architecture using one common vocabulary for all stakeholders
    - Essentially it needs to be described in a multi-dimensional manner
- View based approach
  - Each view represents certain architectural aspects of the system, created for a stakeholder
  - All the views combined together form the consistent whole
- A Structure is the underlying part of a view- essentially the set of elements, and their properties
  - A view corresponding to a structure is created by using these elements and their inter-relationships

Reference Model and Reference Architecture

- A reference model
  - Decomposes the functionality into a set of smaller units
  - How they interact and share data
  - These units co-operatively implement the total functionality
- A reference architecture
  - Derived from the reference model
  - Concrete software elements, mapped to the units of the reference model, that implement the functionality

Inter-relationships

- Not architecture by itself!!

- Actual software system blueprint derived from requirement
- Contains design decisions
- Describes how it is deployed
- Addresses Quality of Service concerns

Benefits of Software Architecture

1. Every stakeholder should understand "unambiguously" what the blueprint is
   - Standard approach, vocabulary, output
   - Common language for communication

2. Streamlining work assignments for multiple teams
   - Avoiding information loss, enforcing traceability

3. Design decisions are made early
   - Quicker to evaluate these decisions and correct it rather than discovering it later (10 – 100 times more costly)
   - Early analysis of QoS and evaluation of architecture
   - Early analysis of meeting quality requirements and compromise between different QoS requirements
   - Early prototyping of important aspects quickly
   - More accurate cost and schedule estimation

4. Improve speed of development
   - Reuse
     - Helps in building a large product line faster by sharing common architecture
     - From one implementation to another similar implementation
   - Based on the architecture, one can quickly decide build-vs –use external components
   - Tool that can automate part of development, testing

Three Structures will be covered

- Module Structure
  - How is the system to be structured as a set of code units (modules)?
- Component-and-connector structures
  - How is the system to be structured as a set of elements that have runtime behavior (components) and interactions (connectors)
  - What are major executing components and how do they interact
- Allocation structures
  - How is the system to relate to non-software structures in it's environment (CPU or cluster of CPUs, File Systems, Networks, Development Teams …)

In Bits and Pieces (Unfortunately)

TOGAF, Zachmann
SEI ATAM, SAAM
ISO 10746RM-ODP
IBM enterprise architecture standard
C4ISR
US Treasury Architecture Development process

Architecture description standards
- ISO 10746 RM-ODP
- IEEE 1471,
- RUP 4+1 views
- ISO/IEC 9126 – software quality

Architecture modeling
- UML and MDA EDOC
- Architecture description language

Behavior modeling
- BPML, BPELWS,
- Statechart, CSP, Petri-net

Network Queuing theory, MVA
Markov modeling for availability analysis
Formal verification techniques
Rule based system for applying architecture heuristics
Engineering best practices

Tools & Systems

* Many open source tools – Acme, XADL
IBM Rational System Architect
Many Quality of service analysis tools

Patterns
- Industry specific
- Insurance Reference Architecture: IAA
- IBM e-Business Architecture patterns

Architecture Style

Thank You

BITS Pilani

Pilani|Dubai|Goa|Hyderabad

SS ZG653 (RL 2.1): Software Architecture

Software Structure and Views

Instructor: Prof. SantonuSarkar

Jan 11, 2015

Views and Architectural Structure- Recap..

- Architecture is a set of Views
  - Each view represents certain architectural aspects of the system, created for a stakeholder
  - All the views combined together form the consistent whole
- A Structure is the underlying part of a view- essentially the set of elements, and their properties
  - A view corresponding to a structure is created by using these elements and their inter-relationships

Many Views exist

- Rational Unified Process/Kruchten 4+1 view (uses UML notations to describe these views)
- Siemens architecture framework- Conceptual, Module, Code, Execution views
- C4ISR framework – Operational, system and technical
- Classical approach – Data flow and control flow views
- RM-ODP (suitable for distributed system development) – 5 viewpoints

Three Structures will be covered

- Module Structure
  - How is the system to be structured as a set of functional units (modules)?
- Component-and-connector structures
  - Here component means a computation unit at runtime
  - Connector is the communication channel between the components
  - Models parallel execution
- Allocation structures
  - How is the system to relate to non-software structures in it's environment (CPU or cluster of CPUs, File Systems, Networks, Development Teams …)

Software Structures

- Module Structure includes
  - Decomposition
    - Sub-modules
    - All sub-mobiles combined together is the module
  - Uses
    - A modules uses the functionality of another module for its behavior
  - Layered
    - Hierarchical organization with restrictions that layer n uses the service of layer n-1
  - Class or generalization
    - Similar to OO concept

Component and Connectors

- Client-Server
  - Components are clients and servers and connectors are how they interact
- Concurrency
  - Opportunities of parallelism, where connectors are logical thread of execution dependency
- Process, or communicating processes
  - Components that are processes and connectors are how they communicate
- Shared data, or repository
  - Components have data store, and connectors describe how data is created, stored, retrieved

Allocation

- Deployment
  - Units are software (processes from component-connector) and hardware processors
  - Relation means how a software is allocated or migrated to a hardware
- Implementation
  - Units are modules (from module view) and connectors denote how they are mapped to files, folders
- Work assignment
  - Assigns responsibility for implementing and integrating the modules to people or team

Architectural Structures

Software Structure	Relations	Useful For
Decomposition	Is a sub-module of	Resource allocation and project structuring; information hiding, encapsulation; configuration control
Uses	Requires the correct presence of	Engineering subsets; engineering extensions
Layered	Requires the correct presence of; uses the services of; provides abstraction to	Incremental development; implementing systems on top of “virtual machines” portability
Class	Is an instance of; shares access methods of	In object-oriented design systems, producing rapid most-alike implementations from a common template

Architectural Structures

Software Structure	Relations	Useful For
Client-Server	Communicates with; depends on	Distributed operation; separation of concerns; performance analysis; load balancing
Process	Runs concurrently with; may run concurrently with; excludes; precedes; etc.	Scheduling analysis; performance analysis
Concurrency	Runs on the same logical thread	Identifying locations where resource contention exists, where threads may fork, join, be created or be killed
Shared Data	Produces data; consumes data	Performance; data integrity; modifiability

Architectural Structures

Software Structure	Relations	Useful For
Deployment	Allocated to; migrates to	Performance, availability, security analysis
Implementation	Stored in	Configuration control, integration, test activities
Work assignment	Assigned to	Project management, best use of expertise, management of commonality

Which Structure to Choose?

- Many opinions exist
- We will consider 4+1 view. This has been institutionalized as Rational Unified Process of Architecture description

End-user
- concerned with functionality
- essentially module structure

Infrastructure team
- concerned with topology, communication, security
- essentially allocation (deployment). Other people also call it as deployment view

Use-case- Binds all the four views together and describes the behavior of the system in a semi-structured text

Architect/Integrator
- concerned with performance, scalability..
- essentially component-connector

Development team
- concerned with software management
- essentially allocation (implementation and work-assignment)

BITS Pilani

Pilani|Dubai|Goa|Hyderabad

SS ZG653 (RL 3.1): Software Architecture

Quality classes and attribute, quality attribute scenario and architectural tactics

Instructor: Prof. SantonuSarkar

Jan 11, 2015

A step back

- What is functionality?
  - Ability of the system to fulfill its responsibilities
- Software Quality Attributes- also called non-functional properties
  - Orthogonal to functionality
  - is a constraint that the system must satisfy while delivering its functionality
- Design Decisions
  - A constraint driven by external factors (use of a programming language, making everything service oriented)

Consider the following requirements

- User interface should be easy to use
  - Radio button or check box? Clear text? Screen layout? --- NOT architectural decisions
- User interface should allow redo/undo at any level of depth
  - Architectural decision
- The system should be modifiable with least impact
  - Modular design is must – Architectural
  - Coding technique should be simple – not architectural

- Need to process 300 requests/sec
  - Interaction among components, data sharing issues--architectural
  - Choice of algorithm to handle transactions -- non architectural

Quality Attributes and Functionality

- Any product (software products included) is sold based on its functionality – which are its features
  - Mobile phone, MS-Office software
  - Providing the desired functionality is often quite challenging
    - Time to market
    - Cost and budget
    - Rollout Schedule
- Functionality DOES NOT determine the architecture. If functionality is the only thing you need
  - It is perfectly fine to create a monolithic software blob!
  - You wouldn't require modules, threads, distributed systems, etc.

Examples of Quality Attributes

- Availability
- Performance
- Security
- Usability
- Functionality
- Modifiability
- Portability
- Reusability
- Integrability
- Testability

- The success of a product will ultimately rest on its Quality attributes
  "Too slow!"-- performance
  "Keeps crashing!" --- availability
  "So many security holes!" --- security
  "Reboot every time a feature is changed!" --- modifiability
  "Does not work with my home theater!" --- integrability

- Needs to be achieved throughout the design, implementation and deployment
- Should be designed in and also evaluated at the architectural level
- Quality attributes are NON-orthogonal
  - One can have an effect (positive or negative) on another
  - Performance is troubled by nearly all other. All other demand more code where-as performance demands the least

Defining and understanding system quality attributes

- Defining a quality attribute for a system
  - System should be modifiable --- vague, ambiguous
- How to associate a failure to a quality attribute
  - Is it an availability problem, performance problem or security or all of them?
- Everyone has his own vocabulary of quality
- ISO 9126 and ISO 25000 attempts to create a framework to define quality attributes

Three Quality Classes

System Quality	Business Quality	Quality of Architecture
Availability	Time to market	Conceptual Integrity
Modifiability	Cost and benefit	Correctness
Performance	Project lifetime	completeness
Security	Targeted market	Buildability
Testability	Rollout schedule	
Usability	Legacy integration	

- We will consider these attributes
- We will use "**<u>Quality Attribute Scenarios</u>**" to characterize them
  - which is a quality attribute specific requirement

Quality Attribute Scenario

Architectural Tactics

- To achieve a quality one needs to take a design decision- called Tactic
  - Collection of such tactics is **architectural strategy**
  - A pattern can be a collection of tactics

Quality Design Decisions

- To address a quality following 7 design decisions need to be taken
  - Allocation of responsibilities
  - Coordination
  - Data model
  - Resource Management
  - Resource Binding
  - Technology choice

Quality Design Decisions

- Responsibility Allocation
  - Identify responsibilities (features) that are necessary for this quality requirement
  - Which non-runtime (module) and runtime (components and connectors) should address the quality requirement
- Coordination
  - Mechanism (stateless, stateful…)
  - Properties of coordination (lossless, concurrent etc.)
  - Which element should and shouldn't communicate
- Data Model
  - What's the data structure, its creation, use, persistence, destruction mechanism
  - Metadata
  - Data organization
- Resource management
  - Identifying resources (CPU, I/O, memory, battery, system lock, thread pool..) and who should manage
  - Arbitration policy
  - Find impact of what happens when the threshold is exceeded
- Binding time decision
  - Use parameterized makefiles
  - Design runtime protocol negotiation during coordination
  - Runtime binding of new devices
  - Runtime download of plugins/apps
- Technology choice

Business Qualities

Business Quality	Details
Time to Market	•Competitive Pressure – short window of opportunity for the product/system •Build vs. Buy decisions •Decomposition of system – insert a subset OR deploy a subset
Cost and benefit	•Development effort is budgeted •Architecture choices lead to development effort •Use of available expertise, technology •Highly flexible architecture costs higher
Projected lifetime of the system	•The product thatneeds to survive for longer time needs to be modifiable, scalable, portable •Such systems live longer; however may not meet the time-to-market requirement
Targeted Market	•Size of potential market depends on feature set and the platform •Portability and functionality key to market share •Establish a large market; a product line approach is well suited
Rollout Schedule	•Phased rollouts; base + additional features spaced in time •Flexibility and customizability become the key
Integration with Legacy System	•Appropriate integration mechanisms •Much implications on architecture

Architectural Qualities

Architectural Quality	Details
Conceptual Integrity	•Architecture should do similar things in similar ways •Unify the design at all levels
Correctness and Completeness	•Essential to ensure system's requirements and run time constraints are met
Build ability	•Implemented by the available team in a timely manner with high quality •Open to changes or modifications as time progresses •Usually measured in cost and time •Knowledge about the problem to be solved

BITS Pilani

Pilani|Dubai|Goa|Hyderabad

SS ZG653 (RL 2.1): Software Architecture

Software Structure and Views

Instructor: Prof. SantonuSarkar

Jan 11, 2015

Views and Architectural Structure- Recap..

- Architecture is a set of Views
  - Each view represents certain architectural aspects of the system, created for a stakeholder
  - All the views combined together form the consistent whole
- A Structure is the underlying part of a view- essentially the set of elements, and their properties
  - A view corresponding to a structure is created by using these elements and their inter-relationships

Many Views exist

- Rational Unified Process/Kruchten 4+1 view (uses UML notations to describe these views)
- Siemens architecture framework- Conceptual, Module, Code, Execution views
- C4ISR framework – Operational, system and technical
- Classical approach – Data flow and control flow views
- RM-ODP (suitable for distributed system development) – 5 viewpoints

Three Structures will be covered

- Module Structure
  - How is the system to be structured as a set of functional units (modules)?
- Component-and-connector structures
  - Here component means a computation unit at runtime
  - Connector is the communication channel between the components
  - Models parallel execution
- Allocation structures
  - How is the system to relate to non-software structures in it's environment (CPU or cluster of CPUs, File Systems, Networks, Development Teams …)

Software Structures

- Module Structure includes
  - Decomposition
    - Sub-modules
    - All sub-mobiles combined together is the module
  - Uses
    - A modules uses the functionality of another module for its behavior
  - Layered
    - Hierarchical organization with restrictions that layer n uses the service of layer n-1
  - Class or generalization
    - Similar to OO concept

Component and Connectors

- Client-Server
  - Components are clients and servers and connectors are how they interact
- Concurrency
  - Opportunities of parallelism, where connectors are logical thread of execution dependency
- Process, or communicating processes
  - Components that are processes and connectors are how they communicate
- Shared data, or repository
  - Components have data store, and connectors describe how data is created, stored, retrieved

Allocation

- Deployment
  - Units are software (processes from component-connector) and hardware processors
  - Relation means how a software is allocated or migrated to a hardware
- Implementation
  - Units are modules (from module view) and connectors denote how they are mapped to files, folders
- Work assignment
  - Assigns responsibility for implementing and integrating the modules to people or team

Architectural Structures

Software Structure	Relations	Useful For
Decomposition	Is a sub-module of	Resource allocation and project structuring; information hiding, encapsulation; configuration control
Uses	Requires the correct presence of	Engineering subsets; engineering extensions
Layered	Requires the correct presence of; uses the services of; provides abstraction to	Incremental development; implementing systems on top of “virtual machines” portability
Class	Is an instance of; shares access methods of	In object-oriented design systems, producing rapid most-alike implementations from a common template

Architectural Structures

Software Structure	Relations	Useful For
Client-Server	Communicates with; depends on	Distributed operation; separation of concerns; performance analysis; load balancing
Process	Runs concurrently with; may run concurrently with; excludes; precedes; etc.	Scheduling analysis; performance analysis
Concurrency	Runs on the same logical thread	Identifying locations where resource contention exists, where threads may fork, join, be created or be killed
Shared Data	Produces data; consumes data	Performance; data integrity; modifiability

Architectural Structures

Software Structure	Relations	Useful For
Deployment	Allocated to; migrates to	Performance, availability, security analysis
Implementation	Stored in	Configuration control, integration, test activities
Work assignment	Assigned to	Project management, best use of expertise, management of commonality

Which Structure to Choose?

- Many opinions exist
- We will consider 4+1 view. This has been institutionalized as Rational Unified Process of Architecture description

End-user
- concerned with functionality
- essentially module structure

Infrastructure team
- concerned with topology, communication, security
- essentially allocation (deployment). Other people also call it as deployment view

Use-case- Binds all the four views together and describes the behavior of the system in a semi-structured text

Architect/Integrator
- concerned with performance, scalability..
- essentially component-connector

Development team
- concerned with software management
- essentially allocation (implementation and work-assignment)

BITS Pilani

Pilani|Dubai|Goa|Hyderabad

SS ZG653 (RL 3.1): Software Architecture

Quality classes and attribute, quality attribute scenario and architectural tactics

Instructor: Prof. SantonuSarkar

Jan 11, 2015

A step back

- What is functionality?
  - Ability of the system to fulfill its responsibilities
- Software Quality Attributes- also called non-functional properties
  - Orthogonal to functionality
  - is a constraint that the system must satisfy while delivering its functionality
- Design Decisions
  - A constraint driven by external factors (use of a programming language, making everything service oriented)

Consider the following requirements

- User interface should be easy to use
  - Radio button or check box? Clear text? Screen layout? --- NOT architectural decisions
- User interface should allow redo/undo at any level of depth
  - Architectural decision
- The system should be modifiable with least impact
  - Modular design is must – Architectural
  - Coding technique should be simple – not architectural

- Need to process 300 requests/sec
  - Interaction among components, data sharing issues--architectural
  - Choice of algorithm to handle transactions -- non architectural

Quality Attributes and Functionality

- Any product (software products included) is sold based on its functionality – which are its features
  - Mobile phone, MS-Office software
  - Providing the desired functionality is often quite challenging
    - Time to market
    - Cost and budget
    - Rollout Schedule
- Functionality DOES NOT determine the architecture. If functionality is the only thing you need
  - It is perfectly fine to create a monolithic software blob!
  - You wouldn't require modules, threads, distributed systems, etc.

Examples of Quality Attributes

- Availability
- Performance
- Security
- Usability
- Functionality
- Modifiability
- Portability
- Reusability
- Integrability
- Testability

- The success of a product will ultimately rest on its Quality attributes
  "Too slow!"-- performance
  "Keeps crashing!" --- availability
  "So many security holes!" --- security
  "Reboot every time a feature is changed!" --- modifiability
  "Does not work with my home theater!" --- integrability

- Needs to be achieved throughout the design, implementation and deployment
- Should be designed in and also evaluated at the architectural level
- Quality attributes are NON-orthogonal
  - One can have an effect (positive or negative) on another
  - Performance is troubled by nearly all other. All other demand more code where-as performance demands the least

Defining and understanding system quality attributes

- Defining a quality attribute for a system
  - System should be modifiable --- vague, ambiguous
- How to associate a failure to a quality attribute
  - Is it an availability problem, performance problem or security or all of them?
- Everyone has his own vocabulary of quality
- ISO 9126 and ISO 25000 attempts to create a framework to define quality attributes

Three Quality Classes

System Quality	Business Quality	Quality of Architecture
Availability	Time to market	Conceptual Integrity
Modifiability	Cost and benefit	Correctness
Performance	Project lifetime	completeness
Security	Targeted market	Buildability
Testability	Rollout schedule	
Usability	Legacy integration	

- We will consider these attributes
- We will use "**<u>Quality Attribute Scenarios</u>**" to characterize them
  - which is a quality attribute specific requirement

Quality Attribute Scenario

Architectural Tactics

- To achieve a quality one needs to take a design decision- called Tactic
  - Collection of such tactics is **architectural strategy**
  - A pattern can be a collection of tactics

Quality Design Decisions

- To address a quality following 7 design decisions need to be taken
  - Allocation of responsibilities
  - Coordination
  - Data model
  - Resource Management
  - Resource Binding
  - Technology choice

Quality Design Decisions

- Responsibility Allocation
  - Identify responsibilities (features) that are necessary for this quality requirement
  - Which non-runtime (module) and runtime (components and connectors) should address the quality requirement
- Coordination
  - Mechanism (stateless, stateful…)
  - Properties of coordination (lossless, concurrent etc.)
  - Which element should and shouldn't communicate
- Data Model
  - What's the data structure, its creation, use, persistence, destruction mechanism
  - Metadata
  - Data organization
- Resource management
  - Identifying resources (CPU, I/O, memory, battery, system lock, thread pool..) and who should manage
  - Arbitration policy
  - Find impact of what happens when the threshold is exceeded
- Binding time decision
  - Use parameterized makefiles
  - Design runtime protocol negotiation during coordination
  - Runtime binding of new devices
  - Runtime download of plugins/apps
- Technology choice

Business Qualities

Business Quality	Details
Time to Market	•Competitive Pressure – short window of opportunity for the product/system •Build vs. Buy decisions •Decomposition of system – insert a subset OR deploy a subset
Cost and benefit	•Development effort is budgeted •Architecture choices lead to development effort •Use of available expertise, technology •Highly flexible architecture costs higher
Projected lifetime of the system	•The product thatneeds to survive for longer time needs to be modifiable, scalable, portable •Such systems live longer; however may not meet the time-to-market requirement
Targeted Market	•Size of potential market depends on feature set and the platform •Portability and functionality key to market share •Establish a large market; a product line approach is well suited
Rollout Schedule	•Phased rollouts; base + additional features spaced in time •Flexibility and customizability become the key
Integration with Legacy System	•Appropriate integration mechanisms •Much implications on architecture

Architectural Qualities

Architectural Quality	Details
Conceptual Integrity	•Architecture should do similar things in similar ways •Unify the design at all levels
Correctness and Completeness	•Essential to ensure system's requirements and run time constraints are met
Build ability	•Implemented by the available team in a timely manner with high quality •Open to changes or modifications as time progresses •Usually measured in cost and time •Knowledge about the problem to be solved

BITS Pilani

Pilani|Dubai|Goa|Hyderabad

SS ZG653 (RL 4.1): Software Architecture

Usability and Its Tactics

Instructor: Prof. Santonu Sarkar

Jan 11, 2015

Usability

- How easy it is for the user to accomplish a desired task and user support the system provides
  - Learnability: what does the system do to make a user familiar
  - Operability:
    - Minimizing the impact of user errors
    - Adopting to user needs
    - Giving confidence to the user that the correct action is being taken?

Usability Scenario Example

WHO

End user

STIMULUS

User Wants to
- Learn system feature
- Use systems efficiently
- Minimize the impact of errors
- Adapt system
- Feel comfortable

IMPACTED PART

Whole System

- **At run time**
- **At configure time**

MITIGATING ACTION
- Learn
  - ✓ Context sensitive help, familiar interface
- Efficient use
  - ✓ Aggregation of data and command, reuse of already entered data, good navigation, search mechanism, multiple activities
- Error impact
  - ✓ Undo, cancel, recover, auto-correct, retrieve forgotten information

MEASURABLE RESPONSE

- Task time
- Number of errors
- User satisfaction
- Gain of user knowledge
- Successful operations
- Amount of time/data lost

End User

Downloads application

Runtime

Uses application productively

Takes 4 mins to be productive

Usability Tactics

Usability is essentially Human Computer Interaction. Runtime Tactics are

User initiative (and system responds)

System initiative

Cancel, undo, aggregation, store partial result

Task model: understands the context of the task user is trying and provide assistance

User model: understands who the user is and takes action

System model: gets the current state of the system and responds

User Initiative and System Response

- Cancel
  - When the user issues cancel, the system must listen to it (in a separate thread)
  - Cancel action must clean the memory, release other resources and send cancel command to the collaborating components
- Undo
  - System needs to maintain a history of earlier states which can be restored
  - This information can be stored as snapshots
- Pause/resume
  - Should implement the mechanism to temporarily stop a running activity, take its snapshot and then release the resource for other's use
- Aggregate (change font of the entire paragraph)
  - For an operation to be applied to a large number of objects
    - Provide facility to group these objects and apply the operation to the group

System Initiated

- Task model
  - Determine the current runtime context, guess what user is attempting, and then help
  - Correct spelling during typing but not during password entry
- System model
  - Maintains its own model and provide feedback of some internal activities
  - Time needed to complete the current activity
- User model
  - Captures user's knowledge of the system, behavioral pattern and provide help
  - Adjust scrolling speed, user specific customization, locale specific adjustment

Usability Tactics and Patterns….

- Design time tactics- UI is often revised during testing. It is best to separate UI from the rest of the application
  - Model view controller architecture pattern
  - Presentation abstraction control
  - Command Pattern
  - Arch/Slinky
    - Similar to Model view controller

Design Checklist

- Allocation of Responsibilities
  - Identify the modules/components responsible for
    - Providing assistance, on-line help
    - Adapt and configure based on user choice
    - Recover from user error
- Coordination Model
  - Check if the system needs to respond to
    - User actions (mouse movement) and give feedback
    - Can long running events be canceled?
- Data model
  - data structures needed for undo, cancel
  - Design of transaction granularity to support undo and cancel
- Resource mgmt
  - Design how user can configure system's use of resource
- Technology selection
  - To achieve usability

Thank You

BITS Pilani

Pilani|Dubai|Goa|Hyderabad

SS ZG653 (RL 4.2): Software Architecture

Availability and Its Tactics

Instructor: Prof. Santonu Sarkar

Jan 20, 2015

Faults and Failure

- Hypothesized cause of error in the software
- Part of the system's total state that can leads to failure
- event that occurs when the delivered service deviates from correct service

- Not every fault causes a failure:
  - Code that is "mostly" correct.
  - Dead or infrequently-used code.
  - Faults that depend on a set of circumstances to occur
- Cost of software failure often far outstrips the cost of the original system
  - data loss
  - down-time
  - cost to fix
- **Primary objective:** Remove faults with the most serious consequences.
- **Secondary objective:** Remove faults that are encountered most often by users.
  - One study showed that removing 60% of software "defects" led to a 3% reliability improvement

Failure Classification

- Transient - only occurs with certain inputs
- Permanent - occurs on all inputs
- Recoverable - system can recover without operator help
- Unrecoverable - operator has to help
- Non-corrupting - failure does not corrupt system state or data
- Corrupting - system state or data are altered

Availability

- Readiness of the software to carry out its task
  - 100% available (which is actually impossible) means it is always ready to perform the intended task
- A related concept is Reliability
  - Ability to “continuously provide” correct service without failure
- Availability vs Reliability
  - A software is said to be available even when it fails but recovers immediately
  - Such a software will NOT be called Reliable

- Thus, Availability measures the fraction of time system is really available for use
  - Takes repair and restart times into account
  - Relevant for non-stop continuously running systems (e.g. traffic signal)

What is Software Reliability

- Probability of failure-free operation of a system over a specified time within a specified environment for a specified purpose
  - Difficult to measure the purpose,
  - Difficult to measure environmental factors.
- It's not enough to consider simple failure rate:
  - Not all failures are created equal; some have much more serious consequences.
  - Might be able to recover from some failures reasonably.

Availability

- Once the system fails
  - It is not available
  - It needs to recover within a short time
- Availability
- Scheduled downtime is typically not considered
  - Availability 100% means it recovers instantaneously
  - Availability 99.9% means there is 0.01% probability that it will not be operational when needed

System Type	Availability (%)	Downtime in a year
Normal workstation	99	3.6 days
HA system	99.9	8.5 hours
Fault-resilient system	99.99	1 hour
Fault-tolerant system	99.999	5 min

Availability Scenarios

WHO

Internal or External to System

STIMULUS

Fault causing
- System does not respond
- Crash
- Delay in response
- Errorneous Response

IMPACTED PART

Infrastructure and/or application

- During normal operation
- During degraded mode of operation

MITIGATING ACTION

When fault occurs it should do one or more of
- ✓ detect and log
- ✓ Notify the relevant stakeholders
- ✓ Disable the source of failure
- ✓ Be unavailable for a predefined time interval
- ✓ Continue to operate in a degraded mode

MEASURABLE RESPONSE

- Specific time interval for availability
- Availability number
- Time interval when it runs in degraded mode
- Time to repair

Two Broad Approaches

- Fault Tolerance
  - Allow the system to continue in presence of faults. Methods are
    - Error Detection
    - Error Masking (through redundancy)
    - Recovery

- Fault Prevention
  - Techniques to avoid the faults to occur

Availability Tactics

Fault detection	Error Masking	Recover From Fault	Fault prevention
• Ping/echo • Heartbeat • Timestamp • Data sanity check • Condition monitoring • Voting • Exception Detection • Self-test	• Active redundancy (Hot) • Passive redundancy (Warm) • Spare (Cold) • Exception handling • Graceful degradation • Ignore faulty behavior	• Rollback • Retry • Reconfiguration • Shadow operation • State resynchronization • Escalating restart • Nonstop forwarding	• Removal of a component to prevent anticipated failure– auto/manual reboot • Create transaction • Software upgrade • Predictive model • Process monitor- that can detect, remove and restart faulty process • Exception prevention

Availability Tactics- Fault Detection

- Ping
  - Client (or fault-detector) pings the server and gets response back
  - To avoid less communication bandwidth- use hierarchy of fault-detectors, the lowest one shares the same h/w as the server
- Heartbeat
  - Server periodically sends a signal
  - Listeners listen for such heartbeat. Failure of heartbeat means that the server is dead
  - Signal can have data (ATM sending the last txn)
- Exception Detection
  - Adding an Exception handler means error masking

More details- Heartbeat

- Each node implements a lightweight process called heartbeat daemon that periodically (say 10 sec) sends heartbeat message to the master node.
- If master receives heartbeat from a node from both connections (a node is connected redundantly for fault-tolerance), everything is ok
- If it gets from one connections, it reports that one of the network connection is faulty
- If it does not get any heartbeat, it reports that the node is dead (assuming that the master gets heartbeat from other nodes)
- Trick: Often heartbeat signal has a payload (say resource utilization info of that node)
  - Hadoop NameNode uses this trick to understand the progress of the job

Detect Fault

- Timer and Timestamping
  - If the running process does not reset the timer periodically, the timer triggers off and announces failure
  - Timestamping: assigns a timestamp (can be a count, based on the local clock) with a message in a decentralized message passing system. Used to detect inconsistency
- Voting (TMR)
  - Three identical copies of a module are connected to a voting system which compares outputs from all the three components. If there is an inconsistency in their outputs when subjected to the same input, the voting system reports error/inconsistency
  - Majority voting, or preferred component wins

Availability Tactics- Error Masking

- Hot spare (Active redundancy)
  - Every redundant process is active
  - When one fails, another one is taken up
  - Downtime is millisec
- Warm restart (Passive redundancy)
  - Standbys keep syncing their states with the primary one
  - When primary fails, backup starts
- Spare copy (Cold)
  - Spares are offline till the primary fails, then it is restarted
  - Typically restarts to the checkpointed position
  - Downtime in minute
  - Used when the MTTF is high and HA is not that critical

Error Masking

- Service Degradation
  - Most critical components are kept live and less critical component functionality is dropped
- Ignore faulty behavior
  - E.g. If the component send spurious messages or is under DOS attack, ignore output from this component
- Exception Handling – this masks or even can correct the error

Availability Tactics- Fault Recovery

- Shadow
  - Repair the component
  - Run in shadow mode to observe the behavior
  - Once it performs correctly, reintroduce it
- State resynch
  - Related to the hot and warm restart
  - When the faulty component is started, its state must be upgraded to the latest state.
    - Update depends on downtime allowed, size of the state, number of messages required for the update..
- Checkpointing and recovery
  - Application periodically “commits” its state and puts a checkpoint
  - Recovery routines can either roll-forward or roll-back the failed component to a checkpoint when it recovers

Availability Tactics- Recovery

- Escalating Restart
  - Allows system to restart at various levels of granularity
    - Kill threads and recreate child processes
    - Frees and reinitialize memory locations
    - Hard restart of the software
- Nonstop forwarding (used in router design)
  - If the main recipient fails, the alternate routers keep receiving the packets
  - When the main recipient comes up, it rebuilds its own state

Availability Tactics- Fault Prevention

- Faulty component removal
  - Fault detector predicts the imminent failure based on process's observable parameters (memory leak)
  - The process can be removed (rebooted) and can be auto-restart
- Transaction
  - Group relevant set of instructions to a transaction
  - Execute a transaction so that either everyone passes or all fails
- Predictive Modeling
  - Analyzes past failure history to build an empirical failure model
  - The model is used to predict upcoming failure
- Software upgrade (preventive maintenance)
  - Periodic upgrade of the software through patching prevents known vulnerabilities

Design Decisions

Responsibility Allocation

- For each service that need to be highly available
  - Assign additional responsibility for fault detection (e.g. crash, data corruption, timing mismatch)
  - Assign responsibilities to perform one or more of:
    - Logging failure, and notification
    - Disable source event when fault occur
    - Implement fault-masking capability
    - Have mechanism to operate on degraded mode

Coordination

- For each service that need to be highly available
  - Ensure that the coordination mechanism can sense the crash, incorrect time
  - Ensure that the coordination mechanism will
    - Log the failure
    - Work in degraded mode

Design Decisions

Data Model

- Identify which data + operations are impacted by a crash, incorrect timing etc.
  - Ensure that these data elements can be isolated when fault occurs
  - E.g. ensure that "write" req. is cached during crash so that during recovery these writes are applied to the system

Resource Management

- Identify which resources should be available to continue operations during fault
- E.g. make the input Q large enough so that can accommodate requests when the server is being recovered from a failure

Design Decisions

- Binding Time
  - Check if late binding can be a source of failure
  - Suppose that a late bound component report its failure in 0.1ms after the failure and the recovery takes 1.5sec. This may not be acceptable
- Technology Choice
  - Determine the technology and tools that can help in fault detection, recovery and then reintroduction
  - Determine the technology that can handle a fault
  - Determine whether these tools have high availability!!

Hardware vs Software Reliability Metrics

- Hardware metrics are not suitable for software since its metrics are based on notion of component failure
- Software failures are often design failures
- Often the system is available after the failure has occurred
- Hardware components can wear out

Software Reliability Metrics

- Reliability metrics are units of measure for system reliability
- System reliability is measured by counting the number of operational failures and relating these to demands made on the system at the time of failure
- A long-term measurement program is required to assess the reliability of critical systems

Time Units

- Raw Execution Time
  - non-stop system
- Calendar Time
  - If the system has regular usage patterns
- Number of Transactions
  - demand type transaction systems

Reliability Metric POFOD

- Probability Of Failure On Demand (POFOD):
  - Likelihood that system will fail when a request is made.
  - E.g., POFOD of 0.001 means that 1 in 1000 requests may result in failure.
- Any failure is important; doesn't matter how many if the failure > 0
- Relevant for safety-critical systems

Reliability Metric ROCOF & MTTF

- Rate Of Occurrence Of Failure (ROCOF):
  - Frequency of occurrence of failures.
  - E.g., ROCOF of 0.02 means 2 failures are likely in each 100 time units.
- Relevant for transaction processing systems
- Mean Time To Failure (MTTF):
  - Measure of time between failures.
  - E.g., MTTF of 500 means an average of 500 time units passes between failures.
- Relevant for systems with long transactions

Rate of Fault Occurrence

- Reflects rate of failure in the system
- Useful when system has to process a large number of similar requests that are relatively frequent
- Relevant for operating systems and transaction processing systems

Mean Time to Failure

- Measures time between observable system failures
- For stable systems MTTF = 1/ROCOF
- Relevant for systems when individual transactions take lots of processing time (e.g. CAD or WP systems)

Failure Consequences

- When specifying reliability both the number of failures and the consequences of each matter
- Failures with serious consequences are more damaging than those where repair and recovery is straightforward
- In some cases, different reliability specifications may be defined for different failure types

Building Reliability Specification

- For each sub-system analyze consequences of possible system failures
- From system failure analysis partition failure into appropriate classes
- For each class send out the appropriate reliability metric

Examples

Failure Class	Example	Metric
Permanent Non-corrupting	ATM fails to operate with any card, must restart to correct	ROCOF = .0001 Time unit = days
Transient Non-corrupting	Magnetic stripe can't be read on undamaged card	POFOD = .0001 Time unit = transactions

THANK YOU

Reliability Metrics - part 1

- Probability of Failure on Demand (POFOD)
  - POFOD = 0.001
  - For one in every 1000 requests the service fails per time unit
- Rate of Fault Occurrence (ROCOF)
  - ROCOF = 0.02
  - Two failures for each 100 operational time units of operation

Reliability Metrics - part 2

- Mean Time to Failure (MTTF)
  - average time between observed failures (aka MTBF)
- Availability = MTTF / (MTTF+MTTR)
  - MTTF = Mean Time To Failure
  - MTTR = Mean Time to Repair
- Reliability = MTBF / (1+MTBF)

Probability of Failure on Demand

- Probability that the system will fail when a service request is made
- Useful when requests are made on an intermittent or infrequent basis
- Appropriate for protection systems service requests may be rare and consequences can be serious if service is not delivered
- Relevant for many safety-critical systems with exception handlers

Specification Validation

- It is impossible to empirically validate high reliability specifications
- No database corruption really means POFOD class < 1 in 200 million
- If each transaction takes 1 second to verify, simulation of one day's transactions takes 3.5 days

Statistical Reliability Testing

- Test data used, needs to follow typical software usage patterns
- Measuring numbers of errors needs to be based on errors of omission (failing to do the right thing) and errors of commission (doing the wrong thing)

Difficulties with Statistical Reliability Testing

- Uncertainty when creating the operational profile
- High cost of generating the operational profile
- Statistical uncertainty problems when high reliabilities are specified

Safety Specification

- Each safety specification should be specified separately
- These requirements should be based on hazard and risk analysis
- Safety requirements usually apply to the system as a whole rather than individual components
- System safety is an an emergent system property

THANK YOU

BITS Pilani

Pilani|Dubai|Goa|Hyderabad

SS ZG653 (RL 5.1): Software Architecture

Modifiability and Its Tactics

Instructor: Prof. Santonu Sarkar

Modifiability

- **Ability to Modify the system based on the change in requirement so that**
  - **the time and cost to implement is optimal**
  - **Impact of modification such as testing, deployment, and change management is minimal**
- **When do you want to introduce modifiability?**
  - **If (cost of modification w/o modifiability mechanism in place) > (cost of modification with modifiability in place)+ Cost of installing the mechanism**

Modifiability Scenarios

WHO

- Enduser
- Developer
- SysAdm

STIMULUS

They want to modify
- Functionality
  - Add, modify, delete
- Quality
  - Capacity

IMPACTED PART

UI, platform or System

- Runtime
- Compile time
- Design time
- Build time

MITIGATING ACTION

When fault occurs it should do one or more of
- ✓ Locate (Impact analysis)
- ✓ Modify
- ✓ Test
- ✓ Deploy again

MEASURABLE RESPONSE

- Volume of the impact of the primary system (number, size)
- Cost of modification
- Time and effort
- Extent of impact to other systems
- New defects introduced

Developer	Tries to change UI	Artifact – Code Environment: Design time	Changes made and unit test done	Completed in 4 hours

Modifiability Tactics

Dependency between two modules (B→ A)

Syntax (compile+runtime)

- Data : B uses the type/format of the data created by A
- Service B uses the API signature provided by A

Semantics of A

- Data: Semantics of data created by A should be consistent with the assumption made by B
- Service: Same …..

Sequence

- Data -- data packets created by A should maintain the order as understood by B
- Control– A must execute 5ms before B. Or an API of A can be called only after calling another API

Interface identity

- Handle of A must be consistent with B, if A maintains multiple interfaces

Location of A

- B may assume that A is in-process or in a different process, hardware..

Quality of service/data provided by A

- Data quality produced by A must be > some accuracy for B to work

Existence of A

- B may assume that A must exist when B is calling A

Resource behavior of A

- B may assume that both use same memory
- B needs to reserve a resource owned by A

Localize Modifications

1. Factoring common service
   - Common services through a specialized module (only implementing module should be impacted)
     - Heavily used in application framework and middleware
   - Reduce Coupling and increase cohesion
2. Anticipate Expected Changes
   - Quite difficult to anticipate, hence should be coupled with previous one
   - Allow extension points to accommodate changes
3. Generalize the Module
   - Allowing it to perform broader range of functions
   - Externalize configuration parameters (could be described in a language like XML)
     - The module reconfigure itself based on the configurable parameters
   - Externalize business rules
4. Limit Possible options
   - Do not keep too many options for modules that are part of the framework

Prevent Ripple Effect Tactics

1. Hide Information (of A)
   - Use interfaces, allow published API based calls only
2. Maintain existing Interface (of A)
   - Add new interfaces if needed
   - Use Wrapper, adapter to maintain same interface
   - Use stub
3. Restrict Communication Paths
   - Too many modules should not depend on A
4. Use an intermediary between B and A
   - Data
     - Repository from which B can read data created by A (blackboard pattern)
     - Publish-subscribe (data flowing through a central hub)
     - MVC pattern
   - Service: Use of design patterns like bridge, mediator, strategy, proxy
   - Identity of A – Use broker pattern which deals with A's identity
   - Location of A – Use naming service to discover A
   - Existence of A- Use factory pattern

Defer Binding Time

1. Runtime registration of A (plug n play)
   - Use of pub-sub
2. Configuration Files
   - To take decisions during startup
3. Polymorphism
   - Late binding of method call
4. Component Replacement (for A)
   - during load time such as classloader
5. Adherence to a defined protocol- runtime binding of independent processes

Design Checklist- Modifiability

- Allocation of Responsibilities
  - Determine the types of changes that can come due to technical, customer or business
  - Determine what sort of additional features are required to handle the change
  - Determine which existing features are impacted by the change
- Coordination Model
  - For those where modifiability is a concern, use techniques to reduce coupling
    - Use publish-subscribe, use enterprise service bus
  - Identify
    - which features can change at runtime
    - which devices, communication paths or protocols can change at runtime
  - And make sure that such changes have limited impact on the system

Design checklist- Modifiability

- Data Model
  - For the anticipated changes, decide which data elements will be impacted, and the nature of impact (creation, modification, deletion, persistence, translation)
  - Group data elements that are likely to change together
  - Design to ensure that changes have minimal impact to the rest of the system
- Resource Management
  - Determine how addition, deletion or modification of a feature or a quality attribute cause
    - New resources to be used, or affect resource usage
    - Changing of resource usage limits
  - Ensure that the resources after modification are sufficient to meet the system requirement
  - Write Resource manager module that encapsulates resource usage policies

Design Checklist- Modifiability

- Binding
  - Determine the latest time at which the anticipated change is required
  - Choose a defer binding if possible
  - Try to avoid too many binding choices
- Choice of Technology
  - Evaluate the technology that can handle modifications with least impact (e.g. enterprise service bus)
  - Watch for vendor lock-in

BITS Pilani

Pilani|Dubai|Goa|Hyderabad

SS ZG653 (RL 5.2): Software Architecture

Performance and Its Tactics

Instructor: Prof. Santonu Sarkar

What is Performance?

- Software system's ability to meet timing requirements when it responds to an event
- Events are
  - interrupts, messages, requests from users or other systems
  - clock events marking the passage of time
- The system, or some element of the system, must **respond to them** in time

Performance Scenarios

WHO	STIMULUS	IMPACTED PART	MITIGATING ACTION	MEASURABLE RESPONSE
• External • Internal	•**Events that are** •Periodic •Sporadic •Stochastic (with avg 1000 txn/min)	System •Running in normal mode •Overloaded	✓ Processes event in time ✓ Changes the service-level	• Latency or deadline by which it must be processed • Throughput- #of events processed/time • Jitter- variation • Miss rate • Data loss
Users	Submits transactions and expect response in 3s		Transactions processed	Average latency of 2s

Events and Responses

- Periodic- comes at regular intervals (real time systems)
- Stochastic- comes randomly following a probability distribution (eCommerce website)
- Sporadic- keyboard event from human

- Latency- time between arrival of stimulus and system response
- Throughput- number of txn processed/unit of time
- Jitter- allowable variation in latency
- #events not processed

Performance Tactics

Why System fails to Respond?

- Resource Consumption
  - CPU, memory, data store, network communication
  - A buffer may be sequentially accessed in a critical section
  - There may be a workflow of tasks one of which may be choked with request

- Blocking of computation time
  - Resource contention
  - Availability of a resource
  - Deadlock due to dependency of resource

Control Resource Demand

- Increase Computation Efficiency: Improving the algorithms used in performance critical areas
- Reduce Overhead
  - Reduce resource consumption when not needed
    - Use of local objects instead of RMI calls
    - Local interface in EJB 3.0
  - Remove intermediaries (conflicts with modifiability)
- Manage
  - event rate: If you have control, don't sample too many events (e.g. sampling environmental data)
  - sampling time: If you don't have control, sample them at a lower speed, leading to loss of request
- Bound
  - Execution: Decide how much time should be given on an event. E.g. iteration bound on a data-dependent algorithm
  - Queue size: Controls maximum number of queued arrivals

Manage Resources

- Increase Resources(infrastructure)
  - Faster processors, additional processors, additional memory, and faster networks
- Increase Concurrency
  - If possible, process requests in parallel
  - Process different streams of events on different threads
  - Create additional threads to process different sets of activities
- Multiple copies
  - Computations : so that it can be performed faster (client-server), MapReduce computation
  - Data:
    - use of cache for faster access and reduce contention
    - Hadoop maintains data copies to avoid data-transfer and improve data locality

Resource Arbitration

- Resources are scheduled to reduce contention
  - Processors, buffer, network
  - Architect needs to choose the right scheduling strategy
- FIFO
- Fixed Priority
  - Semantic importance
    - Domain specific logic such as request from a privileged class gets higher priority
  - Deadline monotonic (shortest job first)
- Dynamic priority
  - Round robin
  - Earliest deadline first- the job which has earliest deadline to complete
- Static scheduling
  - Also pre-emptive scheduling policy

Design Checklist for a Quality Attribute

- Allocate responsibility
  - Modules can take care of the required quality requirement
- Manage Data
  - Identify the portion of the data that needs to be managed for this quality attribute
  - Plan for various data design w.r.t. the quality attribute
- Resource Management Planning
  - How infrastructure should be monitored, tuned, deployed to address the quality concern
- Manage Coordination
  - Plan how system elements communicate and coordinate
- Binding

Performance- Design Checklist- Allocate responsibilities

- Identify which features may involve or cause
  - Heavy workload
  - Time-critical response
- Identify which part of the system that's heavily used
- For these, analyze the scenarios that can result in performance bottleneck
- Furthermore--
  - Assign Responsibilities related to threads of control —allocation and de-allocation of threads, maintaining thread pools, and so forth
  - Assign responsibilities that will schedule shared resources or appropriately select, manage performance-related artifacts such as queues, buffers, and caches

Performance- Design Checklist- Manage Data

- Identify the data that's involved in time critical response requirements, heavily used, massive size that needs to be loaded etc. For those data determine
  - whether maintaining multiple copies of key data would benefit performance
  - partitioning data would benefit performance
  - whether reducing the processing requirements for the creation, initialization, persistence, manipulation, translation, or destruction of the enumerated data abstractions is possible
  - whether adding resources to reduce bottlenecks for the creation, initialization, persistence, manipulation, translation, or destruction of the enumerated data abstractions is feasible.

Performance- Design Checklist- Manage Coordination

- Look for the possibility of introducing concurrency (and obviously pay attention to thread-safety), event priorization, or scheduling strategy
  - Will this strategy have a significant positive effect on performance? Check
  - Determine whether the choice of threads of control and their associated responsibilities introduces bottlenecks

- Consider appropriate mechanisms for example
  - stateful, stateless, synchronous, asynchronous, guaranteed delivery

Performance Design Checklist- Resource Management

- Determine which resources (CPU, memory) in your system are critical for performance.
  - Ensure they will be monitored and managed under normal and overloaded system operation.
- Plan for mitigating actions early, for instance
  - Where heavy network loading will occur, determine whether co-locating some components will reduce loading and improve overall efficiency.
  - Ensure that components with heavy computation requirements are assigned to processors with the most processing capacity.
- Prioritization of resources and access to resources
  - scheduling and locking strategies
- Deploying additional resources on demand to meet increased loads
  - Typically possible in a Cloud and virtualized scenario

Performance Design checklist-Binding

- For each element that will be bound after compile time, determine the
  - time necessary to complete the binding
  - additional overhead introduced by using the late binding mechanism

- Ensure that these values do not pose unacceptable performance penalties on the system.

Performance Design Checklist- Technology choice

- Choice of technology is often governed by the organization mandate (enterprise architecture)
- Find out if the chosen technology will let you set and meet real time deadlines?
  - Do you know its characteristics under load and its limits?
- Does your choice of technology give you the ability to set
  - scheduling policy
  - Priorities
  - policies for reducing demand
  - allocation of portions of the technology to processors
- Does your choice of technology introduce excessive overhead?

Thank You

BITS Pilani

Pilani|Dubai|Goa|Hyderabad

SS ZG653 (RL 6.1): Software Architecture

Security and Its Tactics

Instructor: Prof. Santonu Sarkar

What is Security

A measure of the system's ability to resist unauthorized usage while still providing its services to legitimate users

- Ability to protect data and information from unauthorized access

An attempt to breach this is an "Attack"

- Unauthorized attempt to access, modify, delete data
  - Theft of money by e-transfer, modification records and files, reading and copying sensitive data like credit card number
- Deny service to legitimate users

Important aspects of Security

Security comprises of

Confidentiality

- prevention of the unauthorized disclosure of information. E.g. Nobody except you should be able to access your income tax returns on an online tax-filing site

Integrity

- prevention of the unauthorized modification or deletion of information. E.g. your grade has not been changed since your instructor assigned it

Availability

- prevention of the unauthorized withholding of information – e.g. DoS attack should not prevent you from booking railway ticket

Important aspects of Security

Non repudiation:: An activity (say a transaction) can't be denied by any of the parties involved. E.g. you cannot deny ordering something from the Internet, or the merchant cannot disclaim getting your order.

Assurance:: Parties in an activity are assured to be who they purport to be. Typically done through authentication. E.g. if you get an email purporting to come from a bank, it is indeed from a bank.

Auditing:: System tracks activities so that it can be reconstructed later

Authorization grants a user the privileges to perform a task. For example, an online banking system authorizes a legitimate user to access his account.

Security Scenario

WHO

- **Individual or a system**
  - ✓Correctly identified OR
  - ✓Incorrectly identified
  - ✓OR unknown
- **Who is**
  - ✓Internal or external
  - ✓Authorized or unauthrized
- **Has access to**
  - ✓Limited resource
  - ✓Vast resource

STIMULUS

- **Tries to**
  - ✓Read data
  - ✓Change/delete data
  - ✓Access system services
  - ✓Reduce availability of services

IMPACTED PART

System services, data

- Online/offline
- Within firewall or Open

MITIGATING ACTION

- **Authentication**
  - ✓Authenticates
  - ✓Hides identity
- **Access control to data or services**
  - ✓Blocks access
  - ✓Grants/withdraws permission to access
  - ✓Audits access/ modification attempts
- **Corrective Actions**
  - ✓Encrypts data
  - ✓Detect anomalous situation (high access req.) and informs people/another system
  - ✓Restricts availability

MEASURABLE RESPONSE

- Time required to circumvent security measure with Probability of success
- Probability of detecting attack
- Probability of detecting the individual responsible
- %of services available during attack
- Time to restore data/services
- Extent of damage- how much data is vulnerable
- No of access denials

An insider

Updates payment details

Pay database table

Anomaly detected
Audit trail maintained

Corrective action taken in 1 day

Security Tactics- Close to Physical Security

- Detection:
  - Limit the access through security checkpoints
  - Enforces everyone to wear badges or checks legitimate visitors
- Resist
  - Armed guards
- React
  - Lock the door automatically
- Recover
  - Keep backup of the data in a different place

Security Tactics

Detect Attacks

- Detect Intrusion: compare network traffic or service request patterns *within* a system to
  - a set of signatures or
  - known patterns of malicious behavior stored in a database.
- Detect Service Denial
  - Compare the pattern or signature of network traffic *cominginto* a system to historic profiles of known Denial of Service (DoS) attacks.
- Verify Message Integrity
  - Use checksums or hash values to verify the integrity of messages, resource files, deployment files, and configuration files.
- Detect Message Delay:
  - checking the time that it takes to deliver a message, it is possible to detect suspicious timing behavior.

Resist Attacks

- Identify Actors: identify the source of any external input to the system.
- Authenticate & Authorize Actors:
  - Use strong passwords, OTP, digital certificates, biometric identity
  - Use access control pattern, define proper user class, user group, role based access
- Limit Access
  - Restrict access based on message source or destination ports
  - Use of DMZ

Resist Attacks

- Limit Exposure: minimize the attack surface of a system by allocating limited number of services to each hosts
- Data confidentiality:
  - Use encryption to encrypt data in database
  - User encryption based communication such as SSL for web based transaction
  - Use Virtual private network to communicate between two trusted machines
- Separate Entities: can be done through physical separation on different servers attached to different networks, the use of virtual machines, or an “air gap”.
- Change Default Settings: Force the user to change settings assigned by default.

React to Attacks

- Revoke Access: limit access to sensitive resources, even for normally legitimate users and uses, if an attack is suspected.
- Lock Computer: limit access to a resource if there are repeated failed attempts to access it.
- Inform Actors: notify operators, other personnel, or cooperating systems when an attack is suspected or detected.

Recover From Attacks

- In addition to the Availability tactics for recovery of failed resources there is Audit.

- Audit: keep a record of user and system actions and their effects, to help trace the actions of, and to identify, an attacker.

Design Checklist- Allocation of Responsibilities

- Identify the services that needs to be secured
  - Identify the modules, subsystems offering these services
- For each such service
  - Identify actors which can access this service, and implement authentication and level of authorization for those
  - verify checksums and hash values
  - Allow/deny data associated with this service for these actors
  - record attempts to access or modify data or services
  - Encrypt data that are sensitive
  - Implement a mechanism to recognize reduced availability for this services
  - Implement notification and alert mechanism
  - Implement recover from an attack mechanism

Design Checklist- Manage Data

- Determine the sensitivity of different data fields
- Ensure that data of different sensitivity is separated
- Ensure that data of different sensitivity has different access rights and that access rights are checked prior to access.
- Ensure that access to sensitive data is logged and that the log file is suitably protected.
- Ensure that data is suitably encrypted and that keys are separated from the encrypted data.
- Ensure that data can be restored if it is inappropriately modified.

Design Checklist- Manage Coordination

- For inter-system communication (applied for people also)
  - Ensure that mechanisms for authenticating and authorizing the actor or system, and encrypting data for transmission across the connection are in place.
- Monitor communication
  - Monitor anomalous communication such as
    - unexpectedly high demands for resources or services
    - Unusual access pattern
  - Mechanisms for restricting or terminating the connection.

Design Checklist- Manage Resource

- Define appropriate grant or denial of resources
- Record access attempts to resources
- Encrypt data
- Monitor resource utilization
  - Log
  - Identify suddenly high demand to a particular resource- for instance high CPU utilization at an unusual time
- Ensure that a contaminated element can be prevented from contaminating other elements.
- Ensure that shared resources are not used for passing sensitive data from an actor with access rights to that data to an actor without access rights.
  - .

Design checklist- Binding

- Runtime binding of components can be untrusted. Determine the following
  - Based on situation implement certificate based authentication for a component
    - Implement certification management, validation
  - Define access rules for components that are dynamically bound
  - Implement audit trail for whenever a late bound component tries to access records
  - System data should be encrypted where the keys are intentionally withheld for late bound components

Design Checklist- Technology choice

Choice of technology is often governed by the organization mandate (enterprise architecture)

- Decide tactics first. Based on the tactics, ensure that your chosen technologies support the tactics

- Determine what technology are available to help user authentication, data access rights, resource protection, data encryption
- Identify technology and tools for monitoring and alert

BITS Pilani

Pilani|Dubai|Goa|Hyderabad

SS ZG653 (RL 6.2): Software Architecture

Testability and Its Tactics

Instructor: Prof. Santonu Sarkar

What is Testability

The ease with which software can be made to demonstrate its faults through testing

If a fault is present in a system, then we want it to fail during testing as quickly as possible.

At least 40% effort goes for testing

- Done by developers, testers, and verifiers (tools)

Specialized software for testing

- Test harness
- Simple playback capability
- Specialized testing chamber

Testable Software

Dijkstra's Thesis

- Test can't guarantee the absence of errors, but it can only show their presence.
- Fault discovery is a probability
  - That the next test execution will fail and exhibit the fault
- A perfectly testable code – each component's internal state must be controllable through inputs and output must be observable
- Error-free software does not exist.

Testability Scenario

WHO	STIMULUS	IMPACTED PART	RESPONSE ACTION	MEASURABLE RESPONSE
• **Unit tester (typically unit developers)** • **Integration tester** • **System tester or client acceptance team** • **System user**	•**Milestone in the development process is met** ✓Completion of design ✓Completion of coding ✓Completion of integration	Component or whole system •Design time •Development time •Compile time •Integration time	•Prepare test environment •Access state values •Access computed values	• %executable statements executed (code coverage) • Time to test • Time to prepare test environment • Length of longest dependency chain in test • Probability of failure if fault exists
A unit tester	Performs unit test	Component that has controllable interface After component is complete	Observe the output for inputs provided	Coverage of 85% is achieved in 2 hours

Goal of Testability Tactics

- Using testability tactics the architect should aim to reduce the high cost of testing when the software is modified
- Two categories of tactics
  - Introducing controllability and observability to the system during design
  - The second deals with limiting complexity in the system's design

Testability Tactics

Control and Observe System State

- Specialized Interfaces for testing:
  - to control or capture variable values for a component either through a test harness or through normal execution.
  - Use a special interface that a test harness can use
  - Make use of some metadata through this special interface

- Record/Playback: capturing information crossing an interface and using it as input for further testing.

- Localize State Storage: To start a system, subsystem, or module in an arbitrary state for a test, it is most convenient if that state is stored in a single place.

Control and Observe System State

- Interface and implementation
  - If they are separated, implementation can be replaced by a stub for testing rest of the system
- Sandbox: isolate the system from the real world to enable experimentation that is unconstrained by the worry about having to undo the consequences of the experiment.
- Executable Assertions: assertions are (usually) hand coded and placed at desired locations to indicate when and where a program is in a faulty state.

Manage Complexity

- Limit Structural Complexity:
  - avoiding or resolving cyclic dependencies between components,
  - isolating and encapsulating dependencies on the external environment
  - reducing dependencies between components in general.
- Limit Non-determinism: finding all the sources of non-determinism, such as unconstrained parallelism, and remove them out as far as possible.

Internal Monitoring

- Implement a built-in monitoring mechanism
  - One should be able to turn on or off
    - one example is logging
  - Performed typically by instrumentation- AOP, Preprocessor macro. Instrument the code to introduce recorder at some point

Design Checklist- Allocation of Responsibility

Identify the services are most critical and hence need to be most thoroughly tested.

- Identify the modules, subsystems offering these services

- For each such service
  - Ensure that internal monitoring mechanism like logging is well designed
  - Make sure that the allocation of functionality provides
    - low coupling,
    - strong separation of concerns, and
    - low structural complexity.

Design Checklist- Testing Data

- Identify the data entities that are related to the critical services need to be most thoroughly tested.

- Ensure that creation, initialization, persistence, manipulation, translation, and destruction of these data entities are possible--
  - State Snapshot: Ensure that the values of these data entities can be captured if required, while the system is in execution or at fault
  - Replay: Ensure that the desired values of these data entities can be set (state injection) during testing so that it is possible to recreate the faulty behavior

Design Checklist- Testing Infrastructure

- Is it possible to inject faults into the communication channel and monitoring the state of the communication
- Is it possible to execute test suites and capture results for a distributed set of systems?
- Testing for potential race condition- check if it is possible toexplicitly map
  - processes to processors
  - threads to processes

So that the desired test response is achieved and potential race conditions identified

Design Checklist- Testing resource binding

- Ensure that components that are bound later than compile time can be tested in the late bound context
  - E.g. loading a driver on-demand
- Ensure that late bindings can be captured in the event of a failure, so that you can re-create the system's state leading to the failure.
- Ensure that the full range of binding possibilities can be tested.

Design Checklist- Resource Management

- Ensure there are sufficient resources available to execute a test suite and capture the results
- Ensure that your test environment is representative of the environment in which the system will run
- Ensure that the system provides the means to:
  - test resource limits
  - capture detailed resource usage for analysis in the event of a failure
  - inject new resources limits into the system for the purposes of testing
  - provide virtualized resources for testing

Choice of Tools

- Determine what tools are available to help achieve the testability scenarios
  - Do you have regression testing, fault injection, recording and playback supports from the testing tools?
- Does your choice of tools support the type of testing you intend to carry on?
  - You may want a fault-injection but you need to have a tool that can support the level of fault-injection you want
  - Does it support capturing and injecting the data-state

BITS Pilani

Pilani|Dubai|Goa|Hyderabad

SS ZG653 (RL 6.3): Software Architecture

Interoperability and Its Tactics

Instructor: Prof. Santonu Sarkar

Interoperability

- Ability that two systems can usefully exchange information through an interface
  - Ability to transfer data (syntactic) and interpret data (semantic)
- Information exchange can be direct or indirect
- Interface
  - Beyond API
  - Need to have a set of assumptions you can safely make about the entity exposing the API
- Example- you want to integrate with Google Maps

Why Interoperate?

- The service provided by Google Maps are used by unknown systems
  - They must be able to use Google Maps w/o Google knowing who they can be
- You may want to construct capability from variety of systems
  - A traffic sensing system can receive stream of data from individual vehicles
  - Raw data needs to be processed
  - Need to be fused with other data from different sources
  - Need to decide the traffic congestion
  - Overlay with Google Maps

Interoperability Scenario

WHO	STIMULUS	IMPACTED PART	RESPONSE ACTION	MEASURABLE RESPONSE
• **A system** **initiating request**	•**Request to exchange information**	Component or whole system •It has been already discovered •at runtime •Prior to runtime	• The request is ACCEPTED and information is exchanged • The request is REJECTED and appropriate entities are notified • The request is logged by one or more systems	• %information that has been exchanged successfully • %information that has been REJECTED successfully
Avehicle information system	Sends current location	Traffic monitoring system -- discovered prior to runtime	Combines the location information with other details, overlays with Google Maps, and broadcasts	The vehicle information is included correctly 99.9% of the time

Notion of Interface

- Information exchange
  - Can be as simple as A calling B
  - A and B can exchange implicitly w/o direct communication
  - Operation Dessert Storm 1991: Anti-missile system failed to exchange information (intercept) an incoming ballistic rocket
    - The system required periodic restart in order to recalibrate its position. Since it wasn't restarted, the information wasn't correctly captured due to error accumulation
- Interface
  - Here it also means that a set of assumptions that can be made safely about this entity
  - E.g. it is safe to assume that the API of anti-missile system DOES NOT give information about gradual degradation

Tactics

Interoperability
Locate (Discover service)
Manage interface
Orchestrate
Tailor interface

Interoperability Tactics

- Locate (Discover service)
  - Identify the service through a known directory service. Here service implies a set of capabilities available through an interface
  - By name, location or other attributes

Interoperability Tactics

Manage interface

- Orchestrate
  - Co-ordinate and manage a sequence of services. Example- workflow engines containing scripts of interaction
  - Mediator design pattern for simple orchestration. BPEL language for complex orchestration
- Tailor interface
  - Add or remove capability from an interface (hide a particular function from an untrusted user)
  - Use Decorator design pattern for this purpose

REpresentational State Transfer (REST)

REST is an architectural pattern where services are described using an uniform interface. REST*ful* services are viewed as a hypermedia resource. REST is stateless.

REST Verb	CRUD Operation	Description
POST	CREATE	Create a new resource.
GET	RETRIEVE	Retrieve a representation of the resource.
PUT	UPDATE	Update a resource.
DELETE	DELETE	Delete a resource.

Google Suggest : http://suggestqueries.google.com/complete/search?output=toolbar&hl=en&q=satyajit%20ray

Yahoo Search: http://search.yahooapis.com/WebSearchService/V1/webSearch?appid=YahooDemo&query=accenture

REST vs. SOAP/WSDL

- Simply put, the community has claimed that SOAP and WSDL have become too grandiose and comprehensive to achieve the "agility" touted by SOA (Seeley, R., "Burton sees the future of SOA and it is REST," SearchWebService.com, May 30, 2007)

	SOAP/WSDL	REST
Purpose	Message exchange between two applications/systems	Access and manipulating a hypermedia system
Origin	RPC	WWW
Functionality	Rich	Minimal
Interaction	Orchestrated event-based	Client/server (request/response)
Focus	Process-oriented	Data-oriented
Methods/ operations	Varies depending on the service	Fixed
Reuse	Centrally governed	Little/no governance (focus on ease of use instead)
Interaction context	Can be maintained in both client and server	Only on client
Format	SOAP in, SOAP out	URI (+POX) in, POX out
Transport	Transport independent	HTTP only
Security	WS-Security	HTTP authentication + SSL

Design Checklist- Interoperability

- Allocation of Responsibilities: Check which system features need to interoperate with others. For each of these features, ensure that the designers implement
  - Accepting and rejecting of requests
  - Logging of request
  - Notification mechanism
  - Exchange of information
- Coordination Model: Coordination should ensure performance SLAs to be met. Plan for
  - Handling the volume of requests
  - Timeliness to respond and send the message
  - Currency of the messages sent
  - Handle jitters in message arrival times

Design Checklist-Interoperability

Data Model

- Identify the data to be exchanged among interoperating systems
- If the data can't be exchanged due to confidentiality, plan for data transformation before exchange

Identification of Architectural Component

- The components that are going to interoperate should be available, secure, meet performance SLA (consider design-checklists for these quality attributes)

Design Checklist- Interoperability

Resource Management

- Ensure that system resources are not exhausted (flood of request shouldn't deny a legitimate user)
- Consider communication load
- When resources are to be shared, plan for an arbitration policy

Binding Time

- Ensure that it has the capability to bind unknown systems
- Ensure the proper acceptance and rejection of requests
- Ensure service discovery when you want to allow late binding

Technology Choice

- Consider technology that supports interoperability (e.g. web-services)

Thank You

BITS Pilani

Pilani|Dubai|Goa|Hyderabad

SS ZG653 (RL 7.1): Software Architecture

Introduction to OODesign

Instructor: Prof. Santonu Sarkar

Online shopping application- Pet Store

by Sun Microsystems (Oracle)

- Storefront has the main user interface in a Web front-end. Customers use the Storefront to place orders for pets
  - Register User
  - Login user
  - Browse catalog of products
  - Place order to OPC (asynchronous messaging)
- Order Processing Center (OPC) receives orders from the Storefront.
- Administrators
  - Examine pending orders
  - Approve or deny a pending order
- Supplier
  - View and edit the inventory
  - Fulfills orders from the OPC from inventory and invoices the OPC.

System quality

- Availability
- Modifiability
- Performance
- Security
- Testability
- Usability
- Interoperability

- Show orders in multiple languages
- Help in browsing products
- Easy checkout
- Assurance to customer, supplier and bank
- Guaranteed delivery
- Shopping cart is only modified by the customer
- Order never fails
- System is always ready to sell
- Order is always processed in 2sec.
- System always optimally use the hardware infrastructure and performs load-balancing
- Uses standard protocol to communicate with Suppliers
- Uses secure electronic transaction protocol (SET) for credit card processing
- Quite easy to add new supplier
- Little change necessary to add a third party vendor
- Quite easy to sell books in addition to Pets
- Logging is extensive to trace the root cause of the fault

Example

Programming for this Application

- Everything is an object
  - Pet, Customer, Supplier, Credit card, Customer's address, order processing center
- Pets can be sold, credit card can be charged, clients can be authenticated, order can be fulfilled!
  - Have behavior, properties
- Interact with each other
  - Storefront places order to OPC

What's cool?

- Very close to the problem domain… domain experts can pitch in easily.
- Could group features and related operations together
- It's possible to add new types of Pets (e.g., add Birds in addition to Cat, Dog and Fish)

What is a class

- Class represents an abstract, user-defined type
  - Structure – definition of properties/attributes
    - Commonly known as member variables
  - Behavior – operation specification
    - Set of methods or member functions

- An object is an instance of a class. It can be instantiated (or created) w/o a class

Object State

- **Properties/Attribute Values at a particular moment represents the state of an object**
- **Object may or may not change state in response to an outside stimuli.**
- **Objects whose state can not be altered after creation are known as immutable objects and class as immutable class [String class in Java]**
- **Objects whose state can be altered after creation are known as mutable objects and class as mutable class [StringBuffer class in Java]**

States of Three Different INSTANCES of "Dog" class

Labrador	***Golden Retriever***	***Pug***
Age: 1 yr	***Age: 6months***	***Age: 1.5yr***
Price : 3500	***Price : 2000***	***Price : 1500***
Sex:Male	***Sex:Female***	***Sex: Female***

Object-Oriented Programming

- A program is a bunch of objects telling each other what to do, by sending messages
  - In Java, C++, C# one object (say o1) invokes a method of another object (o2), which performs operations on o2
  - You can create any type of objects you want
- OO different from procedural?
  - No difference at all: Every reasonable language is ultimate the same!!
  - Very different if you want to build a large system
    - Increases understandability
    - Less chance of committing errors
    - Makes modifications faster
    - Compilers can perform stronger error detections

Type System

- Classes are user-defined data-types
- Primitive types
  - int, float, char, boolean in Java (bool in C#), double, short, long, string in C#
- Unified type
  - C# keyword object – mother of all types (root)
    - Everything including primitive types are objects
  - Java JDK gives java.lang.Object– not a part of the language
    - Primitive types are not objects

References and Values

Value

- Primitive types are accessed by value
- C++ allow a variable to have object as its value
- C# uses struct to define types whose variables are values
- No explicit object creation/deletion required
  - Faster, space decided during compilation

Reference

- Java allows a variable to have reference to an object only
- C++ uses pointers for references
- Needs explicit object creation
  - Slower, space allocated during runtime from heap
- Java performs escape analysis for faster allocation

Modules

- You need an organization when you deal with large body of software – 20MLOC, 30000 classes or files!
- Notion of module introduced in 1970
  - Group similar functionality together
  - Hide implementation and expose interface
  - Earlier languages like Modula, Ada introduced this notion
  - In OO language “class” was synonymous to a module
- But they all faced the problem of managing 20M lines of C or C++ code
- ANSI C++, Java, Haskell, C#, Perl, Python, PHP all support modules
  - Namespace (C++, C#)
  - Package (Java, Perl)

Module- aka namespace, package

- A set of classes grouped into a module
  - A module is decomposed into sub-modules
  - Containment hierarchy of modules – forms a tree

- A fully qualified, unique name= module+name of the class
  - namespace
  - package

- Import- A class can selectively use one or more classes in a module or import the entire module

Inheritance

- Parent (called Base class) and children classes (Derived classes)
  - A Derived class inherits the methods and member variables of the Base -- also called ISA relationship
  - A child can have multiple parents – Multiple inheritance
  - Hierarchy of inheritance (DAG)

- The Derived class can
  - Use the inherited variables and methods – reuse
  - Add new methods – extends the functionality
  - Modify derived method- called **overriding** the base class member function

Introducing Interface

What is it?

- A published declaration of a set of services
  - An interface is a collection of constants and method declarations
  - No implementation, a separate class needs to implement an interface
- A class can implement more than one interfaces

Why?

- Provide capability for unrelated classes to implement a set of common methods
- Standardize interaction
- Extension- let's the designer to defer the design
- User does not know who implemented it
  - It is easy to change implementation without impacting the user

Interface Definition

- An interface in C++, C# is a class that has at least one pure virtual method
  - A pure virtual method only has specification, but no body
  - This is called abstract base class
  - One can have an abstract class where some methods are concrete (with implementation) and some as pure virtual which could be clumsy

- Java uses keyword “interface” and is more clean
  - Interface only provides method declarations

- Java also allows to define an abstract base class just like C++

Creating Objects

- With built-in types like **int** or **char**, we just say

  int i; char c; --- and we get them

- When we define a class A-- user-defined type
  - We need to explicitly
    1. tell that we want a new object of type A (operator new)
    2. Initialize the object (you need to decide) after creation -- constructor method
       - Special method that compiler understands. The constructor method name must be same as the class name
       - C++ allows constructor method for struct also
- Constructor methods can be overloaded

Destroying Objects

- If an object goes “out of scope,” it can no longer be used (its name is no longer known) it is necessary to free the memory occupied by the object
  - Otherwise there will be a “memory leak”

1. Just before freeing the memory
   - It is necessary to perform clean-up tasks (you need to decide) just before getting deleted
     - » Destructor method describes these tasks
     - » Special name for destructor method ~<ClassName> in C++
     - » finalize() method in Java
     - » does not have any return value
2. Then free the memory
   - In C++, we need to explicitly delete this object (delete operator)
   - Java uses references and “garbage collection” automatically.

Thank You

Next class - UML

BITS Pilani

Pilani|Dubai|Goa|Hyderabad

SS ZG653 (RL 7.2): Software Architecture

Introduction to UML

Instructor: Prof. Santonu Sarkar

Unified Modeling Language (Introduction)

- **Modeling Language for specifying, Constructing, Visualizing and documenting software system and its components**
- **Model -> Abstract Representation of the system [Simplified Representation of Reality]**
- **UML supports two types of models:**

- Static - Dynamic

UML

- Unified Modeling Language is a standardized general purpose modeling language in the field of object oriented software engineering

- The standard is managed, and was created by, the Object Management Group.

- UML includes a set of graphic notation techniques to create visual models of object-oriented software-intensive systems

UML Diagrams overview

- Structure diagrams emphasize the things that must be present in the system being modeled- extensively used for documenting software architecture
- Behavior diagrams illustrate the behavior of a system, they are used extensively to describe the functionality of software systems.

Structural Diagrams

- **Class diagram**: system's classes, their attributes, and the relationships
- Component diagram: A system, comprising of components and their dependencies
- Composite structure diagram: decomposition of a class into more finer elements and their interactions
- **Deployment diagram**: describes the hardware used in system implementations and the execution environments and artifacts deployed on the hardware.
- Object diagram: shows a complete or partial view of the structure of an example modeled system at a specific time.
- Package diagram: describes how a system is split up into logical groupings by showing the dependencies among these groupings.
- Profile diagram: operates at the metamodel level

Behavioral Diagrams

- **Activity diagram**: describes the business and operational step-by-step workflows of components in a system. An activity diagram shows the overall flow of control.
- **State machine diagram**: describes the states and state transitions of part of the system.
- **Use case diagram**: describes the functionality provided by a system in terms of actors, their goals represented as use cases, and any dependencies among those use cases

Behavioral Model- Interactions

- Communication diagram: shows the interactions between objects or parts in terms of sequenced messages. They represent a combination of information taken from Class, Sequence, and Use Case Diagrams describing both the static structure and dynamic behavior of a system.
- Interaction overview diagram: provides an overview in which the nodes represent communication diagrams.
- **Sequence diagram**: Interaction among objects through a sequence of messages. Also indicates the lifespans of objects relative to those messages
  - Timing diagrams: a specific type of sequence diagram where the focus is on timing constraints.

Petstore Shopping System

Use Case

Storefront has the main user interface in a Web front-end. Customers use the Storefront to place orders for pets

- Register User
- Login user
- Browse catalog of products
- Place order to OPC (asynchronous messaging)

Order Processing Center (OPC) receives orders from the Storefront.

Administrator

- Examine pending orders
- Approve or deny a pending order

Supplier

- View and edit the inventory
- Fulfills orders from the OPC from inventory and invoices the OPC.

Class Notation

Visibility Modes : Private(-), Protected (#) , Public (+) and Package Private()

Class Relationships

Generalization Example

- isA, is-a-type of relation ship
- A PostalAddress, or an EmailAddress is an Address
- There can be ThirdPartyProduct or a Refurbished product in the online store

```
public class PostalAddress extends AbstractAddress  {

 private String Street;
 private String city;
 private String state;
 private Integer pincode;
}
```

```
public class ElectronicAddress
extends AbstractAddress  {

  private String email;
  private String facebookid;

  public void authenticateAddress() {
  }
}
```

Different forms of association

1. Strongest (Composition)
   - Implies total ownership, if the owner is destroyed, the parts are also destroyed
   - Inner classes will certainly be a composition
2. Aggregation
   - Implies has-a part ownership
   - If the owner is destroyed, the parts still exist
3. Weakest (Association)
   - General form of dependency based on the usage of features

Composition

- Total ownership
- A CreditCard exclusively belongs to one Client, and one client can have many credit cards.
- A client exclusively owns her shopping cart.
- The shopping cart and the credit card will no longer exist if the client is removed

Composition Code snippet

```
public class ClientAccount {
  private String name;
  private String id;
  private Integer yearofBirth;
  private Date dateofReg;
  private Vector  myAbstractAddress;
  private Vector  myCreditCard;
  private ShoppingCart myShoppingCart;

  public void addNewAddress() {  }
  public void addNewCreditCard() {  }
  public void setShippingAddress() {  }
}
```

```
public class CreditCard {
    private String number;
    private Date expDate;
    private ClientAccount
                    myClientAccount;

  public void validate() {   }
  public void chargeAmount() {    }
}
```

```
public class ShoppingCart {
    private Vector  myShoppingCartController;
    private ClientAccount myClientAccount;
}
```

Aggregation Relationship

- Relatively weaker composition
- Students will exist even when the Professor stops taking the class
- Ducks will exist without the Pond

Association or Dependency Relation

- **A product category in the online shop can have many products**
- **However, a product belongs to only one category.**
- **Both of them independently exist.**

```
public class Product {
   private String id;
   private String name;
   private String description;
   private Integer unitcost;
   private Category category;

  public void retrieveProductImage() {
  }
}
```

```
public class Category {
   private String id;
   private String name;
   private Vector<Product> myProduct;

   public void getProducts() {
   }
}
```

Dependency Example 2

```
class B
{
  public void doB()
  {
    System.out.println("Hello");
  }
}
class A
{
  public void doS()
  {
    B b1 = new B();
    b1.doB();
  }
} //End of class Test
```

Sequence Diagrams

Sequence diagram is drawn to represent (i) objects participating in an interaction and (ii) what messages have exchanged among those objects

Thank You

BITS Pilani

Pilani|Dubai|Goa|Hyderabad

SS ZG653 (RL 8.1): Software Architecture

Documenting Architecture with UML

Instructor: Prof. Santonu Sarkar

February 28, 2022

Unified Modeling Language (Introduction)

- **Modeling Language for specifying, Constructing, Visualizing and documenting software system and its components**
- **Model -> Abstract Representation of the system [Simplified Representation of Reality]**
- **UML supports two types of models:**

- Static **- Dynamic**

UML

- Unified Modeling Language is a standardized general purpose modeling language in the field of object oriented software engineering

- The standard is managed, and was created by, the Object Management Group.

- UML includes a set of graphic notation techniques to create visual models of object-oriented software-intensive systems

Documenting Architecture

- UML has not been designed specifically for architecture, though practitioners use UML for architecture description
  - It is upto the architect to augment UML for architecture
  - UML provides no direct support for module-structure, component-connector structure or allocation structure

Three Structures- Recap

- Module Structure
  - Code units grouped into modules
  - Decomposition
    - Larger modules decomposed into smaller modules
  - Use
    - One module uses functionality of another module
    - Layered
      - Careful control of uses relation

Illustration- Module Views

Decomposition

Layering

- Packages are used as modules
- Other approaches
  - One can use class, or interface to denote a module
- Relations
  - One can use various other UML relations to denote uses, layering or generalization

Three Structures- Recap

- Component-Connector Structure
  - Processes
    - Components are processes and relations are communications among them
  - Concurrency
    - Relationships between components- control flow dependency, and parallelism
  - Client-Server
    - Components are clients or servers, connectors are protocols
  - Shared data
    - Components have data store, and connectors describe how data is created, stored, retrieved

Illustration- CNC Views

- No standard representation exists
- UML components are used with stereotypes
- Other approaches
  - One can use class, interface, or package to denote a component
- Relations
  - One can use various other UML relations such as association class

Three Structures- Recap

- Allocation Structure
  - Deployment
    - Units are software (processes from component-connector) and hardware processors
    - Relation means how a software is allocated or migrated to a hardware
  - Implementation
    - Units are modules (from module view) and connectors denote how they are mapped to files, folders
  - Work assignment
    - Assigns responsibility for implementing and integrating the modules to people or team

Illustration-Allocation Views

- UML Deployment diagram is a good option for deployment structure
- No specific recommendation for work assignment and implementation

Thank You

BITS Pilani

Pilani|Dubai|Goa|Hyderabad

SS ZG653 (RL 8.2): Software Architecture

Introduction to Agile Methodology

Instructor: Prof. Santonu Sarkar

What is Agile Methodology

- Collaborative
  - Forms a pair for any development task to avoid error
  - Involves stakeholders from the beginning
- Interactive and feedback oriented
  - Teams interact frequently
  - Quick, and repeated integration of the product
  - Constant feedback from the stakeholder (customer)
- Iterative
  - Requirement, design, coding, testing goes through many iterations each having short duration
  - Refactoring is a part of the development process
- Test driven
  - Before building the component, define the test cases
  - Continuously test

– Scott Amber, Kent Beck

A Brief Overview

- There are 7 disciplines performed in an iterative manner
- At each iteration the software (or a part of the software) is built, tested
- Software architecture is more “agile” and it is never frozen
- UML based modeling is performed

Discipline Overview

- Model
  - Business Model
  - Analysis and Design (Architecture)
- Implementation
- Test
- Deployment
- Config Management
- Project Management
- Environment

Steps of Architecture Modeling in Agile

- Feature driven
  - Prioritize. Elaborate critical features more
- Model the architecture (UML)
- Suggested viewpoints for Agile
  - Usage scenarios
  - User interface and system interface
  - Network, deployment, hardware
  - Data storage, and transmission
  - Code distribution
- Suggested quality concerns
  - Reuse
  - Reliability, availability, serviceability, performance
  - Security
  - Internationalization, regulation, maintaince

Class Responsibilities and Collaborators (CRC) Card

What

- It is a physical (electronic) card
- One card for one class
  - Indicates the responsibilities of a class
- Collaboration
  - Sometimes a class can fulfill all its assigned responsibilities on its own
  - But sometimes, it needs to collaborate with other classes in order to fulfill its own responsibilities

Why?

- A good technique to identify a class and its responsibility during functional architecture design
- Highly collaborative and interactive process for a team of designers
- The team can do it fast

CRC Card

Class Name	Collaborators
Responsibilities assigned to this Class	If this class can not fulfill any of its assigned task on its own then which other classes it has to collaborate

CRC Card Example 1

```
class Box
{
private double length;
private double width;
private double height;
Box(double l, double w, double h)    {    length = l; width = w; height = h;    }
public double getLength()            {    return length;                        }
public double getWidth()             {    return width;                         }
public double getHeight()            {    return height;                        }
public double area()                 {    return 2*(length*width + width * height + height * length);    }
public double volume()               {    return length * width * height;                                 }
} // End of class BOX
```

Write CRC Cards for Box Class

Box	Collaborators
Responsibilities 1. Getting length 2. Getting width 3. Getting height 4. Computing area and volume	<<None>>

CRC Card Example 2

```
class B
{
  public void doB()
  {
    System.out.println("Hello");
  }
}
class A
{
  public void doS()
  {
      B b1 = new B();
      b1.doB();
  }
} //End of class Test
```

Write CRC Cards for Classes A & B

How do you create CRC Model?

Thank You

BITS Pilani

Pilani|Dubai|Goa|Hyderabad

SS ZG653 (RL 8.3): Software Architecture

Introduction to Unified Process

Instructor: Prof. Santonu Sarkar

What is (Rational) Unified Process

- While UML provided the necessary technology for OO software design
- Unified process gives a framework to build the software using UML
- Iterative approach
- Five main phases
  - Inception
    - Establish a justification and define project scope
    - Outline the use cases and key requirements that will drive the design tradeoffs Outline one or more candidate architectures
    - Identify risks and prepare a preliminary project schedule alongwith cost estimate
  - Elaboration (Architecture and Design)
  - Construction (Actual implementation)
  - Transition (initial release)
  - Production (Actual deployment)

– Rumbaugh, Booch, Jacobson

Architecture Activities

- Architecture Focused all the time
- Starts during inception and described in detail during the elaboration phase

"Development-iterative" by Dutchguilder - Own work. Licensed under Public Domain via Commons – https://commons.wikimedia.org/wiki/File:Development-iterative.png#/media/File:Development-iterative.png

Feature Driven Design

- Starts during Inception-Elaboration Phase
- Mostly used in the context of Agile development
- The requirement is modeled as a set of features
  - A feature is a client-valued functionality that can be implemented and demonstrated quickly
- 'Feature' template <*action*> **the** <*result*> (**by**|**for**|**of**|**to**) **a(n)** <*object*>
  - Add a product to a shopping-cart
  - Store the shipping-information for the customer

From Feature to Architecture

- Once a set of features are collected
- Similar features are grouped together into a module
- Set of clusters created out of the features, forms a set of modules
- This becomes the basic Module Structure
  - Recall the software architecture and views…

Use-case Model

- Use-cases are used to describe an usage scenario from the user's point of view
  - Create a basic use-case diagram
  - Elaborate each use-case
- To create Use-Case
  - Define actors (who will use this use-case)
  - Describe the scenario
    - Preconditions
    - Main tasks
    - Exceptions
    - Variation in the actors interaction
    - What system information will the actor acquire, produce or change?
    - Will the actor have to inform about the changes?
    - Does the actor wish to be informed about any unexpected changes?

First step towards Module Views

- Analysis Classes are not the final implementation level classes. They are more coarse grained. They are typically modules. They manifest as
  - External Entities
    - Other systems, devices that produce or consume information related to this system
  - Things
    - Report, letters, signals
  - Structure
    - Sensors, four-wheeled car,… that define a class of objects

Analysis Classes manifest as…

- Event Occurrences
  - Transfer of fund, Completion of Job
- Roles
  - People who interact with the systems (Manager, engineer, etc.)-- Actors
- Organizational Units
  - Divisions or groups that are important for this system
- Places
  - Location that establish the context of the overall functionality of the system

How to get these classes?

- Get the noun phrases. They are the potential objects.
- Apply the following heuristics to get a legitimate analysis class
  - It should retain information for processing
  - It should have a set of identifiable operations
  - Multiple attributes should be there for a class
  - Operations should be applicable for all instances of this class
  - Attributes should be meaningful for all instances of this class

Thank You

BITS Pilani

Pilani|Dubai|Goa|Hyderabad

SS ZG653 (RL 9.1): Software Architecture

Introduction to Patterns

Instructor: Prof. SantonuSarkar

Christopher Alexander

Source:
http://en.wikipedia.org/wiki/Christopher_Alexander

- ***The Timeless Way of Building*** is a 1979 book that ties life and architecture together
- Much of SW Architecture derives from it
- ***A Pattern Language: Towns, Buildings, Construction*** is a 1977 book on architecture

What is a (Architecture) Pattern

- A set of components (or subsystems), their responsibilities, interactions, and the way they collaborate
  - Constraints or rules that decide the interaction
  - To solve a recurring architectural problem in a generic way
  - Synonymous to architecture style

– Buschmann, F. et al, Pattern Oriented Software Architecture – Volume1, Wiley, 1996

Properties of Patterns

- Addresses a recurring design problem that arises in specific design situations and presents a solution to it
- Document existing, well-proven design experience
- Identify and Specify abstractions at the high(est) level
- Provide a common vocabulary and understanding of <u>design principles</u>
- Helps to build complex systems
- Manage software complexity

A note on Design Principles

- A set of guidelines that helps to get a good design

- Robert Martin's book on Agile Software Development says
  - Avoid Rigidity (hard to change)
  - Avoid Fragility (whenever I change it breaks)
  - Avoid Immobility (can't be reused)

OO Design Principles

- Open close
  - Open to extension and close for modification
  - Template and strategy pattern
- Dependency inversion
  - Decouple two module dependencies (A→ B)
    - A holds the interface of B. Implementer of B implments the interface.
  - Adapter pattern
- Liskov's Substitution
  - Superclass can be replaced by subclass
- Interface Segregation
  - Don't pollute an interface. Define for a specific purpose
- Single responsibility
  - One class only one task

Pattern – Three-part Schema

Context

- A scenario or situation where design problem arises
  - Describe situations in which the problem occurs
- Ideally the scenario should be generic, but it may not always be possible
  - Give a list of all known situations
- Example
  - Developing Messaging solution for mobile applications
  - Developing software for a Man Machine Interface

Problem

- Starts with a generic problem statement; captures the central theme
- Completed by *forces;* aspect of the problem that should be considered when solving it
  - It is a Requirement
  - It can be a Constraint
  - It can be a Desirable property
- Forces complement or contradict
- Example
  - Ease of modifying the User Interface (Personalization)

Solution

- Configuration to balance forces
  - Structure with components and relationships
  - Run-time behavior
- Structure: Addresses static part of the solution
- Run-time: Behavior while running – addresses the dynamic part
- Example
  - Building blocks for the application
  - Specific inputs events and their processing

Reference Model

- An ideal solution for a domain, comprising of only functional elements without any technology or operational platform
  - NGOSS framework in Telecom
  - Open financial services architecture based on the use of intelligent mobile devices," Electronic Commerce Research and Applications, 2008.

Architecture Style/Pattern

- A set of components (or subsystems), their responsibilities, interactions, and the way they collaborate
- address one or more quality concerns

Operational Platform and Infrastructure Pattern

- A topology of nodes where the functional blocks will be deployed at runtime
- A topology of hardware devices
  - Multi-tier pattern, IBM runtime patterns, availability patterns

Design Pattern

- software-centric solution to implement the application logic, data and the interaction. The solutions include (but not limited to) package structure, analysis patterns for data modeling
- GoF, Fowler's analysis pattern

Idioms

- Programming language level best practices

Abstract

Implementation

Pattern System

A pattern system for software architecture is a collection of patterns for software architecture, together with guidelines for their implementation, combination and practical use of software development

Pattern System

- Support the development of high-quality software systems; Functional and non-functional requirements
- It should comprise a sufficient base of patterns
- It should describe all its patterns uniformly
- It should expose the various relationships between patterns
- It should organize its constituent patterns
- It should support the construction of software systems
- It should support its own evolution

Pattern Classification

- It should be simple and easy to learn
- It should consist of only a few classification criteria
- Each classification criterion should reflect natural properties of patterns
- It should provide a ‘roadmap’
- The schema should be open to integration of new patterns

Problem Categories

Category	Description
Mud to Structure	Includes patterns that support suitable decomposition of an overall system task into cooperating subtasks
Distributed Systems	Includes patterns that provide infrastructures for systems that have components located in different processes or in several subsystems and components
Interactive Systems	Includes patterns that help to structure human-computer interaction
Adaptable Systems	Includes patterns that provide infrastructures for the extension and adaptation of application in response to evolving and changing functional requirements
Structural Decomposition	Includes patterns that support a suitable decomposition of subsystems and complex components into cooperating parts
Organization of Work	Includes patterns that define how components collaborate to provide a complex service

Problem Categories

Category	Description
Creation	Includes patterns that help with instantiating objects and recursive object structures
Service Variation	Comprises patterns that support changing the behavior of an object or component
Service Extension	Includes patterns that help to add new services to an object or object structure dynamically
Adaptation	Provides patterns that help with interface and data conversion
Access Control	Includes patterns that guard and control access to services or components
Management	Includes patterns for handling homogenous collections of objects, services and components in their entirety
Communication	Includes patterns that help organize communication between components
Resource handling	Includes patterns that help manage shared components and objects

	Architectural Patterns	Design Patterns	Idioms
Mud to Structure	Layers, Pipes and Filters, Blackboard		
Distributed Systems	Broker, Pipes and Filters, Microkernel		
Interactive Systems	MVC, PAC		
Adaptable Systems	Microkernel, Reflection		
Creation		Abstract Factory, Prototype, Builder	Singleton, Factory Method
Structural Decomposition		Whole-Part, Composite	
Organisation of work		Master-Slave, Chain of Responsibility, Command, Mediator	
Access Control		Proxy, Façade, Iterator	
Service Variation		Bridge, Strategy, State	Template method
Service Extension		Decorator, Visitor	
Management		Command Processor, View Handler, Memento	
Adaptation		Adapter	
Communication		Publisher-subscriber, Forwarder-Receiver, Client-Dispatcher-Server	
Resource Handling		Flyweight	Counted Pointer

Mud to Structure

Mud to Structure

- Before we start a new system, we collect requirement from customer → transform those into specifications
  - Requirements → Architecture (Optimistic View)
- "Ball of mud" is the realization
- Cutting the ball along only one aspect (like along lines visible in the application domain may not be of help)
  - Need to consider functional and non-funcational attributes

Architectural Patterns

Mud to Structure	Layers Pipes and Filters Blackboard
Distributed Systems	Broker
Interactive Systems	Model-View-Controller Presentation-Abstraction-Control
Adaptable Systems	Microkernel Reflection

Thank You

BITS Pilani

Pilani|Dubai|Goa|Hyderabad

SS ZG653 (RL 9.3) : Software Architecture

Layering Pattern

Instructor: Prof. SantonuSarkar

Petstore example again…

Suppose that the store should provide the capability for a user to
Browse the catalog of products
Select a product and put it in shopping cart

Product is stored in a Table

Name	Category	Age	Price
Labrador	Dog	1	3500
Pug	Dog	1.5	1500
Goldfish1	Fish	0.5	50

When you implement, it will look like Flipkart or Amazon…

When application runs, it displays products based on category

User Selects a productand puts it shopping cart

Many users accessing the application

Application logic is deciding product price, managing users

Product database

What you need at a minimum?

- Three sets of classes
  - One set manages display of products, ease of selection, navigation
  - Another set manages the product management, pricing
  - Another set manages the database access
- UI Layer classes
- Business Layer classes
- Database Layer classes

Layers Architectural Pattern

Helps to structure application that can be decomposed into groups of subtasks in which each group of subtasks is at a particular level of abstraction

Layers

- Implementing protocols
- Conceptually different issues split into separate, interacting layers
- Functionality decomposed into layers; helps replace layer(s) with better or different implementation

Layers – 3 part schema

Context	**A large system that requires decomposition**
Problem	Mix of low- and high-level issues, where high-level operations rely on low-level ones A typical pattern of communication flow consists of requests moving from high level to low level, and answers to requests, incoming data and notification about events traveling in the opposite direction
Forces	•Code changes should not ripple through the system •Stable interfaces; standardization •Exchangeable parts •Grouping of responsibilities for better understandability and maintainability
Solution	Structure the system into appropriate number of layers

OSI 7-Layer Model

Application	Layer 7	Provides miscellaneous protocols for common activities
Presentation	Layer 6	Structures information and attaches semantics
Session	Layer 5	Provides dialog control and synchronization facilities
Transport	Layer 4	Breaks messages into packets and guarantees delivery
Network	Layer 3	Selects route from sender to receiver
Data Link	Layer 2	Detects and corrects errors in bit sequences
Physical	Layer 1	Transmits bits: velocity, bit-code, connection etc.

Layers

- Services of Layer J are only used by Layer J+1
- No further direct dependencies between layers
- Each layer may contain many components

Dynamics

Scenario - I

Scenario - II

Dynamics

Scenario – III (layer of caching, where An intermediate layer returns the data)

Scenario – IV (In network protocol design, intermediate layer responds based on a notification

Dynamics

- Scenario V
  - Involves two stacks communication with each other

Implementation Guideline

Define abstraction criteria

Level of abstractions define the layers. Heuristics can be

- Most generic components are in lowest layer whereas the domain-specific components are in top layer

- More stable components (which hardly undergoes change) are in lower layer. Use degree of stability to decide layers

- Distance from hardware
  - User-visible elements
  - Specific Application Modules
  - Common Service Levels
  - OS Interface Level
  - Hardware

Determine the no. of abstraction levels

- Typically each abstraction level is one layer
- Map the abstraction levels to layers
- Use mechanisms to keep number of layers to optimum number (say 3 layers for a typical self-service based application)
  - Too Few Layers → Can Result in Poor Structure
  - Too Many Layers → Impose Unnecessary Overhead

Complete Layer specification

A) Name the layer and assign tasks
   - Highest layers are system functionality perceived by the user
   - Lower layers are helpers
   - In bottom up approach – create generic tasks at the lowest level- sort of infrastructure
   - Requires experience to achieve this

B) Specify the services
   - Strict separation of layers
   - No component should spread over two layers
   - Inverted pyramid of use

Construct Each Layer

- Specify layer interface
  - Use a black box approach
  - Layer N treats Layer N-1 as a black box
- Structure each layer
  - Identify components inside each layer
  - Bridge or strategy pattern can help
    - Supports multiple implementations of services provided by a layer
    - Supports Dynamic exchange of algorithms used by a user

Inter layer communication

- Identify communication mechanism
  - Push: upper layer invokes a service of lower one
  - Pull mechanism
- Layer decoupling
  - Lower layer not aware of higher layer & vice versa
    - Changes in Layer J can ignore the presence and identity of Layer J+1 [Suitable for Top-up communication]
  - What happens in Bottom up scenario?
    - Use of call backs
    - Upper layer registers with lower layer
    - Lower layer maintains mapping between event and callback functions
  - (reactor and commnd pattern)

Design an error handling strategy

- Define an efficient strategy
- Handling may be expensive – errors need to propagate

Benefits

Benefits

- Reuse of layers
- Support for standardization
- Dependencies are kept local
- Exchangeability

Liabilities

- Cascades of changing behavior
- Lower efficiency
- Unnecessary work
- Difficulty in establishing the correct granularity

Examples

Your application

Middleware- J2EE

- Can replace vendor (oracle, IBM,JBOSS)

JVM

- Can replace the vendor

OS

- Switch from Windows to Unix

Presentation

Application logic

- Application logic, business rules

Domain layer

- Conceptual model of domain elements

Database

- Tables, indexes

System services

Resource Mgmt

- Secuity monitor, process mgr, I/O mgr, virtual memory mgr

Kernel

- Interrupt, exception, thread scheduling & dispatching

Hardware abstraction

- Hides h/w differences between different processor families

Virtual Machines

Information Systems

Operating system- Windows NT

Can you find (at least 2) more popular uses and document them?

Layers

Pattern	Description
Context	A large system that requires decomposition
Problem	Mix of low- and high-level issues, where high-level operations rely on low-level ones A typical pattern of communication flow consists of requests moving from high level to low level, and answers to requests, incoming data and notification about events traveling in the opposite direction Forces •Code changes should not ripple through the system •Stable interfaces; standardization •Exchangeable parts •Grouping of responsibilities for better understandability and maintainability
Solution	Structure the system into appropriate number of layers
Variants	Relaxed Layered System Layering Through Inheritance
Benefits	Reuse of layers Support for standardization Dependencies are kept local Exchangeability
Liabilities	Cascades of changing behavior Lower efficiency Unnecessary work Difficulty in establishing the correct granularity

Thank You

BITS Pilani

Pilani|Dubai|Goa|Hyderabad

SS ZG653 (RL 10.1): Software Architecture

Pipe and Filter Pattern

Instructor: Prof. Santonu Sarkar

Pipes and Filters

A structure for systems that process a stream of data

Filter

- Has interfaces from which a set of inputs can flow in and a set of outputs can flow out
- processing step is encapsulated in a filter component
- Independent entities
- Does not share state with other filters.
- Does not know the identity to upstream and downstream filters
- All data does not need to be processed for next filter to start working

Pipes

- Data is passed through pipes between adjacent filters
- Stateless data stream
- Source end feeds filter input and sink receives output.

Recombining filters allows you to build families of related systems

Pipes and Filters – 3 part schema

Context	**Processing Data Streams**
Problem	System that must process or transform a stream of input data. Multi-stage operations on data (workflow) Many developers may work on different stages Requirements may change
Forces	• Future enhancements – exchange processing steps or recombination • Reuse desired, hence small processing steps • Non adjacent processing steps do not share information • Different sources of data exist (different sensor data) • Store final result in various ways • Explicit storage of intermediate results should be automatically done • Multiprocessing the steps should be possible
Solution	Pipes and filters – data source to data sink

Simple case

```
ls scores| grep -e July | sort
```

- Data source – file containing scores
- Filters
  - Listing of scores
  - Filtering only July scores
  - Sorting of records

Known Example –Compiler Design

Various Components

Data Source

- Represents input to the system
- Sequence of data of the same structure or type

- Filters are processing units
- Enriches – computing and adding information
- Refine – concentrating or extracting information
- Transforms – delivering data in some other representation

Data Sink

- collects results from end of the pipeline
- Active: pulls results from preceding processing stage
- Passive: allows preceding filter to push or write the results into it

Scenario-1

- Push pipeline [Activity starts with the Data source]
- Filter activity started by writing data to the filters
- Passive Filter [Use direct calls to the adjacent pipeline]

Scenario-2

- Pull pipeline
- Control flow is started by the data sink calling for data

Scenario 3

– Push-pull mixed pipeline

Scenario 4- Multiprocess

- All filters actively pull, compute and push data in a loop
- Each filter runs its own thread of control
- Filters are synchronized by buffering pipe between them

Implementation

#	Steps
1	Divide the system's task into a sequence of processing stages
2	Define the data format to be passed along each pipe
3	Decide how to implement each pipe connection
4	Design and implement the filters
5	Design the error handling
6	Set up the processing pipeline

Initial Steps

1: Divide the systems tasks into sequence of processing stages

- Each stage must depend on the output of the predecessor
- All stages conceptually connected by data flow

2: Define data format to be passed along each pipe

- Define a uniform format results in the highest flexibility because it makes recombination of filters easy
- Define the end of input marking

Design Pipe and Filter

3. Pipe

- Decision determines active or passive filter
- Using a separate pipe mechanism that synchronises adjacent active filters provide a more flexible solution

4. Filter

- Design Depends on
  - Task it must perform
  - Adjacent pipe
- Active or Passive filters
  - Active filter pulls data from a pipe
  - Passive ones get the data
- Implemented as threads or processes
- Filter reuse
  - Each filter should do one thing well
  - Can read from global or external files for flexible configuration

Final Steps

5: Design error handling

- Never neglect error handling
- No global state shared; error handling hard to address
- Strategies in case of error – depend on domain

6: Setup processing pipeline

- Use of standardised main program
- Use of user inputs or choice

Variants

- Tee and Join pipeline
  - Filters with more then one input and/or more than one output

Benefits

- No intermediate files necessary, but possible
- Filter addition, replacement, and reuse
  - Possible to hook any two filters together
- Rapid prototyping of pipelines
- Concurrent execution
- Certain analyses possible
  - Throughput, latency, deadlock

Liabilities

- Sharing state information is expensive or inflexible
- Data transformation overhead
- Error handling can be a problem
- Does not work well with interactive applications
- Lowest common denominator on data transmission determines the overall throughput

Pipe and Filter in Cloud based Service

- Most PaaS service providers (Amazon, Azure, Google) provides message oriented service orchestration
- Pipe-n-Filter is a common pattern
- Azure
  - The components having worker role are the filters
  - Pipe is the queuing service
- Amazon
  - EC2 instances are filters, communicating via SQS pipes

Thank You

BITS Pilani

Pilani|Dubai|Goa|Hyderabad

SS ZG653 (RL 10.2): Software Architecture

Blackboard Architecture

Instructor: Prof. Santonu Sarkar

Context and Problem

- A set of heterogeneous specialized modules which dynamically change their strategies as a response to unpredictable events
  - Non-deterministic strategies
- Problem
  - When there is no deterministic solutions to process raw data, and it is required to interchange algorithms processing some intermediate computation
  - Solutions to partial problems require different representation
  - No predetermined strategy is present to solve a problem (in functional decomposition sequence of activations are more hard-coded)
  - Dealing with uncertain knowledge

Forces

- A complete search of the solution space is not possible
- Different algorithms to be used for partial solutions
- One algorithm uses results of another algorithm
- Input, intermediate data, output can have different representation
- No strict sequence between algorithms, one can run them concurrently if required

Examples

- Speech recognition (HEARSAY project 1980)
- Vehicle identification and tracking
- Robot control (navigation, environment learning, reasoning, destination route planning)
- Modern machine learning algorithms for complex task (Jeopardy challenge)
- Adobe OCR text recognition
- Modern compilers tend to be more Blackboard oriented

Blackboard Pattern

- Two kinds of components
  - Central data structure — blackboard
  - Components operating on the blackboard
- System control is entirely driven by the blackboard state

Components of Blackboard

- The blackboard is the shared data structure where solutions are built
- The control plan encapsulates information necessary to run the system
  - It is accessed and up dated by control knowledge sources
- DomainKS are concerned with the solving of domain specific problems
- Control KS adapt the current control plan to the current situation
- The control component selects, configures and executes knowledge sources

Solution Structure

Runs in a loop
- Monitors change in blackboard
- Activates next KS
- Selection strategy may depend on ControlData

Highly specialized modules
- Each is different
- Has a set of triggering conditions
- Executable code that retrieves data from the blackboard and updates blackboard
- Don't interact with each other

Shared datastore containing partial solutions

Updates control Data Such as:
- progress estimation,
- computation cost to execute a KS
- Control plan

Automated Robo Navigation

- Robot’s high level goal is to visit a set of places as so on as possible
  - The successive sub-goals are
    - to decide on a sequence of places to visit,
    - to compute the best route, and
    - to navigate with a constraint of rapidity.

Benefits

Benefits

- Experimentation- try with different strategies,
- Support for modifiability- each KS is strictly decoupled
- Reuse of KS
- Fault-tolerance even when the data is noisy

Liabilities

- Difficulty in testing
- No good solution guaranteed
- Computational overhead in rejecting wrong solutions
- High development effort
- Concurrent access to blackboard must be synchronized, parallelization is difficult

Thank You

BITS Pilani

Pilani|Dubai|Goa|Hyderabad

SS ZG653 (RL 11.1) : Software Architecture

Distributed Systems

Instructor: Prof. Santonu Sarkar

Distributed Systems

BROKER PATTERN

Context

Complex environment comprises of distributed systems

- You want to take advantage of computing power of many CPUs, or a cluster of low-cost systems
- A software may be available only on a specific computer
- Due to security reasons, you want to run different parts in different systems
- Some services are provided by business partners over the internet

Problem with distributed components

- To build a complex sw system as a set of decoupled, interoperating components rather than a monolith.
  - Greater flexibility, maintainability, changeability
  - Partitioning into independent components makes system distributable and scalable.
- Require a flexible means of inter-process communication
  - If participating components handle communication, there can be several issues
    - System depends on which communication mechanism used
    - Clients need to know location of servers

Forces

- It should be possible to distribute components during deployment– application should unaware of
  - Whether the service is collocated or remote
  - If remote, where the location of the server is
- Need to exchange, add, or remove components at run-time
  - Must not depend on system-specific details to guarantee portability and interoperability
- Architecture should hide system-specific and implementation-specific details from users of components and services
  - Specifically communication issues, data transfer, security issues

Broker Pattern: Solution

- Introduce a broker component to achieve better decoupling of clients and servers
  - Servers: register themselves with the broker and make their services available to clients through method interfaces.
  - Clients: access the functionality of servers by sending requests via the broker
- The Broker:
  - Locating the appropriate server and forwarding a request to that server
  - Transmitting results and exceptions back to the client

Broker Pattern: Solution -- 2

- Reduces the development complexity
  - Introduces object model where distributed services are encapsulated within objects.
- Broker systems offer a path to the integration of two core technologies:
  - Distribution
  - Object oriented design
- Object oriented design from single applications to distributed applications that can
  - run on heterogeneous machines and
  - written in different programming languages.

Broker Pattern: Structure

- Participating components
  - Clients
  - Servers
  - Brokers
  - Bridges
  - Client-side proxies
  - Server-side proxies

Broker

Necessary abstraction that makes distribution possible by providing the contract about the service that the server is going to provide without exposing the implementation details on the server side

Broker pattern: Implementation

1. Define an object model or use an existing one
2. Decide which kind of component-interoperability the system should offer
3. Specify the APIs the broker component provides for collaborating with clients and servers
4. Use proxy objects to hide implementation details from clients and servers
5. Design the broker component in parallel with steps 3 and 4
   - broken down into nine steps
6. Develop IDL compilers

Scenario 1

The client simply invoke **performFunction()** on the client proxy as if it were a local call

It has no impact at all to the client even when it is changed

Works well, but you can't change the server location at run-time

Broker as service locator

- Broker resides at a well-known location and then expose that location to the client
- Broker is responsible for locating the server for the client.
- Broker also implements a repository for
  - adding and removing server components
  - Makes it possible to add, remove, or exchange server components at run time
- Once the server is located, client and server talks directly

Broker behavior server look-up

Broker as Intermediary

- In some situations, direct communication between client and server is not desirable
  - For security reasons you may want to host all the servers in your company's private network, behind a firewall, and
  - only allow access to them from the broker
- Broker forwards all the requests and responses between the server and the client instead of direct communication

Broker as intermediary

Broker Known Uses- CORBA

- CORBA is the oldest amongst the middleware technologies used in today's IT world
- CORBA stands for Common Object Request Broker Architecture and is defined by
  - its interfaces
  - their semantics and
  - protocols used for communication (Internet Inter-Orb Protocol IIOP)
  - CORBA supports the basic Broker pattern.
- For the basic functionality CORBA supports the so called Dynamic Invocation Interface (DII) on the client-side
- From IDL both client proxy (client stub) and the server proxy (called skeleton)
- Various ORB extensions support a wide variety of advanced features
  - CORBA supports client-side asynchrony via standardized interface. Server-side asynchrony is only supported proprietarily.

Broker Known Uses- RMI

- Sun's Java Remote Method Invocation (RMI) is based on the Transparent Broker variant pattern
- The client-side proxy (so called stub) and the server-side invoker (so called skeleton) have to be created manually by an additional compilation step
- In contrast to CORBA the ServiceInterface is not written in an abstract IDL, but in Java.
- RMI is limited to the usage of Java
- To establish interoperability RMI-IIOP is provided
- RMI doesn't support client-side or server-side asynchrony out of the box- you have to implement
- A central naming service (so called RMI registry) allows clients to look up servant identifiers

Broker Known Uses- .NET

- Microsoft's .NET Remoting platform implements the Transparent Broker variant pattern to handle remote communication.
  - Since the .NET platform supports reflection to acquire type information, the client proxy is created automatically at runtime behind the scene, completely transparent for the application developer.
  - No separate source code generation or compilation step required.
  - The interface description for the client proxy can be provided by MSIL-Code or by a WSDL-Description of the interface itself.
  - The client proxy is responsible of creating the invocation request, but is not in charge of any communication related aspects.
- The remote communication functionality of .NET Remoting is encapsulated within a framework consisting of marshalers (so called Formatters in .NET Remoting) and Transport Channels, which abstract from the underlying transport layer.
  - Flexible, allows any custom extensions to fulfil for example QoS requirements.
  - Supports the client-side asynchrony broker variants. Lifecycle management strategies for servants are also included within the framework.
  - Doesn't have a central naming or lookup system. Clients have to know the object reference of the servant in advance. However different strategies exist to avoid the hardcoding of the server destination inside the client application code

Benefits

- **Location Independence**-- Clients do not have to care where an object is located, though for remote objects, they always have to use the more complex interface, unless a Transparent Broker is used.
- **Type System Transparency**—Differences in type systems are coped with by a intermediate network protocol. The marshaler translates between programming language specific types and the common network protocol.
- **Isolation**-- Separating all the communication-related code into its own layer isolates it from the application. You can decide to run the application distributed or all on one computer without having to change any application code.
- **Separation of Concerns** —The communication and marshaling concerns are properly encapsulated in the requestor, invoker, and marshaler.
- **Resource Management**—The management of network and other communication resources such as connections, transfer buffers and threads is encapsulated within the Broker Participants and therefore seperated from the application logic.
- **Portability** —Platform dependencies which typically arise from low level I/O and IP communication are encapsulated within the Broker Participants and therefore separated from the application logic.

Liabilities

- Error Handling—Clients have to cope with the inherent unreliability and the associated errors of network communication.
- Overhead —Developers can easily forget about the location of objects, which can cause overhead if the expenses of remote communication are not considered
- Performance
- Lower fault tolerance (server fails, broker fails, ...)
- Testing and debugging

BITS Pilani

Pilani|Dubai|Goa|Hyderabad

SS ZG653 (RL 12.1) : Software Architecture

Interactive Systems

Instructor: Prof. Santonu Sarkar

The Need: Example

You are going to build a web-based software application for this example

Grade A: 5
Grade B: 15
Grade C: 50
Grade D: 25
Grade F: 5

Grade	% of Students
A	5
B	15
C	50
D	25
F	5

- Such an application typically retrieves data and displays it for the user.
- After the user changes the data, the system stores the updates in the data store.

Context and Problem

- Context
  - Interactive application with flexible human-computer interface
- Problem
  - Because the flow of information is between the data store and UI, one may be inclined to data and UI to reduce the amount of coding and to improve application performance.
  - However, the major problem is that UI tends to change much more frequently than the data storage system.
  - Another problem is that business applications tend to incorporate complex business logic which also gets mixed up

Forces

- Same data with different presentation needed
- Display, behavior and data-manipulation to be reflected immediately
- UI is a different element altogether
  - Changes are very frequent, more than data and business logic
    - One should be able to test only the UI part
  - Skillset: HTML page designer skills are different from core app development. It is desirable to separate UI with the rest of the app
- Changing look-feel (even device dependency) shouldn't affect the core app
- In web-app, one UI action can trigger many processing and then outcome may need to be collated into one

Model View Controller

Event (keyboard, mouse) passed to the Controller

- **Multiple Controllers possible.**
- **Each controller interprets input events and takes action**

- **Multiple Views**
- **Each view manages display of information**

- **Captures data**
- **Sends data to View**
- **Responds to Controller for changing data**

Model-View-Controller

1. User makes change in UI, which comes to controller
2. Controller interprets the change, informs model for the change
3. Model makes change in data
4. Model notifies all the relevant views regarding the change
5. View gets latest data and updates the display

Model-View-Controller

1. User makes change in UI, which comes to controller
2. Controller interprets the change, informs model for the change
3. Model makes change in data
4. Model notifies all the relevant views regarding the change
5. View gets latest data and updates the display

MVC Dynamics- User Event

MVC Initialization

Other Dynamic Scenarios

- Update to display alone i.e. no change in the controller
- System exit. Sequence of deletion or destruction of objects
- Scenario of update with multiple View-Controller pairs
- …

RL 12.2

MVC IMPLEMENTATION

Implementation

	#	Steps
Fundamental steps for realising a MVC	1	Separate human-computer interaction from the core functionality
	2	Implement the set-up of MVC setup part
	3	Design and implement Model
	4	Design and implement views
	5	Design and implement controllers

Initial Part

1: Separate Human-Computer Interaction

- Analyze and separate the core functionality of your system and data from input and output
- Decide on the functionality to be exposed to View(s) and controller(s)
- Decide how many views and controllers you need

2. Set-up MVC

- Write code that calls Model initialization method
- Write code to call view creation
- Write the Start the event processing mechanism

3: Design the Model

- Encapsulate the data and functionality to access and modify data
  - Bridge to the core business logic of the system
- Publish-Subscribe design pattern
  - Implement a registry that holds references of observers (Views and Controllers)
  - Provide APIs for an observer to subscribe and unsubscribe
  - Implement notify() which will be called every time the (other) parts of the system change the model's state (and data)
    - In turn calls update() of each observer (a view or a controller)

4: Design and Implement Views

- Design the appearance of the View(s)
  - Presents information to the user
  - Each important data entity should have a view
- Each view may have its own controller (sometimes a set of views can also share a controller)
  - Creates a controller using the Factory Method design pattern (makeController() in View class)
  - View exposes a method for controller to use directly bypassing the model (a scenario when model's state is not changed by the action of a user)
- Implement update() method
  - retrieves data from the model and presents it on the screen
- Initialization
  - Register with model
  - Set up relationship with the controller
- <u>Look for efficiency of fetching the data from model to build the view</u>
  - View to decide based on changes if "Draw" needs to be called

5: Design and Implement Controllers

- Initialization procedure
  - Binds the controller to it's View and Model and enables event processing
  - Subscribes to the change-propagation mechanism
  - Set up relationship with the View
- Implement event processing
  - accept user input as events; events delivery to the controller is platform dependent
  - Event translated into requests for the model or the associated view
- Controller behavior dependent on state of model
  - Registers for change propagation
  - Implements its update() procedure

Variants

- Document View
  - Document = Model
  - View = View Controller
- Loose coupling of View and Controller enables multiple simultaneous and synchronized but different views of the same document

RL 12.3

MVC IN AJAX BASED APPLICATIONS

AJAX Based Web Applications

- Traditional web based UI is thin-client based
  - Browser sends HTTP GET/POST request to the server
  - Entire web page is refreshed
  - Client side Javascript is used for field validations
  - One request may entail retrieving data from many servers
- AJAX running on a browser
  - Makes asynchronous calls to the server without refreshing the primary HTML
  - No longer a thin client, provides a richer user interface

AJAX in Action

- Since web server expects requests to come serially
  - Ajax controller serializes the request and then sends them
- Can provide a rich user interface over browser
- Many SaaS cloud applications make use of AJAX
- AJAX is a perfect example of MVC pattern

AJAX and MVC

- Controller
  - Some part is in the server (traditional Struts framework) and some part moves to the client as AJAX controller
  - It is possible to reflect the changes in the model immediately, unlike a traditional Web application
- View is the HTML+Javascript
- Model is the business logic that changes the underlying data
  - Model functions can change without impacting the UI or the controller

Benefits

- Multiple views of the same model
- Synchronized views
- ‘Pluggable’ views and controllers
- Exchangeability of ‘look-and-feel’
- Framework potential

Liabilities

- Increased complexity
- Potential for excessive number of updates
- Intimation connection between view and controller
- Close coupling of views and controllers to a model
- Inefficiency of data access in view
- Difficulty of using MVC with modern user-interface tools

THANK YOU

BITS Pilani

Pilani|Dubai|Goa|Hyderabad

SS ZG653 (RL 13.1): Software Architecture

Adaptable Systems

Instructor: Prof. Santonu Sarkar

Adaptable Systems

MICROKERNEL

Microkernel – 3 part schema

- Context
  - The development of several applications that use similar programming interfaces that build on the same core functionality
- Problem
  - Developing software for an application domain that needs to cope with a broad spectrum of similar standards and technologies
  - Continuous evolution (software and hardware), platform must be extensible, portable and adaptable
- Solution- Microkernel
  - Encapsulate the fundamental services of in a microkernel component
    - Often encapsulates the hardware dependent aspects
  - Allows other components to interact with microkernel through well-defined API
  - Size should be really small

Microkernel

Participating Components

- Internal Servers
  - Implements additional services, typically device drivers
  - Runs as a process or a shared lib
- External Servers
  - More complex functionality for the clients. Typically run as separate processes
- Adapters
  - Provides interface between external server and client
- Clients
- Microkernel
  - Only core functionality + communication mechanism
  - Interacts with Internal server

Microkernel

- Main component of the pattern, which is optimized for memory
- Provides core mechanisms
- Encapsulates system specific dependencies : Hardware dependent parts
- Maintain system resources such as Processes, Files
- Provides a set of atomic services- known as mechanisms
  - More complex functionality can be built by combining atomic services

- In OS design, Microkernel would run in the privileged mode
- Everything else runs in user mode

Server

Internal Server

- Also known as subsystem
- Extends functionality provided by Microkernel
- Microkernel invokes services based on client requests
  - Example: Device drivers
- Consider as extensions which are accessible to microkernel only

External Server (aka Personality)

- Implements it's own 'view' of the underlying microkernel
  - E.g. OS/2 flavor on top of microkernel
  - MacOS on top of microkernel
- Different servers implement different policies for specific application domains
- Perform:
  - Receives service requests from client applications using the communication facilities provided by the microkernel
  - interprets these requests
  - executes the appropriate services
  - returns results to the clients

Client

Client

- Represents an application
- Associated with exactly one external server
  - No direct communication, through Adapter only
- Accesses the services provided by the external server
  - thru adapter as a service request

Adapter

- Protect the client from implementation details of the microkernel
  - Uses communication mechanism provided by the microkernel
- Service requests from client
  - forwards the call to the appropriate server
  - Invokes methods of external servers on behalf of clients

Microkernel Layered design

Client Request

External Server Request

- External server requests a service to microkernel
- This service is provided by an internal server
- Internal server can be a separate process or a shared library

RL 13.2

MICROKERNEL DETAILED

Implementation

#	Steps
1	Analyze the application domain
2	Analyze the external servers
3	Categorize the servers
4	Partition the categories
5	Find a consistent and complete set of operations and abstractions
6	Determine strategies for request transmission and retrieval
7	Structure the microkernel component
8	Specify the programming interfaces for the microkernel
9	Microkernel is responsible for all the system resources
10	Design and implement the internal servers
11	Implement the external servers
12	Implement the adapters
13	Develop client applications

Client Requirement

1. Analyze Application Domain

- Identify the functionality required
- Perform analysis to identify policies the external servers need to offer for the application domain

2. Analyze External Servers

- Analyze the policies the external servers are going to provide to define
  - High level interfaces required to be provided
  - List of services and service categories necessary

Category of Core Services

3. Categorize Services

- Identify all core service functionality
- Group them into a set of semantically-independent categories
  - Example: memory mgmt, process mgmt, low level I/O, communication
- Identify categories which are not directly related to application domain
  - Page handler for processes, file system mgmt
- Focus is to build the system infrastructure

4. Partition categories

- Separate the categories of services that should belong to microkernel and internal servers
  - Typically based on frequency of use or hardware dependent will be part of microkernel
  - Page fault handlers, drivers and file system form part of internal servers

Core Service Design

5. Identify operations for each category

- Find a consistent and complete set of operations and abstractions
- Microkernel provides mechanisms for the policies of external server; each policy is implemented thru use of services the microkernel offers
- Example
  - Creating and terminating processes and threads

6. Determine communication strategies

- Specify the facilities that microkernel should provide for communication between components
  - Synchronous/Asynchronous
  - Relationship: one-to-one, many-to-one or a many-to-many
- Build core services of message passing or shared memory, based on which the other services can be built

Microkernel

7:Structure the microkernel component

- Separate system-specific parts from system-independent parts
- Use Layer pattern
  - Lowermost – system specific
  - Uppermost – system independent; focusing on the services which microkernel will expose

8:Specify the microkernel API

- Decide how the interfaces should be available externally
- Microkernel as a
  - Separate process
  - Module physically shared by other components
- Main Quality attribute: Efficiency and not make microkernel a bottle neck

Resource Management

9:Design Microkernel resource mgmt

- Implement strategies for sharing, locking, allocation and deallocation of resources
- Maintain the information about resources and allow access to them in a coordinated and systematic way

10:Design & Implement Internal Servers

- Separate processes or shared libraries centered around resources
  - Active servers are implemented as processes
  - Passive servers are implemented as shared libraries
- Graphics card driver → shared libraries
- Page fault handler → separate processes

Client Access

11:Implement external servers

- Each external server is implemented as a separate process that provides it's own service interface
- Internal architecture depends on the policies it comprises
- Specify how external servers dispatch requests to their internal procedures

12:Implement adapters

- Decide on strategy of adapter; one adapter for all clients or every client associated with one adapter (trade off)
- Adapter needs to package all relevant information into a request and forwards the request to appropriate external server
- Use "Proxy" pattern

Variants

- Microkernel system with indirect Client-Server connections
- Distributed Microkernel System

Example

- While Traditional BSD kernel was monolithic, microkernel architecture is loosely coupled
- During startup microkernel-based system requires device drivers (internal server), which are not part of the kernel.
  - This means that they are packaged with the kernel in the boot image, and the kernel supports a bootstrap protocol that defines how the drivers are located and started;

https://en.wikipedia.org/wiki/Microkernel

Microsoft's Singularity OS

- Microkernel provides memory mgmt, process creation, communication, scheduling and I/O
  - Kernel calls only passes value
  - A process can't change the state of another process
- Code in Singularity is either verified by the compiler or trusted
- Software isolated processes
  - Encapsulates applications for failure isolation

http://research.microsoft.com/pubs/52716/tr-2005-135.pdf

Relationship with Other Patterns

- Broker
  - Coupling
    - Broker is decentralized
    - Microkernel is more tightly coupled
  - Client communiction
    - In broker, clients communicate via message passing
    - Microkernel uses adapter strategy so that clients do not call the service directly
- Layer
  - Microkernel can be thought of as a variant of layered architecture
  - Each layer is a VM
    - Lowest layer – internal server
    - Middle layer external server+adapter
    - Topmost layer – client application

Pros and Cons

Benefits

- Portability
- Flexibility and Extensibility
- Separation of policy and mechanism
- Easy to ensure security as everything else can run in user mode
- Scalability
- Reliability
- Transparency

Liability

- Performance is worse than a monolithic kernel
  - Much more IPC due to internal and external servers, unlike a monolithic solution
- Complexity of design and implementation

Known Uses

- Amoeba operating System
- Chorus
- Windows NT
- Symbian
- iOS – based on Darwin which is a microkernel architecture
- Find (at least 2) more popular uses and document them
- **Conceptual Architecture of the Linux Kernel** http://docs.huihoo.com/linux/kernel/a1/index.html

BITS Pilani

Pilani|Dubai|Goa|Hyderabad

SS ZG653 (RL 13.3): Software Architecture

(Adaptable Systems) Reflection Pattern

Instructor: Prof. Santonu Sarkar

March 3, 2015

Basic Idea

- Reflection pattern allows runtime discovery of interfaces and dynamic calling of discovered interfaces

Context and Problem

- Context
  - Support for variation and change in the structure of the system
  - Create architecture that's open to change
- Problem
  - How to build systems that support unanticipated changes?

Solution

- Encapsulate information about properties and variant aspects of the application's structure, behavior, and state into a set of meta-objects.
  - Separate the meta-objects from the core application logic via a two-layer architecture:
  - The meta level contains the meta-objects
  - The base level contains the application logic.
- Base-level objects consult an appropriate meta-object before they execute behavior or access state that potentially can vary.

Examples

- Java offers Reflection through capabilities provided in the Java Reflection API
- Broker Pattern – CORBA and RMI
  - offers Reflection in terms of its Dynamic Invocation Interface (DII) and Interface Repository
  - RMI

Reflection Pattern in CORBA

- For the basic functionality CORBA supports the so called Dynamic Invocation Interface (DII) on the client-side
- The CORBA Interface Repository is simply another CORBA object that stores the contents of IDL defined types
- Once you add an IDL type to the repository, any other CORBA object can query the repository at runtime and:
  - dynamically understand an interface and make calls on it
  - dynamically manipulate an interface by adding or deleting interface methods

RMI and Reflection

- Sun's Java Remote Method Invocation (RMI) is based on the Transparent Broker variant pattern
- Java reflection is used to determine the properties, events and methods
- This is turn is used to serialize an object while transmitting from one distributed component to another

Java Reflection Mechanism

- Java to support Remote Method Invocation, object serialization, and JavaBeans
  - Reflection is used by JavaBeans to determine the properties, events, and methods that are supported by a particular bean
  - Java.lang.Class is the entrypoint for reflection
- How Java reflection is used?
  - **JUnit** – uses reflection to parse @Test annotation to get the test methods and then invoke it.
  - **Spring** – dependency injection, read more at Spring Dependency Injection
  - **Tomcat** web container to forward the request to correct module by parsing their web.xml files and request URI.
  - **Eclipse** auto completion of method names
  - **Struts**
  - **Hibernate**

More on Java reflection

- Class Level
  - Getting info regarding class, superclass, members, package, modifiers, methods, implemented interfaces
- Field Level
  - Getting info on field types, get/set info
- Method and constructor Level
  - Getting public methods and constructor information
  - Invocation of methods,
  - Instantiation of object by calling constructor
- Annotation Level

Power of Java Reflection

- Load a class
- Determine if it is a class or interface
- Determine its superclass and implemented interfaces
- Instantiate a new instance of a class
- Determine class and instance methods
- Invoke class and instance methods
- Determine and possibly manipulate fields
- Determine the modifiers for fields, methods, classes, and interfaces

Class and Interface

- Load a class

```
Class c =
  Class.forName
  ("Classname")
```

- Determine if a class or interface

```
c.isInterface ()
```

- Determine Inheritance and Implementation
  - Superclass

```
Class c1 =
  c.getSuperclass ()
```

  - Superinterface

```
Class[] c2 =
  c.getInterfaces ()
```

- Determine implemented interfaces

```
Class[] c2=c.getInterfaces()
```

- Determine constructors

```
Constructor[] c0 =
  c.getDeclaredConstructors()
```

- Instantiate an instance
  - Default constructor

```
Object o=c.newInstance()
```

  - Non-default constructor

```
Constructor c =
  c.getConstructor
  (class[]{…})
Object i = c.newInstance
  (Object[] {…})
```

Methods and Members

- Determine methods

```
Methods[] m =
  c.getDeclaredMetho
  ds ()
```

- Find a specific method

```
Method m =
  c.getMethod
  ("methodName",

  new Class[] {…})
```

- Invoke a method

```
m.invoke (c, new
  Object[] {…})
```

- Determine modifiers

```
Modifiers[] mo =
  c.getModifiers ()
```

- Determine fields

```
Class[] f =
  c.getDeclaredFields()
```

- Find a specific field

```
Field f = c.getField()
```

- Modify a specific field
  - Get the value of a specific field : `f.get (o)`
  - Set the value of a specific field: `f.set (o, value)`

Reflection Liabilities

- **Poor Performance**
  - Since reflection resolve the types dynamically, it involves extra processing like scanning to find the class to load and introspect, causing slow performance.
- **Security Restrictions**
  - Reflection requires runtime permissions that might not be available for system running under security manager. This can cause you application to fail at runtime because of security manager.
- **Security Issues**
  - Using reflection we can access part of code that we are not supposed to access, for example we can access private fields of a class and change it's value. This can be a serious security threat and cause your application to behave abnormally.
- **High Maintenance**
  - Reflection code is hard to understand and debug, also any issues with the code can't be found at compile time because the classes might not be available, making it less flexible and hard to maintain.

Patterns and Tactics

- Tactics are building blocks using which patterns are created
- A pattern without any tactic is not useful

Architectural Pattern and Modifiability									
	Localize Change			Reduce ripple effect			Defer Binding		
PATTERN	Increase semantic coherence	Factor common services	Generalize	Maintain existing interface-wrapper	Restrict communication path	Use intermediary	Use runtime registration	Config file- startup time decision	Runtime binding
Layering	X	X	X		X	X			
Pipe-n-filter	X				X				
Blackboard	X	X	X		X	X			
Broker	X	X	X		X	X	X		X
MVC					X				X
Microkernel	X	X			X	X			
Reflection	X								

BITS Pilani

Pilani|Dubai|Goa|Hyderabad

SS ZG653 (RL 14): Software Architecture

Design Pattern

Instructor: Prof. Santonu Sarkar

March 17, 2015

Design Patterns

- **A Pattern presents a proven advice in standard format**
- **A design pattern gives advice about a problem in software design.**
- **Attributes of design Pattern:**

1. **Name** **[name of the design pattern]**
2. **Context** **[situation where it can be applied]**
3. **Problem** **[The exact problem to be solved]**
4. **Solution** **[The solution outline in the form of class diagram]**

	Architectural Patterns	Design Patterns	Idioms
Mud to Structure	Layers, Pipes and Filters, Blackboard		
Distributed Systems	Broker, Pipes and Filters, Microkernel		
Interactive Systems	MVC, PAC		
Adaptable Systems	Microkernel, Reflection		
Creation		Abstract Factory, Prototype, Builder	Singleton, Factory Method
Structural Decomposition		Whole-Part, Composite	
Organisation of work		Master-Slave, Chain of Responsibility, Command, Mediator	
Access Control		Proxy, Façade, Iterator	
Service Variation		Bridge, Strategy, State	Template method
Service Extension		Decorator, Visitor	
Management		Command Processor, View Handler, Memento	
Adaptation		Adapter	
Communication		Publisher-subscriber, Forwarder-Receiver, Client-Dispatcher-Server	
Resource Handling		Flyweight	Counted Pointer

From GoF classification

Creational	Factory Pattern	About creation
	Factory Method	About creation, also called Idioms
	Singleton	
Structural	Adapter	About structural decomposition
	Composite	
	Decorator	About service extension
	Proxy	About access control
Behavioral	Iterator	
	Observer	Communication
	Visitor	About service extension
	Strategy	Service variation
	Command	Organization of task

What is Creational Pattern?

- Provides mechanisms to create objects in a more generic manner
- Direct object creation sometimes can make the application susceptible to frequent changes.
- Creational pattern is a more indirect way of object creation
- Provide different ways (patterns) to remove explicit references in the concrete classes from the code that needs to instantiate them.

Why Creational Patterns?

- A system should be independent of how its objects and products are created.
- Hiding the implementations of a class library or product, revealing only their interfaces.
- Creating different representation of independent complex objects shouldn't have change impact
- A class wants its subclass to implement the object it creates.
- The class instantiations are specified at run-time.
- There must be a single instance and client can access this instance at all times.
- Instance should be extensible without modifying the code.

Three Creational Patterns

- Factory method pattern
  - A method that creates a type of object
  - Centralize the creation of objects
- Factory Pattern
  - Centralize the decision of what factory (the class which actually creates the business object) to instantiate
- Singleton
  - Restricts instantiation of only object for a business logic class

The FACTORY METHOD Pattern

- Context
  - A type (the creator) creates objects of another type (the product).
  - Subclasses of the creator type need to create different kinds of product objects.
  - Clients do not need to know the exact type of product objects.

- Solution

–Define an (abstract) creator type that contains main logic of handling an abstract product.

–Define an abstract method, called the factory method, in the creator type.
The factory method yields a product object.

–Each concrete creator class implements the factory method so that it returns an object of a concrete product class.

–Define an (abstract) product type that models a common product

–Create concrete products

Factory Method Pattern

`DocumentMgmtFramework`

- knows when to create a Document (when user selects the menu item
- but DOES NOT anticipate what type of Document to create

Factory Method Pattern Example

- Framework classes are all abstract and it does not change when new document types are added
  - DocumentMgmtFramework can also provide a default creation of a Document
- Client provides new types of documents and document creation factory.
  - Later on you can add another type of document w/o changing DocumentMgmtFramework

Benefits

- Introduces a separation between the application and a family of classes
- Weak coupling instead of tight coupling hiding concrete classes from the application framework
- Simple way of extending the family of products with minor changes in application code
- It provides customization hooks

Abstract Factory Pattern

- Context
  - Instead of one product, there could be multiple families of products which needs to be instantiated
  - System needs to be independent from the way the products it works with are created
- Solution
  - Create a factory to instantiate objects specific to a particular family of products
  - One factory per product family

Abstract Factory Pattern

Each style (Windows, vs OSX) defines different looks and behaviors for each type of controls: like Buttons, Edit Boxes and Fields.
In order to avoid the hardociding it for each type of control we define an abstract class GUIFactory.
Depending on a configuration parameter in MyApp one of the concrete factories: WindowsGUIFactory or OSXGUIFacory will be created.
Button creation or Editbox creation requests will be delegated to the concrete factory which will return the specific flavor

Creational Patterns and Reflection

- Many of the object-oriented design patterns can benefit from reflection
- Reflection extends the decoupling of objects that design patterns offer
- Can significantly simplify design patterns

- Factory
- Factory Method
  - One can either create different factory method class for each type of object OR
  - Create a nested if-then-else to create all possible types of object.

Factory Method Without Reflection

```
public static Document createDocument(String s) {
    Document temp = null;
    if (s.equals ("Html"))
        temp = new HtmlDocument();
    else if (s.equals ("PDF"))
            temp = new PDFDocument();
    else          // …
            // continues for each kind of shape
    return temp;
}
```

Factory With Reflection

```
public static Shape getFactoryShape (String s) {
    Shape temp = null;
    try
    {
      temp=(Shape) Class.forName(s).
                   newInstance();
    }
    catch (Exception e) {
    }
    return temp;
}
```

Singleton Pattern

- Sometimes it's important to have only one instance for a class
  - For example, in a system there should be only one window manager (or only a file system or only a server object).
  - Usually singletons are used for centralized management of internal or external resources and
  - Provide a global point of access

Singleton Example

```
SingletonServer.getInstance()
      .doMainTask();
```

```
public static SingletonServer getInstance() {
    if (instance==null)
       instance= new SingletonServer();
    return instance;
}
```

- A static member in the Singleton class
- A private constructor
  - The constructor should not be accessible from the outside of the class to ensure the only way of instantiating the class would be only through getInstance()
- A static public method that returns a reference to the static member
  - getInstance() method is used also to provide a global point of access to the object

Applicability Example

- Logger Class
  - A logger is usualy implemented as a singletons, and provides a global logging access point in all the application components without being necessary to create an object each time a logging operations is performed.
- Configuration Classes
  - Provides the configuration settings for an application
  - Not only a global access point, but one can keep the instance as a read-only cache object
- Server object

THANK YOU

SS ZG653 (RL 15): Software Architecture

Design Patterns- Structure

BITS Pilani

Pilani|Dubai|Goa|Hyderabad

Instructor: Prof. Santonu Sarkar

March 18, 2015

Design Patterns

STRUCTURAL PATTERNS

What is it?

- A set of design patterns
  - Provides easy and generalized techniques to realize relationships between entities
- We shall study
  - Decorator pattern
  - Adapter pattern
  - Composite pattern
  - Proxy pattern

15.1

DECORATOR PATTERN

Decorator Pattern

1. A component object need to be enhanced (a window to have a scrollbar) or additional responsibilities to be added
2. The decorated object can be used in the same way as the undecorated object
3. The original component class does not want to take on the responsibility of the decoration
4. There may be many possible decorations

Decorator Pattern Class Diagram

- Define an interface that is an abstraction for the component
- Concrete component classes realize this interface type
- Decorator classes also realize this interface type
  1. Decorator object manages the component object that it decorates
  2. When implementing a method from the component interface type, the decorator class applies the method to the decorated component
  3. Combines the result with the effect of the decoration

Using a Decorator

```
public class MyGUIApp {
   public static void main(String[] args) {
// create a new window
        Window w = new ConcreteWindow();
        w.renderWindow();
// sometime later, you find that too much text!!
// So you need scrolling functionality through decoration
        w = new ScrollableWindow(w);
// now window object has additional behavior
        w.renderWindow();
   }
}
```

GUI Main function use the decorator pattern by

- Creating a Decorated object (ScrollableWindow)
- Invoking the main method `w.renderWindow()` defined in the interface

Usage in JDK

- FileIO
  - Reader is the interface
  - A BufferedReader is a Decorator

```
Reader r = new BufferedReader(new FileReader
("readme.txt"));
r.read();
```

Consequences

- Decoration is more convenient for adding functionalities to objects instead of entire classes at runtime
- With decoration it is also possible to remove the added functionalities dynamically
- Decoration adds functionality to objects at runtime which would make debugging system functionality harder

15.2

ADAPTER PATTERN

Introducing Adapter Pattern

- Like any adapter in the real world, an Adapter it is a bridge between two objects
- One class expecting some type of object and you have an object offering the same features, but exposing a different interface
- Certainly both of them should be used instead of re-implementing
  - Changing existing classes is not an option
- Therefore, why not create an adapter...

Adapter pattern class diagram

1. Define an adapter class that implements the Target interface
   1. Adapter lets classes work together, that could not otherwise because of incompatible interfaces

2. The adapter class holds a reference to the Adaptee. It translates target methods to Adaptee methods.

3. Adaptee is wrapped into an adapter class object

4. Composition plays an important role

Usage of Adapter Pattern

```
public class PlayMyAudio {
  public static void main(String[] args) {
      MediaPlayerInterface player = new MediaAdapter();
      player.play("beyond the horizon.avi");
      player.play("alone.mp4");
  }
}
```

- 3rd parties libraries and frameworks - most of the applications using third party libraries use adapters as a middle layer between the application and the 3rd party library to decouple the application from the library
- Suppose that you change this 3rd party library. Then only an adapter for the new library is required without having to change the application code.

How much should the Adapter work?

- it should do only that much which is necessary in order to adapt
- If the Target and Adaptee are similar then the adapter has just to delegate the requests from the Target to the Adaptee
- If Target and Adaptee are not similar, then
  - the adapter might have to convert the data structures between those
  - implement the operations required by the Target but not implemented by the Adaptee

15.3

COMPOSITE PATTERN

Primitive vs Composite Objects

Primitive Objects are atomic objects which can not be put into any other object

Composite Objects are collection objects which can contain other objects (primitive or composite)

Moreover, Composite lets clients treat individual objects and compositions of objects uniformly

A typical GUI Application

Primitive Objects → **Button, Radio Button, Check Box, Text Fields,**

Composite Objects → **JFrame, JPanel**

Composite Pattern Requirement

- There are some operations that needs to be applied on both primitive as well as composite objects
- Interface for operation (whether for primitive type or composite type) is same
- Any operation for primitive type object is atomic in nature but the same operation for a composite type object is recursive in nature

Composite Pattern Class Diagram

- **STEP 1: Create an interface for Common Operations**
- **STEP 2: Both Primitive and Composite Objects Realize the same interface**
- **STEP 3: Make Composite Objects as a collection of primitive objects**

Using Composite Pattern

```
public class MyGUIApp {
  public static void main(String[] args) {
// create a rectangle shape
   GUIShape addButton= new Button(); GUIShape remButton= new Button();
   GUIShape startButton= new Button();
   GUIShape cancelButton= new Button();
   GUIShape ipL= new Label(); GUIShape ipT= new TextBox();

   CompositeShape complexComp= new CompositeShape();
   complexComp.addShape(addButton); complexComp.addShape(remButton);
   complexComp.addShape(ipL); complexComp.addShape(ipT);

   CompositeShape veryComplexShape= new CompositeShape();
   veryComplexShape.addShape(complexComp);
   veryComplexShape.addShape(startButton);
   veryComplexShape.addShape(cancelButton);

   veryComplexShape.render();
  }
}
```

Consequences

- The composite pattern defines class hierarchies consisting of primitive objects and composite objects. Primitive objects can be composed into more complex objects, which in turn can be composed
  - `complexComp`
  - `veryComplexShape.addShape(complexComp);`
- Clients treat primitive and composite objects uniformly through a component interface (GUIShape) which makes client code simple

- Adding new components can be easy and client code does not need to be changed since client deals with the new components through the component interface (GUIShape)

15.4

PROXY PATTERN

Motivation and intent

- We need the ability to control the access to an object.
  - For example if we need to use only a few methods of some costly objects we'll initialize those objects when we need them entirely.
  - Till that situation arises, we can use some light objects exposing the same interface as the heavy objects.
  - These light objects are proxy objects
  - They will instantiate those heavy objects when they are really need

- This ability to control the access to an object
  - When a costly object needs to be instantiated and initialized
  - Different access rights to an object, as well as
  - Providing a sophisticated means of accessing and referencing objects running in other processes, on other machines.

Class Diagram

A client obtains a reference to a Proxy, the client then handles the proxy in the same way it handles RealSubject and thus invoking the method doOpeationA().

- Maintains a reference that allows the Proxy to access the RealSubject.
- Implements the same interface implemented by the Real Subject so that the Proxy can be substituted for the RealSubject.
- Controls access to the RealSubject and may be responsible for its creation and deletion

This interface is implemented by both Real subject and proxy

the Real subject that the proxy represents

Client calling Proxy

```
public class SimpleClient{

    public static void main(String[] args){
        ServerInterface server = new ServerProxy();
         server.setup();
         server.doOperationA();
         server.doOperationB();
        try {
            executor.runCommand("ls -ltr");
            executor.runCommand(" rm -rf abc.pdf");
        } catch (Exception e) {
            System.out.println("Exception Message::"+e.getMessage());
        }

    }

}
```

Examples

- Remote Proxy--Java RMI
  - An object on one machine (executing in one JVM) called a client can invoke methods on a remote object in another machine (another JVM)
  - The proxy (also called a stub) resides on the client machine and the client invokes the proxy in as if it is invoking the object itself
  - The proxy itself will handle communication to the remote object, invoke the method on that remote object, and would return the result if any to the client
- Proxy contains machine address, process id, object id

Examples

- Virtual Proxy: Lazy loading when the data size is huge
  - In place of a complex or heavy object, use a skeleton representation.
  - When an underlying image is huge in size, just represent it using a virtual proxy object and on demand load the real object

- Protection Proxy: Restricts access
  - Checks access rights before invoking the actual operation
  - Accessing public e-mail, social networking, data storage etc. in a corporate setup

Examples

- Firewall proxy
  - Proxy is a process on a firewall machine
  - Clients requests to outside world from within the firewall, is intercepted by this process
  - Depending on the security policy, it allows or stops the request
  - Incoming requests are also intercepted by proxy
    - Checks for compliance before passing to the server
  - Both inside and outside entities are oblivious to proxy until their requests are denied!!

THANK YOU

SS ZG653 (RL 16): Software Architecture

Design Patterns- Behavior

BITS Pilani

Pilani|Dubai|Goa|Hyderabad

Instructor: Prof. SantonuSarkar

March 24, 2015

Design Patterns

BEHAVIORAL PATTERNS

What is it?

- A set of design patterns
  - that identify common communication between objects. These patterns increase flexibility in carrying out communication
- We shall study
  - Iterator pattern- Access elements of an aggregate sequentially
  - Observer pattern- Publish/Subscribe or Event Listener
  - Strategy pattern- Select Algorithms on the fly
  - Visitor pattern- Separate an algorithm from an object
  - Command pattern- Encapsulate an action, parameters, state

16.1

ITERATOR PATTERN

Iterator Pattern

- Intent
  - Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation
    - An aggregate object is an object that contains other objects for the purpose of grouping as a unit
    - Also called a container or a collection
    - Examples are a linked list and a hash table.
  - Also Known As Cursor

- Motivation
  - A list should allow a way to traverse its elements without exposing its internal structure
  - It should allow different traversal methods
  - It should allow multiple traversals to be in progress concurrently

What are Iterators

- Iterators helps to iterate or traverse the collections[Moving from the first element collection to its last element]
- Example of Collections : Arrays, Linked Lists etc..
- Given a list of numbers say 4, -5, 10, 6, 8, 20, -10, 30

Array Representation

0	1	2	3	4	5	6	7
4	-5	10	6	8	20	-10	30

Linked List Representation

Iterators cont….

Traversal in Java

```
LinkedList list = new
LinkedList()
ListIteratorLtr =
list.listIterator();
while(Ltr.hasNext() {
  Object current= Ltr.next();
  ....................
}
```

- Advantages of Iterators
  - Iterators does not expose internal structure [only return elements for use]
  - More than one iterators can be attached to a single collection.

Traversal in C

```
Link Ltr = list.head;
while(Ltr !=  null) {
 Object current= Ltr.data;
Ltr= Ltr.next;
}
```

- Disadvantages w/o iterator
  - Internal structure links exposed to user
  - Only one way traversal at a time

Solution

- Define an iterator that fetches one element at a time
- Each iterator object keeps track of the position of the next element
- If there are several aggregate/iterator variations, it is best if the aggregate and iterator classes realize common interface types.

16.2

OBSERVER PATTERN

Observer Pattern

- Context
  - Certain objects need to be informed about changes in occurred in other objects whenever certain event occurs
- Example (Event mgmt)
  - A real time stock update system that publishes stock prices whenever a change in stock price happen
  - Clients can be web-based application or a smart phone application
  - These clients need to be alerted whenever such change occurs
- Example (MVC Architecture Pattern)
  - Button notifies action listeners when Button is Pressed
  - Views attach themselves to model in order to be notified
  - Action listeners attach themselves to button in order to be notified

Observer Pattern Class Diagram

Subject which can be abstract class or interface

Concrete observers

```
p.getLatestNews();
```

```
for(InformationSubscriber s : subscribers) {
o.update(this);
}
```

Interaction

- Define an observer interface type
  - Concrete observers can model different communication channels or entities interested in certain class of events
  - Also called Subscriber
  - Use **Factory pattern** to create new observer
- The subject maintains a collection of observers
  - A specialized subject can model a specific type of event
  - The subject supplies methods for attaching and detaching observers.
  - Also called Publisher
- Whenever an event occurs, the subject notifies all observers

Implementation Issues

- Many to Many
  - Many subscribers may need to observe many events (subjects)
  - During notification, the subject reference is passed as a parameter (recall the class diagram)

- Triggering update
  - If updates are very frequent there can be many consecutive updates
  - Observer can be made responsible to initiate notify operation when it needs (better efficiency)

- Just before notify() is called…
  - Subject state should be updated properly so that the observer gets the updated state

Push vs Pull

- Push
  - subjects send detailed information about the change to the observer whether it uses it or not
    - Could be inefficient when a large amount of data needs to be sent and it is not used
  - Send only the information required by the observer.
    - Subject should be able to distinguish between different types of observers and to know the required data of each of them, ➔ subject is more coupled to observer
- Pull
  - The subject just notifies the observers about the change
  - Observer pulls the required data
    - Communication is done in 2 steps– extra overhead

16.3

STRATEGY PATTERN

Strategy Pattern

Context:

- A class can benefit from different variants for an algorithm
- Clients sometimes want to replace standard algorithms with custom versions

Strategy Pattern Overview

Interaction

- Define an interface type that is an abstraction for the algorithm
- Actual strategy classes realize this interface type.
- Clients can supply strategy objects
- Whenever the algorithm needs to be executed, the context class calls the appropriate method (doWork()) of the strategy object

Strategy pattern class diagram

The Overall "Context"

```
public void start() {
  System.out.println(this.name + ": Based on current position" +
      "the behaviour object decide the next move:");
  int command = strategy.moveCommand();
  // ... send the command to mechanisms
  // DO other stuff
}
```

Different concrete strategies (algorithms) to move the robot

Usage of Strategy Pattern

```
public class RunRobot{

public static void main(String[] args){
  Robot r1 = new Robot("Big Robot");
  Robot r2 = new Robot(“D2");

  r1.setBehavior(new AgressiveMove());
  r2.setBehaviour(new DefensiveMove());

  r1.start();
  r2.start();
  if (someCondition) {
   r1.setBehavior(new NormalMove());
  r1.start();
 }
}
```

Examples

- Saving Files in Different Formats
  - file.setFormat(<Name of Format>);
  - file.save();

- Compress Files using different algorithms
  - Various Encryption/Decryption Strategies

- Different types of sort- merge sort, bubble sort
- Different types of interest calculation logic

Strategy Pattern – Subtle points

- Different implementation to accomplish the same thing, so that one implementation can replace the other while the caller does not change

- Strategy doesn't exist as a standalone object- it works meaningfully, as a support object in a given context
  - Robo movement, interest calculation while computing total asset

- You typically choose one strategy at a time depending on the internal state of the Context

Implementation Issues

- Usually each strategy need data from the context
  - Create a data class to hold the data and pass it to strategy
    - Special care to design data class- what fields should be included?
    - In the future some new strategy may require data from context which are not included in this data class!
  - Pass the context object to the strategy
    - The strategy object can use the data directly in the context
    - Tight coupling
- The client needs to know "concrete" strategy
  - The context object can use Factory pattern to create the strategy object
  - Client has only to send a parameter (like a string) to the Context for a specific algorithm

16.4

VISITOR PATTERN

When do we need?

- Applicability
  - Similar operations have to be performed on objects of different types grouped in a structure (a collection or a more complex structure).
  - Several distinct and unrelated operations needed to be performed on the structure
  - New operations may have to be added without affecting the structure

- Example
  - Different traversal algorithms to be performed on a graph structure
  - Compiler code generation
  - Generating different types of reports on a structure
  - Dynamic pricing of various items we purchase in an online shopping

Visitor Pattern Class Diagram

declare the visit operations for all the types of concrete visitableclasses

Implements a specific activity (price computation) for all visitables

```
public int visit(Book book) {
    int cost=0;
    if(book.getPrice() > 50){
        cost = book.getPrice()-5; // Apply flat discount
    }else cost = book.getPrice();
    return cost;
}

public int visit(Fruit fruit) {
    int cost = fruit.getPricePerKg()*fruit.getWeight();
    return cost;
}

public int visit(Pet p) {
    int cost = p.getUnitPrice()*p.getUnits();
    cost += addTax(cost);
    return cost;
}
```

This is ObjectStructure

- Another class that is a collection of visitable objects
- Offers mechanisms to iterate over visitable

Visitor Pattern usage

```
public static void main(String[] args) {
   ShoppingCartmyCart = new ShoppingCar();
myCart.add(new Book(20, "1234"));
myCart.add(new Book(100, "5678"));
myCart.add(new Fruit(10, 2, "Banana"));
myCart.add(new Fruit(5, 5, "Apple");
myCart.add(new Pet(100, 2, “Fish”)};
   Visitor pricecalculator = new ShoppingCartVisitor();
   intsum=0;
   for(Visitableitem : shoppingCart.getList() ){
       item.accept( pricecalculator);
      sum += item.getStateAfterVisit();
   }
   System.out.println("Total Cost = "+ sum);
}
```

Consequences

- Flexible design for adding new visitors to extend existing functionality without changing existing code

- If a new Visitable class – say “ElectronicItem” is added all the implemented visitors need to be modified
  - What is the separation of visitors and visitable?
    - Visitors dependent on visitableBUT visitablesare not dependent on visitors.

- Part of the dependency problems can be solved by using reflection with a performance cost

Visitor with Reflection Pattern

- Suppose we need to add a new visitable class "ElectronicItem" in our structure
- Make Visitor – abstract class
- Create one visit(Visitable v) method instead of many visit() methods
  - Use reflection to get the exact concrete class of v (ElectronicItem)
  - Use reflection to check if there is any method in the concrete visitor class that implements visit(ElectronicItem). If so call this method
  - Otherwise execute a default visit method

Visitor and Composite Pattern

- Can be used in addition with the composite pattern
- The object structure can be a composite structure

- In the implementation of the accept() method of the composite object
  - the accept methods of the component object has to be invoked

Difference between Iterator and Visitor

- Both are used to traverse object structures
- The main difference is
  - Iterator works on collections, usually containing objects of the same type. The visitor pattern can be used on complex structure such as hierarchical structures or composite structures.
    - The accept method of a complex object should call the accept method of all the child objects.
  - In one case the visitor defines the operations that should be performed, while the iterator is used by the client to iterate through the objects
    - The operations are defined by the client itself

16.5

COMMAND PATTERN

Motivation and intent

- Encapsulate a request in an object
- Allows the parameterization of clients with different requests
- Allows saving the requests in a queue and call them one by one

Applicability

- Need to parameterize objects depending on the action they must perform
- Need to queue commands and execute them later
- Need to provide undoable actions (the Execute method can memorize the state and allow going back to that state)

- Need to build a system that comprises of high level operations, and each high level operation requires many primitive operations

- Need to decouple the invoker from the action performer
  - Also known as Producer - Consumer design pattern

Class Diagram

The invoker who asks the command to carry out request

Abstract Command to execute an operation

Receiver who knows how to execute an operation

Concrete command, implements execute() by invoking appropriate operations on Receiver

Command Pattern usage

```
public static void
main(String[] args) {
StockTraderstock = new
StockTrader();
BuyStockbsc = new
BuyStock(stock);
SellStockssc = new
SellStock(stock);
TradingAgentagent = new
TradingAgent();

// Buy Shares
agent.placeNewOrder(bsc);
// Sell Shares
agent.placeNewOrder(ssc);
}
```

- Client creates orders for buy and sell (concrete command)
- The agent (invoker) maintains a queue of commands to execute
  - It may execute a buy command on Monday if the request comes on Sunday
- Buy command interacts with actual stock exchange (Receiver), gets the result and sends it back

Implementation Issues

- How intelligent should a command be?
  - Just a link between receiver, does nothing else
  - Implements everything, does not need receiver
- Undo and Redo
  - Command can keep a snapshot of the receiver state before calling receiver (sometimes difficult due to memory issues)
  - Stores set of performed operations
    - Receiver and command need to implement inverse operations
- Asynchronous calls
  - Invoker sends requests to which are to be received in a separate threads
  - Threads may be limited, invoker can use thread pool

Command Pattern – Subtle Points

- May look similar to Strategy pattern
  - Unlike strategy, it is not NOT about multiple implementation of the same algorithm
- Command encapsulates the state and the action together
  - It acts like a standalone object (unlike strategy)
  - It can be passed around
  - The actual command performed can be undone/redone as it maintains the state
- In the application context, you typically use many command objects at any time
  - In strategy you tend to use either/or

THANK YOU

SS ZG653 (RL 17): Software Architecture

Architecture of Next Gen Systems-

Introduction to Cloud Computing

Instructor: Prof. Santonu Sarkar

BITS Pilani

Pilani|Dubai|Goa|Hyderabad

March 31, 2015

17.1

CLOUD COMPUTING PARADIGM

Characteristics of Cloud Computing

According to
National Institute of Standard & Technology

On Demand and Self-Service	• Consumer can provision the computing resource without provider's intervention
Ubiquitous Access	• Resource can be accessed through network on heterogeneous platforms • Location independent access
Resource Pooling	• Internally resource needs to be pooled and shared among consumers
Elasticity	• Rapid and elastic scale up/down of resource
Measured Service	• Monitoring and measuring of service
Multi-tenancy	• Single application to support distinct class of users • Each class has its own dataset and access rights

Service Oriented Software-Characteristics

Cloud Computing

- Consumer can provision the computing resource without provider's intervention
- Resource can be accessed through network
- Internally resource needs to be pooled and shared among consumers
- Rapid and elastic scale up/down of resource
- Monitoring and measuring of service

Next Generation of Service

- On demand
- Continuous adaptation
- Self-serviced & personalized
- Always available through any client platform
- Rapid scaling
- Multi-tenancy (better resource consumption)
- Ensure accountability and trust
- Affordable mode of payment

Four Key Infrastructure Components

- Infrastructure Sharing
- Provisioning of right sized infrastructure on-demand

- Customers and providers form a larger ecosystem
- Emergence of social network
- Customers decide the feature

- Prolific adoption of mobile devices
- Increase in mobile compute power
- Next gen services will mobile device for service consumption

- Massive data
  - From infrastructure
  - From social ecosystem
- Next gen service will process and analyze this data for its own evolution

Examples

In 2012, > 200 courses, 33 universities, 1.9M students accessing video/audio lectures and presentation

Sep 2012- effected by DDOS attack

100M users, 100 PB
All of their user files were publicly accessible for 4 hrs due to authentication fault

170 million Gmail users. Two major failures in Dec 2012 when Gmail upgraded its attachment capability to 10GB.

185,000 applications on Force.com
6M Txn/day

1B active users, 500M uses mobile devices

500M active users, handles 1.6B queries per day

1B Txn/day.
They need to detect credit card fraud within 5ms.

8 petabytes/yr, 20hrs of new video/min

Data Deluge- structured, unstructured text, audio, and video

Next Gen Service uses various means to interact with its environment

- Sensors in various form
- Users connected via wired and wireless network
- Create, share and communicate information among the service community- subscriber, provider, manufacturer

Data is massive

- 1800 exabyte today-- of which 70% is created by individuals
- Facebook → more than 30 petabytes of user content
- amazon → petabytes of user-reco & browsing data for recommendation

Unprecedented Impact

- Informed and near optimal decisions on all relevant aspects of the service
- Service offering, product design can be improved through realtime feedback
- Distribution, production and maintenance can be highly fine tuned
- Empowers consumers like never before

Scenarios

- US Healthcare can save upto 300B through informed decision making
- Europe govt admin service can save € 100B through operational efficiency in tax collection
- Personalized suggestions, optimized real-time pricing – Amazon

(http://www.emc.com/collateral/analyst-reports/diverse-exploding-digital-universe.pdf)
(http://hmi.ucsd.edu/pdf/HMI_2009_ConsumerReport_Dec9_2009.pdf),
(http://www2.sims.berkeley.edu/research/projects/how-much-info-2003/printable_report.pdf)
http://www.mckinsey.com/mgi/publications/big_data/index.asp
http://www.reuters.com/article/2011/07/19/idUS319973276120110719

Traditional versus Cloud Computing (Layered Rubric)

Cloud Computing - Examples

- Mobile- Andriod, iPhone, Windows Mobile
- OS- gOS
- Thin client- Zonbu, gOS based applications

Software As a Service

- Identity : OAuth, OpenID
- Integration : Amazon Simple Queue
- Payments :Amazon Flexi Pay, PayPal
- Mapping : Google Maps
- Search :Google Custom Search, Yahoo! BOSS
- Human: Mechanical Turk

- Peer-peer : Skype, Bittorent
- Web app : Facebook
- SaaS: Google Apps, SAP, Salesforce
- Microsoft Online service

Platform As a Service

- Database :Amazon SimpleDB, Google BigTable
- Network : MobileMe iDisk, CloudNAS
- Synch : MobileMe push

- App Framework: Google App Engine, Ruby on Rails
- .NET : Azure Services Platform
- Proprietary : Force.com

Infrastructure As a Service

- Full virtualization (GoGrid, Skytap)
- Grid computing (Sun Grid, HP)
- Management (RightScale, Elastra, HP)
- Compute (Amazon Elastic Compute Cloud)

System Virtualization

- Virtual Machine Monitor (VMM) - Hypervisor
  - Underlying software platform that provides virtualization support
  - Manages several VMs
  - Two main services
    - Page mapping
    - Scheduling
- Virtual machine
  - A software that simulates a machine environment
- Guest Operating System
  - An operating system and its applications running inside a VM

Hypervisor

- According to [1], a VMM should have
  - Equivalence
  - Efficiency
  - Resource Control
- Furthermore a VMM provides
  - mediation between VM and hardware
    - Strong isolation (performance, fault and security)
  - encapsulation of virtual machine
    - Failure and recovery (Reliability)
    - Check pointing (Reliability)
    - Live migration (Availability)

1. G. J. Popek and R. P. Goldberg, "Formal Requirements for Virtualizable Third generation Architectures," *Communications of ACM,* vol. 17, no. 7, 1974

Ideal VMM Scenario

- Hardware needed by monitor
  - Ex: VMM must control real hardware interrupts
- Access to hardware would allow VM to compromise isolation boundaries
  - Ex: access to MMU would allow VM to write any page
- So…
  - During virtualization- Ring 0 is occupied by VMM.
  - OS goes to Ring 3, it can't call privileged instructions- trap is generated!!
  - All sensitive instructions must always pass control to the VMM
    - All access to the virtual System ISA by the guest must be emulated by the VMM in software.
    - System state kept in memory.
    - System instructions are implemented as functions in the VMM

Virtualization Problem with x86

- A processor or mode of a processor is strictly virtualizable if, when executed in a lesser privileged mode:
  - all privileged instruction execution attempt causes a trap
  - Rest of the instructions execute identically
- So that.. guest OS runs in a deprivileged mode and all privileged instructions creates a trap into VMM
  - VMM emulates instructions against virtual state
- But x86 architecture is not virtualizable
  - ISA includes privileged instructions like POPF that read or modify privileged state but don’t trap in user mode
  - Hence VMM won’t know that a VM calls POPF

Full Virtualization

- VMM fully abstracts CPU, memory, and IO
- OS is de-privileged to Ring-1 which OS is not aware of. OS runs un-modified with regular device drivers
- Binary instructions coming from OS using binary translation

- Not every instruction is translated
- Only the ones that are privileged

Solution 2 - Paravirtualization

- Requires a modified guest OS to run with a hypervisor
- Modified device drivers also are aware of hypervisor
- Instead of privileged system calls to CPU, guest OS makes hyper calls directly to hypervisor – no need for BT

Challenge

- Guest OS must be modified – difficult with proprietary OSes like Windows

More on Paravirtualization

- Paravirtualization Exports Simpler Architecture
- Modified Guest OS is aware of virtualization layer
  - Remove non-virtualizable parts of architecture
  - Avoid rediscovery of knowledge in hypervisor
- Excellent performance and simple, but poor compatibility

Virtualization Basics- Memory Mgmt

- Memory divided into pages- some are in physical memory and the rest are in disk (swap)
  - Page table maps consumer application address to machine address or disk location
- Virtualized Environment
  - VM page table maps target address to an address within VM.
  - This is converted to physical address by Hypervisor page table

Memory Management

- Guest OS sees flat "physical" address space.
- Page tables within guest OS:
  - Translate from virtual to physical addresses.
- Next level mapping is done by VMM
  - Physical addresses to machine addresses.

Virtualized Address Spaces

- Now, we need two-level translations:
  - Guest VA → Guest PA → Host MA
  - SLOW!! Each memory access needs to be translated in software

Network- in a Nutshell

- Recall every device, and a VM on a network can have its own IP address (32bit number for IPv4)
  - IP message has a header (source IP addr, destination IP addr) and the content
  - IPv6 has 128 bit header but it has source and destination IP
- Gateway
  - An organization uses a gateway to manage incoming and outgoing traffic
  - Public IP: www.google.com (unique within internet)
  - Private IP: restricted within the organization (10.2.1.1)

Network Address Translation (NAT)

- Outgoing message
  - Gateway records the private IP of the source, replace the header with its own public IP
  - Sends it to external network
- Incoming message
  - Gateway intercepts, overwrites the destination IP (which is the gateway) with internal private IP
  - Sends it to internal network

Infrastructure as a Service

- Variety of clusters
  - A cluster manager manages node, and persistent object manager manages files
  - Cluster manager controls
    - execution of VMs on a node
    - Virtual networking between VMs and VM– external user
  - Node manager uses the hypervisor to manage VM execution, inspection, and termination

Networking in IaaS

- Virtual Resource Manager acts as a Gateway
- If a VM has a public IP address, IaaS automatically reallocates this to another VM if this VM fails
- Auto Scaling
  - On demand, based on the workload, new VMs can be spawned

Data (Massive) and Storage

- Forces against Traditional RDBMS
  - Billions of pages need to be collected and indexed. The data access is through simple indexes, sophisticated RDBMS indexing and optimization is not required
  - Data schema is not fixed
  - To get data from multiple table, join is required which is expensive when data is large
  - It is okay to have temporary inconsistency of data, but availability is a MUST

HDFS

Component and Connector View

- One NameNode managing multiple DataNodes
- Data stored in DataNodes in blocks of 64MB
- Each block is replicated for Reliability

HBase

- Hbase- key-value based database like Google's BigTable
  - Trillions of data can be stored
  - Supports table but no schema, one column is key
  - A data-value is indexed by
    - Row value, column name, timestamp
    - Supported time based versioning
- Web page crawling
  - Row value can be URL
  - Columns can be attributes of web-page
  - URL/timestamp can be the index

MongoDB

- Stores data as object- containing all information about some concept
  - Without bothering whether a data field of one object is related to another object
  - Two objects can have all data fields common, some common or no commonality
  - Links in data items can serve as connection between object implicity…
- Stored in binary JSON
- A field may be indexed or not indexed. No concept of primary or secondary keys

DATA CENTER AND CLOUD OS

Data Centers with Virtualization

- Data centers must be virtualized to serve as cloud providers
- There are open-source (Eucalyptus, OpenStack) and commercial (VMWare, Rackspace) virtual infrastructure (VI) managers and OSes.
  - Specially tailored for virtualizing data centers which often own a large number of servers in clusters
- Create and manage VMs, aggregate them into virtual clusters
- Dashboards for management and control

Amazon EC2

- Amazon's Elastic Compute Cloud (EC2) is a good example of a web service that provides elastic computing power in a cloud. EC2 permits customers to create VMs and to manage user accounts over the time of their use.

Various Cloud Platforms

OS, License	Resources being virtualized	Client API	Hypervisor	Public Cloud Interface	Features
Nimbus Linux, Apache v2	VM creation, virtual cluster, www .nimbusproject.org/	EC2 WS, WSRF, CLI	Xen, KVM	EC2	Network virtualization
Eucalyptus Linux, BSD	Virtual networking www .eucalyptus.com/	EC2 WS, CLI	Xen, KVM	EC2	Network virtualization
OpenNebula Linux, Apache v2	Management of VM, host, virtual network, and scheduling tools, www.opennebula.org/	XML-RPC, CLI, Java	Xen, KVM	EC2	Network virtualization
vSphere 4 Linux, Windows, proprietary	Virtualizing OS for data centers www .vmware.com/products/vsphere/	CLI, GUI, Portal, WS	VMWare ESXi	VMWare vCloud	Network virtualization, storage, HA, DRM, Live migration
Openstack, Linux, Apache v2	VM creation, storage virtualization, key management	CLI, WS	KVM-QEMU, Xen, VMWare	EC2	Network, storage, Live migration

Eucalyptus

- Interact with EC2.
- Provide a storage API to emulate the Amazon S3 API for storing user data and VM images.
- Installed on Linux-based platforms

- Instance Manager controls the execution, inspection, and terminating of VM instances
- Group Manager gathers information about and schedules VM execution on specific instance managers, as well as manages virtual instance network.
- Cloud Manager: entry-point into the cloud for users and administrators. It queries node managers for information about resources, makes scheduling decisions, and implements them by making requests to group managers.

Openstack

- Compute(*Nova*): manages resource allocation and lifecycle of VMs.
- Network (*Neutron*): Provides network connectivity between interface devices. It provides API-driven network and helps users manage IP addresses for their VMs.
- Storage: *Swift*- allows data to be stored and retrieved as objects. *Cinder* provides persistent block storage of data to VMs. *Glance* is the repository of all virtual disk images and uses Swift as a backend.
- Identity Management: *Keystone* provides the authentication, authorization, and access control for all interaction between services
- Dashboard (*Horizon*): Web user interface for OpenStack services.

Openstack capabilities

- Dashboard for monitoring and control
  - Assign physical resources like pCPU, RAM, storage to an image

- Different image types and snapshots
  – Linux, Windows and AWS EC2 images
  - Different container types, e.g. Single container (OVF), components (AMI, ARI, AKI)
- VM Lifecycle Management
  - Terminate instance.
    Reboot instance.
    Log actions in instance.
    Console interface for command line etc. Edit image description.
  - Create a snapshot of instance current state

CLOUD ARCHITECTURE

Private and Public Cloud

Characteristics	Public clouds	Private clouds
Technology leverage and ownership	Owned by service providers	Leverage existing IT infrastructure and personnel; owned by individual organization
Management of provisioned resources	Creating and managing VM instances within proprietary infrastructure; promote standardization, preserves capital investment, application flexibility	Client managed; achieve customization and offer higher efficiency
Workload distribution methods and loading policies	Handle workload without communication dependency; distribute data and VM resources; surge workload is off-loaded	Handle workload dynamically, but can better balance workloads; distribute data and VM resources
Security and data privacy enforcement	Publicly accessible through remote interface	Access is limited; provide pre-production testing and enforce data privacy and security policies
Example platforms	Google App Engine, Amazon AWS, Microsoft Azure	IBM RC2

Infrastructure as a Service- Most Basic Cloud Service

1. Amazon EC2, Rackspace etc. offer computers, as physical or more often as VMs, and other resources.
2. VMs run on a hypervisor, such as Xen or KVM.
3. You buy a VM of a certain capacity, often with a pre-installed OS and then install your own application software
   - Login via ssh or through browser
4. Amazon/Rackspace bills you as per the number of hours the VM is running
   - to reflect the amount of resources allocated and consumed.

Amazon CloudFormation (and underlying services such as Amazon EC2), Rackspace Cloud, Terremark, and Google Compute Engine

Platform as a Service & Software as a Service

- Microsoft Azure, Force.com etc. deliver a computing platform typically including
  - OS, programming language execution environment, database, and web server, email service etc.
- You develop and run your software on this platform
  - No cost and complexity of buying and managing the underlying hardware and software layers.
- Amazon Elastic Beanstalk, Cloud Foundry, Heroku, Force.com, EngineYard, Mendix, Google App Engine, Microsoft Azure and OrangeScape

- Microsoft or Salesforce.com install and operate application software in the cloud
  - you access the software (Word, Excel, Salesforce Software) typically through browser based environment
- Pricing model: Typically a monthly or yearly flat fee per user
- Google Apps, innkeypos, Quickbooks Online, Limelight Video Platform, Salesforce.com, and Microsoft Office 365

IAAS vs PAAS

Amazon Cloud
Google Cloud
Development SDK
Internet
Provisioning Services
VM
VM
VM
VM
VM
VM
VM
VM
VM
....
Application Deployment and Distributed Execution Platform
....
Simple DB
Simple Queue
S3 – Simple Storage
Google Datastore
Map Reduce
Big Table

Amazon Web Service (AWS) – Leader in IaaS

- EC2 (Elastic compute cloud) allows
  - Rent VMs and add more on-demand
  - Can create, launch, and terminate server instances as needed, paying by the hour for active servers.
- S3 (simple storage service) provides the object-oriented storage service for users.
- EBS (Elastic block service) provides the block storage interface which can be used to support traditional applications.
- Amazon DevPay is a simple to use online billing and account management service that makes it easy for businesses
- MPI clusters uses hardware-assisted virtualization instead of para-virtualization and users are free to create a new AMIs
- AWS import/export allows one to ship large volumes of data to and from EC2 by shipping physical discs.
- Small-business companies can put their business on the Amazon cloud platform. Using AWS they can service a large number of internet users and make profits through those paid services

Amazon EC2

- VM Access
  - Web service interfaces to launch instances with a variety of operating systems (bundled into AMIs)
  - Load a VM with custom application environment
  - Manage network's access permissions
- Scaling really fast
  - Obtain and boot new server instances
  - Increase or decrease capacity
- On-Demand Instances
  - Pay-by-the hour
  - Start and stop as you wish
- Reserved Instances
  - Pay-by-the-year
  - Reserved for you, can start-or-stop as needed
- Spot Instances
  - Bid for unused EC2 capacity with a spot price SP
  - If market price < SP you get your instance
  - If market price > SP, instance automatically terminates

Amazon VM Lifecycle

- Standard EC2 Instance (Once terminated, data is lost)
- EBS-backed EC2 Instance (Data persists as long as EBS volume is Intact)

- Amazon Elastic Block Store (EBS) offers persistent storage for Amazon EC2 instances
- Persists the image file independently from the life of an instance
- You can create point-in-time consistent snapshots of your volumes
  - Can be used as the starting point for new Amazon EBS volumes
  - Can protect data for long term durability
  - Can be easily shared with co-workers and other AWS developers
- Snapshots are stored in Amazon S3, and automatically replicated across multiple available zones

THANK YOU

SS ZG653 (RL 18): Software Architecture

Architecture of Next Gen Systems- Software Architecture for Cloud Computing

Instructor: Prof. Santonu Sarkar

BITS Pilani

Pilani|Dubai|Goa|Hyderabad

March 31, 2015

18.1

QUALITY ATTRIBUTES

Quality of Service

- Architecting a system meant for cloud platform is no different from other distributed platform
  - Issues of modifiability, interoperability, and testability are the same
- What's significantly different
  - Availability
  - Performance
  - Security

Availability

- Services expected to always available- but there are many failure points
  - VM, PM, Network
- Architect should plan for a failure
  - EC2 provides 99.95% availability, but how to plan for 0.05% failure?

- Netflix – real time video streaming service
  - hosts on EC2 with additional application level availability
  - EC2 has 4 days of sporadic outage in 2011, but Netflix was unaffected

Failure Estimation

- A subsystem is dependent of all the modules to work
  - Probability that the subsystem will work without failure: $\prod_{i=1}^{n} p(i)$ where $p(i)$ is the probability of the ith module working without failure
- For a redundant system where it is sufficient to have one module working
  - the probability of the subsystem being operational: $1 - \prod_{i=1}^{n}(1 - p(i))$
- When you need at least k modules to work
  - and each has the identical probability of working without failure: $\sum_{j=k}^{n} \binom{n}{j} p^{j}(1-p)^{n-j}$

Netflix case study- Availability

- Spare Copy tactic
  - Use of stateless service- any service request can be executed by any available server
  - Thus everyone can be a spare copy!

- Active redundancy tactic
  - Amazon's availability zone: Multiple redundant hot copies of data spread across Amazon provided availability zone

- Graceful degradation tactic
  - Fail fast- use timeout so as to isolate from a failing component
  - Fallback- critical services degrade to a lower quality representation (similar to OS)
  - Feature removal- non-critical features are removed from being used

Performance

- Assurance of business transaction SLA
- Cloud can elastically add additional resource
  - Increase the VM capacity on-demand
  - Add additional VMs to divert the load
- As an architect
  - It is necessary to understand Application's resource demand and projected resource usage
  - Application should predict the load and possibly ask for more resource from the cloud resource manager

Security

- Multi-tenancy in Cloud can increase security threat possibility
- A PM is hosting other VMs alongwith yours
  - Information stealing
    - Each tenant's virtual resource is ultimately mapped to a physical resource. Data from one tenant's physical resource can get copied/moved to another tenant
  - VM escape
    - Though rare, an attacker can exploit hypervisor software error and access information by accessing VM address space
  - Side channel attack
    - Attacker can deduce the information about keys and other data by monitoring timing activity of another VM
  - Denial of Service
    - Malicious tenant grabs maximum resources so that other VMs starve for resource and that impacts service quality

Testing strategy Netflix- Simian Army

- Uses test environments (called monkeys) to test
  - Latency: induces artificial delays in the communication to simulate service degradation and measures impact
  - Fault-Injection: Injects faults at several places of a running system to analyze the fault-tolerance
- Helper utilities
  - Conformity: checks if a VM instance don't adhere to best practices and shuts it down
    - E.g. Security: detects security violations or improper configuration
      - Improper configuration of security group, use of SSL, use of DRM
  - Health: monitors CPU load to detect overloaded VMs
  - Janitor: Removes unused resources

18.2

MULTI-TENANT ARCHITECTURE

Need existed long before SaaS

- A Bank with many branches
  - Each branch should be able to access only its own data
  - Global changes should be possible- by adding a field to the master table
  - Local branch specific fields should also be possible to add
  - Global processing – like inter-bank reconciliation
  - Fine grained data centric access
    - High value transactions should be visible to special people in the bank
    - Special customer names should be invisible by default

- This is exactly the need of today's multi-tenant systems

Multi-Tenancy

- Usually meant for hosted Software as a Service applications
- Multi-tenancy - single code base; multiple customers
  - *software development economies of scale driving down costs*
  - *rapid and invisible upgrades*

Single schema model

- Many early SaaS applications
  - Extra column: OU_ID indicates organization id
  - Each query is appended with OU_ID filter
- Reengineering an existing application using single schema model is difficult
- Difficult to have customer specific extensions
  - Additional metadata table needs to be created and handled in the application

Other fields of the table					OU_ID
					Org1
					Org1
					Org1
					Org2
					Org2

Multi-tenancy-Single Schema

1. fetchCustomer (cname, ou)

2

3. fetchCustomfields (“Customer”, cname, ou)

4. update()

ID	Name	Address	Value	OU_ID
100				503
101				490
102				503

Customer table

Entity	Custom_field	OU_ID	Key	Value
Customer	Hobby	503	100	Golf
Customer	Hobby	503	102	Tennis
Customer	Birthday	503	102	10/10/72
Customer	Risk	490	101	High

Customer fields table

Multi Schema Model

- Application computes the OU_ID and then switches to the appropriate database
- Separate schema for each customer
  - One customer type has name, address, hobby, birthday
  - Another customer type has name, address, risk
- Separate Metadata table for customization
  - Does not contain field-name-value pair for each customized record
  - Just the customization information

Multi-tenancy- Multiple Schema

Entity	Custom Field	OU_ID
Customer	Hobby	503
Customer	Birthday	503
Customer	Risk	490

1. fetchSchema (“Customer”, ou)

2

3. update()

```
fetchCustomer(cname, ou) {
}
```

ID	Name	Address	Hobby	Birthday	Value

Customer schema for 503

ID	Name	Address	Risk	Value

Customer schema for 490

Multi-tenancy using Cloud Data Store

- Google App Engine supports multi-tenancy anyway
  - User1's datastore does not interfere with User2
  - Though underlying data structure is the same for User1, and User2
- Suppose that you as application developer want to implement your own multi-tenancy on Google App Engine
- Once you create a "Customer" kind, you can't create it again!
  - You can leverage dynamic schema property of GFS to support this
  - If you use Python, at runtime you can create a class using `type()` function
    - Create subclasses of Customer such as Customer_503, Customer_490

Customer_503

ID	Name	Address	Hobby	Birthday	Value

Customer_490

ID	Name	Address	Risk	Value

Images and Content Ref:

http://martinfowler.com/articles/microservices.html

http://microservices.io/patterns/microservices.html

http://techblog.netflix.com/2012/07/embracing-differences-inside-netflix.html

Book: Releast It!

18.3

MICROSERVICES

Microservice

- An approach to develop an application as a suite of small components (modules)
  - Implements a business capability, independently deployable
  - Independently replaceable and upgradeable, hence loosely coupled
- (Micro) Service
  - Collection of components provide a business functionality
  - Published Interface (read IEEE Software article by Fowler)
  - Out-of-process, runs in its own process and communicating lightly- REST style
- Application
  - Collection of services, each service can be one or more processes
    - They are always built together
    - Use their own database
  - Many languages
  - Centralized management is limited and written in a different languages

Monolith vs Microservice

Design Principle

- Avoid Conway's Law (design of system is a copy of the organization's communication structure)
  - Create a service that is based on business functionality (with its own UI, business logic, database)
- Smart endpoint, dumb pipe (i.e. as simple as a reliable message bus)
  - Classic Pipe-Filter model, Each service is a Filter,
  - Receive req. → process → produce response
  - REST, not BPEL, or SOAP
- Decentralized Data Management
- Infrastructure Automation
- Design for Failure

Data storage

monolith - single database

microservices - application databases

Design of Data Persistence

- Multiple types of data store, not necessarily Relational
- Cons
  - Multiple technology -> Intergration problem (new interfaces)
  - Consistency and availability issues
- Pros
  - Faster. When relational databases are used inappropriately, they exert a significant drag on application development.
  - Typical Design flaws:
    - Aplication was essentially composing and serving web pages. They only looked up page elements by ID
    - they had no need for transactions, and no need to share their database.
    - Much better suited to a key-value store than the relational db
- Example: Gurdian website is rebuilt using MongoDB recently

Event Based Communication

Transaction Consistency

To maintain consistency a microservice must atomically publish an event

- Use datastore as a message
  - Update database: new entity state & event
  - Consume event & mark event as consumed
- Eventually consistent mechanism
  - Replay events to recreate the state
  - No more 2PC
- **Implementation Difficulty**
  - event publishing code and business logic is tangled

EventTable

e1	101	acctOpen	500
e1	101	acctCredit	250
e1	101	acctDebit	300

Architecture to Handle Failure

- Since services can fail at any time
  - it's important to be able to detect the failures quickly and,
  - if possible, automatically restore service
- Microservices based application put a lot of emphasis on real-time monitoring to check
  - architectural elements (how many requests per second is the database getting) and
  - business relevant metrics (such as how many orders per minute are received).
- Semantic monitoring
  - can provide an early warning system of something going wrong
  - Can trigger development teams to follow up and investigate.
  - This is particularly important here as the application consists of a set micro-services collaborating through messages and events
  - Monitoring is vital to spot bad emergent behavior quickly so it can be fixed.

Fault-Tolerant Architecture Pattern

- In Service oriented design, integration with external systems is essential
- Many a times the failure is due to the problem of the external system
  - failures take several forms, ranging from various network errors to semantic errors
  - Error responses will be difficult to decipher
  - Errors can be protocol violation, slow response, or outright hang.
- Fault-tolerant programming
  - Circuit Breaker, Timeouts will help in
    - Avoiding cascading failure
    - Can reduce the risk of DDOS attack on your service

Patterns- Circuit Breaker

- Critical calls (incoming or outgoing) is wrapped with a circuit breaker- which is in a normal "closed" state
  - If the call succeeds, nothing extraordinary happens.
  - If it fails, however, the circuit breaker makes a note of the failure.
  - Once the # failures (or frequency of failures, in more sophisticated cases) exceeds a threshold the circuit breaker trips and "opens" the circuit
    - Then the call fails immediately and i) a notificaiton is sent to the caller as well as an alert is sent
    - After sometime, the state is reset to CLOSED thinking that the problem is fixed

Pattern- Bulkhead

- Foo and Bar are two (micro)services who depend on a common service Baz
- If foo() crashes, or consumes excessive resource, or compromised, or triggers a bug in Baz, Bar() will also suffer. Hence Partition Bazz

Capacity Planning

- If Foo, Bar and Bazz use the same resource pools and if there resource demands are known, a shared pool is more cost effective
- Use VMs for load-balancing and implementing this pattern

Pattern - Timeout

- The timeout is a simple mechanism to break from waiting for an answer beyond certain point.
  - Typically external call, network calls
  - Asynchronous calls are often too complex for a particular situation
- Well-placed timeouts provide fault isolation
- Unfortunately, with higher levels of abstraction, explicit timeout is not present in any Program level APIs
- Vendor-provided client libraries are devoid of timeouts
  - These libraries hides a socket communication and prevent setting timeouts.
- Timeouts can also be relevant within a single application-specifically in multi-threaded application

Scalable Deployment

Monolithic Single System (all customer data A-Z)

Set of redundant, load balanced clones (clones of customer data)

Example of bulkhead

18.4

CAP THEOREM

CAP Theorem

- No distributed system can achieve all of the three properties
  - Consistency
    - The data will be consistent throughout the distributed system
  - Availability
    - The data will always be available
  - Partition Tolerance
    - The system will not fail when network is partitioned

CAP Theorem- Explanation

- Imagine a distributed system which keeps track of a single piece of data using three nodes—A, B, and C—and which claims to be both consistent and available. Misfortune strikes, and that system is partitioned into: {A,B} and {C}. Now a write request arrives at C.
    1. Accept the write, knowing that neither A nor B will know about this new data until the partition heals.
    2. Refuse the write, knowing that the client might not be able to contact A or B until the partition heals.
- **Choosing Consistency Over Availability**
    - It has to guarantee atomic reads and writes by refusing to respond to some requests. It may decide to shut down entirely (like the clients of a single-node data store), refuse writes (like Two-Phase Commit), or only respond to reads and writes for pieces of data whose "master" node is inside the partition component
- **Choosing Availability Over Consistency**
    - It will respond to all requests, potentially returning stale reads and accepting conflicting writes. These inconsistencies are often resolved via causal ordering mechanisms like vector clocks and application-specific conflict resolution procedures.
        - Dynamo systems usually offer both of these
        - Cassandra's hard-coded Last-Writer-Wins conflict resolution being the main exception

Architect's Decision

- How do we tradeoff between C, A, and P?
- Eventual Consistency
  - Nodes across partitions are allowed to be inconsistent- for some time- then it becomes consistent
  - Architect's decision- Design challenge
    - No more than n% of data should be stale
    - It shouldn't take more than t sec. to be consistent
- Availability and Partition tolerance (latency) is preferred than consistency
  - Facebook user should always be able to respond but they are okay with delayed newsfeed
  - Amazon shopping cart does not require ACID property
  - Twitter timelines may be delayed
  - Fund-transfer between Banks- delay of 24hrs is okay

Architect's Role

- Mechanism for eventual consistency
  - Caching, replication, message retry, timeout
- Quality Attribute Tradeoff
  - NoSQL databases must trade-off between
    - Availability, latency, consistency , partitioning
  - Additionally
    - Interoperability, security

THANK YOU

Reference Chapter 16
Software Architecture in Practice
Third Edition
Len Bass
Paul Clements
Rick Kazman

BITS Pilani
Pilani|Dubai|Goa|Hyderabad

Software Architecture

Designing & Documenting the Architecture #1

Architecture and Requirements

Harvinder S Jabbal
Module 19 (RL6.1)

Architecture and Requirements

- Define Architecturally Significant Requirements (ASR)
- Gathering (ASR) from Requirement document, Business goals & Stakeholders
- Capturing ASR in an Utility Tree for further refinement
- Architecture Design strategy and Attribute –Driven Design Method

innovate achieve lead

BITS Pilani

Pilani|Dubai|Goa|Hyderabad

Define Architecturally Significant Requirements (ASR)

Requirements

- Architectures exist to build systems that satisfy requirements.
- But, to an architect, not all requirements are created equal.

- An *architecturally significant requirement* (ASR) is a requirement that will have a profound effect on the architecture.

- How do we find those?

© Len Bass, Paul Clements, Rick Kazman, distributed under Creative Commons Attribution License

ASRs and Requirements Documents

- An obvious location to look for candidate ASRs is in the requirements documents or in user stories.
- Requirements should be in requirements documents!
- Unfortunately, this is not usually the case.
- Why?

© Len Bass, Paul Clements, Rick Kazman, distributed under Creative Commons Attribution License

Don't Get Your Hopes Up

- Many projects don't create or maintain the detailed, high-quality requirements documents.
- Standard requirements pay more attention to functionality than quality attributes.
- Most of what is in a requirements specification does not affect the architecture.
- No architect just sits and waits until the requirements are "finished" before starting work. The architect *must* begin while the requirements are still in flux.

When does the Architect Start?

© Len Bass, Paul Clements, Rick Kazman, distributed under Creative Commons Attribution License

Don't Get Your Hopes Up

- Quality attributes, when captured at all, are often captured poorly.
  - "The system shall be modular"
  - "The system shall exhibit high usability"
  - "The system shall meet users' performance expectations"
- Much of what is useful to an architect is not in even the best requirements document.
  - ASRs often derive from business goals in the development organization itself
  - Developmental qualities (such as teaming) are also out of scope

Most ASRs are not obvious.

© Len Bass, Paul Clements, Rick Kazman, distributed under Creative Commons Attribution License

innovate achieve lead

BITS Pilani

Pilani|Dubai|Goa|Hyderabad

Gathering (ASR) from
Requirement document,
Business goals & Stakeholders

Sniffing Out ASRs

Design Decision Category	Look for Requirements Addressing . . .
Allocation of Responsibilities	Planned evolution of responsibilities, user roles, system modes, major processing steps, commercial packages
Coordination Model	Properties of the coordination (timeliness, currency, completeness, correctness, and consistency) Names of external elements, protocols, sensors or actuators (devices), middleware, network configurations (including their security properties) Evolution requirements on the list above
Data Model	Processing steps, information flows, major domain entities, access rights, persistence, evolution requirements
Management of Resources	Time, concurrency, memory footprint, scheduling, multiple users, multiple activities, devices, energy usage, soft resources (buffers, queues, etc.) Scalability requirements on the list above
Mapping among Architectural Elements	Plans for teaming, processors, families of processors, evolution of processors, network configurations
Binding Time Decisions	Extension of or flexibility of functionality, regional distinctions, language distinctions, portability, calibrations, configurations
Choice of Technology	Named technologies, changes to technologies (planned and unplanned)

© Ka[illegible] under Creative Commons Attribution License

Requirements that can affect:

ALLOCATION OF RESPONSIBILITY

Requirements that can affect: COORDINATION MODEL

Properties of coordination

- timeliness
- currency
- completeness
- correctness
- Consistence

Names of

- external elements
- protocols
- sensors
- actuators (devices),
- middleware.
- network configurations (including their security properties)

Evolution requirements on the list at the left

Requirements that can affect: DATA MODEL

- Processing steps
- information flow
- major domain enteties
- access rights
- persistence
- evolution requirements

Requirements that can affect:

MANAGEMENT OF RESOURCES

- Time
- concurrency
- memory footprint
- scheduling
- multiple users
- multiple activities
- devices
- energy usage
- soft resources (buffers, queues etc)
- Scalability requirements on the list above

Requirements that can affect:

MAPPING AMONG ARCHITECTURAL ELEMENTS

Plans for

- teaming
- processors
- families of processors
- evolution of processors
- network configuration

Requirements that can affect:

BINDING TIME DECISIONS

- Extension of or flexibility of functionality
- regional distinctions
- portability
- calibration
- configurations

Requirements that can affect:

CHOICE OF TECHNOLOGY

Named technologies

changes in technologies (planned and unplanned)

Gathering ASRs from Stakeholders

- Say your project won't have the QAs nailed down by the time you need to start your design work.
- What do you do?
- Stakeholders often have *no idea* what QAs they want in a system
  - if you insist on quantitative QA requirements, you're likely to get numbers that are arbitrary.
  - at least some of those requirements will be very difficult to satisfy.
- Architects often have very good ideas about what QAs are reasonable to provide.
- Interviewing the relevant stakeholders is the surest way to learn what they know and need.

© Len Bass, Paul Clements, Rick Kazman, distributed under Creative Commons Attribution License

Gathering ASRs from Stakeholders

The results of stakeholder interviews should include

- a list of architectural drivers
- a set of QA scenarios that the stakeholders (as a group) prioritized.

This information can be used to:

- **refine** system and software **requirements**
- understand and clarify the system's **architectural drivers**
- provide rationale for why the architect subsequently made certain **design decisions**
- guide the development of **prototypes and simulations**
- influence the **order in which the architecture** is developed.

© Len Bass, Paul Clements, Rick Kazman, distributed under Creative Commons Attribution License

Quality Attribute Workshop

- The QAW is a facilitated, stakeholder-focused method to generate, prioritize, and refine **quality attribute scenarios** before the software architecture is completed.
- The QAW is focused on **system-level concerns** and specifically the role that software will play in the system.

Q A W

© Len Bass, Paul Clements, Rick Kazman, distributed under Creative Commons Attribution License

QAW Steps

Step 1: QAW Presentation and Introductions.

- QAW facilitators describe the motivation for the QAW and explain each step of the method.

Step 2: Business/Mission Presentation.

- The stakeholder representing the business concerns behind the system presents the system's business context, broad functional requirements, constraints, and known quality attribute requirements.
- The quality attributes that will be refined in later steps will be derived largely from the business/mission needs presented in this step.

Step 3: Architectural Plan Presentation.

- The architect will present the system architectural plans as they stand.
- This lets stakeholders know the current architectural thinking, to the extent that it exists.

Step 4: Identification of Architectural Drivers.

- The facilitators will share their list of key architectural drivers that they assembled during Steps 2 and 3, and ask the stakeholders for clarifications, additions, deletions, and corrections.
- The idea is to reach a consensus on a distilled list of architectural drivers that includes overall requirements, business drivers, constraints, and quality attributes.

© Len Bass, Paul Clements, Rick Kazman, distributed under Creative Commons Attribution License

QAW Steps

Step 5: Scenario Brainstorming.

- Each stakeholder expresses a scenario representing his or her concerns with respect to the system.
- Facilitators ensure that each scenario has an explicit stimulus and response.
- The facilitators ensure that at least one representative scenario exists for each architectural driver listed in Step 4.

Step 6: Scenario Consolidation.

- Similar scenarios are consolidated where reasonable.
- Consolidation helps to prevent votes from being spread across several scenarios that are expressing the same concern.

Step 7: Scenario Prioritization.

- Prioritization of the scenarios is accomplished by allocating each stakeholder a number of votes equal to 30 percent of the total number of scenarios

Step 8: Scenario Refinement.

- The top scenarios are refined and elaborated.
- Facilitators help the stakeholders put the scenarios in the six-part scenario form of source-stimulus-artifact-environment-response-response measure.

© Len Bass, Paul Clements, Rick Kazman, distributed under Creative Commons Attribution License

ASRs from Business Goals

FIGURE 3.2 Some business goals may lead to quality attribute requirements (which lead to architectures), or lead directly to architectural decisions, or lead to nonarchitectural solutions.

© Len Bass, Paul Clements, Rick Kazman, distributed under Creative Commons Attribution License

A General Scenario for Business Goals

A Goal-subject (example stakeholder)

..... has the goal that ....

a goal-object (example Individual/System/Organisation)

..... Achieves ....

a goal (example improve business processes)

.... In the context of .....

the environment (example competitive)

... And will be considered to be satisfied if ...

Goal-measure (based on goal (categories)

... Is which is measured based on .

Value

... of output vis-a-vis resources you are willing to expend.

Examples of Business Goals For the following goal-objects

- Individual
- System
- Portfolio
- Organisation's Employees
- Organisation's Shareholders
- Organisation
- Nation
- Society

Goal-Object: Individual Business Goals

Personalwealth

power

honor/face/reputation

game and gambling spirit

maintain or improve

reputation(personal)

family interests

Goal-Object: System Business Goals->

Manage flexibility	distributed development	portability	Opensystems/standards	testability
product lines	integrability	interoperability	ease of installation and ease of repair	flexibility/configurabiiity
performance	reliability/availability	ease of use	security	safety
scalability/extendibility	functionality	system constraints	internationalization	reduce time to market

Goal-Object: Portfolio Business Goals->

Reduce cost of development	costleadership,	differentiation,	reducecostof	retirement
smooth transition to follow-on systems	replace legacy systems	replace labor with automation	diversify operational sequence	eliminate intermediate stages
automate tracking of business events	collect/communicate/retrieve operational knowledge	improve decision making	coordinate across distance	align task and process
manage on basis of process measurements	operate effectively within the competitive environment, the technological environment, or the customerenvironment	Create something new	provide the best quality products and services possible	be the leading innovator in the industry

Goal-Object: Organisation's Employees Business Goals->

Provide high rewards and benefits to employees,

create a pleasant and friendly work place,

have satisfied employees

fulfill responsibility toward employees

maintain jobs of work force on legacy systems

Goal-Object: Organisation's Shareholders

Business Goals->

Maximize dividends for the shareholders

Goal-Object: Organisation Business Goals->

Growth of the business	continuity of the business	maximize profits over the short run	maximize profits over the long run
survival of the organization	maximize the companys net assets and reserves	be a market leader	maximize the market share
expand or retain market share	enter new markets	maximize the companys rate of growth	keep tax payments to a minimum
increase sales growth	maintain or improve reputation	achieve business goals through financial objectives	run a stable organization

Goal-Object: Nation Business Goals->

Patriotism

national pride

national security

national welfare

Goal-Object: Society Business Goals->

Run an ethical organization

responsibility toward society

be a socially responsible company,

be of service to the community

operate effectively within social environment

operate effectively within legal environment

Categories of Business Goals, to Aid in Elicitation

BUSINESS GOAL	• GOAL-MEASURE
Contributing to the growth and continuity of the organisation	• Time that business remains viable
Meeting Financial objectives	• Financial Performance vs. objectives
Meeting personal objectives	• Promotion or raise achieved in period
Meeting responsibilities to employees	• Employee satisfaction; turnover rate
Meeting responsibilities to society	• Contribution to trade deficit
Meeting responsibilities to state	• Stock price, dividends
Meeting responsibilities to shareholders	• Market Share
Managing market position	• Time to carry out business
Improving business processes	• Quality measures of products
Managing the quality and reputation of the products	• Technology-related problems
Managing change in environmental factors	• Time window for achievement

Kazman, distributed under Creative Commons Attribution License

Expressing Business Goals
Business goal scenario, 7 parts

1. Goal-source

The people or written artifacts providing the goal.

2. Goal-subject

The stakeholders who own the goal and wish it to be true.	Each stakeholder might be an individual or the organization itself

3. Goal-object

The entities to which the goal applies.

4. Environment

The context for this goal	Environment may be social, legal, competitive, customer, and technological

© Len Bass, Paul Clements, Rick Kazman, distributed under Creative Commons Attribution License

Expressing Business Goals
Business goal scenario, 7 parts

5. Goal

Any business goal articulated by the goal-source.

SEE LIST IN PREVIOUS SLIDE

6. Goal-measure

A testable measurement to determine how one would know if the goal has been achieved. The goal-measure should usually include a time component, stating the time by which the goal should be achieved.

7. Pedigree and value

The degree of confidence the person who stated the goal has in it	The goal's volatility and value.

© Len Bass, Paul Clements, Rick Kazman, distributed under Creative Commons Attribution License

PALM: A Method for Eliciting Business Goals

PALM is a seven-step method.

Nominally carried out over a day and a half in a workshop.

Attended by architect(s) and stakeholders who can speak to the relevant business goals.

© Len Bass, Paul Clements, Rick Kazman, distributed under Creative Commons Attribution License

PALM Steps

PALM overview presentation

Overview of PALM, the problem it solves, its steps, and its expected outcomes.

Business drivers presentation.

Briefing of business drivers by project management	What are the goals of the customer organization for this system?	What are the goals of the development organization?

Architecture drivers presentation

Briefing by the architect on the driving business and quality attribute requirements: the ASRs.

Business goals elicitation

Business goals are elaborated and expressed as scenarios.	Consolidate almost-alike business goals to eliminate duplication.	Participants prioritize the resulting set to identify the most important goals.

© Len Bass, Paul Clements, Rick Kazman, distributed under Creative Commons Attribution License

PALM Steps

Identification of potential quality attributes from business goals.

For each important business goal scenario, participants describe a quality attribute that (if architected into the system) would help achieve it. If the QA is not already a requirement, this is recorded as a finding.

Assignment of pedigree to existing quality attribute drivers.

For each architectural driver identify which business goals it is there to support.	If none, that's recorded as a finding.	Otherwise, we establish its pedigree by asking for the source of the quantitative part.

Exercise conclusion

Review of results, next steps, and participant feedback.

© Len Bass, Paul Clements, Rick Kazman, distributed under Creative Commons Attribution License

innovate achieve lead

BITS Pilani

Pilani|Dubai|Goa|Hyderabad

Capturing ASR in an Utility Tree for further refinement

Capturing ASRs in a Utility Tree

An ASR must have the following characteristics:

A profound impact on the architecture

- Including this requirement will very likely result in a different architecture than if it were not included.

A high business or mission value

- If the architecture is going to satisfy this requirement it must be of high value to important stakeholders.

© Len Bass, Paul Clements, Rick Kazman, distributed under Creative Commons Attribution License

Utility Tree

A way to record ASRs all in one place.

Establishes priority of each ASR in terms of

- Impact on architecture
- Business or mission value

ASRs are captured as scenarios.

Root of tree is placeholder node called “Utility”.

Second level of tree contains broad QA categories.

Third level of tree refines those categories.

© Len Bass, Paul Clements, Rick Kazman, distributed under Creative Commons Attribution License

Utility Tree Example (excerpt)

utility

Quality Attribute	Attribute Refinement	ASR
Performance	Transaction response time	A user updates a patient's account in response to a change-of-address notification while the system is under peak load, and the transaction completes in less than 0.75 second. (H,M)
		A user updates a patient's account in response to a change-of-address notification while the system is under double the peak load, and the transaction completes in less than 4 seconds. (L,M)
	Throughput	At peak load, the system is able to complete 150 normalized transactions per second. (M,M)
Usability	Proficiency training	A new hire with two or more years' experience in the business becomes proficient in Nightingale's core functions in less than 1 week. (M,L)
		A user in a particular context asks for help, and the system provides help for that context, within 3 seconds. (H,M)
	Normal operations	A hospital payment officer initiates a payment plan for a patient while interacting with that patient and completes the process without the system introducing delays. (M,M)
Configurability	User-defined changes	A hospital increases the fee for a particular service. The configuration team makes the change in 1 working day; no source code needs to change. (H,L)
Maintainability	Routine changes	A maintainer encounters search- and response-time deficiencies, fixes the bug, and distributes the bug fix with no more than 3 person-days of effort. (H,M)
		A reporting requirement requires a change to the report-generating metadata. Change is made in 4 person-hours of effort. (M,L)
	Upgrades to commercial components	The database vendor releases a new version that must be

Key: Utility
H=high
M=medium
L=low

© Len Bass, Paul Clements, Rick Kazman, distributed under Creative Commons Attribution License

Utility Tree: Next Steps

- A QA or QA refinement without any ASR is not necessarily an error or omission
  - Attention should be paid to searching for unrecorded ASRs in that area.
- ASRs that rate a (H,H) rating are the ones that deserve the most attention
  - A very large number of these might be a cause for concern: Is the system is achievable?
- Stakeholders can review the utility tree to make sure their concerns are addressed.

© Len Bass, Paul Clements, Rick Kazman, distributed under Creative Commons Attribution License

Tying the Methods Together

Requirement Documents

↓

Stakeholders Interview

For important stakeholders who have been overlooked

↓

Quality Attribute Workshop

Capture inputs from Stakeholders

↓

PALM (Pedigree Attribute eLicitation Method)

Capture Business goals behind the system

↓

Quality Attribute Utility Tree

Repository of scenarios

Summary

- Architectures are driven by architecturally significant requirements: requirements that will have profound effects on the architecture.
  - Architecturally significant requirements may be captured from requirements documents, by interviewing stakeholders, or by conducting a Quality Attribute Workshop.
- Be mindful of the business goals of the organization.
  - Business goals can be expressed in a common, structured form and represented as scenarios.
  - Business goals may be elicited and documented using a structured facilitation method called PALM.
- A useful representation of quality attribute requirements is in a utility tree.
  - The utility tree helps to capture these requirements in a structured form.
  - Scenarios are prioritized.
  - This prioritized set defines your “marching orders” as an architect.

© Len Bass, Paul Clements, Rick Kazman, distributed under Creative Commons Attribution License

Thank you……..

Credits

- **Chapter Reference from Text T1:** 16, 17, 18
- Slides have been adapted from Authors Slides Software Architecture in Practice – Third Ed.
    - Len Bass
    - Paul Clements
    - Rick Kazman

© Len Bass, Paul Clements, Rick Kazman, distributed under Creative Commons Attribution License

Reference Chapter 17
Software Architecture in Practice
Third Edition
Len Bass
Paul Clements
Rick Kazman

BITS Pilani
Pilani|Dubai|Goa|Hyderabad

Software Architecture

Designing & Documenting the Architecture #1

Designing an Architecture

Harvinder S Jabbal
Module RL6.2

Designing & Documenting the Architecture #1

- Define Architecturally Significant Requirements (ASR)
- Gathering (ASR) from Requirement document, Business goals & Stakeholders
- Capturing ASR in an Utility Tree for further refinement
- Architecture Design strategy and Attribute –Driven Design Method

innovate achieve lead

BITS Pilani

Pilani|Dubai|Goa|Hyderabad

Architecture Design strategy and Attribute –Driven Design Method

Chapter Outline

Design Strategy

The Attribute-Driven Design Method

The Steps of ADD

Summary

© Len Bass, Paul Clements, Rick Kazman, distributed under Creative Commons Attribution License

innovate achieve lead

BITS Pilani

Pilani|Dubai|Goa|Hyderabad

Design Strategy

Design Strategy-

Ideas key to architectural design methods

Idea 1
- Decomposition

Idea 2
- Designing to Architecturally Significant Requirements

Idea 3
- Generate and Test

© Len Bass, Paul Clements, Rick Kazman, distributed under Creative Commons Attribution License

Idea 1: Decomposition

- Architecture determines quality attributes
- Important quality attributes are characteristics of the *whole* system.
- Design therefore begins with the whole system
  - The whole system is decomposed into parts
  - Each part may inherit all of part of the quality attribute requirements from the whole

© Len Bass, Paul Clements, Rick Kazman, distributed under Creative Commons Attribution License

Design Doesn't Mean Green Field

- If you are given components to be used in the final design, then the decomposition must accommodate those components.

© Len Bass, Paul Clements, Rick Kazman, distributed under Creative Commons Attribution License

Idea 2: Designing to Architecturally Significant Requirements

- Remember architecturally significant requirements (ASRs)?

- These are the requirements that you must satisfy with the design
  - There are a small number of these
  - They are the most important (by definition)

© Len Bass, Paul Clements, Rick Kazman, distributed under Creative Commons Attribution License

How Many ASRs Simultaneously?

- If you are inexperienced in design then design for the ASRs one at a time beginning with the most important.

- As you gain experience, you will be able to design for multiple ASRs simultaneously.

© Len Bass, Paul Clements, Rick Kazman, distributed under Creative Commons Attribution License

What About Other Quality Requirements?

If your design does not satisfy a particular non ASR quality requirement then either

- **IMPROVE THE DESIGN**
  - **Adjust your design** so that the ASRs are still satisfied and so is this additional requirement or
- **WEAKEN THE REQUIREMENT**
  - **Weaken the additional requirement** so that it can be satisfied either by the current design or by a modification of the current design or
- **CHANGE PRIORITIES**
  - **Change the priorities** so that the one not satisfied becomes one of the ASRs or
- **DECLARE NON-SATISFIABLE**
  - **Declare the additional requirement non-satisfiable** in conjunction with the ASRs.

© Len Bass, Paul Clements, Rick Kazman, distributed under Creative Commons Attribution License

Idea 3: Generate and Test

View the current design as a hypothesis.

- Assume requirement will be satisfied

Ask whether the current design satisfies the requirements

- Test if requirement is satisfied.

If not, then generate a new hypothesis

© Len Bass, Paul Clements, Rick Kazman, distributed under Creative Commons Attribution License

Raises the Following Questions

- Where does initial hypothesis come from?
- How do I test a hypothesis?
- When am I done?
- How do I generate the next hypothesis?

- You already know most of the answers; it is just a matter of organizing your knowledge.

© Len Bass, Paul Clements, Rick Kazman, distributed under Creative Commons Attribution License

Initial Hypothesis come from "collateral" that are available to the project

Desirable sources	
Existing systems	Frameworks

Less desirable sources		
Patterns and tactics	Domain decomposition	Design checklists

Why "less desirable"?
The less desirable ones do not cover all of the requirements. They typically omit many of the quality attribute requirements.

© Len Bass, Paul Clements, Rick Kazman, distributed under Creative Commons Attribution License

How Do I Test a Hypothesis?

Use the analysis techniques already covered

Design checklists from quality attribute discussion.

- Example- coordination model to support capturing activity to support testabilitycollect

Architecturally significant requirements

- Does the hypothesis provide a solution for the ASR.

© Len Bass, Paul Clements, Rick Kazman, distributed under Creative Commons Attribution License

What is the output of the tests?

List of requirements

either responsibilities not met by current design

Or quality not met by current design.

How Do I Generate the Next Hypothesis?

Add missing responsibilities.

Be mindful of the side effects of a tactic.

Use tactics to adjust quality attribute behavior of hypothesis.

The choice of tactics will depend on which quality attribute requirements are not met.

© Len Bass, Paul Clements, Rick Kazman, distributed under Creative Commons Attribution License

When Am I Done?

All ASRs are satisfied and/or…

You run out of budget for design activity

- In this case, use the best hypothesis so far.
- Begin implementation
- Continue with the design effort although it will now be constrained by implementation choices.

© Len Bass, Paul Clements, Rick Kazman, distributed under Creative Commons Attribution License

innovate achieve lead

BITS Pilani

Pilani|Dubai|Goa|Hyderabad

The Attribute-Driven Design Method

The Attribute-Driven Design Method

Packaging of many of the techniques already discussed.

An iterative method. At each iteration you

- Choose a part of the system to design.
- Marshal all the architecturally significant requirements for that part.
- Generate and test a design for that part.

ADD does not result in a complete designbut the main design approach

- Set of containers with responsibilities
- Interactions and information flow among containers

Does not produce an API or signature for containers.

- Gives a “workable” architecture early and quickly

© Len Bass, Paul Clements, Rick Kazman, distributed under Creative Commons Attribution License

ADD Inputs and Outputs

The Steps of ADD

The Steps of ADD

1. Choose an element of the system to design.
2. Identify the ASRs for the chosen element.
3. Generate a design solution for the chosen element.
4. Inventory remaining requirements and select the input for the next iteration.
5. Repeat steps 1–4 until all the ASRs have been satisfied.

© Len Bass, Paul Clements, Rick Kazman, distributed under Creative Commons Attribution License

Step 1:

• Choose an Element of the System to Design

- For green field designs, the element chosen is usually the whole system.
- For legacy designs, the element is the portion to be added.
- After the first iteration:

- *Initial iteration will be broad with less depth.*
- *Gradually get fine-grained.*

© Len Bass, Paul Clements, Rick Kazman, distributed under Creative Commons Attribution License

Which Element Comes Next?

Two basic refinement strategies:	Which one to choose?	If using new technology =>	If a team needs work =>	Otherwise =>
Breadth first Depth first	It depends ☺	depth first: explore the implications of using that technology.	depth first: generate requirements for that team.	breadth first.

© Len Bass, Paul Clements, Rick Kazman, distributed under Creative Commons Attribution License

Step 2

• Identify the ASRs for the Chosen Element

- If the chosen element is the whole system, then use a utility tree (as described earlier).
- If the chosen element is further down the decomposition tree, then generate a utility tree from the requirements for that element.

© Len Bass, Paul Clements, Rick Kazman, distributed under Creative Commons Attribution License

Step 3

- Generate a Design Solution for the Chosen Element

Choose element for design

List ASRs that apply to this element

↓

Take each ASR in turn

Develop a solution by choosing a candidate design approach

↓

Initial candidate design

Inspired by pattern	Augmented by one/more tactics

↓

Refine the candidate design

Use design checklists for the quality attribute

↓

Arrive at design decision

This becomes a cpnstraint to all future steps.

© Len Bass, Paul Clements, Rick Kazman, distributed under Creative Commons Attribution License

Step 4 • Select the Input for the Next Iteration

Ensure that requirement has been satisfied,

- if not:
- BACKTRACK.

ASR not yet satisfied should relate to

- Quality Attribute Requirement/
- Functional responsibility /
- constraint of the parent element

then add responsibilities to satisfy the requirement.

- Add them to container with similar requirements
- If no such container may need to create new one or add to container with dissimilar responsibilities (coherence)
- If container has too many requirements for a team, split it into two portions. Try to achieve loose coupling when splitting.

© Len Bass, Paul Clements, Rick Kazman, distributed under Creative Commons Attribution License

For each Quality Attribute Requirements, responsibility and constraint.

If the quality attribute requirement has been satisfied,	• it does not need to be further considered.
If the quality attribute requirement has not been satisfied then either	• Delegate it to one of the child elements • Split it among the child elements
If the quality attribute cannot be satisfied,	• see if it can be weakened. • If it cannot be satisfied or weakened then it cannot be met.

© Len Bass, Paul Clements, Rick Kazman, distributed under Creative Commons Attribution License

Constraints

Constraints are treated as quality attribute requirements have been treated.

Satisfied	Delegated	Split	Unsatisfiable

© Len Bass, Paul Clements, Rick Kazman, distributed under Creative Commons Attribution License

Step 5

• Repeat Steps 1–4 Until All ASRs are Satisfied

At end of step 3, each child element will have associated with it a set of:

functional requirements (responsibilities),	quality attribute requirements, and	constraints.

This sets up the child element for the next iteration of the method.

ADD PROCESS CAN BE TERMINATED IF

All requirements satisfied	High degree of trust between architect and implementation team.	Contractual arrangement satisfied.	Project's design budget exhausted.

© Len Bass, Paul Clements, Rick Kazman, distributed under Creative Commons Attribution License

Summary

Summary

Designing the architecture is a matter of		
Determining the ASRs	Performing generate and test one an element to decompose it to satisfy the ASRs	Iterating until requirements are satisfied.

© Len Bass, Paul Clements, Rick Kazman, distributed under Creative Commons Attribution License

Thank you……..

Credits

- **Chapter Reference from Text T1:** 16, 17, 18
- Slides have been adapted from Authors Slides Software Architecture in Practice – Third Ed.
    - Len Bass
    - Paul Clements
    - Rick Kazman

Reference Chapter 18
Software Architecture in Practice
Third Edition
Len Bass
Paul Clements
Rick Kazman

BITS Pilani
Pilani|Dubai|Goa|Hyderabad

Software Architecture

Designing & Documenting the Architecture #2

Harvinder S Jabbal
Module RL7.0

Designing & Documenting the Architecture #2

Documenting the Architecture with relevant views

- Module,
- C&C,
- Allocation &
- Quality

Documenting beyond views,

- Documenting behaviours with sequence diagram

BITS Pilani, Deemed to be University under Section 3 of UGC Act, 1956

innovate achieve lead

BITS Pilani

Pilani|Dubai|Goa|Hyderabad

Documenting the Architecture with relevant views – Module, C&C, Allocation & Quality

Architecture Documentation

Even the best architecture will be useless if the people who need it

- do not know what it is;
- cannot understand it well enough to use, build, or modify it;
- misunderstand it and apply it incorrectly.

All of the effort, analysis, hard work, and insightful design on the part of the architecture team will have been wasted.

Chapter Outline

1. Uses and Audiences for Architecture Documentation
2. Notations for Architecture Documentation
3. Views
4. Choosing the Views
5. Combining Views
6. Building the Documentation Package
7. Documenting Behavior
8. Architecture Documentation and Quality Attributes
9. Documenting Architectures That Change Faster Than You Can Document Them
10 Documenting Architecture in an Agile Development Project
11. Summary

BITS Pilani, Deemed to be University under Section 3 of UGC Act, 1956

innovate achieve lead

BITS Pilani

Pilani|Dubai|Goa|Hyderabad

1. Uses and Audience for Architecture Documentation

1. Uses and Audience for Architecture Documentation

innovate achieve lead

Architecture documentation must

- be sufficiently transparent and accessible to be quickly understood by new employees
- be sufficiently concrete to serve as a blueprint for construction
- have enough information to serve as a basis for analysis.

Architecture documentation is both prescriptive and descriptive.

- For some audiences, it prescribes what *should* be true, placing constraints on decisions yet to be made.
- For other audiences, it describes what *is* true, recounting decisions already made about a system's design.

Understanding stakeholder uses of architecture documentation is essential

- Those uses determine the information to capture.

BITS Pilani, Deemed to be University under Section 3 of UGC Act, 1956

Three Uses for Architecture Documentation

Education

- Introducing people to the system
  - New members of the team
  - External analysts or evaluators
  - New architect

Primary vehicle for communication among stakeholders

- Especially architect to developers
- Especially architect to future architect!

Basis for system analysis and construction

- documentation serves as the basis for architecture evaluation.

BITS Pilani, Deemed to be University under Section 3 of UGC Act, 1956

2. Notations

2. Notations

Informal notations

- Views are depicted (often graphically) using general-purpose diagramming and editing tools
- The semantics of the description are characterized in natural language
- They cannot be formally analyzed

Semiformal notations

- Standardized notation that prescribes graphical elements and rules of construction
- Lacks a complete semantic treatment of the meaning of those elements
- Rudimentary analysis can be applied
- UML is a semiformal notation in this sense.

Formal notations

- Views are described in a notation that has a precise (usually mathematically based) semantics.
- Formal analysis of both syntax and semantics is possible.
- Architecture description languages (ADLs)
- Support automation through associated tools.

BITS Pilani, Deemed to be University under Section 3 of UGC Act, 1956

Choosing a Notation

innovate achieve lead

Tradeoffs

- Typically, more formal notations take more time and effort to create and understand, but offer reduced ambiguity and more opportunities for analysis.
- Conversely, more informal notations are easier to create, but they provide fewer guarantees.

Different notations are better (or worse) for expressing different kinds of information.

- UML class diagram will not help you reason about schedulability, nor will a sequence chart tell you very much about the system's likelihood of being delivered on time.
- Choose your notations and representation languages knowing the important issues you need to capture and reason about.

BITS Pilani, Deemed to be University under Section 3 of UGC Act, 1956

3. Views

3. Views

Views let us divide a software architecture into a number of (we hope) interesting and manageable representations of the system.

Principle of architecture documentation:

- *Documenting an architecture is a matter of documenting the relevant views and then adding documentation that applies to more than one view.*

BITS Pilani, Deemed to be University under Section 3 of UGC Act, 1956

Which Views? The Ones You Need!

Different views support different goals and uses.

We do not advocate a particular view or collection of views.

The views you should document depend on the uses you expect to make of the documentation.

Each view has a cost and a benefit; you should ensure that the benefits of maintaining a view outweigh its costs.

BITS Pilani, Deemed to be University under Section 3 of UGC Act, 1956

Overview of Module Views

Elements

- Modules, which are implementation units of software that provide a coherent set of responsibilities.

Relations

- *Is part of*, which defines a part/whole relationship between the submodule—the part—and the aggregate module—the whole.
- *Depends on*, which defines a dependency relationship between two modules. Specific module views elaborate what dependency is meant.
- *Is a*, which defines a generalization/specialization relationship between a more specific module—the child—and a more general module—the parent.

Overview of Module Views

Constraints

- Different module views may impose specific topological constraints, such as limitations on the visibility between modules.

Usage

- Blueprint for construction of the code
- Change-impact analysis
- Planning incremental development
- Requirements traceability analysis
- Communicating the functionality of a system and the structure of its code base
- Supporting the definition of work assignments, implementation schedules, and budget information
- Showing the structure of information that the system needs to manage

BITS Pilani, Deemed to be University under Section 3 of UGC Act, 1956

Module Views

- It is unlikely that the documentation of any software architecture can be complete without at least one module view.

BITS Pilani, Deemed to be University under Section 3 of UGC Act, 1956

Overview of C&C Views

Elements

- *Components.* Principal processing units and data stores. A component has a set of *ports* through which it interacts with other components (via connectors).
- *Connectors*. Pathways of interaction between components. Connectors have a set of roles (interfaces) that indicate how components may use a connector in interactions.

Relations

- *Attachments*. Component ports are associated with connector roles to yield a graph of components and connectors.
- *Interface delegation.* In some situations component ports are associated with one or more ports in an "internal" subarchitecture. The case is similar for the roles of a connector

Overview of C&C Views

Constraints

- Components can only be attached to connectors, not directly to other components.
- Connectors can only be attached to components, not directly to other connectors.
- Attachments can only be made between compatible ports and roles.
- Interface delegation can only be defined between two compatible ports (or two compatible roles).
- Connectors cannot appear in isolation; a connector must be attached to a component.

Usage

- Show how the system works.
- Guide development by specifying structure and behavior of runtime elements.
- Help reason about runtime system quality attributes, such as performance and availability.

BITS Pilani, Deemed to be University under Section 3 of UGC Act, 1956

Notations for C&C Views

UML components are good match for C&C components.

Suggested Reading:

http://agilemodeling.com/artifacts/componentDiagram.htm

BITS Pilani, Deemed to be University under Section 3 of UGC Act, 1956

Notations for C&C Views

UML connectors are not rich enough to represent many C&C connectors.	• UML connectors cannot have substructure, attributes, or behavioral descriptions.
Represent a “simple” C&C connector using a UML connector—a line.	• Many commonly used C&C connectors have well-known, application-independent semantics and implementations, such as function calls or data read operations. • You can use a stereotype to denote the type of connector.
Connector roles cannot be explicitly represented with a UML connector.	• The UML connector element does not allow the inclusion of interfaces. • Label the connector ends and use these labels to identify role descriptions that must be documented elsewhere.
Represent a “rich” C&C connector	• using a UML component, or by annotating a line UML connector with a tag that explains the meaning of the complex connector.

Overview of Allocation Views

Elements

- *Software element* and *environmental element.*
- A software element has properties that are *required* of the environment.
- An environmental element has properties that are *provided* to the software.

Relations

- *Allocated to*. A software element is mapped (allocated to) an environmental element. Properties are dependent on the particular view.

BITS Pilani, Deemed to be University under Section 3 of UGC Act, 1956

Overview of Allocation Views

Constraints

- Varies by view

Usage

- Reasoning about performance, availability, security, and safety.
- Reasoning about distributed development and allocation of work to teams.
- Reasoning about concurrent access to software versions.
- Reasoning about the form and mechanisms of system installation.

BITS Pilani, Deemed to be University under Section 3 of UGC Act, 1956

Quality Views

A *quality view* can be tailored

- for specific stakeholders or to address specific concerns.

A quality views is formed

- by extracting the relevant pieces of structural views and packaging them together.

BITS Pilani, Deemed to be University under Section 3 of UGC Act, 1956

Quality Views: Examples

- Show the components that have some security role or responsibility, how those components communicate, any data repositories for security information, and repositories that are of security interest.
- The view's context information would show other security measures (such as physical security) in the system's environment.
- The behavior part of a security view
  - Show how the operation of security protocols and where and how humans interact with the security elements.
  - Capture how the system would respond to

Security view

- Especially helpful for systems that are globally dispersed and heterogeneous.
- Show all of the component-to-component channels, the various network channels, quality-of-service parameter values, and areas of concurrency.
- Used to analyze certain kinds of performance and reliability (such as deadlock or race condition detection).
- The behavior part of this view could show (for example) how network bandwidth is dynamically allocated.

Communications view

Quality Views: Examples

- Could help illuminate and draw attention to error reporting and resolution mechanisms.
- Show how components detect, report, and resolve faults or errors.
- It would help identify the sources of errors

Exception or *error-handling view*

- Models mechanisms such as replication and switchover.
- Depicts timing issues and transaction integrity.

Reliability view

- Shows those aspects of the architecture useful for inferring the system's performance.
- Show network traffic models, maximum latencies for operations, and so forth.

Performance view

innovate achieve lead

BITS Pilani

Pilani|Dubai|Goa|Hyderabad

Sample separator slide for the presentation

4. Choosing the Views

You can determine which views are required, when to create them, and how much detail to include if you know the following:

What people, and with what skills, are available	Which standards you have to comply with	What budget is on hand	What the schedule is	What the information needs of the important stakeholders are	What the driving quality attribute requirements are	What the approximate size of the system is

5. Combining Views

5. Combining Views

At a minimum, expect to have		
at least one module view,	at least one C&C view,	and for larger systems, at least one allocation view in your architecture document.

Method for Choosing the Views

Step 1. Build a stakeholder/view table.

- Rows: List the stakeholders for your project's software architecture documentation
- Columns: Enumerate the views that apply to your system.
  - Use the structures discussed in Chapter 1, the views discussed in this chapter, and the views that your design work in ADD has suggested as a starting list of candidates.
  - Include the views or view sketches you have as a result of your design work so far.
- Some views (such as decomposition, uses, and work assignment) apply to every system, while others (various C&C views, the layered view) only apply to some systems.
- Fill in each cell to describe how much information the stakeholder requires from the view: none, overview only, moderate detail, or high detail.

BITS Pilani, Deemed to be University under Section 3 of UGC Act, 1956

Method for Choosing the Views

Step 2. Combine views to reduce their number

- Look for marginal views in the table; those that require only an overview, or that serve very few stakeholders.
- Combine each marginal view with another view that has a stronger constituency.
- These views often combine naturally:
  - *Various C&C views.* Because C&C views all show runtime relations among components and connectors of various types, they tend to combine well.
  - *Deployment view with either SOA or communicating-processes views.* An SOA view shows services, and a communicating-processes view shows processes. In both cases, these are components that are deployed onto processors.
  - *Decomposition view and any of work assignment, implementation, uses, or layered views.* The decomposed modules form the units of work, development, and uses. In addition, these modules populate layers.

Method for Choosing the Views

Step 3. Prioritize and stage.

- The decomposition view (one of the module views) is a particularly helpful view to release early.
  - High-level decompositions are often easy to design
  - The project manager can start to staff development teams, put training in place, determine which parts to outsource, and start producing budgets and schedules.
- You don't have to satisfy all the information needs of all the stakeholders to the fullest extent.
  - Providing 80 percent of the information goes a long way, and this might be "good enough" so that the stakeholders can do their job.
  - Check with the stakeholder if a subset of information would be sufficient.
- You don't have to complete one view before starting another.
  - People can make progress with overview-level information
  - A breadth-first approach is often the best.

BITS Pilani, Deemed to be University under Section 3 of UGC Act, 1956

innovate achieve lead

BITS Pilani

Pilani|Dubai|Goa|Hyderabad

6. Building the Documentation Package

6. Building the Documentation Package

Documentation package consists of	
Views	Documentation beyond views

Documenting a View

Section 1: The Primary Presentation.

- The *primary presentation* shows the elements and relations of the view.
- The primary presentation should contain the information you wish to convey about the system—in the vocabulary of that view.
- The primary presentation is most often graphical.
  - It might be a diagram you've drawn in an informal notation using a simple drawing tool, or it might be a diagram in a semiformal or formal notation imported from a design or modeling tool that you're using.
  - If your primary presentation is graphical, make sure to include a key that explains the notation.
  - Lack of a key is the most common mistake that we see in documentation in practice.
- Occasionally the primary presentation will be textual, such as a table or a list.
  - If that text is presented according to certain stylistic rules, these rules should be stated or incorporated by reference, as the analog to the graphical notation key.

BITS Pilani, Deemed to be University under Section 3 of UGC Act, 1956

Documenting a View

Section 2: The Element Catalog.

- The *element catalog* details at least those elements depicted in the primary presentation.
  - For instance, if a diagram shows elements A, B, and C, then the element catalog needs to explain what A, B, and C are.
  - If elements or relations relevant to this view were omitted from the primary presentation, they should be introduced and explained in the catalog.
- Parts of the catalog:
  - *Elements and their properties*. This section names each element in the view and lists the properties of that element. Each view introduced in Chapter 1 listed a set of suggested properties associated with that view.
  - *Relations and their properties*. Each view has specific relation types that it depicts among the elements in that view.
  - *Element interfaces*. This section documents element interfaces.
  - *Element behavior*. This section documents element behavior that is not obvious from the primary presentation.

Documenting a View

Section 3: Context Diagram.

- A *context diagram* shows how the system or portion of the system depicted in this view relates to its environment.
- The purpose of a context diagram is to depict the scope of a view.
- Entities in the environment may be humans, other computer systems, or physical objects, such as sensors or controlled devices.

Section 4: Variability Guide.

- A *variability guide* shows how to exercise any variation points that are a part of the architecture shown in this view.

Section 5: Rationale.

- *Rationale* explains why the design reflected in the view came to be.
- The goal of this section is to explain why the design is as it is and to provide a convincing argument that it is sound.
- The choice of a pattern in this view should be justified here by describing the architectural problem that the chosen pattern solves and the rationale for choosing it over another.

View Template

Template for a View

Section 1. Primary Presentation

Section 2. Element Catalog

- Section 2.A. Elements and Their Properties
- Section 2.B. Relations and Their Properties
- Section 2.C. Element Interfaces
- Section 2.D. Element Behavior

Section 3. Context Diagram

Section 4. Variability Guide

Section 5. Rationale

innovate achieve lead

BITS Pilani

Pilani|Dubai|Goa|Hyderabad

Documenting beyond views,

Documenting Information Beyond Views

Document control information.

List the • issuing organization, • the current version number, • date of issue and • status, • a change history, and • the procedure for submitting change requests to the document.	Usually captured in the front matter

Documenting Information Beyond Views

Section 1: Documentation Roadmap. The documentation map tells the reader what information is in the documentation and where to find it.

- *Scope and summary*. Explain the purpose of the document and briefly summarize what is covered.
- *How the documentation is organized*. For each section in the documentation, give a short synopsis of the information that can be found there.
- *View overview*. Describes the views that the architect has included in the package. For each view::
  - The name of the view and what pattern it instantiates, if any.
  - A description of the view's element types, relation types, and property types.
  - A description of language, modeling techniques, or analytical methods used in constructing the view.
- *How stakeholders can use the documentation*.
  - This section shows how various stakeholders might use the documentation to help address their concerns.
  - Include short scenarios, such as "A maintainer wishes to know the units of software that are likely to be changed by a proposed modification."
  - To be compliant with ISO/IEC 42010-2007, you must consider the concerns of at least users, acquirers, developers, and maintainers.

Documenting Information Beyond Views

Section 2: How a View Is Documented.

- Explain the standard organization you're using to document views—either the one described in this chapter or one of your own.

Section 3: System Overview.

- Short prose description of the system's function, its users, and any important background or constraints.
- Provides your readers with a consistent mental model of the system and its purpose.
- This might be a pointer to your project's concept-of-operations document for the system.

Documenting Information Beyond Views

Section 4: Mapping Between Views.

- Helping a reader understand the associations between views will help that reader gain a powerful insight into how the architecture works as a unified conceptual whole.
- The associations between elements across views in an architecture are, in general, many-to-many.
- View-to-view associations can be captured as tables.
  - The table should name the correspondence between the elements across the two views.
  - Examples
    - "is implemented by" for mapping from a component-and-connector view to a module view
    - "implements" for mapping from a module view to a component-and-connector view
    - "included in" for mapping from a decomposition view to a layered view

Documenting Information Beyond Views

Section 5: Rationale.

- Documents the architectural decisions that apply to more than one view.
  - Documentation of background or organizational constraints or major requirements that led to decisions of system-wide import.
  - Decisions about which fundamental architecture patterns are used.

Section 6: Directory.

- Set of reference material that helps readers find more information quickly.
  - Index of terms
  - Glossary
  - Acronym list.

innovate achieve lead

BITS Pilani

Pilani|Dubai|Goa|Hyderabad

7 Documenting behaviours

7. Documenting Behavior

Behavior documentation complements each views by describing how architecture elements in that view interact with each other.

Behavior documentation enables reasoning about

- a system's potential to deadlock
- a system's ability to complete a task in the desired amount of time
- maximum memory consumption
- and more

Behavior has its own section in our view template's element catalog.

BITS Pilani, Deemed to be University under Section 3 of UGC Act, 1956

Notations for Documenting Behavior

Trace-oriented languages

Traces are sequences of activities or interactions that describe the system's response to a specific stimulus when the system is in a specific state.

A trace describes a particular sequence of activities or interactions between structural elements of the system.

Examples

- use cases
- sequence diagrams
- communication diagrams
- activity diagrams
- message sequence charts
- timing diagrams
- Business Process Execution Language

BITS Pilani, Deemed to be University under Section 3 of UGC Act, 1956

Use Case Diagram

From *Documenting Software Architectures: Views and Beyond*, 2nd edition, Addison Wesley, 2010

Use Case Description

Name: Make a basic call

Description: Making a point-to-point connection between two phones.

Primary actors: Caller

Secondary actors: Callee

Flow of events:

The use case starts when a caller places a call via a terminal, such as a cell phone. All terminals to which the call should be routed then begin ringing. When one of the terminals is answered, all others stop ringing and a connection is made between the caller's terminal and the terminal that was answered. When either terminal is disconnected—someone hangs up—the other terminal is also disconnected. The call is now terminated, and the use case is ended.

Exceptional flow of events:

The caller can disconnect, or hang up, before any of the ringing terminals has been answered. If this happens, all ringing terminals stop ringing and are disconnected, ending the use case.

From *Documenting Software Architectures: Views and Beyond*, 2nd edition, Addison Wesley, 2010

Sequence Diagram

Communication Diagram

From *Documenting Software Architectures: Views and Beyond*, 2nd edition, Addison Wesley, 2010

Activity Diagram

From *Documenting Software Architectures: Views and Beyond*, 2nd edition, Addison Wesley, 2010

Notations for Documenting Behavior

Comprehensive languages

- *Comprehensive models* show the complete behavior of structural elements.
- Given this type of documentation, it is possible to infer all possible paths from initial state to final state.
- The state machine formalism represents the behavior of architecture elements because each state is an abstraction of all possible histories that could lead to that state.
- State machine languages allow you to complement a structural description of the elements of the system with constraints on interactions and timed reactions to both internal and environmental stimuli.

BITS Pilani, Deemed to be University under Section 3 of UGC Act, 1956

State Machine

State Machine

From *Documenting Software Architectures: Views and Beyond*, 2nd edition, Addison Wesley, 2010

innovate achieve lead

BITS Pilani

Pilani|Dubai|Goa|Hyderabad

8. Documenting Quality Attributes

8. Documenting Quality Attributes

innovate achieve lead

Where do quality attributes show up in the documentation? There are five major ways:

Rationale that explains the choice of design approach should include a discussion about the quality attribute requirements and tradeoffs.	Architectural elements providing a service often have quality attribute bounds assigned to them, defined in the interface documentation for the elements, or recorded as *properties* that the elements exhibit.	Quality attributes often impart a "language" of things that you would look for. Someone fluent in the "language" of a quality attribute can search for the kinds of architectural elements) put in place to satisfy that quality attribute requirement.	Architecture documentation often contains a *mapping to requirements* that shows how requirements (including quality attribute requirements) are satisfied.	Every quality attribute requirement will have a constituency of stakeholders who want to know that it is going to be satisfied. For these stakeholders, the roadmap tells the stakeholder where in the document to find it.

9. Documenting Architectures That Change Faster Than You Can Document Them

9. Documenting Architectures That Change Faster Than You Can Document Them

An architecture that changes at runtime, or as a result of a high-frequency release-and-deploy cycle, change much faster than the documentation cycle.

Nobody will wait until a new architecture document is produced, reviewed, and released.

In this case:

- *Document what is true about all versions of your system.* Record those invariants as you would for any architecture. This may make your documented architecture more a description of constraints or guidelines that any compliant version of the system must follow.
- *Document the ways the architecture is allowed to change.* This will usually mean adding new components and replacing components with new implementations. The place to do this is called the variability guide.

innovate achieve lead

BITS Pilani

Pilani|Dubai|Goa|Hyderabad

10. Documenting Architecture in an Agile Development Project

10. Documenting Architecture in an Agile Development Project

Adopt a template or standard organization to capture your design decisions.

Plan to document a view if (but only if) it has a strongly identified stakeholder constituency.

Fill in the sections of the template for a view, and for information beyond views, when (and in whatever order) the information becomes available. But only do this if writing down this information will make it easier (or cheaper or make success more likely) for someone downstream doing their job.

Don't worry about creating an architectural design document and then a finer-grained design document. Produce just enough design information to allow you to move on to code.

Don't feel obliged to fill up all sections of the template, and certainly not all at once. Write "N/A" for the sections for which you don't need to record the information (perhaps because you will convey it orally).

Agile teams sometimes make models in brief discussions by the whiteboard. Take a picture and use it as the primary presentation.

Summary

- You must understand the uses to which the writing is to be put and the audience for the writing.
- Architectural documentation serves as a means for communication among various stakeholders, not only up the management chain and down to the developers but also across to peers.
- An architecture is a complicated artifact, best expressed by focusing on views.
- You must choose the views to document, must choose the notation to document these views, and must choose a set of views that is both minimal and adequate.
- You must document not only the structure of the architecture but also the behavior.

BITS Pilani, Deemed to be University under Section 3 of UGC Act, 1956

Thank you……..

Credits

- **Chapter Reference from Text T1: 16, 17, 18**
- Slides have been adapted from Authors Slides Software Architecture in Practice – Third Ed.
    - Len Bass
    - Paul Clements
    - Rick Kazman