BITS Pilani

Pilani | Dubai | Goa | Hyderabad

What is Software Architecture

Harvinder S Jabbal
CSIS, Work Integrated Learning Programs

innovate achieve lead

BITS Pilani

Pilani|Dubai|Goa|Hyderabad

SEZG651/ SSZG653 Software Architectures CS01/1

What is Software Architecture

- What Software Architecture Is and What It Isn't
- Architectural Structures and Views
- Architectural Patterns
- What Makes a "Good" Architecture?

What is Software Architecture?

The software architecture of a system is the set of structures needed to reason about the system, which comprise software elements, relations among them, and properties of both.

Definition

This definition stands in contrast to other definitions that talk about the system's "early" or "major" design decisions.

- Many architectural decisions are made early, but not all are.
- Many decisions are made early that are not architectural.
- It's hard to look at a decision and tell whether or not it's "major."

Structures, on the other hand, are fairly easy to identify in software, and they form a powerful tool for system design.

Architecture Is a Set of Software Structures

- A structure is a set of elements held together by a relation.
- Software systems are composed of many structures, and no single structure holds claim to being the architecture.
- There are three important categories of architectural structures.
  1. Module
  2. Component and Connector
  3. Allocation

Module Structures

- Some structures partition systems into implementation units, which we call modules.
- Modules are assigned specific computational responsibilities, and are the basis of work assignments for programming teams.
- In large projects, these elements (modules) are subdivided for assignment to sub-teams.

Component-and-connector Structures

- Other structures focus on the way the elements interact with each other at runtime to carry out the system's functions.
- We call runtime structures *component-and-connector (C&C) structures*.
- In our use, a component is always a runtime entity.
  - Suppose the system is to be built as a set of services.
  - The services, the infrastructure they interact with, and the synchronization and interaction relations among them form another kind of structure often used to describe a system.
  - These services are made up of (compiled from) the programs in the various implementation units – modules.

Allocation Structures

Allocation structures describe the mapping from software structures to the system's environments

- organizational
- developmental
- installation
- Execution

For example

- Modules are assigned to teams to develop, and assigned to places in a file structure for implementation, integration, and testing.
- Components are deployed onto hardware in order to execute.

Which Structures are Architectural?

- A structure is architectural if it supports reasoning about the system and the system's properties.
- The reasoning should be about an attribute of the system that is important to some stakeholder.

These include

- functionality achieved by the system
- the availability of the system in the face of faults
- the difficulty of making specific changes to the system
- the responsiveness of the system to user requests,
- many others.

Architecture is an Abstraction

- An architecture comprises software elements and how the elements relate to each other.
  - An architecture specifically omits certain information about elements that is not useful for reasoning about the system.
  - It omits information that has no ramifications outside of a single element.
  - An architecture selects certain details and suppresses others.
  - Private details of elements—details having to do solely with internal implementation—are not architectural.
- The architectural abstraction lets us look at the system in terms of its elements, how they are arranged, how they interact, how they are composed, what their properties are that support our system reasoning, and so forth.
- This abstraction is essential to taming the complexity of an architecture.
- We simply cannot, and do not want to, deal with all of the complexity all of the time.

Every System has a Software Architecture

- Every system comprises elements and relations among them to support some type of reasoning.
- But the architecture may not be known to anyone.
  - Perhaps all of the people who designed the system are long gone
  - Perhaps the documentation has vanished (or was never produced)
  - Perhaps the source code has been lost (or was never delivered)
- An architecture can exist independently of its description or specification.
- Documentation is critical.

Architecture Includes Behavior

- The behavior of each element is part of the architecture insofar as that behavior can be used to reason about the system.
- This behavior embodies how elements interact with each other, which is clearly part of the definition of architecture.
- Box-and-line drawings that are passed off as architectures are not architectures at all.
  - When looking at the names of the a reader may well imagine the functionality and behavior of the corresponding elements.
  - But it relies on information that is not present – and could be wrong!
- This does not mean that the exact behavior and performance of every element must be documented in all circumstances.
  - Some aspects of behavior are fine-grained and below the architect's level of concern.
- To the extent that an element's behavior influences another element or influences the acceptability of the system as a whole, this behavior must be considered, and should be documented, as part of the software architecture.

Physiological Structures

- The neurologist, the orthopedist, the hematologist, and the dermatologist all have different views of the structure of a human body.
- Ophthalmologists, cardiologists, and podiatrists concentrate on specific subsystems.
- The kinesiologist and psychiatrist are concerned with different aspects of the entire arrangement's behavior.
- Although these views are pictured differently and have different properties, all are inherently related, interconnected.
- Together they describe the architecture of the human body.
- So it is with software!

Physiological Structures

Structures and Views

- A *view* is a representation of a coherent set of architectural elements, as written by and read by system stakeholders.
  - A view consists of a representation of a set of elements and the relations among them.
- A *structure* is the set of elements itself, as they exist in software or hardware.
- In short, a view is a representation of a structure.
  - For example, a module *structure* is the set of the system's modules and their organization.
  - A module *view* is the representation of that structure, documented according to a template in a chosen notation, and used by some system stakeholders.
- Architects design structures. They document views of those structures.

Module Structures

- Module structures embody decisions as to how the system is to be structured as a set of code or data units that have to be constructed or procured.
- In any module structure, the elements are modules of some kind (perhaps classes, or layers, or merely divisions of functionality, all of which are units of implementation).
- Modules are assigned areas of functional responsibility; there is less emphasis in these structures on how the software manifests at runtime.
- Module structures allow us to answer questions such as these:
  - What is the primary functional responsibility assigned to each module?
  - What other software elements is a module allowed to use?
  - What other software does it actually use and depend on?
  - What modules are related to other modules by generalization or specialization (i.e., inheritance) relationships?

Component-and-connector Structures

- Component-and-connector structures embody decisions as to how the system is to be structured as a set of elements that have runtime behavior (components) and interactions (connectors).
- Elements are runtime comopnents such as services, peers, clients, servers, filters, or many other types of runtime element)
- Connectors are the communication vehicles among components, such as call-return, process synchronization operators, pipes, or others.
- Component-and-connector views help us answer questions such as these:
  - What are the major executing components and how do they interact at runtime?
  - What are the major shared data stores?
  - Which parts of the system are replicated?
  - How does data progress through the system?
  - What parts of the system can run in parallel?
  - Can the system's structure change as it executes and, if so, how?
- Component-and-connector views are crucially important for asking questions about the system's runtime properties such as performance, security, availability, and more.

Allocation structures

- Allocation structures show the relationship between the software elements and elements in one or more external environments in which the software is created and executed.
- Allocation views help us answer questions such as these:
  - What processor does each software element execute on?
  - In what directories or files is each element stored during development, testing, and system building?
  - What is the assignment of each software element to development teams?

Structures Provide Insight

- Structures play such an important role in our perspective on software architecture because of the analytical and engineering power they hold.
- Each structure provides a perspective for reasoning about some of the relevant quality attributes.
- For example:
  - The module structure, which embodies what modules use what other modules, is strongly tied to the ease with which a system can be extended or contracted.
  - The concurrency structure, which embodies parallelism within the system, is strongly tied to the ease with which a system can be made free of deadlock and performance bottlenecks.
  - The deployment structure is strongly tied to the achievement of performance, availability, and security goals.
  - And so forth.

Some Useful Module Structures

Decomposition structure

- The units are modules that are related to each other by the *is-a-submodule-of* relation.
- It shows how modules are decomposed into smaller modules recursively until the modules are small enough to be easily understood.
- Modules often have products (such as interface specifications, code, test plans, etc.) associated with them.
- The decomposition structure determines, to a large degree, the system's modifiability, by assuring that likely changes are localized.
- This structure is often used as the basis for the development project's organization, including the structure of the documentation, and the project's integration and test plans.
- The units in this structure tend to have names that are organization-specific such as "segment" or "subsystem."

Some Useful Module Structures

Uses structure.

- The units here are also modules, perhaps classes.
- The units are related by the *uses* relation, a specialized form of dependency.
- A unit of software *uses* another if the correctness of the first requires the presence of a correctly functioning version (as opposed to a stub) of the second.
- The uses structure is used to engineer systems that can be extended to add functionality, or from which useful functional subsets can be extracted.
- The ability to easily create a subset of a system allows for incremental development.

Some Useful Module Structures

Layer structure

- The modules in this structure are called *layers*.
- A layer is an abstract “virtual machine” that provides a cohesive set of services through a managed interface.
- Layers are *allowed to use* other layers in a strictly managed fashion.
  - In strictly layered systems, a layer is only allowed to use a single other layer.
- This structure is imbues a system with portability, the ability to change the underlying computing platform.

Some Useful Module Structures

- Class (or generalization) structure
- The module units in this structure are called *classes*.
- The relation is *inherits from* or *is an instance of*.
- This view supports reasoning about collections of similar behavior or capability
  - e.g.,the classes that other classes inherit from and parameterized differences
- The class structure allows one to reason about reuse and the incremental addition of functionality.
- If any documentation exists for a project that has followed an object-oriented analysis and design process, it is typically this structure.

Some Useful Module Structures

Data model

- The data model describes the static information structure in terms of data entities and their relationships.
  - For example, in a banking system, entities will typically include Account, Customer, and Loan.
  - Account has several attributes, such as account number, type (savings or checking), status, and current balance.

Some Useful C&C Structures

- The relation in all component-and-connector structures is attachment, showing how the components and the connectors are hooked together.
- The connectors can be familiar constructs such as “invokes.”
- Useful C&C structures include:
  - Service structure
    - The units are services that interoperate with each other by service coordination mechanisms such as SOAP.
    - The service structure helps to engineer a system composed of components that may have been developed anonymously and independently of each other.
  - Concurrency structure
    - This structure helps determine opportunities for parallelism and the locations where resource contention may occur.
    - The units are components
    - The connectors are their communication mechanisms.
    - The components are arranged into logical threads.

Some Useful Allocation Structures

Deployment structure

- The deployment structure shows how software is assigned to hardware processing and communication elements.
- The elements are software elements (usually a process from a C&C view), hardware entities (processors), and communication pathways.
- Relations are allocated-to, showing on which physical units the software elements reside, and migrates-to if the allocation is dynamic.
- This structure can be used to reason about performance, data integrity, security, and availability.
- It is of particular interest in distributed and parallel systems.

Some Useful Allocation Structures

Implementation structure

- This structure shows how software elements (usually modules) are mapped to the file structure(s) in the system's development, integration, or configuration control environments.
- This is critical for the management of development activities and build processes.

Work assignment structure

- This structure assigns responsibility for implementing and integrating the modules to the teams who will carry it out.
- Having a work assignment structure be part of the architecture makes it clear that the decision about who does the work has architectural as well as management implications.
- The architect will know the expertise required on each team.
- This structure will also determine the major communication pathways among the teams: regular teleconferences, wikis, email lists, and so forth.

Relating Structures to Each Other

- Elements of one structure will be related to elements of other structures, and we need to reason about these relations.
  - A module in a decomposition structure may be manifested as one, part of one, or several components in one of the component-and-connector structures.
- In general, mappings between structures are many to many.

Modules vs. Components

FIGURE 1.2 Two views of a client-server system

Architectural Patterns

- Architectural elements can be composed in ways that solve particular problems.
  - The compositions have been found useful over time, and over many different domains
  - They have been documented and disseminated.
  - These compositions of architectural elements, called architectural patterns.
  - Patterns provide packaged strategies for solving some of the problems facing a system.
- An architectural pattern delineates the element types and their forms of interaction used in solving the problem.

- A common module type pattern is the Layered pattern.
  - When the uses relation among software elements is strictly unidirectional, a system of layers emerges.
  - A layer is a coherent set of related functionality.
  - Many variations of this pattern, lessening the structural restriction, occur in practice.

Architectural Patterns

Common component-and-connector type patterns:

- Shared-data (or repository) pattern.
  - This pattern comprises components and connectors that create, store, and access persistent data.
  - The repository usually takes the form of a (commercial) database.
  - The connectors are protocols for managing the data, such as SQL.
- Client-server pattern.
  - The components are the clients and the servers.
  - The connectors are protocols and messages they share among each other to carry out the system's work.

Architectural Patterns

Common allocation patterns:

- Multi-tier pattern
  - Describes how to distribute and allocate the components of a system in distinct subsets of hardware and software, connected by some communication medium.
  - This pattern specializes the generic deployment (software-to-hardware allocation) structure.
- Competence center pattern and platform pattern
  - These patterns specialize a software system's work assignment structure.
  - In competence center, work is allocated to sites depending on the technical or domain expertise located at a site.
  - In platform, one site is tasked with developing reusable core assets of a software product line, and other sites develop applications that use the core assets.

What Makes a "Good" Architecture?

- There is no such thing as an inherently good or bad architecture.
- Architectures are either more or less fit for some purpose
- Architectures can be evaluated but only in the context of specific stated goals.

- There are, however, good rules of thumb.

Structural "Rules of Thumb"

- The architecture should be the product of a single architect or a small group of architects with an identified technical leader.
  - This approach gives the architecture its conceptual integrity and technical consistency.
  - This recommendation holds for Agile and open source projects as well as "traditional" ones.
  - There should be a strong connection between the architect(s) and the development team.
- The architect (or architecture team) should base the architecture on a prioritized list of well-specified quality attribute requirements.
- The architecture should be documented using views. The views should address the concerns of the most important stakeholders in support of the project timeline.
- The architecture should be evaluated for its ability to deliver the system's important quality attributes.
  - This should occur early in the life cycle and repeated as appropriate.
- The architecture should lend itself to incremental implementation,
  - Create a "skeletal" system in which the communication paths are exercised but which at first has minimal functionality.

Structural "Rules of Thumb"

- The architecture should feature well-defined modules whose functional responsibilities are assigned on the principles of information hiding and separation of concerns.
  - The information-hiding modules should encapsulate things likely to change
  - Each module should have a well-defined interface that encapsulates or "hides" the changeable aspects from other software
- Unless your requirements are unprecedented your quality attributes should be achieved using well-known architectural patterns and tactics specific to each attribute.
- The architecture should never depend on a particular version of a commercial product or tool. If it must, it should be structured so that changing to a different version is straightforward and inexpensive.
- Modules that produce data should be separate from modules that consume data.
  - This tends to increase modifiability
  - Changes are frequently confined to either the production or the consumption side of data.

Structural "Rules of Thumb"

- Don't expect a one-to-one correspondence between modules and components.
- Every process should be written so that its assignment to a specific processor can be easily changed, perhaps even at runtime.
- The architecture should feature a small number of ways for components to interact.
  - The system should do the same things in the same way throughout.
  - This will aid in understandability, reduce development time, increase reliability, and enhance modifiability.
- The architecture should contain a specific (and small) set of resource contention areas, the resolution of which is clearly specified and maintained.

Summary

- The software architecture of a system is the set of structures needed to reason about the system, which comprise software elements, relations among them, and properties of both.
- A structure is a set of elements and the relations among them.
- A view is a representation of a coherent set of architectural elements. A view is a representation of one or more structures.

Summary

- There are three categories of structures:
  - Module structures show how a system is to be structured as a set of code or data units that have to be constructed or procured.
  - Component-and-connector structures show how the system is to be structured as a set of elements that have runtime behavior (components) and interactions (connectors).
  - Allocation structures show how the system will relate to nonsoftware structures in its environment (such as CPUs, file systems, networks, development teams, etc.).
- Structures represent the primary engineering leverage points of an architecture.
- Every system has a software architecture, but this architecture may be documented and disseminated, or it may not be.
- There is no such thing as an inherently good or bad architecture. Architectures are either more or less fit for some purpose.

Why Software Architecture is Important?

BITS Pilani
Pilani | Dubai | Goa | Hyderabad

Harvinder S Jabbal
CSIS, Work Integrated Learning Programs

innovate achieve lead

BITS Pilani

Pilani|Dubai|Goa|Hyderabad

SEZG651/ SSZG653 Software Architectures CS01/02A

Why is Software Architecture Important?

1. An architecture will inhibit or enable a system's driving quality attributes.
2. The decisions made in an architecture allow you to reason about and manage change as the system evolves.
3. The analysis of an architecture enables early prediction of a system's qualities.
4. A documented architecture enhances communication among stakeholders.
5. The architecture is a carrier of the earliest and hence most fundamental, hardest-to-change design decisions.
6. An architecture defines a set of constraints on subsequent implementation.
7. The architecture dictates the structure of an organization, or vice versa.
8. An architecture can provide the basis for evolutionary prototyping.
9. An architecture is the key artifact that allows the architect and project manager to reason about cost and schedule.
10. An architecture can be created as a transferable, reusable model that form the heart of a product line.
11. Architecture-based development focuses attention on the assembly of components, rather than simply on their creation.
12. By restricting design alternatives, architecture channels the creativity of developers, reducing design and system complexity.
13. An architecture can be the foundation for training a new team member.

Inhibiting or Enabling a System's Quality Attributes

Whether a system will be able to exhibit its desired (or required) quality attributes is substantially determined by its architecture.

This is the most important message of this course!

- Performance: You must manage the time-based behavior of elements, their use of shared resources, and the frequency and volume of inter-element communication.
- Modifiability: Assign responsibilities to elements so that the majority of changes to the system will affect a small number of those elements.
- Security: Manage and protect inter-element communication and control which elements are allowed to access which information; you may also need to introduce specialized elements (such as an authorization mechanism).
- Scalability: Localize the use of resources to facilitate introduction of higher-capacity replacements, and you must avoid hardcoding in resource assumptions or limits.
- Incremental subset delivery: Manage inter-component usage.
- Reusability: Restrict inter-element coupling, so that when you extract an element, it does not come out with too many attachments to its current environment.

Reasoning About and Managing Change

- About 80 percent of a typical software system's total cost occurs after initial deployment
  - accommodate new features
  - adapt to new environments,
  - fix bugs, and so forth.
- Every architecture partitions possible changes into three categories
  - A *local* change can be accomplished by modifying a single element.
  - A *nonlocal* change requires multiple element modifications but leaves the underlying architectural approach intact.
  - An *architectural* change affects the fundamental ways in which the elements interact with each other and will probably require changes all over the system.
- Obviously, local changes are the most desirable
- A good architecture is one in which the most common changes are local, and hence easy to make.

Predicting System Qualities

- If we know that certain kinds of architectural decisions lead to certain quality attributes in a system, we can make those decisions and rightly expect to be rewarded with the associated quality attributes.
- When we examine an architecture we can look to see if those decisions have been made, and confidently predict that the architecture will exhibit the associated qualities.
- The earlier you can find a problem in your design, the cheaper, easier, and less disruptive it will be to fix.

Enhancing Communication Among Stakeholders

- Software architecture represents a common abstraction of a system that most, if not all, of the system's stakeholders can use as a basis for creating mutual understanding, negotiating, forming consensus, and communicating with each other.
- The architecture—or at least parts of it—is sufficiently abstract that most nontechnical people can understand it adequately.
- Each stakeholder of a software system—customer, user, project manager, coder, tester, and so on—is concerned with different characteristics of the system that are affected by its architecture. For example:
  - The user is concerned that the system is fast, reliable, and available when needed.
  - The customer is concerned that the architecture can be implemented on schedule and according to budget.
  - The manager is worried (in addition to concerns about cost and schedule) that the architecture will allow teams to work largely independently, interacting in disciplined and controlled ways.
  - The architect is worried about strategies to achieve all of those goals.
- Architecture provides a common language in which different concerns can be expressed, negotiated, and resolved at a level that is intellectually manageable even for large, complex systems.

Earliest Design Decisions

- Software architecture is a manifestation of the earliest design decisions about a system.
- These early bindings carry enormous weight with respect to the system's remaining development, its deployment, and its maintenance life.
- Each decision constrains the many decisions that follow.
- What are these early design decisions embodied by software architecture?
  - Will the system run on one processor or be distributed across multiple processors?
  - Will the software be layered? If so, how many layers will there be? What will each one do?
  - Will components communicate synchronously or asynchronously? Will they interact by transferring control or data or both?
  - Will the system depend on specific features of the operating system or hardware?
  - Will the information that flows through the system be encrypted or not?
  - What communication protocol will we choose?
- Imagine the nightmare of having to change any of these decisions.

Defining Constraints on an Implementation

- An implementation exhibits an architecture if it conforms to the design decisions prescribed by the architecture.
  - The implementation must be implemented as the set of prescribed elements
  - These elements must interact with each other in the prescribed fashion
  - Each element must fulfill its responsibility to the other elements as dictated by the architecture.
- Each of these prescriptions is a constraint on the implementer.
- Element builders may not be aware of the architectural tradeoffs—the architecture (or architect) simply constrains them in such a way as to meet the tradeoffs.
  - Example: an architect assigns performance budget to the pieces of software involved in some larger piece of functionality.
  - If each software unit stays within its budget, the overall transaction will meet its performance requirement.
  - Implementers of each of the constituent pieces may not know the overall budget, only their own.

Influencing the Organizational Structure

- Architecture prescribes the structure of the system being developed.
- That structure becomes engraved in the structure of the development project (and sometimes the structure of the entire organization).
- The architecture is typically used as the basis for the work-breakdown structure.
- The work-breakdown structure in turn dictates
  - units of planning, scheduling, and budget
  - interteam communication channels
  - configuration control and file-system organization
  - integration and test plans and procedures;
  - much more
- The maintenance activity will also reflect the software structure, with teams formed to maintain specific structural elements from the architecture.
- If these responsibilities have been formalized in a contractual relationship, changing responsibilities could become expensive or even litigious.

Enabling Evolutionary Prototyping

- Once an architecture has been defined, it can be analyzed and prototyped as a skeletal system.
  - A skeletal system is one in which at least some of the infrastructure— how the elements initialize, communicate, share data, access resources, report errors, log activity, and so forth—is built before much of the system's functionality has been created.
- This approach aids the development process because the system is executable early in the product's life cycle.
- The fidelity of the system increases as stubs are instantiated, or prototype parts are replaced with complete versions of these parts of the software.
- This approach allows potential performance problems to be identified early in the product's life cycle.
- These benefits reduce the potential risk in the project.

Improving Cost and Schedule Estimates

- One of the duties of an architect is to help the project manager create cost and schedule estimates early in the project life cycle.
- Top-down estimates are useful for setting goals and apportioning budgets.
- Cost estimations that are based on a bottom-up understanding of the system's pieces are typically more accurate than those that are based purely on top-down system knowledge.
  - Each team or individual responsible for a work item will be able to make more-accurate estimates for their piece than a project manager and will feel more ownership in making the estimates come true.
- The best cost and schedule estimates will typically emerge from a consensus between the top-down estimates (created by the architect and project manager) and the bottom-up estimates (created by the developers).

Transferable, Reusable Model

- Reuse of architectures provides tremendous leverage for systems with similar requirements.
  - Not only can code be reused, but so can the requirements that led to the architecture in the first place, as well as the experience and infrastructure gained in building the reused architecture.
  - When architectural decisions can be reused across multiple systems, all of the early-decision consequences are also transferred.
- A software product line or family is a set of software systems that are all built using the same set of reusable assets.
  - Chief among these assets is the architecture that was designed to handle the needs of the entire family.
  - The architecture defines what is fixed for all members of the product line and what is variable.
  - The architecture for a product line becomes a capital investment by the organization.

Using Independently Developed Components

- Architecture-based development often focuses on components that are likely to have been developed separately, even independently, from each other.
- The architecture defines the elements that can be incorporated into the system.
- Commercial off-the-shelf components, open source software, publicly available apps, and networked services are example of interchangeable software components.
- The payoff can be
  - Decreased time to market
  - Increased reliability (widely used software should have its bugs ironed out already)
  - Lower cost (the software supplier can amortize development cost across their customer base)
  - Flexibility (if the component you want to buy is not terribly specialpurpose, it's likely to be available from several sources, thus increasing your buying leverage)

Restricting Design Vocabulary

- As useful architectural patterns are collected, we see the benefit in voluntarily restricting ourselves to a relatively small number of choices of elements and their interactions.
  - We minimize the design complexity of the system we are building.
  - Enhanced reuse
  - More regular and simpler designs that are more easily understood and communicated
  - More capable analysis
  - Shorter selection time
  - Greater interoperability.
- Architectural patterns guide the architect and focus the architect on the quality attributes of interest in large part by restricting the vocabulary of design.
  - Properties of software design follow from the choice of an architectural pattern.

Basis for Training

- The architecture can serve as the first introduction to the system for new project members.
- Module views are excellent for showing someone the structure of a project
  - Who does what, which teams are assigned to which parts of the system, and so forth.
- Component-and-connector views are excellent for explaining how the system is expected to work and accomplish its job.

Summary

1. An architecture will inhibit or enable a system's driving quality attributes.
2. The decisions made in an architecture allow you to reason about and manage change as the system evolves.
3. The analysis of an architecture enables early prediction of a system's qualities.
4. A documented architecture enhances communication among stakeholders.
5. The architecture is a carrier of the earliest and hence most fundamental, hardest-to-change design decisions.
6. An architecture defines a set of constraints on subsequent implementation.
7. The architecture dictates the structure of an organization, or vice versa.

Summary

8. An architecture can provide the basis for evolutionary prototyping.
9. An architecture is the key artifact that allows the architect and project manager to reason about cost and schedule.
10. An architecture can be created as a transferable, reusable model that form the heart of a product line.
11. Architecture-based development focuses attention on the assembly of components, rather than simply on their creation.
12. By restricting design alternatives, architecture channels the creativity of developers, reducing design and system complexity.
13. An architecture can be the foundation for training a new team member.

BITS Pilani

Pilani | Dubai | Goa | Hyderabad

The Many Contexts of Software Architecture

Harvinder S Jabbal
CSIS, Work Integrated Learning Programs

innovate achieve lead

BITS Pilani

Pilani|Dubai|Goa|Hyderabad

SEZG651/ SSZG653 Software Architectures CS01/02B

Chapter Outline

Architecture in a Technical Context

Architecture in a Project Life-Cycle Context

Architecture in a Business Context

Architecture in a Professional Context

Stakeholders

How Is Architecture Influenced?

What Do Architectures Influence?

Summary

Contexts of Software Architecture

- Sometimes we consider software architecture the center of the universe!
- Here, though, we put it in its place relative to four contexts:
  - Technical. What technical role does the software architecture play in the system or systems of which it's a part?
  - Project life cycle. How does a software architecture relate to the other phases of a software development life cycle?
  - Business. How does the presence of a software architecture affect an organization's business environment?
  - Professional. What is the role of a software architect in an organization or a development project?

Technical Context

- The most important technical context factor is the set of quality attributes that the architecture can help to achieve.
- The architecture's current technical environment is also an important factor.
  - Standard industry practices
  - Software engineering techniques prevalent in the architect's professional community.
- Today's information systems are web-based, object-oriented, service-oriented, mobility-aware, cloud-based, social-networking-friendly.
  - It wasn't always so.
  - It won't be so ten years from now.

Project Life-cycle Context

- Software development processes are standard approaches for developing software systems.
- They impose a discipline on software engineers and, more important, teams of software engineers.
- They tell the members of the team what to do next.
- There are four dominant software development processes:
  - Waterfall
  - Iterative
  - Agile
  - Model-driven development

Architecture Activities

- All of these processes include design among their obligations.
- Architecture is a special kind of design, so architecture finds a home in each one.
- No matter the software development process, there are activities involved in creating a software architecture, using that architecture to realize a complete design, and then implementing or managing the evolution of a target system or application:
  - 1. Making a business case for the system
  - 2. Understanding the architecturally significant requirements
  - 3. Creating or selecting the architecture
  - 4. Documenting and communicating the architecture
  - 5. Analyzing or evaluating the architecture
  - 6. Implementing and testing the system based on the architecture
  - 7. Ensuring that the implementation conforms to the architecture

Business Context

Architectures and systems are not constructed frivolously.

They serve some business purposes.

These purposes may change over time.

Architecture and Business Goals

- Systems are created to satisfy the business goals of one or more organizations.
  - Development organizations want to make a profit, or capture market, or stay in business, or help their customers do their jobs better, or keep their staff gainfully employed, or make their stockholders happy, or a little bit of each.
  - Customers have their own goals for acquiring a system, usually involving some aspect of making their lives easier or more productive. Other organizations involved in a project's life cycle, such as subcontractors or government regulatory agencies, have their own goals dealing with the system.
- Architects need to understand who the vested organizations are and what their goals are. Many of these goals will have a profound influence on the architecture.

Architecture and Business Goals

- Every quality attribute—such as a user-visible response time or platform flexibility or ironclad security or any of a dozen other needs—should originate from some higher purpose that can be described in terms of added value.
  - “Why do you want this system to have a really fast response time?”
  - This differentiate the product from its competition and let the developing organization capture market share.
- Some business goals will not show up in the form of requirements.
- Still other business goals have no effect on the architecture whatsoever.
  - A business goal to lower costs might be realized by asking employees to work from home, or turn the office thermostats down in the winter, or using less paper in the printers.

Architecture and 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.

Professional Context

You will perform many *duties* beyond directly producing an architecture.

- You will need to be involved in supporting management and dealing with customers.

Architects need more than just technical *skills*.

- Architects need to explain to one stakeholder or another the chosen priorities of different properties, and why particular stakeholders are not having all of their expectations fulfilled.
- Architects need diplomatic, negotiation, and communication skills.
- Architects need the ability to communicate ideas clearly
- You will need to manage a diverse workload and be able to switch contexts frequently.
- You will need to be a leader in the eyes of developers and management.

Architects need up-to-date *knowledge.*

- You will need to know about (for example) patterns, or database platforms, or web services standards.
- You will need to know business considerations.

Stakeholders

- A stakeholder is anyone who has a stake in the success of the system
- Stakeholders typically have different specific concerns that they wish the system to guarantee or optimize.
- You will need to know and understand the nature, source, and priority of constraints on the project as early as possible. Therefore, you must identify and actively engage the stakeholders to solicit their needs and expectations.
- Early engagement of stakeholders allows you to understand the constraints of the task, manage expectations, negotiate priorities, and make tradeoffs.

Stakeholders

- Know your stakeholders!
- Talk to them, engage them, listen to them, and put yourself in their shoes.

- See Table for a list of example stakeholders and their interests and concerns.

Stakeholders

How is Architecture Influenced?

- Requirements influence the architecture, of course.
- But the requirements specification only begins to tell the story.
- A software architecture is a result of business and social influences, as well as technical ones.
- The existence of an architecture in turn affects the technical, business, and social environments that subsequently influence future architectures.
- In particular, each of the contexts for architecture plays a role in influencing an architect and the architecture.

How is Architecture Influenced?

FIGURE 3.4 Influences on the architect

What Do Architectures Influence?

Technical context

- The architecture can affect stakeholder requirements for the next system
- It gives the customer the opportunity to receive a system (based on the same architecture) in a more reliable, timely, and economical manner than if built from scratch.
- A customer may in fact be willing to relax some of their requirements to gain these economies.
- Shrinkwrapped software has clearly affected people's requirements by providing solutions that are not tailored to any individual's precise needs but are instead inexpensive and (in the best of all possible worlds) of high quality.

What Do Architectures Influence?

Project context

- The architecture affects the structure of the developing organization.
- An architecture prescribes the units of software that must be implemented (or otherwise obtained) and integrated to form the system.
- These units are the basis for the development project's structure.
- Teams are formed for individual software units; and the development, test, and integration activities all revolve around the units.
- Teams become embedded in the organization's structure.

What Do Architectures Influence?

Business context

- The architecture can affect the business goals of the developing organization.
- A successful system built from an architecture can enable a company to establish a foothold in a particular market segment.
- The architecture can provide opportunities for the efficient production and deployment of similar systems, and the organization may adjust its goals to take advantage of its newfound expertise to plumb the market.

What Do Architectures Influence?

Professional context

- The process of system building will affect the architect's experience with subsequent.
- A system that was successfully built around a particular technical approach will make the architect more inclined to build systems using the same approach in the future.
- Architectures that fail are less likely to be chosen for future projects.

Architecture Influence Cycle

FIGURE 3.5 Architecture Influence Cycle

Summary

- Architectures exist in four different contexts.
  - Technical. The technical context includes the achievement of quality attribute requirements.
  - Project life cycle. Regardless of the software development methodology you use, you must perform specific activities.
  - Business. The system created from the architecture must satisfy the business goals of a wide variety of stakeholders.
  - Professional. You must have certain skills and knowledge to be an architect, and there are certain duties that you must perform as an architect.
- An architecture has influences that lead to its creation, and its existence has an impact on the architect, the organization, and, potentially, the industry.
- We call this cycle the Architecture Influence Cycle.

BITS Pilani

Pilani | Dubai | Goa | Hyderabad

L01.
Understanding Quality Attributes

Harvinder S Jabbal
CSIS, Work Integrated Learning Programs

innovate achieve lead

BITS Pilani

Pilani|Dubai|Goa|Hyderabad

SEZG651/ SSZG653 Software Architectures Module 2-CS 02

Understanding Quality Attributes

- Architecture and Requirements
- Functionality
- Quality Attribute Considerations
- Specifying Quality Attribute Requirements
- Achieving Quality Attributes through Tactics
- Guiding Quality Design Decisions

Architecture and Requirements

System requirements can be categorized as:

- Functional requirements. These requirements state what the system must do, how it must behave or react to run-time stimuli.
- Quality attribute requirements. These requirements annotate (qualify) functional requirements. Qualification might be how fast the function must be performed, how resilient it must be to erroneous input, how easy the function is to learn, etc.
- Constraints. A constraint is a design decision with zero degrees of freedom. That is, it's a design decision that has already been made for you.

Architectural Description

ISO/IEC/IEEE 42010: 2011

ISO/IEC/IEEE 42010 - *International Standard for Systems and Software Engineering – Architectural Description*, 2011

Functionality

- Functionality is the ability of the system to do the work for which it was intended.
- Functionality has a strange relationship to architecture:
  - functionality does not determine architecture; given a set of required functionality, there is no end to the architectures you could create to satisfy that functionality
  - functionality and quality attributes are orthogonal

Quality Attribute Considerations

- If a functional requirement is "when the user presses the green button the Options dialog appears”:
  - a performance QA annotation might describe how quickly the dialog will appear;
  - an availability QA annotation might describe how often this function will fail, and how quickly it will be repaired;
  - a usability QA annotation might describe how easy it is to learn this function.

Quality Attribute Considerations

There are three problems with previous discussions of quality attributes:

1. The definitions provided for an attribute are not testable. It is meaningless to say that a system will be “modifiable”.
2. Endless time is wasted on arguing over which quality a concern belongs to. Is a system failure due to a denial of service attack an aspect of availability, performance, security, or usability?
3. Each attribute community has developed its own vocabulary.

Quality Attribute Considerations

- A solution to the first two of these problems (untestable definitions and overlapping concerns) is to use *quality attribute scenarios* as a means of characterizing quality attributes.
- A solution to the third problem is to provide a discussion of each attribute—concentrating on its underlying concerns—to illustrate the concepts that are fundamental to that attribute community.

Specifying Quality Attribute Requirements

- We use a common form to specify all quality attribute requirements as scenarios.
- Our representation of quality attribute scenarios has these parts:
  1. **Stimulus**
  2. **Stimulus source**
  3. **Response**
  4. **Response measure**
  5. **Environment**
  6. **Artifact**

Specifying Quality Attribute Requirements

1. **Source of stimulus**. This is some entity (a human, a computer system, or any other actuator) that generated the stimulus.
2. **Stimulus**. The stimulus is a condition that requires a response when it arrives at a system.
3. **Environment**. The stimulus occurs under certain conditions. The system may be in an overload condition or in normal operation, or some other relevant state. For many systems, “normal” operation can refer to one of a number of modes.
4. **Artifact**. Some artifact is stimulated. This may be a collection of systems, the whole system, or some piece or pieces of it.
5. **Response**. The response is the activity undertaken as the result of the arrival of the stimulus.
6. **Response measure**. When the response occurs, it should be measurable in some fashion so that the requirement can be tested.

Specifying Quality Attribute Requirements

- We distinguish *general* quality attribute scenarios (“general scenarios”)—those that are system independent and can, potentially, pertain to any system—from *concrete* quality attribute scenarios (concrete scenarios)—those that are specific to the particular system under consideration.

Specifying Quality Attribute Requirements

Example general scenario for availability:

Achieving Quality Attributes Through Tactics

- There are a collection of primitive design techniques that an architect can use to achieve a quality attribute response.
- We call these architectural design primitive *tactics*.
- Tactics, like design patterns, are techniques that architects have been using for years. We do not *invent* tactics; we simply capture what architects do in practice.

Achieving Quality Attributes Through Tactics

Why do we do this? There are three reasons:

1. Design patterns are complex; they are a bundle of design decisions. But patterns are often difficult to apply as is; architects need to modify and adapt them. By understanding tactics, an architect can assess the options for augmenting an existing pattern to achieve a quality attribute goal.
2. If no pattern exists to realize the architect's design goal, tactics allow the architect to construct a design fragment from "first principles".
3. By cataloguing tactics, we make design more systematic. You frequently will have a choice of multiple tactics to improve a particular quality attribute. The choice of which tactic to use depends on factors such as tradeoffs among other quality attributes and the cost to implement.

Guiding Quality Design Decisions

- Architecture design is a systematic approach to making design decisions.
- We categorize the design decisions that an architect needs to make as follows:
  1. Allocation of responsibilities
  2. Coordination model
  3. Data model
  4. Management of resources
  5. Mapping among architectural elements
  6. Binding time decisions
  7. Choice of technology

Allocation of Responsibilities

Decisions involving allocation of responsibilities include:

- identifying the important responsibilities including basic system functions, architectural infrastructure, and satisfaction of quality attributes.
- determining how these responsibilities are allocated to non-runtime and runtime elements (namely, modules, components, and connectors).

Coordination Model

Decisions about the coordination model include:

- identify the elements of the system that must coordinate, or are prohibited from coordinating
- determining the properties of the coordination, such as timeliness, currency, completeness, correctness, and consistency
- choosing the communication mechanisms that realize those properties. Important properties of the communication mechanisms include stateful vs. stateless, synchronous vs. asynchronous, guaranteed vs. non-guaranteed delivery, and performance-related properties such as throughput and latency

Data Model

Decisions about the data model include:

- choosing the major data abstractions, their operations, and their properties. This includes determining how the data items are created, initialized, accessed, persisted, manipulated, translated, and destroyed.
- metadata needed for consistent interpretation of the data
- organization of the data. This includes determining whether the data is going to be kept in a relational data base, a collection of objects or both

Management of Resources

Decisions for management of resources include:

- identifying the resources that must be managed and determining the limits for each
- determining which system element(s) manage each resource
- determining how resources are shared and the arbitration strategies employed when there is contention
- determining the impact of saturation on different resources.

Mapping Among Architectural Elements

Useful mappings include:

- the mapping of modules and runtime elements to each other—that is, the runtime elements that are created from each module; the modules that contain the code for each runtime element
- the assignment of runtime elements to processors
- the assignment of items in the data model to data stores
- the mapping of modules and runtime elements to units of delivery

Binding Time

- The decisions in the other categories have an associated binding time decision. Examples of such binding time decisions include:
  - For allocation of responsibilities, you can have build-time selection of modules via a parameterized build script.
  - For choice of coordination model, you can design run-time negotiation of protocols.
  - For resource management you can design a system to accept new peripheral devices plugged in at run-time.
  - For choice of technology, you can build an app-store for a smart phone that automatically downloads the appropriate version of the app.

Choice of Technology

Choice of technology decisions involve:

- deciding which technologies are available to realize the decisions made in the other categories
- determining whether the tools to support this technology (IDEs, simulators, testing tools, etc.) are adequate
- determining the extent of internal familiarity and external support for the technology (such as courses, tutorials, examples, availability of contractors)
- determining the side effects of choosing a technology such as a required coordination model or constrained resource management opportunities
- determining whether a new technology is compatible with the existing technology stack

Summary

Requirements for a system come in three categories.

1. Functional. These requirements are satisfied by including an appropriate set of responsibilities within the design.
2. Quality attribute. These requirements are satisfied by the structures and behaviors of the architecture.
3. Constraints. A constraint is a design decision that's already been made.

To express a quality attribute requirement we use a quality attribute scenario. The parts of the scenario are:

1. Source of stimulus.
2. Stimulus
3. Environment.
4. Artifact.
5. Response.
6. Response measure.

Summary

- An architectural tactic is a design decision that affects a quality attribute response. The focus of a tactic is on a single quality attribute response.
- Architectural patterns can be seen as “packages” of tactics.
- The seven categories of architectural design decisions are:
  1. Allocation of responsibilities
  2. Coordination model
  3. Data model
  4. Management of resources
  5. Mapping among architectural elements
  6. Binding time decisions
  7. Choice of technology

BITS Pilani

Pilani | Dubai | Goa | Hyderabad

L02
Availability

Harvinder S Jabbal
CSIS, Work Integrated Learning Programs

innovate achieve lead

BITS Pilani

Pilani|Dubai|Goa|Hyderabad

SEZG651/ SSZG653 Software Architectures Module 2-CS 02

Outline

What is Availability?

Availability General Scenario

Tactics for Availability

A Design Checklist for Availability

Summary

Credits

These Slides are based on

- Software Architecture in Practice by
  - Len Bass, Paul Clement and Rick Kazman
  - Pearson © 2013

What is Availability?

- Availability refers to a property of software that it is there and ready to carry out its task when you need it to be.
- This is a broad perspective and encompasses what is normally called reliability.
- Availability builds on reliability by adding the notion of recovery (repair).
- Fundamentally, availability is about minimizing service outage time by mitigating faults.

Availability General Scenario

Portion of Scenario	Possible Values
Source	Internal/external: people, hardware, software, physical infrastructure, physical environment
Stimulus	Fault: omission, crash, incorrect timing, incorrect response
Artifact	System's processors, communication channels, persistent storage, processes
Environment	Normal operation, startup, shutdown, repair mode, degraded operation, overloaded operation
Response	Prevent the fault from becoming a failure Detect the fault: • log the fault • notify appropriate entities (people or systems) Recover from the fault • disable source of events causing the fault • be temporarily unavailable while repair is being effected • fix or mask the fault/failure or contain the damage it causes • operate in a degraded mode while repair is being effected
Response Measure	Time or time interval when the system must be available Availability percentage (e.g. 99.999%) Time to detect the fault Time to repair the fault Time or time interval in which system can be in degraded mode Proportion (e.g., 99%) or rate (e.g., up to 100 per second) of a certain class of faults that the system prevents, or handles without failing

Sample Concrete Availability Scenario

The heartbeat monitor determines that the server is nonresponsive during normal operations. The system informs the operator and continues to operate with no downtime.

Goal of Availability Tactics

- A failure occurs when the system no longer delivers a service consistent with its specification
  - this failure is observable by the system's actors.
- A fault (or combination of faults) has the potential to cause a failure.
- Availability tactics enable a system to endure faults so that services remain compliant with their specifications.
- The tactics keep faults from becoming failures or at least bound the effects of the fault and make repair possible.

Goal of Availability Tactics

Availability Tactics

Detect Faults

- Ping/echo: asynchronous request/response message pair exchanged between nodes, used to determine reachability and the round-trip delay through the associated network path.
- Monitor: a component used to monitor the state of health of other parts of the system. A system monitor can detect failure or congestion in the network or other shared resources, such as from a denial-of-service attack.
- Heartbeat: a periodic message exchange between a system monitor and a process being monitored.

Detect Faults

- Timestamp: used to detect incorrect sequences of events, primarily in distributed message-passing systems.
- Sanity Checking: checks the validity or reasonableness of a component's operations or outputs; typically based on a knowledge of the internal design, the state of the system, or the nature of the information under scrutiny.
- Condition Monitoring: checking conditions in a process or device, or validating assumptions made during the design.

Detect Faults

- Voting: to check that replicated components are producing the same results. Comes in various flavors: replication, functional redundancy, analytic redundancy.
- Exception Detection: detection of a system condition that alters the normal flow of execution, e.g. system exception, parameter fence, parameter typing, timeout.
- Self-test: procedure for a component to test itself for correct operation.

Recover from Faults (Preparation & Repair)

- Active Redundancy (hot spare): all nodes in a *protection group* receive and process identical inputs in parallel, allowing redundant spare(s) to maintain synchronous state with the active node(s).
  - A protection group is a group of nodes where one or more nodes are “active,” with the remainder serving as redundant spares.
- Passive Redundancy (warm spare): only the active members of the protection group process input traffic; one of their duties is to provide the redundant spare(s) with periodic state updates.
- Spare (cold spare): redundant spares of a protection group remain out of service until a fail-over occurs, at which point a power-on-reset procedure is initiated on the redundant spare prior to its being placed in service.

Recover from Faults (Preparation & Repair)

- Exception Handling: dealing with the exception by reporting it or handling it, potentially masking the fault by correcting the cause of the exception and retrying.
- Rollback: revert to a previous known good state, referred to as the “rollback line”.
- Software Upgrade: in-service upgrades to executable code images in a non-service-affecting manner.

Recover from Faults (Preparation & Repair)

- Retry: where a failure is transient retrying the operation may lead to success.
- Ignore Faulty Behavior: ignoring messages sent from a source when it is determined that those messages are spurious.
- Degradation: maintains the most critical system functions in the presence of component failures, dropping less critical functions.
- Reconfiguration: reassigning responsibilities to the resources left functioning, while maintaining as much functionality as possible.

Recover from Faults (Reintroduction)

- Shadow: operating a previously failed or in-service upgraded component in a “shadow mode” for a predefined time prior to reverting the component back to an active role.
- State Resynchronization: partner to active redundancy and passive redundancy where state information is sent from active to standby components.
- Escalating Restart: recover from faults by varying the granularity of the component(s) restarted and minimizing the level of service affected.
- Non-stop Forwarding: functionality is split into supervisory and data. If a supervisor fails, a router continues forwarding packets along known routes while protocol information is recovered and validated.

Prevent Faults

- Removal From Service: temporarily placing a system component in an out-of-service state for the purpose of mitigating potential system failures
- Transactions: bundling state updates so that asynchronous messages exchanged between distributed components are *atomic*, *consistent*, *isolated*, and *durable*.
- Predictive Model: monitor the state of health of a process to ensure that the system is operating within nominal parameters; take corrective action when conditions are detected that are predictive of likely future faults.

Prevent Faults

- Exception Prevention: preventing system exceptions from occurring by masking a fault, or preventing it via smart pointers, abstract data types, wrappers.
- Increase Competence Set: designing a component to handle more cases—faults—as part of its normal operation.

Design Checklist for Availability

Allocation of Responsibilities

- Determine the system responsibilities that need to be highly available. Ensure that additional responsibilities have been allocated to detect an omission, crash, incorrect timing, or incorrect response.
- Ensure that there are responsibilities to:
  - log the fault
  - notify appropriate entities (people or systems)
  - disable source of events causing the fault
  - be temporarily unavailable
  - fix or mask the fault/failure
  - operate in a degraded mode

BITS Pilani, Deemed to be University under Section 3 of UGC Act, 1956

Design Checklist for Availability

Coordination Model

- Determine the system responsibilities that need to be highly available. With respect to those responsibilities
- Ensure that coordination mechanisms can detect an omission, crash, incorrect timing, or incorrect response. Consider, e.g., whether guaranteed delivery is necessary. Will the coordination work under degraded communication?
- Ensure that coordination mechanisms enable the logging of the fault, notification of appropriate entities, disabling of the source of the events causing the fault, fixing or masking the fault, or operating in a degraded mode
- Ensure that the coordination model supports the replacement of the artifacts (processors, communications channels, persistent storage, and processes). E.g., does replacement of a server allow the system to continue to operate?
- Determine if the coordination will work under conditions of degraded communication, at startup/shutdown, in repair mode, or under overloaded operation. E.g., how much lost information can the coordination model withstand and with what consequences?

Design Checklist for Availability

Data Model

- Determine which portions of the system need to be highly available. Within those portions, determine which data abstractions could cause a fault of omission, a crash, incorrect timing behavior, or an incorrect response.
- For those data abstractions, operations, and properties, ensure that they can be disabled, be temporarily unavailable, or be fixed or masked in the event of a fault.
- E.g., ensure that write requests are cached if a server is temporarily unavailable and performed when the server is returned to service.

Design Checklist for Availability

Mapping Among Architectural Elements

- Determine which artifacts (processors, communication channels, storage, processes) may produce a fault: omission, crash, incorrect timing, or incorrect response.
- Ensure that the mapping (or re-mapping) of architectural elements is flexible enough to permit the recovery from the fault. This may involve a consideration of
- which processes on failed processors need to be re-assigned at runtime
- which processors, data stores, or communication channels can be activated or re-assigned at runtime
- how data on failed processors or storage can be served by replacement units
- how quickly the system can be re-installed based on the units of delivery provided
- how to (re-) assign runtime elements to processors, communication channels, and data stores
- When employing tactics that depend on redundancy of functionality, the mapping from modules to redundant components is important. E.g., it is possible to write a module that contains code appropriate for both the active and back-up components in a protection group.

Design Checklist for Availability

Resource Management

- Determine what critical resources are necessary to continue operating in the presence of a fault: omission, crash, incorrect timing, or incorrect response. Ensure there are sufficient remaining resources in the event of a fault to log the fault; notify appropriate entities (people or systems); disable source of events causing the fault; fix or mask the fault/failure; operate normally, in startup, shutdown, repair mode, degraded operation, and overloaded operation.
- Determine the availability time for critical resources, what critical resources must be available during specified time intervals, time intervals during which the critical resources may be in a degraded mode, and repair time for critical resources. Ensure that the critical resources are available during these time intervals.
- For example, ensure that input queues are large enough to buffer anticipated messages if a server fails so that the messages are not permanently lost.

Design Checklist for Availability

Binding Time

- Determine how and when architectural elements are bound. If late binding is used to alternate between components that can themselves be sources of faults (e.g. processes, processors, communication channels), ensure the chosen availability strategy is sufficient to cover faults introduced by all sources. E.g.
- If late binding is used to switch between processors that will be the subject of faults, will the fault detection and recovery mechanisms work for all possible bindings?
- If late binding is used to change the definition or tolerance of what constitutes a fault (e.g., how long a process can go without responding before a fault is assumed), is the recovery strategy chosen sufficient to handle all cases? For example, if a fault is flagged after 0.1 ms, but the recovery mechanism takes 1.5 seconds to work, that might be an unacceptable mismatch.
- What are the availability characteristics of the late binding mechanism itself? Can it fail?

Design Checklist for Availability

Choice of Technology

- Determine the available technologies that can (help) detect faults, recover from faults, re-introduce failed components.
- Determine what technologies are available that help the response to a fault (e.g., event loggers).
- Determine the availability characteristics of chosen technologies themselves: What faults can they recover from? What faults might they introduce into the system?

Summary

- Availability refers to the ability of the system to be available for use when a fault occurs.
- The fault must be recognized (or prevented) and then the system must respond.
- The response will depend on the criticality of the application and the type of fault
  - can range from “ignore it” to “keep on going as if it didn’t occur.”

Summary

- Tactics for availability are categorized into detect faults, recover from faults and prevent faults.
- Detection tactics depend on detecting signs of life from various components.
- Recovery tactics are retrying an operation or maintaining redundant data or computations.
- Prevention tactics depend on removing elements from service or limiting the scope of faults.
- All availability tactics involve the coordination model.

BITS Pilani

Pilani | Dubai | Goa | Hyderabad

L03 Interoperability

Harvinder S Jabbal
CSIS, Work Integrated Learning Programs

innovate achieve lead

BITS Pilani

Pilani|Dubai|Goa|Hyderabad

SEZG651/ SSZG653 Software Architectures Module 2-CS 02

Credits

These Slides are based on

- Software Architecture in Practice by
  - Len Bass, Paul Clement and Rick Kazman
  - Pearson © 2013

Outline

- What is Interoperability?
  - Interoperability General Scenario
- Tactics for Interoperability
  - A Design Checklist for Interoperability
- Summary

What is Interoperability?

- Interoperability is about the degree to which two or more systems can usefully exchange meaningful information.
- Like all quality attributes, interoperability is not a yes-or-no proposition but has shades of meaning.

Interoperability General Scenario

Portion of Scenario	Possible Values
Source	A system
Stimulus	A request to exchange information among system(s).
Artifact	The systems that wish to interoperate
Environment	System(s) wishing to interoperate are discovered at run time or known prior to run time.
Response	One or more of the following: • the request is (appropriately) rejected and appropriate entities (people or systems) are notified • the request is (appropriately) accepted and information is exchanged successfully • the request is logged by one or more of the involved systems
Response Measure	One or more of the following: • percentage of information exchanges correctly processed • percentage of information exchanges correctly rejected

Sample Concrete Interoperability Scenario

- Our vehicle information system sends our current location to the traffic monitoring system.
- The traffic monitoring system combines our location with other information, overlays this information on a Google Map, and broadcasts it.
- Our location information is correctly included with a probability of 99.9%.

Goal of Interoperability Tactics

- For two or more systems to usefully exchange information they must
  - Know about each other. That is the purpose behind the locate tactics.
  - Exchange information in a semantically meaningful fashion. That is the purpose behind the manage interfaces tactics. Two aspects of the exchange are
    - Provide services in the correct sequence
    - Modify information produced by one actor to a form acceptable to the second actor.

Goal of Interoperability Tactics

Interoperability Tactics

Locate

- Discover service:
  - Locate a service through searching a known directory service.
  - There may be multiple levels of indirection in this location process
    - ✓– i.e. a known location points to another location that in turn can be searched for the service.

Manage Interfaces

- Orchestrate: uses a control mechanism to coordinate, manage and sequence the invocation of services. Orchestration is used when systems must interact in a complex fashion to accomplish a complex task.
- Tailor Interface: add or remove capabilities to an interface such as translation, buffering, or data-smoothing.

Design Checklist for Interoperability

Allocation of Responsibilities

- Determine which of your system responsibilities will need to interoperate with other systems.
- Ensure that responsibilities have been allocated to detect a request to interoperate with known or unknown external systems
- Ensure that responsibilities have been allocated to
- accept the request
- exchange information
- reject the request
- notify appropriate entities (people or systems)
- log the request (for interoperability in an untrusted environment, logging for non-repudiation is essential)

Design Checklist for Interoperability

Coordination Model

- Ensure that the coordination mechanisms can meet the critical quality attribute requirements. Considerations for performance include:
- Volume of traffic on the network both created by the systems under your control and generated by systems not under your control.
- Timeliness of the messages being sent by your systems
- Currency of the messages being sent by your systems
- Jitter of the messages arrival times.
- Ensure that all of the systems under your control make assumptions about protocols and underlying networks that are consistent with the systems not under your control.

Design Checklist for Interoperability

Data Model

- Determine the syntax and semantics of the major data abstractions that may be exchanged among interoperating systems.
- Ensure that these major data abstractions are consistent with data from the interoperating systems. (If your system's data model is confidential and must not be made public, you may have to apply transformations to and from the data abstractions of systems with which yours interoperates.)

Design Checklist for Interoperability

Mapping Among Architectural Elements

- For interoperability, the critical mapping is that of components to processors. Beyond the necessity of making sure that components that communicate externally are hosted on processors that can reach the network, the primary considerations deal with meeting the security, availability, and performance requirements for the communication.
- These will be dealt with in their respective chapters.

Design Checklist for Interoperability

Resource Management

- Ensure that interoperation with another system (accepting a request and/or rejecting a request) can never exhaust critical system resources (e.g., can a flood of such requests cause service to be denied to legitimate users?).
- Ensure that the resource load imposed by the communication requirements of interoperation is acceptable.
- Ensure that if interoperation requires that resources be shared among the participating systems, an adequate arbitration policy is in place.

Design Checklist for Interoperability

Binding Time

- Determine the systems that may interoperate, and when they become known to each other. For each system over which you have control
- Ensure that it has a policy for dealing with binding to both known and unknown external systems.
- Ensure that it has mechanisms in place to reject unacceptable bindings and to log such requests.
- In the case of late binding, ensure that mechanisms will support the discovery of relevant new services or protocols, or the sending of information using chosen protocols.

Design Checklist for Interoperability

Choice of Technology

- For any of your chosen technologies, are they “visible” at interface boundary of a system? If so, what interoperability effects do they have? Do they support, undercut, or have no effect on the interoperability scenarios that apply to your system? Ensure the effects they have are acceptable.
- Consider technologies that are designed to support interoperability, e.g. Web Services. Can they be used to satisfy the interoperability requirements for the systems under your control?

Summary

- Interoperability refers to the ability of systems to usefully exchange information.
- Achieving interoperability involves the relevant systems locating each other and then managing the interfaces so that they can exchange information.

L04
Modifiability

Harvinder S Jabbal
CSIS, Work Integrated Learning Programs

innovate achieve lead

BITS Pilani

Pilani|Dubai|Goa|Hyderabad

SEZG651/ SSZG653 Software Architectures Module 2-CS 02

Chapter Outline

What is Modifiability?

Modifiability General Scenario

Tactics for Modifiability

A Design Checklist for Modifiability

Summary

What is Modifiability?

Modifiability is about change and our interest in it is in the cost and risk of making changes.

To plan for modifiability, an architect has to consider three questions:

- What can change?
- What is the likelihood of the change?
- When is the change made and who makes it?

Modifiability General Scenario

Portion of Scenario	Possible Values
Source	End user, developer, system administrator
Stimulus	A directive to add/delete/modify functionality, or change a quality attribute, capacity, or technology
Artifacts	Code, data, interfaces, components, resources, configurations, …
Environment	Runtime, compile time, build time, initiation time, design time
Response	One or more of the following: • make modification • test modification • deploy modification
Response Measure	Cost in terms of: • number, size, complexity of affected artifacts • effort • calendar time • money (direct outlay or opportunity cost) • extent to which this modification affects other functions or quality attributes • new defects introduced

Sample Concrete Modifiability Scenario

The developer wishes to change the user interface by modifying the code at design time. The modifications are made with no side effects within three hours.

Goal of Modifiability Tactics

Tactics to control modifiability have as their goal controlling the complexity of making changes, as well as the time and cost to make changes.

Goal of Modifiability Tactics

Modifiability Tactics

Reduce Size of a Module

Split Module: If the module being modified includes a great deal of capability, the modification costs will likely be high. Refining the module into several smaller modules should reduce the average cost of future changes.

Increase Cohesion

Increase Semantic Coherence: If the responsibilities A and B in a module do not serve the same purpose, they should be placed in different modules. This may involve creating a new module or it may involve moving a responsibility to an existing module.

Reduce Coupling

Encapsulate: Encapsulation introduces an explicit interface to a module. This interface includes an API and its associated responsibilities, such as “perform a syntactic transformation on an input parameter to an internal representation.”

Use an Intermediary: Given a dependency between responsibility A and responsibility B (for example, carrying out A first requires carrying out B), the dependency can be broken by using an intermediary.

Reduce Coupling

Restrict Dependencies: restricts the modules which a given module interacts with or depends on.

Refactor: undertaken when two modules are affected by the same change because they are (at least partial) duplicates of each other.

Abstract Common Services: where two modules provide not-quite-the-same but similar services, it may be cost-effective to implement the services just once in a more general (abstract) form.

Defer Binding

In general, the later in the life cycle we can bind values, the better.

If we design artifacts with built-in flexibility, then exercising that flexibility is usually cheaper than hand-coding a specific change.

However, putting the mechanisms in place to facilitate that late binding tends to be more expensive.

Design Checklist for Modifiability

Allocation of Responsibilities	Determine which changes or categories of changes are likely to occur through consideration of changes in technical, legal, social, business, and customer forces. For each potential change or category of changes . Determine the responsibilities that would need to be added, modified, or deleted to make the change. . Determine what other responsibilities are impacted by the change. . Determine an allocation of responsibilities to modules that places, as much as possible, responsibilities that will be changed (or impacted by the change) together in the same module, and places responsibilities that will be changed at different times in separate modules.

Design Checklist for Modifiability

Coordination Model	Determine which functionality or quality attribute can change at runtime and how this affects coordination; for example, will the information being communicated change at run-time, or will the communication protocol change at run-time? If so, ensure that such changes affect a small number set of modules. Determine which devices, protocols, and communication paths used for coordination are likely to change. For those devices, protocols, and communication paths, ensure that the impact of changes will be limited to a small set of modules. For those elements for which modifiability is a concern, use a coordination model that reduces coupling such as publish/subscribe, defers bindings such as enterprise service bus, or restricts dependencies such as broadcast.

Design Checklist for Modifiability

Data Model	Determine which changes (or categories of changes) to the data abstractions, their operations, or their properties are likely to occur. Also determine which changes or categories of changes to these data abstractions will involve their creation, initialization, persistence, manipulation, translation, or destruction. For each change or category of change, determine if the changes will be made by an end user, system administrator, or developer. For those changes made by an end user or administrator, ensure that the necessary attributes are visible to that user and that the user has the correct privileges to modify the data, its operations, or its properties. For each potential change or category of change • determine which data abstractions need to be added, modified, or deleted • determine whether there would be any changes to the creation, initialization, persistence, manipulation, translation, or destruction of these data abstractions • determine which other data abstractions are impacted by the change. For these additional abstractions, determine whether the impact would be on their operations, properties, creation, initialization, persistence, manipulation, translation, or destruction. • ensure an allocation of data abstractions that minimizes the number and severity of modifications to the abstractions by the potential changes Design your data model so that items allocated to each element of the data model are likely to change together.

Design Checklist for Modifiability

Mapping Among Architectural Elements	Determine if it is desirable to change the way in which functionality is mapped to computational elements (e.g. processes, threads, processors) at runtime, compile time, design time, or build time. Determine the extent of modifications necessary to accommodate the addition, deletion, or modification of a function or a quality attribute. This might involve a determination of, for example: • execution dependencies • assignment of data to databases • assignment of runtime elements to processes, threads, or processors Ensure that such changes are performed with mechanisms that utilize deferred binding of mapping decisions.

Design Checklist for Modifiability

Resource Management	Determine how the addition, deletion, or modification of a responsibility or quality attribute will affect resource usage. This involves, for example, . determining what changes might introduce new resources or remove old ones or affect existing resource usage. . determining what resource limits will change and how Ensure that the resources after the modification are sufficient to meet the system requirements. Encapsulate all resource managers and ensure that the policies implemented by those resource managers utilize are themselves encapsulated and bindings are deferred to the extent possible.

Design Checklist for Modifiability

Binding Time	For each change or category of change • Determine the latest time at which the change will need to be made. • Choose a defer-binding mechanism (see Section 7.2.4) that delivers the appropriate capability at the time chosen. • Determine the cost of introducing the mechanism and the cost of making changes using the chosen mechanism • Do not introduce so many binding choices that change is impeded because the dependencies among the choices are complex and unknown.

Design Checklist for Modifiability

Choice of Technology	Determine what modifications are made easier or harder by your technology choices. . Will your technology choices help to make, test, and deploy modifications? . How easy is it to modify your choice of technologies itself (in case some of these technologies change or become obsolete)? Choose your technologies to support the most likely modifications. For example, an Enterprise Service Bus makes it easier to change how elements are connected but may introduce vendor lock in.

Summary

Modifiability deals with change and the cost in time or money of making a change, including the extent to which this modification affects other functions or quality attributes.

Tactics to reduce the cost of making a change include making modules smaller, increasing cohesion, and reducing coupling. Deferring binding will also reduce the cost of making a change.

BITS Pilani

Pilani | Dubai | Goa | Hyderabad

L05 Performance

Harvinder S Jabbal
CSIS, Work Integrated Learning Programs

innovate achieve lead

BITS Pilani

Pilani|Dubai|Goa|Hyderabad

SEZG651/ SSZG653 Software Architectures Module 2-CS 02

Chapter Outline

What is Performance?

Performance General Scenario

Tactics for Performance

A Design Checklist for Performance

Summary

What is Performance?

Performance is about time and the software system's ability to meet timing requirements.

When events occur – interrupts, messages, requests from users or other systems, or clock events marking the passage of time – the system, or some element of the system, must respond to them in time.

Characterizing the events that can occur (and when they can occur) and the system or element's time-based response to those events is the essence is discussing performance.

Performance General Scenario

Portion of Scenario	Possible Values
Source	Internal or external to the system
Stimulus	Arrival of a periodic, sporadic, or stochastic event
Artifact	System or one or more components in the system.
Environment	Operational mode: normal, emergency, peak load, overload.
Response	Process events, change level of service
Response Measure	Latency, deadline, throughput, jitter, miss rate

Sample Concrete Performance Scenario

Users initiate transactions under normal operations. The system processes the transactions with an average latency of two seconds.

Goal of Performance Tactics

Tactics to control Performance have as their goal to generate a response to an event arriving at the system within some time-based constraint.

Goal of Performance Tactics

Performance Tactics

Control Resource Demand

Manage Sampling Rate: If it is possible to reduce the sampling frequency at which a stream of data is captured, then demand can be reduced, typically with some loss of fidelity.

Limit Event Response: process events only up to a set maximum rate, thereby ensuring more predictable processing when the events are actually processed.

Prioritize Events: If not all events are equally important, you can impose a priority scheme that ranks events according to how important it is to service them.

Control Resource Demand

Reduce Overhead: The use of intermediaries (important for modifiability) increases the resources consumed in processing an event stream; removing them improves latency.

Bound Execution Times: Place a limit on how much execution time is used to respond to an event.

Increase Resource Efficiency: Improving the algorithms used in critical areas will decrease latency.

Manage Resources

Increase Resources: Faster processors, additional processors, additional memory, and faster networks all have the potential for reducing latency.

Increase Concurrency: If requests can be processed in parallel, the blocked time can be reduced. Concurrency can be introduced by processing different streams of events on different threads or by creating additional threads to process different sets of activities.

Maintain Multiple Copies of Computations: The purpose of replicas is to reduce the contention that would occur if all computations took place on a single server.

Manage Resources

Maintain Multiple Copies of Data: keeping copies of data (possibly one a subset of the other) on storage with different access speeds.

Bound Queue Sizes: control the maximum number of queued arrivals and consequently the resources used to process the arrivals.

Schedule Resources: When there is contention for a resource, the resource must be scheduled.

Design Checklist for Performance

Allocation of Responsi-bilities	Determine the system's responsibilities that will involve heavy loading, have time-critical response requirements, are heavily used, or impact portions of the system where heavy loads or time critical events occur. For those responsibilities, identify • processing requirements of each responsibility and determine whether they may cause bottlenecks • additional responsibilities to recognize and process requests appropriately including • Responsibilities that result from a thread of control crossing process or processor boundaries. • Responsibilities to manage the threads of control — allocation and de-allocation of threads, maintaining thread pools, and so forth. • Responsibilities for scheduling shared resources or managing performance-related artifacts such as queues, buffers, and caches. For the responsibilities and resources you identified, ensure that the required performance response can be met (perhaps by building a performance model to help in the evaluation).

BITS Pilani, Deemed to be University under Section 3 of UGC Act, 1956

Design Checklist for Performance

Coordination Model	Determine the elements of the system that must coordinate with each other—directly or indirectly—and choose communication and coordination mechanisms that • supports any introduced concurrency (for example, is it thread-safe?), event priorization, or scheduling strategy • ensures that the required performance response can be delivered • can capture periodic, stochastic, or sporadic event arrivals, as needed • have the appropriate properties of the communication mechanisms, for example, stateful, stateless, synchronous, asynchronous, guaranteed delivery, throughput, or latency.

Design Checklist for Performance

Data Model	Determine those portions of the data model that will be heavily loaded, have time critical response requirements, are heavily used, or impact portions of the system where heavy loads or time critical events occur. For those data abstractions, 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.

Design Checklist for Performance

Mapping Among Architectural Elements	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. Determine where introducing concurrency (that is, allocating a piece of functionality to two or more copies of a component running simultaneously) is feasible and has a significant positive effect on performance. Determine whether the choice of threads of control and their associated responsibilities introduces bottlenecks.

Design Checklist for Performance

Resource Management	Determine which resources in your system are critical for performance. For these resources ensure they will be monitored and managed under normal and overloaded system operation. For example • system elements that need to be aware of, and manage, time and other performance-critical resources • process/thread models • prioritization of resources and access to resources • scheduling and locking strategies • deploying additional resources on demand to meet increased loads

Design Checklist for Performance

Binding Time	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.

Design Checklist for Performance

Choice of Technology	Will your choice of technology let you set and meet hard 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 • other performance related parameters Does your choice of technology introduce excessive overhead for heavily used operations?

Summary

Performance is about the management of system resources in the face of particular types of demand to achieve acceptable timing behavior.

Performance can be measured in terms of throughput and latency for both interactive and embedded real time systems.

Performance can be improved by reducing demand or by managing resources more appropriately.

L06 Security

SoftArch_Session04.pdf

BITS Pilani
Pilani|Dubai|Goa|Hyderabad

Harvinder S Jabbal
CSIS, Work Integrated Learning Programs

innovate achieve lead

BITS Pilani

Pilani|Dubai|Goa|Hyderabad

SEZG651/ SSZG653 Software Architectures Module 2-CS 02

Chapter Outline

What is Security?

Security General Scenario

Tactics for Security

A Design Checklist for Security

Summary

What is Security?

Security is a measure of the system's ability to protect data and information from unauthorized access while still providing access to people and systems that are authorized.

An action taken against a computer system with the intention of doing harm is called an *attack* and can take a number of forms.

It may be an unauthorized attempt to access data or services or to modify data, or it may be intended to deny services to legitimate users.

What is Security?

Security has three main characteristics, called CIA:

- Confidentiality is the property that data or services are protected from unauthorized access. For example, a hacker cannot access your income tax returns on a government computer.
- Integrity is the property that data or services are not subject to unauthorized manipulation. For example, your grade has not been changed since your instructor assigned it.
- Availability is the property that the system will be available for legitimate use. For example, a denial-of-service attack won't prevent you from ordering a book from an online bookstore.

Other characteristics that support CIA are

- Authentication verifies the identities of the parties to a transaction and checks if they are truly who they claim to be. For example, when you get an e-mail purporting to come from a bank, authentication guarantees that it actually comes from the bank.
- Nonrepudiation guarantees that the sender of a message cannot later deny having sent the message and that the recipient cannot deny having received the message. For example, you cannot deny ordering something from the Internet, or the merchant cannot disclaim getting your order.
- 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 General Scenario

Portion of Scenario	Possible Values
Source	Human or another system which may have been previously identified (either correctly or incorrectly) or may be currently unknown. A human attacker may be from outside the organization or from inside the organization.
Stimulus	Unauthorized attempt is made to display data, change or delete data, access system services, change the system's behavior, or reduce availability.
Artifact	System services; data within the system; a component or resources of the system; data produced or consumed by the system
Environment	The system is either online or offline, connected to or disconnected from a network, behind a firewall or open to a network, fully operational, partially operational, or not operational
Response	Transactions are carried out in a fashion such that • data or services are protected from unauthorized access; • data or services are not being manipulated without authorization; • parties to a transaction are identified with assurance; • the parties to the transaction cannot repudiate their involvements; • the data, resources, and system services will be available for legitimate use. The system tracks activities within it by • recording access or modification, • recording attempts to access data, resources or services, • notifying appropriate entities (people or systems) when an apparent attack is occurring.
Response Measure	One or more of the following • how much of a system is compromised when a particular component or data value is compromised, • how much time passed before an attack was detected, • how many attacks were resisted, • how long does it take to recover from a successful attack, • how much data is vulnerable to a particular attack

Sample Concrete Security Scenario

A disgruntled employee from a remote location attempts to modify the pay rate table during normal operations. The system maintains an audit trail and the correct data is restored within a day.

Goal of Security Tactics

One method for thinking about system security is to think about physical security.

Secure installations have limited access to them (e.g., by using security checkpoints), have means of detecting intruders (e.g., by requiring legitimate visitors to wear badges), have deterrence mechanisms such as armed guards, have reaction mechanisms such as automatic locking of doors and have recovery mechanisms such as off-site back up.

This leads to our four categories of tactics: detect, resist, react, and recover.

Goal of Security Tactics

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: comparison of the pattern or signature of network traffic *coming into* a system to historic profiles of known Denial of Service (DoS) attacks.

Verify Message Integrity: use techniques such as 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

Resist Attacks

Identify Actors: identify the source of any external input to the system.

Authenticate Actors: ensure that an actor (user or a remote computer) is actually who or what it purports to be.

Authorize Actors: ensuring that an authenticated actor has the rights to access and modify either data or services.

Limit Access: limiting access to resources such as memory, network connections, or access points.

Resist Attacks

Limit Exposure: minimize the attack surface of a system by having the fewest possible number of access points.

Encrypt Data: apply some form of encryption to data and to communication.

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 for Security

Allocation of Responsibilities	Determine which system responsibilities need to be secure. For each of these responsibilities ensure that additional responsibilities have been allocated to: • identify the actor • authenticate the actor • authorize actors • grant or deny access to data or services • record attempts to access or modify data or services • encrypt data • recognize reduced availability for resources or services and inform appropriate personnel and restrict access • recover from an attack • verify checksums and hash values

Design Checklist for Security

Coordination Model	Determine mechanisms required to communicate and coordinate with other systems or individuals. For these communications, ensure that mechanisms for authenticating and authorizing the actor or system, and encrypting data for transmission across the connection are in place. Ensure also that mechanisms exist for monitoring and recognizing unexpectedly high demands for resources or services as well as mechanisms for restricting or terminating the connection.

Design Checklist for Security

Data Model	Determine the sensitivity of different data fields. For each data abstraction • 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 for Security

Mapping Among Architectural Elements	Determine how alternative mappings of architectural elements may change how an individual or system may read, write, or modify data, access system services or resources, or reduce their availability. Determine how alternative mappings may affect the recording of access to data, services or resources and the recognition of high demands for resources. For each such mapping, ensure that there are responsibilities to • identify an actor • authenticate an actor • authorize actors • grant or deny access to data or services • record attempts to access or modify data or services • encrypt data • recognize reduced availability for resources or services, inform appropriate personnel, and restrict access • recover from an attack

Design Checklist for Security

Resource Management	Determine the system resources required to identify and monitor a system or an individual who is internal or external, authorized or not authorized, with access to specific resources or all resources. Determine the resources required to authenticate the actor, grant or deny access to data or resources, notify appropriate entities, record attempts to access data or resources, encrypt data, recognize high demand for resources, inform users or systems, and restrict access. For these resources consider whether an external entity can access or exhaust a critical resource; how to monitor the resource; how to manage resource utilization; how to log resource utilization and ensure that there are sufficient resources to perform necessary security operations. 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 for Security

Binding Time	Determine cases where an instance of a late bound component may be untrusted. For such cases ensure that late bound components can be qualified, that is, if ownership certificates for late bound components are required, there are appropriate mechanisms to manage and validate them; that access to late bound data and services can be managed; that access by late bound components to data and services can be blocked; that mechanisms to record the access, modification, and attempts to access data or services by late bound components are in place; and that system data is encrypted where the keys are intentionally withheld for late bound components

Design Checklist for Security

Choice of Technology	Determine what technologies are available to help user authentication, data access rights, resource protection, data encryption. Ensure that your chosen technologies support the tactics relevant for your security needs.

Summary

Attacks against a system can be characterized as attacks against the confidentiality, integrity, or availability of a system or its data.

This leads to many of the tactics used to achieve security. Identifying, authenticating, and authorizing actors are tactics intended to determine which users or systems are entitled to what kind of access to a system.

No security tactic is foolproof and systems *will* be compromised. Hence, tactics exist to detect an attack, limit the spread of any attack, and to react and recover from an attack.

L07
Testability

Harvinder S Jabbal
CSIS, Work Integrated Learning Programs

innovate achieve lead

BITS Pilani

Pilani|Dubai|Goa|Hyderabad

SEZG651/ SSZG653 Software Architectures Module 2-CS 02

Chapter Outline

What is Testability?

Testability General Scenario

Tactics for Testability

A Design Checklist for Testability

Summary

What is Testability?

Software testability refers to the ease with which software can be made to demonstrate its faults through (typically execution-based) testing.

Specifically, testability refers to the probability, assuming that the software has at least one fault, that it will fail on its *next* test execution.

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

What is Testability?

For a system to be properly testable, it must be possible to *control* each component's inputs (and possibly manipulate its internal state) and then to *observe* its outputs (and possibly its internal state).

Testability General Scenario

Portion of Scenario	Possible Values
Source	Unit testers, integration testers, system testers, acceptance testers, end users, either running tests manually or using automated testing tools
Stimulus	A set of tests are executed due to the completion of a coding increment such as a class, layer or service; the completed integration of a subsystem; the complete implementation of the system; or the delivery of the system to the customer.
Environment	Design time, development time, compile time, integration time, deployment time, run time
Artifacts	The portion of the system being tested
Response	One or more of the following: execute test suite and capture results; capture activity that resulted in the fault; control and monitor the state of the system
Response Measure	One or more of the following: effort to find a fault or class of faults, effort to achieve a given percentage of state space coverage; probability of fault being revealed by the next test; time to perform tests; effort to detect faults; length of longest dependency chain in test; length of time to prepare test environment; reduction in risk exposure (size(loss) * prob(loss))

Sample Concrete Testability Scenario

The unit tester completes a code unit during development and performs a test sequence whose results are captured and that gives 85% path coverage within 3 hours of testing.

Goal of Testability Tactics

The goal of tactics for testability is to allow for easier testing when an increment of software development has completed.

Anything the architect can do to reduce the high cost of testing will yield a significant benefit.

There are two categories of tactics for testability:

- The first category deals with adding controllability and observability to the system.
- The second deals with limiting complexity in the system's design.

Goal of Testability Tactics

Testability Tactics

Control and Observe System State

Specialized Interfaces: to control or capture variable values for a component either through a test harness or through normal execution.

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

Abstract Data Sources: Abstracting the interfaces lets you substitute test data more easily.

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.

Limit Complexity

Limit Structural Complexity: avoiding or resolving cyclic dependencies between components, isolating and encapsulating dependencies on the external environment, and reducing dependencies between components in general.

Limit Non-determinism: finding all the sources of non-determinism, such as unconstrained parallelism, and weeding them out as far as possible.

Design Checklist for Testability

Allocation of Responsibilities	Determine which system responsibilities are most critical and hence need to be most thoroughly tested. Ensure that additional system responsibilities have been allocated to do the following: • execute test suite and capture results (external test or self-test) • capture (log) the activity that resulted in a fault or that resulted in unexpected (perhaps emergent) behavior that was not necessarily a fault • control and observe relevant system state for testing Make sure the allocation of functionality provides high cohesion, low coupling, strong separation of concerns, and low structural complexity.

Design Checklist for Testability

Coordination Model	Ensure the system's coordination and communication mechanisms: • support the execution of a test suite and capture of the results within a system or between systems • support capturing activity that resulted in a fault within a system or between systems • support injection and monitoring of state into the communication channels for use in testing, within a system or between systems • do not introduce needless non-determinism

Design Checklist for Testability

Data Model	Determine the major data abstractions that must be tested to ensure the correct operation of the system. • Ensure that it is possible to capture the values of instances of these data abstractions. • Ensure that the values of instances of these data abstractions can be set when state is injected into the system, so that system state leading to a fault may be re-created. • Ensure that the creation, initialization, persistence, manipulation, translation, and destruction of instances of these data abstractions can be exercised and captured

Design Checklist for Testability

Mapping Among Architectural Elements	Determine how to test the possible mappings of architectural elements (especially mappings of processes to processors, threads to processes, modules to components) so that the desired test response is achieved and potential race conditions identified. In addition, determine whether it is possible to test for illegal mappings of architectural elements.

Design Checklist for Testability

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 (or better yet, identical to) 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

Design Checklist for Testability

Binding Time	Ensure that components that are bound later than compile time can be tested in the late bound context. 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 for Testability

Choice of Technology	Determine what technologies are available to help achieve the testability scenarios that apply to your architecture. Are technologies available to help with regression testing, fault injection, recording and playback, and so on? Determine how testable the technologies are that you have chosen (or are considering choosing in the future) and ensure that your chosen technologies support the level of testing appropriate for your system. For example, if your chosen technologies do not make it possible to inject state, it may be difficult to re-create fault scenarios.

Summary

Ensuring that a system is easily testable has payoffs both in terms of the cost of testing and the reliability of the system.

Controlling and observing the system state are a major class of testability tactics.

Complex systems are difficult to test because of the large state space in which their computations take place, and because of the larger number of interconnections among the elements of the system. Consequently, keeping the system simple is another class of tactics that supports testability.

L08
Usability

Harvinder S Jabbal
CSIS, Work Integrated Learning Programs

innovate achieve lead

BITS Pilani

Pilani|Dubai|Goa|Hyderabad

SEZG651/ SSZG653 Software Architectures Module 2-CS 02

Chapter Outline

What is Usability?

Usability General Scenario

Tactics for Usability

A Design Checklist for Usability

Summary

What is Usability?

Usability is concerned with how easy it is for the user to accomplish a desired task and the kind of user support the system provides.

Over the years, a focus on usability has shown itself to be one of the cheapest and easiest ways to improve a system's quality (or, more precisely, the user's *perception* of quality).

What is Usability?

Usability comprises the following areas:

- Learning system features.
- Using a system efficiently.
- Minimizing the impact of errors.
- Adapting the system to user needs.
- Increasing confidence and satisfaction.

Usability General Scenario

Portion of Scenario	Possible Values
Source	End user, possibly in a specialized role
Stimulus	End user tries to use a system efficiently, learn to use the system, minimize the impact of errors, adapt the system, or configure the system
Environment	Runtime or configuration time
Artifacts	System or the specific portion of the system with which the user is interacting.
Response	The system should either provide the user with the features needed or anticipate the user's needs.
Response Measure	One or more of the following: task time, number of errors, number of tasks accomplished, user satisfaction, gain of user knowledge, ratio of successful operations to total operations, or amount of time or data lost when an error occurs.

Sample Concrete Usability Scenario

The user downloads a new application and is using it productively after two minutes of experimentation.

Goal of Usability Tactics

Researchers in human-computer interaction have used the terms "user initiative," "system initiative," and "mixed initiative" to describe which of the human-computer pair takes the initiative in performing certain actions and how the interaction proceeds.

Usability scenarios can combine initiatives from both perspectives.

We use this distinction between user and system initiative to discuss the tactics that the architect uses to achieve the various scenarios.

Goal of Usability Tactics

Usability Tactics

Support User Initiative

Cancel: the system must listen for the cancel request; the command being canceled must be terminated; resources used must be freed; and collaborating components must be informed.

Pause/Resume: temporarily free resources so that they may be re-allocated to other tasks.

Undo: maintain a sufficient amount of information about system state so that an earlier state may be restored, at the user's request.

Aggregate: ability to aggregate lower-level objects into a group, so that a user operation may be applied to the group, freeing the user from the drudgery.

Support System Initiative

Maintain Task Model: determines context so the system can have some idea of what the user is attempting and provide assistance.

Maintain User Model: explicitly represents the user's knowledge of the system, the user's behavior in terms of expected response time, etc.

Maintain System Model: system maintains an explicit model of itself. This is used to determine expected system behavior so that appropriate feedback can be given to the user.

Design Checklist for Usability

Allocation of Responsibilities	Ensure that additional system responsibilities have been allocated, as needed, to assist the user in • learning how to use the system • efficiently achieving the task at hand • adapting and configuring the system • recovering from user and system errors

Design Checklist for Usability

Coordination Model	Determine whether the properties of system elements' coordination—timeliness, currency, completeness, correctness, consistency—affect how a user learns to use the system, achieves goals or completes tasks, adapts and configures the system, recovers from user and system errors, increases confidence and satisfaction. For example, can the system respond to mouse events and give semantic feedback in real time? Can long-running events be canceled in a reasonable amount of time?

Design Checklist for Usability

Data Model	Determine the major data abstractions that are involved with user-perceivable behavior. Ensure these major data abstractions, their operations, and their properties have been designed to assist the user in achieving the task at hand, adapting and configuring the system, recovering from user and system errors, learning how to use the system, and increasing satisfaction and user confidence For example, the data abstractions should be designed to support undo and cancel operations: the transaction granularity should not be so great that canceling or undoing an operation takes an excessively long time.

Design Checklist for Usability

Mapping Among Architectural Elements	Determine what mapping among architectural elements is visible to the end user (for example, the extent to which the end user is aware of which services are local and which are remote). For those that are visible, determine how this affects the ways in which, or the ease with which the user will learn how to use the system, achieve the task at hand, adapt and configure the system, recover from user and system errors, and increase confidence and satisfaction.

Design Checklist for Usability

Resource Management	Determine how the user can adapt and configure the system's use of resources. Ensure that resource limitations under all user-controlled configurations will not make users less likely to achieve their tasks. For example, attempt to avoid configurations that would result in excessively long response times. Ensure that the level of resources will not affect the users' ability to learn how to use the system, or decrease their level of confidence and satisfaction with the system.

Design Checklist for Usability

Binding Time	Determine which binding time decisions should be under user control and ensure that users can make decisions that aid in usability. For example, if the user can choose, at run-time, the system's configuration, or its communication protocols, or its functionality via plug-ins, you need to ensure that such choices do not adversely affect the user's ability to learn system features, use the system efficiently, minimize the impact of errors, further adapt and configure the system, or increase confidence and satisfaction.

Design Checklist for Usability

Choice of Technology	Ensure the chosen technologies help to achieve the usability scenarios that apply to your system. For example, do these technologies aid in the creation of on-line help, training materials, and user feedback collection. How usable are any of your chosen technologies? Ensure the chosen technologies do not adversely affect the usability of the system (in terms of learning system features, using the system efficiently, minimizing the impact of errors, or adapting/configuring the system, increase confidence and satisfaction).

Summary

Architectural support for usability involves both allowing the user to take the initiative in circumstances such as cancelling a long running command, undoing a completed command, and aggregating data and commands.

To predict user or system response, the system must keep a model of the user, the system, and the task.

Other Quality Attributes

Chapter Outline

Other Important Quality Attributes

Other Categories of Quality Attributes

Software Quality Attributes and System Quality Attributes

Using Standard Lists of Quality Attributes

Dealing with “X-ability”

Summary

Other Important Quality Attributes

Variability: is a special form of modifiability. It refers to the ability of a system and its supporting artifacts to support the production of a set of variants that differ from each other in a preplanned fashion.

Portability: is also a special form of modifiability. Portability refers to the ease with which software that built to run on one platform can be changed to run on a different platform.

Development Distributability: is the quality of designing the software to support distributed software development.

Other Important Quality Attributes

Scalability: Horizontal scalability (scaling out) refers to adding more resources to logical units such as adding another server to a cluster. Vertical scalability (scaling up) refers to adding more resources to a physical unit such as adding more memory to a computer.

Deployability: is concerned with how an executable arrives at a host platform and how it is invoked.

Mobility: deals with the problems of movement and affordances of a platform (e.g. size, type of display, type of input devices, availability and volume of bandwidth, and battery life).

Other Important Quality Attributes

Monitorability: deals with the ability of the operations staff to monitor the system while it is executing.

Safety: Software safety is about the software's ability to avoid entering states that cause or lead to damage, injury, or loss of life, and to recover and limit the damage when it does enter into bad states. The architectural concerns with safety are almost identical with those for availability (i.e. preventing, detecting, and recovering from failures).

Other Categories of Quality Attributes

Conceptual Integrity: refers to consistency in the design of the architecture. It contributes to the understandability of the architecture. Conceptual integrity demands that the same thing is done in the same way through the architecture.

Marketability: Some systems are marketed by their architectures, and these architectures sometimes carry a meaning all their own, independent of what other quality attributes they bring to the system (e.g. service-oriented or cloud-based).

Other Categories of Quality Attributes

Quality in Use: qualities that pertain to the use of the system by various stakeholders. For example

- Effectiveness: a measure whether the system is correct
- Efficiency: the effort and time required to develop a system
- Freedom from risk: degree to which a product or system affects economic status, human life, health, or the environment

Software Quality Attributes and System Quality Attributes

Physical systems, such as aircraft or automobiles or kitchen appliances, that rely on software embedded within are designed to meet a whole other litany of quality attributes: weight, size, electric consumption, power output, pollution output, weather resistance, battery life, and on and on.

The software architecture can have a substantial effect on the system's quality attributes.

Standard Lists of Quality Attributes

Standard Lists of Quality Attributes

Advantages:

- Can be helpful checklists to assist requirements gatherers in making sure that no important needs were overlooked.
- Can serve as the basis for creating your own checklist that contains the quality attributes of concern in your domain, your industry, your organization, your products, …

Standard Lists of Quality Attributes

Disadvantages:

- No list will ever be complete.
- Lists often generate more controversy than understanding.
- Lists often purport to be *taxonomies*. But what is a denial-of-service attack?
- They force architects to pay attention to every quality attribute on the list, even if only to finally decide that the particular quality attribute is irrelevant to their system.

Dealing with "X-ability"

Suppose you must deal with a quality attribute for which there is no compact body of knowledge, e.g. green computing.

What do you do?

1. Model the quality attribute
2. Assemble a set of tactics for the quality attribute
3. Construct design checklists

Summary

There are many other quality attributes than the seven that we cover in detail.

Taxonomies of attributes may offer some help, but their disadvantages often outweigh their advantages.

You may need to design or analyze a system for a “new” quality attribute. While this may be challenging, it is doable.

BITS Pilani

Pilani | Dubai | Goa | Hyderabad

L01 Architecture requirements and design

Harvinder S Jabbal
CSIS, Work Integrated Learning Programs

innovate achieve lead

BITS Pilani

Pilani|Dubai|Goa|Hyderabad

SEZG651/ SSZG653 Software Architectures Module 3-CS 04

Requirements, Designing, Documentation

- Architecture and Requirements
- Designing an Architecture
- Documenting Software Architecture

Architecture and Requirements

- Gathering ASRs from Requirements Documents
- Gathering ASRs by Interviewing Stakeholders
- Gathering ASRs by Understanding the Business Goals
- Capturing ASRs in a Utility Tree
- Tying the Methods Together

Designing an Architecture

Designing an Architecture

- Design Strategy
- The Attribute-Driven Design Method
- The Steps of ADD

Documenting Software Architecture

Documenting Software Architecture

- Uses and Audiences for Architecture Documentation
- Notations for Architecture Documentation
- Views
- Choosing the Views
- Combining Views
- Building the Documentation Package
- Documenting Behavior
- Architecture Documentation and Quality Attributes
- Documenting Architectures That Change Faster Than You Can Document Them
- Documenting Architecture in an Agile Development Project

Architectures in Agile Projects

Chapter Outline

How Much Architecture?

Agility and Architecture Methods

A Brief Example of Agile Architecting

Guidelines for the Agile Architect

Summary

-- Charles Darwin

It is not the strongest of the species that survives,

not the most intelligent that survives. It is the

one that is the most adaptable to change.

Architecture in Agile Environment

- How Much Architecture?
- Agility and Architecture Methods
- A Brief Example of Agile Architecting
- Guidelines for the Agile Architect

Agility

Agile processes were a response to a need for projects to

- be more responsive to their stakeholders
- be quicker to develop functionality that users care about
- show more and earlier progress in a project's life cycle
- be less burdened by documenting aspects of a project that would inevitably change.

These needs are not in conflict with architecture.

Agility

The question for a software project is not "Should I do Agile or architecture?" Instead ask:

- "How much architecture should I do up front versus how much should I defer until the project's requirements have solidified somewhat?"
- "How much of the architecture should I formally document, and when?"

Agile and architecture are happy companions for many software projects.

Manifesto for Agile Software Development

We are uncovering better ways of developing software by doing it and helping others do it. Through this work we have come to value:

Individuals and interactions	• over processes and tools
Working software	• over comprehensive documentation
Customer collaboration	• over contract negotiation
Responding to change	• over following a plan

That is, while there is value in the items on the right,

• we value the items on the left more.

Twelve Agile Principles

1. Our highest priority is to satisfy the customer through early and continuous delivery of valuable software.
2. Welcome changing requirements, even late in development. Agile processes harness change for the customer's competitive advantage.
3. Deliver working software frequently, from a couple of weeks to a couple of months, with a preference to the shorter timescale.
4. Business people and developers must work together daily throughout the project.
5. Build projects around motivated individuals. Give them the environment and support they need, and trust them to get the job done.
6. The most efficient and effective method of conveying information to and within a development team is face-to-face conversation.
7. Working software is the primary measure of progress.
8. Agile processes promote sustainable development. The sponsors, developers, and users should be able to maintain a constant pace indefinitely.
9. Continuous attention to technical excellence and good design enhances agility.
10. Simplicity—the art of maximizing the amount of work not done—is essential.
11. The best architectures, requirements, and designs emerge from self-organizing teams.
12. At regular intervals, the team reflects on how to become more effective, then tunes and adjusts its behavior accordingly.

Agile

- These processes were initially employed on small- to medium-sized projects with short time frames and enjoyed considerable success.
- They were not often used for larger projects, particularly those with distributed development.

How Much Architecture?

There are two activities that can add time to the project schedule:

- Up-front design work on the architecture and up-front risk identification, planning, and resolution work
- Rework due to fixing defects and addressing modification requests.

Intuitively, these two trade off against each other.

Boehm and Turner plotted these two values against each other for three hypothetical projects:

- One project of 10 KSLOC
- One project of 100 KSLOC
- One project of 1,000 KSLOC

How Much Architecture?

How Much Architecture?

- These lines show that there is a sweet spot for each project.
  - For the 10KSLOC project, the sweet spot is at the far left. Devoting much time to up-front work is a waste for a small project.
  - For the 100 KSLOC project, the sweet spot is around 20 percent of the project schedule.
  - For the 1,000 KSLOC project, the sweet spot is around 40 percent of the project schedule.
- A project with a million lines of code is enormously complex.
- It is difficult to imagine how Agile principles alone can cope with this complexity if there is no architecture to guide and organize the effort.

Agility and Documentation

Write for the reader!

If the reader doesn't need it, don't write it.

- But remember that the reader may be a maintainer or other newcomer not yet on the project!

Agility and Architecture Evaluation

- Architecture evaluation work as part of an Agile process? Absolutely.
- Meeting stakeholders' important concerns is a cornerstone of Agile philosophy.
- Our approach to architecture evaluation is exemplified by the Architecture Tradeoff Analysis Method (ATAM).
  - ATAM does not endeavor to analyze all, or even most, of an architecture.
  - The focus is determined by a set of quality attribute scenarios that represent the most important of the concerns of the stakeholders.
- It is easy to tailor a lightweight architecture evaluation.

An Example of Agile Architecting

WebArrow Web-conferencing system

To meet stakeholder needs, architect and developers found that they needed to think and work in two different modes at the same time:

- *Top-down*—designing and analyzing architectural structures to meet the demanding quality attribute requirements and tradeoffs
- *Bottom-up*—analyzing a wide array of implementation-specific and environment-specific constraints and fashioning solutions to them

Experiments to Make Tradeoffs

- To analyze architectural tradeoffs, the team adopted an agile architecture discipline combined with a rigorous program of experiments.
  - These experiments are called “spikes” in Agile terminology.

Experiments to Make Tradeoffs

The experiments (spikes) answered questions such as:

- Would moving to a distributed database from local flat files negatively impact feedback time (latency) for users?
- What (if any) scalability improvement would result from using mod_perl versus standard Perl?
- How difficult would the development and quality assurance effort be to convert to mod_perl?
- How many participants could be hosted by a single meeting server?
- What was the correct ratio between database servers and meeting servers?

Lesson

- Making architecture processes agile does not require radical re-invention of either Agile practices or architecture methods.
- The WebArrow team's emphasis on experimentation proved the key factor.

Guidelines for the Agile Architect

Barry Boehm and colleagues have developed the Incremental Commitment Model to blend agility and architecture.

This model is based upon the following principles:

- Commitment and accountability of success-critical stakeholders
- Stakeholder "satisficing" (meeting an acceptability threshold) based on success-based negotiations and tradeoffs
- Incremental and evolutionary growth of system definition and stakeholder commitment
- Iterative system development and definition
- Interleaved system definition and development allowing early fielding of core capabilities, continual adaptation to change, and timely growth of complex systems without waiting for every requirement and subsystem to be defined
- Risk management—risk-driven anchor point milestones, which are key to synchronizing and stabilizing all of this concurrent activity

Authors' Advice

- If you are building a large, complex system with relatively stable and well-understood requirements and/or distributed development, doing a large amount of architecture work up front will pay off.
- On larger projects with *unstable* requirements, start by quickly designing a candidate architecture even if it leaves out many details.
  - Be prepared to change and elaborate this architecture as circumstances dictate, as you perform your spikes and experiments, and as functional and quality attribute requirements emerge and solidify.
- On smaller projects with uncertain requirements, at least try to get agreement on the major patterns to be employed. Don't spend too much time on architecture design, documentation, or analysis up front.

Summary

- The Agile Manifesto and principles value close-knit teams, with continuous, frequent delivery of working software.
- Agile processes were initially employed on small- to medium-sized projects with short time frames. They were seldom used for larger projects, particularly with distributed development.
- Large-scale successful projects need a blend of agile and architecture.
- Agile architects take a middle ground, proposing an initial architecture and running with that, until its technical debt becomes too great, at which point they need to refactor.
- Boehm and Turner found that projects have a “sweet spot” where up-front architecture planning pays off.

BITS Pilani

Pilani | Dubai | Goa | Hyderabad

Documenting Architecture Views

Harvinder S Jabbal

SSZG653 Software Architectures

innovate achieve lead

BITS Pilani

Pilani|Dubai|Goa|Hyderabad

SEZG651/ SSZG653 Software Architectures Module 4-CS 05/2

Architecture Views

Views

- Representation of a coherent set of architectural elements , as written by and read by system stakeholders.

Documenting

- “Documenting an architecture is a matter of documenting the relevant **views** and then adding a documentation that applies to more than one view.”

Solution to a Problem

- Problem
  - Architecture documents do not address the concerns of all stakeholders .
  - Deferent Stakeholders : end-user, system engineers, developers and project managers.
  - Architecture documents contained complex diagrams some times they are hard to be represented on the documentation.
- Solution
  - Using different notations for several **Views** each one addressing one specific set for concerns.

4+1 Model -
Philippe Kruchten, Rational Software Corp.

Figure 1 — The "4+1" view model

Logical View

- **The logical view**, which is the object model of the design (when an object-oriented design method is used)

Viewer: End-user

considers: Functional requirements- What are the services must be provided by the system to the users.

Notation: The Booch notation .

Tool: Rational Rose

Notation for Logical View-

Philippe Kruchten, Rational Software Corp.

Notation for the logical view

The notation for the logical view is derived from the Booch notation[4]. It is considerably simplified to take into account only the items that are architecturally significant. In particular, the numerous adornments are not very useful at this level of design. We use Rational Rose® to support the logical architecture design.

Figure 2 — Notation for the logical blueprint

Logical view Example PABX-

Philippe Kruchten, Rational Software Corp.

Process View

The process view, which captures the concurrency and synchronization aspects of the design**(The process decomposition).**

viewer: Integrators

considers: Non - functional requirements (scalability, concurrency, and performance)

style: Garlan and Shaw ‘s Architecture styles.

Process (cont.)

Uses multiple levels of abstractions.

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

- Major Tasks: Architecture relevant tasks.
- Minor or helper Tasks: (Buffering)

Notation-

Philippe Kruchten, Rational Software Corp.

Figure 4 — Notation for the Process blueprint

Process View example PABX (partial)-

Philippe Kruchten, Rational Software Corp.

Development View

The *development* view, which describes the **static** organization of the software in its development environment.

Viewer: Programmers and Software Managers

considers: software module organization. (Hierarchy of layers, software management, reuse, constraints of tools).

Notation: the Booch notation.

Style: layered style

Notation for Development blueprint-

Philippe Kruchten, Rational Software Corp.

Figure 5 — Notation for the Development blueprint

Development View – Layered Style

Philippe Kruchten, Rational Software Corp.

Figure 6 — The 5 layers of Hughes Air Traffic Systems (HATS)

Physical View

the physical view, which describes the mapping(s) of the software onto the hardware and reflects its distributed aspect.

Viewer: System Engineers

Considers: Non-functional requirement (reliability, availability and performance). regarding to underlying hardware.

There may be two architecture:

- Test and development
- deployment

Notation for Physical view-

Philippe Kruchten, Rational Software Corp.

Figure 7 — Notation for the Physical blueprint

Physical blueprint PABX-

Philippe Kruchten, Rational Software Corp.

Figure 8 — Physical blueprint for the PABX

Physical view example-

Philippe Kruchten, Rational Software Corp.

A small PABX physical architecture with process allocation.

Physical view example-

Philippe Kruchten, Rational Software Corp.

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

Scenarios

(Putting all “4 views” together)

Viewer: All users and Evaluators.

Considers: System consistency and validity

Notation: Similar to logical view

Scenario example-

Philippe Kruchten, Rational Software Corp.

Scenario for a Local call – selection phase

Correspondence between the views

The **views** are interconnected.

Start with Logical view and Move to Development / Process view and then finally go to Physical view.

From logical to Process view

Two strategies :

- Inside-out: starting from Logical structure
- Outside-in: starting from physical structure

From logical to Process view

Figure 12: Mapping from Logical to Process view

From Logical to development

They are very close, but the larger the project, the greater the distance between these views.

Grouping to subsystems depending on:

- The team organization.
- The class categories which includes the packages.
- The Line of codes.

Iterative process

Not all architectures need all views.

A scenario-driven approach to develop the system is used to handle the iterative.

Documenting the architecture:

- **Software architecture document**: follows closely “4+1” views.
- **Software design guidelines:** it captured the most important design decisions that must be respected to **maintain** the architectural **integrity**.

Software Design Document

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

Figure 13 — Outline of a Software Architecture Document

Annotation:

- “4+1 views” methodology successfully used in the industry
  - Air Traffic Control
  - Telecom
- This paper missing the tools to integrate these views which lead to an inconsistency problem.
- The inconsistency problem is more tangible in the maintenance of the architecture.

Summary

View	*Logical*	*Process*	*Development*	*Physical*	*Scenarios*
Components	Class	Task	Module, Subsystem	Node	Step, Scripts
Connectors	association, inheritance, containment	Rendez-vous, Message, broadcast, RPC, etc.	compilation dependency, "with" clause, "include"	Communica-tion medium, LAN, WAN, bus, etc.	
Containers	Class category	Process	Subsystem (library)	Physical subsystem	Web
Stakeholders	End-user	System designer, integrator	Developer, manager	System designer	End-user, developer
Concerns	Functionality	Performance, availability, S/W fault-tolerance, integrity	Organization, reuse, portability, line-of-product	Scalability, performance,av ailability	Understand-ability
Tool support	Rose	UNAS/SALE DADS	Apex, SoDA	UNAS, Openview DADS	Rose

Table 1 — Summary of the "4+1" view model

BITS Pilani

Pilani | Dubai | Goa | Hyderabad

Architectural Structures and Views

Harvinder S Jabbal
SSZG653 Software Architectures

innovate achieve lead

BITS Pilani

Pilani|Dubai|Goa|Hyderabad

SEZG651/ SSZG653 Software Architectures Module 4-CS 05/1

Structure & View

Structure & View

- A **view** is a representation of a coherent set of architectural elements, as written by and read by system stakeholders.
  - It consists of a representation of a set of elements and the relations among them.
- A **structure** is the set of elements itself, as they exist in software or hardware.

example

- A **module structure** is the set of the system's modules and their organization.
- A **module view** is the representation of that structure, as documented by and used by some system stakeholders.
- These terms are often used interchangeably, but we will adhere to these definitions.

Architectural structures

Module structures.

- Here the elements are modules, which are units of implementation.
- Modules represent a code-based way of considering the system.
- They are assigned areas of functional responsibility.
- There is less emphasis on how the resulting software manifests itself at runtime.
- Module structures allow us to answer questions such as
  - What is the primary functional responsibility assigned to each module?
  - What other software elements is a module allowed to use?
  - What other software does it actually use?
  - What modules are related to other modules by generalization or specialization (i.e., inheritance) relationships?

Component-and-connector structures.

- Here the elements are
  - runtime components (which are the principal units of computation) and
  - connectors (which are the communication vehicles among components).
- Component-and-connector structures help answer questions such as
  - What are the major executing components and how do they interact?
  - What are the major shared data stores?
  - Which parts of the system are replicated?
  - How does data progress through the system?
  - What parts of the system can run in parallel?
  - How can the system's structure change as it executes?

Allocation structures.

- Allocation structures show the relationship between
  - the software elements and
  - the elements in one or more external environments in which the software is created and executed.
- They answer questions such as
  - What processor does each software element execute on?
  - In what files is each element stored during development, testing, and system building?
  - What is the assignment of software elements to development teams?

architectural design

3 structures
3 broad decision types

- How is the system to be structured as a set of code units (modules)?
- How is the system to be structured as a set of elements that have
  - runtime behavior (components) and
  - interactions (connectors)?
- How is the system to relate to nonsoftware structures in its environment
  - (i.e., CPUs, file systems, networks, development teams, etc.)?

SOFTWARE STRUCTURES

SOFTWARE STRUCTURES

Module

Decomposition.

- The units are modules related to each other by the "is a submodule of " relation, showing how larger modules are decomposed into smaller ones recursively until they are small enough to be easily understood.
- Modules in this structure represent a common starting point for design, as the architect enumerates what the units of software will have to do and assigns each item to a module for subsequent (more detailed) design and eventual implementation.
- Modules often have associated products (i.e., interface specifications, code, test plans, etc.).
- The decomposition structure provides a large part of the system's modifiability, by ensuring that likely changes fall within the purview of at most a few small modules.
- It is often used as the basis for the development project's organization, including the structure of the documentation, and its integration and test plans. The units in this structure often have organization-specific names.
- Certain U.S. Department of Defense standards, for instance, define Computer Software Configuration Items (CSCIs) and Computer Software Components (CSCs), which are units of modular decomposition.

Uses.

- The units of this important but overlooked structure are also modules, or (in circumstances where a finer grain is warranted) procedures or resources on the interfaces of modules.
- The units are related by the uses relation.
- One unit uses another if the correctness of the first requires the presence of a correct version (as opposed to a stub) of the second.
- The uses structure is used to engineer systems that can be easily extended to add functionality or from which useful functional subsets can be easily extracted.
- The ability to easily subset a working system allows for incremental development.

Layered.

- When the uses relations in this structure are carefully controlled in a particular way, a system of layers emerges, in which a layer is a coherent set of related functionality.
- In a strictly layered structure, layer n may only use the services of layer n - 1.
- Many variations of this (and a lessening of this structural restriction) occur in practice, however.
- Layers are often designed as abstractions (virtual machines) that hide implementation specifics below from the layers above, engendering portability.

Class, or generalization.

- The module units in this structure are called classes.
- The relation is "inherits-from" or "is-an-instance-of."
- This view supports reasoning about collections of similar behavior or capability (i.e., the classes that other classes inherit from) and parameterized differences which are captured by subclassing.
- The class structure allows us to reason about re-use and the incremental addition of functionality.

Component and Connector

Process, or communicating processes.

- Like all component-and-connector structures, this one is orthogonal to the module-based structures and deals with the dynamic aspects of a running system.
- The units here are processes or threads that are connected with each other by communication, synchronization, and/or exclusion operations.
- The relation in this (and in all component-and-connector structures) is attachment, showing how the components and connectors are hooked together.
- The process structure is important in helping to engineer a system's execution performance and availability.

Concurrency.

- This component-and-connector structure allows the architect to determine opportunities for parallelism and the locations where resource contention may occur.
- The units are components and the connectors are "logical threads."
- A logical thread is a sequence of computation that can be allocated to a separate physical thread later in the design process.
- The concurrency structure is used early in design to identify the requirements for managing the issues associated with concurrent execution.

Shared data, or repository.

- This structure comprises components and connectors that create, store, and access persistent data.
- If the system is in fact structured around one or more shared data repositories, this structure is a good one to illuminate.
- It shows how data is produced and consumed by runtime software elements, and it can be used to ensure good performance and data integrity.

Client-server.

- If the system is built as a group of cooperating clients and servers, this is a good component-and-connector structure to illuminate.
- The components are the clients and servers, and the connectors are protocols and messages they share to carry out the system's work.
- This is useful for separation of concerns (supporting modifiability), for physical distribution, and for load balancing (supporting runtime performance).

Allocation

Deployment.

- The deployment structure shows how software is assigned to hardware-processing and communication elements.
- The elements are software (usually a process from a component-and-connector view), hardware entities (processors), and communication pathways.
- Relations are "allocated-to," showing on which physical units the software elements reside, and "migrates-to," if the allocation is dynamic.
- This view allows an engineer to reason about performance, data integrity, availability, and security.
- It is of particular interest in distributed or parallel systems.

Implementation.

- This structure shows how software elements (usually modules) are mapped to the file structure(s) in the system's development, integration, or configuration control environments.
- This is critical for the management of development activities and build processes.

Work assignment.

- This structure assigns responsibility for implementing and integrating the modules to the appropriate development teams.
- Having a work assignment structure as part of the architecture makes it clear that the decision about who does the work has architectural as well as management implications.
- The architect will know the expertise required on each team.
- Also, on large multi-sourced distributed development projects, the work assignment structure is the means for calling out units of functional commonality and assigning them to a single team, rather than having them implemented by everyone who needs them.

elements and relations in each structure

Software Structure	Relations	Useful for
Decomposition	Is a submodule of; shares secret with	Resource allocation and project structuring and planning; 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 almost-alike implementations from a common template

elements and relations in each structure

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

elements and relations in each structure

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

Kruchten's four views

WHICH STRUCTURES TO CHOOSE?

Logical.

The elements are "key abstractions," which are manifested in the object-oriented world as objects or object classes. This is a module view.

Process.

This view addresses concurrency and distribution of functionality. It is a component-and-connector view.

Development.

This view shows the organization of software modules, libraries, subsystems, and units of development. It is an allocation view, mapping software to the development environment.

Physical.

This view maps other elements onto processing and communication nodes and is also an allocation view (which others call the deployment view).

Thank you

Ref. Text Book

Software Architecture
Module 5
Techniques in a typical layered Architecture

BITS Pilani

Harvinder S Jabbal
SEZG651/SSZG653 Software Architectures

Contents

- Typical layers in Layered architecture
- Techniques used in different layers

Typical layered architecture

Figure 1
The logical architecture view of a layered system

Example of layers in Judiciary system

Typical layered architecture

Benefits

- What are the benefits of layered architecture?

Characteristics & benefits

- Clearly defined functional layers (separation of concerns)
- Loose coupling
- Reusable lower layer components
- Exchangeable parts

Techniques used in different layers

Presentation layer techniques

- Cache frequently used data such as Product catalog (Client side caching vs Server side caching)
- Use asynchronous communication between UI & Webserver to update parts of the web page without loading the whole web page (ex. AJAX)
- Different rendering for different form-factors: Responsive design

Client side caching vs Server side caching

Client-side caching: Here we store frequently used data in the browser of the client machine – example user preference such as payment method, delivery address, etc.

Server-side caching: Here we store frequently used data needed by different back-end modules on the server-side – example: Product codes and description, promotions, etc.

Combination: In practice, a combination the above 2 techniques is used.

Memcached

memcached is a high-performance, distributed memory object caching system, generic in nature, but originally intended for use in speeding up dynamic web applications by alleviating database load.

You can think of it as a short-term memory for your applications.

memcached allows you to take memory from parts of your system where you have more than you need and make it accessible to areas where you have less than you need.

Asynchronous communication with web server

Example use cases:

- Show product list depending on the product category selected by user
- Validating a user input such as loan amount - whether it is within permissible limits, depending on user profile
- Displaying a chat panel
- Reloading Captcha

Asynchronous communication with web server

AJAX Architecture

AJAX stands for **A**synchronous **Ja**vaScript and **X**ML.

AJAX is a technique for creating better, faster, and more interactive web applications with the help of XML, HTML, CSS, and Java Script.

Some famous web applications that use AJAX:

Google Maps (Drag entire map)
Google Suggest (Google suggests as you type)

Responsive web design: Examples

Notice how the layout changes depending on screen size

Responsive web design is an approach that renders web pages well on a variety of devices or screen sizes.

Consider different form factors – Use Responsive design

Ref: Wikipedia

Flexible grids

Allows varying layout depending on screen size

HTML5 and CSS3 help in specifying the display arrangement

Presentation layer

Have you used any other techniques in Presentation layer?

Business layer

Figure 1
A typical application showing the business layer and the components it may contain

Business layer techniques

- Use Application façade to hide internal modules
- Implement session management
- Use Work flow engine, Rules engine, etc. for modifiability

Facade: Making sub-system easier to use

Intent:

- Provide a unified interface to a set of interfaces in a subsystem.
- Facade defines a higher-level interface that makes the subsystem easier to use.
- It typically involves a single wrapper class which contains a set of members required by client.

Example of Application façade: Flight booking

Session management

- Client based state mgmt. (cookies)
- Server based session mgmt. (persistent)

EJB: Stateless & Stateful session beans

Stateful session Bean: All requests from a Client goes to the same instance of the Bean.

Bean maintains the state of the session

Stateless session Bean: Subsequent request from a client to a Bean may go to another instance of the Bean

Hence Client needs to main the state of the session

Work flow engine

Insurance policy processing

Example Activities

Once an activity is completed, the engine invokes the next step in the process
The next step / activity could be a manual one or an automated one

Popular engines: BizTalk Server, Oracle BPEL processor, IBM WebSphere Process Server

Aspect oriented design for cross cutting concerns

Aspect Oriented design aims to encapsulate cross-cutting concerns into aspects (enhancing modularity)

Examples of aspects:

- Logging & instrumentation
- Auditing: Who is doing what
- Caching
- Security

Aspect Oriented Design

Figure 1
A typical application showing the business layer and the components it may contain

Different aspects of the system

They cut across layers

Modules in different layers call the modules in the cross-cutting sub-system at different points in time

Logging & instrumentation

- **Log actionable information** such as errors
  - “Database is unavailable”
  - Information useful to diagnose errors

- Logging is expensive. Log only the most essential data

- **Instrumentation** helps in understanding the usage & behaviour of the system
  - Request rate
  - Error rate
  - Duration of requests
  - Queue length

Ref: https://peter.bourgon.org/blog/2016/02/07/logging-v-instrumentation.html

Business layer

We saw:

- Façade
- Session mangement
- Work flow engine
- Aspect oriented design

Have you used any other technique in Business layer?

Data layer: Prominent techniques used

- Use DB connection pool, if there are too many users
- Use multiple copies of data for faster access
- Use transactions to achieve atomicity
- Use Object - Relational mapping
- Use stored procedures to improve performance
- Use parameterized SQL queries to reduce SQL injection attacks

Object-Relational mapping

- Helps in mapping objects in our program to database table

- Hibernate is a framework that provides an Object/Relational Mapping (O/RM)

SQL Injection attacks

Do not concatenate user entered string to SQL string

- Malicious input can result in the following:
- SELECT product-data FROM table WHERE product-name = ‘**Toothpaste**’ **OR ‘x’ = ‘x’;**
- **This will return all product data**

Use parameterized SQL queries

- SELECT product-data FROM table WHERE product-name = ?
- ? Is a parameter which contains the value entered by user

Data layer

Have you used any other technique in Data layer?

Services layer

Provides services to external systems

Example: A bank may provide following services to external systems:

- Get Balance
- Request chequebook

Figure 1
Overall view of a typical application showing the service layer

Service layer

Situations that need to be handled by service layer

- Handle same request coming twice
- Handle message coming out of sequence
- Handle communication failure (using retry mechanism or by queuing work and sending once communication is restored)

Question

Have you used any other techniques in a layered architecture?

Exercise 1

What are some of the components of Departure Control System in

- Presentation layer
- Business layer
- Data layer
- Service layer

Exercise: Identify components in different layers of DCS

Exercise 2

What software mechanism will you use to improve the performance in the following situation:

- One of the frequently used screens in Aadhaar system is citizen registration.
- This screen has a fields “State” and “District” and “Town”, which are Drop-down fields
- It takes time to loading this screen due to the large number of states, districts and towns in India

Solution

- Cache states, districts and towns in the backend
- Use AJAX for dynamic loading of districts and towns

Exercise 3:

- Suppose you are developing a hotel reservation system.
- You want to provide is an interface to externals applications such as Makemytrip.com, to inquire about availability of rooms and book rooms.
- What layer would you build and what will be the component (s) in that layer?

Answer

Services layer is needed to provide interface to external systems.

This layer will have components such as

- Get room availability (from date, to date) return (# rooms available by type of room)
- Reserve room (# of rooms, type of room, from date, to date) Return (Success / Failure)

Exercise 4

Suppose you are building a complex logistics system for a shipping & container company

Users wanting to send goods via containers will login to the system and place their requests

A number of modules need to be called for this:

- Find out the nearest location of an empty container
- Find a the best transporter to pick up the goods
- Find a ship that is scheduled leave to the desired destination and has spare capacity to load the container

As an architect you want to hide these modules from the client layer.

What technique will you use and in which layer?

Answer

- In the Business layer, provide a unified API – façade - to place a shipping request

- The façade will in turn make use of modules to search for appropriate container, appropriate transporter, appropriate ship, etc.

Appendix

SSL

1. Browser connects to a web server (website) secured with SSL (https). Browser requests that the server identify itself.
2. Server sends a copy of its SSL Certificate, including the server's public key.
3. Browser checks the certificate root against a list of trusted CAs and that the certificate is unexpired, unrevoked, and that its common name is valid for the website that it is connecting to. If the browser trusts the certificate, it creates, encrypts, and sends back a symmetric session key using the server's public key.
4. Server decrypts the symmetric session key using its private key and sends back an acknowledgement encrypted with the session key to start the encrypted session.
5. Server and Browser now encrypt all transmitted data with the session key.

Logging: Apache Log4j

- Logging is an important component of the software development.
- A well-written logging code offers quick debugging, easy maintenance, and structured storage of an application's runtime information.
- Logging does have its drawbacks also. It can slow down an application.

SSL

1. Browser requests that the server identify itself.
2. Server sends a copy of its SSL Certificate
3. Browser checks the certificate. and sends back a symmetric session key
4. Server sends back an acknowledgement.
5. Server and Browser now encrypt all transmitted data with the session key.

IPSec – Internet Protocol Security

IPSec is an Internet Engineering Task Force (IETF) standard suite of protocols that provides data authentication, integrity, and confidentiality as data is transferred between communication points across IP networks.

IPSec provides data security at the IP packet level.

Session management

- A web **session** is a sequence of network HTTP request and response transactions associated to the same user.

- Modern and complex web applications require the **retaining of information or status about each user** for the duration of multiple requests.

Reference Chapter 21
Software Architecture in Practice
Third Edition
Len Bass
Paul Clements
Rick Kazman

BITS Pilani
Pilani|Dubai|Goa|Hyderabad

Architecture Evaluation through Architecture Trade off Analysis Method

Harvinder S Jabbal
SEZG651/SSZG653 Software Architectures

innovate achieve lead

BITS Pilani

Pilani|Dubai|Goa|Hyderabad

SEZG651/ SSZG653 Software Architectures Module 5-CS 06

Chapter Outline

- Evaluation Factors
- The Architecture Tradeoff Analysis Method (ATAM)
- Lightweight Architecture Evaluation
- Summary

Evaluation Factors

Three Forms of Evaluation

Evaluation by the designer within the design process.

Evaluation by peers within the design process.

Analysis by outsiders once the architecture has been designed.

Evaluation by the Designer

Every time the designer makes a key design decision or completes a design milestone, the chosen and competing alternatives should be evaluated.

Evaluation by the designer is the "test" part of the "generate-and-test" approach to architecture design.

How much analysis? This depends on the importance of the decision. Factors include:

- *The importance of the decision*. The more important the decision, the more care should be taken in making it and making sure it's right.
- *The number of potential alternatives.* The more alternatives, the more time could be spent in evaluating them. Try to eliminate alternatives quickly so that the number of viable potential alternatives is small.
- *Good enough as opposed to perfect.* Many times, two possible alternatives do not differ dramatically in their consequences. In such a case, it is more important to make a choice and move on with the design process than it is to be absolutely certain that the best choice is being made. Again, do not spend more time on a decision than it is worth.

Peer Review

Architectural designs can be peer reviewed, just as code can.

A peer review can be carried out at any point of the design process where a candidate architecture, or at least a coherent reviewable part of one, exists.

Allocate at least several hours and possibly half a day.

Peer Review Steps

Step 1

- *The reviewers determine a number of quality attribute scenarios to drive the review.* These scenarios can be developed by the review team or by additional stakeholders.

Step 2

- *The architect presents the portion of the architecture to be evaluated.* The reviewers individually ensure that they understand the architecture. Questions at this point are specifically for understanding.

Step 3

- *For each scenario, the designer walks through the architecture and explains how the scenario is satisfied.* The reviewers ask questions to determine (a) that the scenario is, in fact, satisfied and (b) whether any of the other scenarios being considered will not be satisfied.

Step 4

- *Potential problems are captured.* Real problems must either must be fixed or a decision must be explicitly made by the designers and the project manager that they are willing to accept the problems and its probability of occurrence.

Analysis by Outsiders

Outside evaluators can cast an objective eye on an architecture.

"Outside" is relative; this may mean

- outside the development project
- outside the business unit where the project resides but within the same company
- outside the company altogether.

Outsiders are chosen because they possess specialized knowledge or experience, or long experience successfully evaluating architectures.

Managers tend to be more inclined to listen to problems uncovered by an outside team.

An outside team tends to be used to evaluate complete architectures.

Contextual Factors for Evaluation

What artifacts are available?

- To perform an architectural evaluation, there must be an artifact that describes the architecture.

Who sees the results?

- Some evaluations are performed with the full knowledge and participation of all of the stakeholders. Others are performed more privately.

Who performs the evaluation?

- Evaluations can be carried out by an individual or a team.

Which stakeholders will participate?

- The evaluation process should provide a method to elicit the goals and concerns that the important stakeholders have regarding the system. Identifying the individuals who are needed and assuring their participation in the evaluation is critical.

What are the business goals?

- The evaluation should answer whether the system will satisfy the business goals.

innovate achieve lead

BITS Pilani

Pilani|Dubai|Goa|Hyderabad

The Architecture Tradeoff Analysis Method

The Architecture Tradeoff Analysis Method

The Architecture Tradeoff Analysis Method (ATAM) has been used for over a decade to evaluate software architectures in domains ranging from automotive to financial to defense.

The ATAM is designed so that evaluators need not be familiar with the architecture or its business goals, the system need not yet be constructed, and there may be a large number of stakeholders.

Participants in the ATAM

The evaluation team.

- External to the project whose architecture is being evaluated.
- Three to five people; a single person may adopt several roles in an ATAM.
- They need to be recognized as competent, unbiased outsiders.

Project decision makers.

- These people are empowered to speak for the development project or have the authority to mandate changes to it.
- They usually include the project manager, and if there is an identifiable customer who is footing the bill for the development, he or she may be present (or represented) as well.
- The architect is always included – the architect must willingly participate.

Architecture stakeholders.

- Stakeholders have a vested interest in the architecture performing as advertised.
- Stakeholders include developers, testers, integrators, maintainers, performance engineers, users, builders of systems interacting with the one under consideration, and, possibly, others.
- Their job is to articulate the specific quality attribute goals that the architecture should meet.
- Expect to enlist 12 to 15 stakeholders for the evaluation of a large enterprise-critical architecture.

ATAM Evaluation Team Roles

Role

Responsibilities

Team leader

- Sets up the evaluation; coordinates with client, making sure client's needs are met; establishes evaluation contract; forms evaluation team; sees that final report is produced and delivered (although the writing may be delegated)

Evaluation leader

- Runs evaluation; facilitates elicitation of scenarios; administers scenario selection/prioritization process; facilitates evaluation of scenarios against architecture; facilitates on-site analysis

Scenario scribe

- Writes scenarios on flipchart or whiteboard during scenario elicitation; captures agreed-on wording of each scenario, halting discussion until exact wording is captured

Proceedings scribe

- Captures proceedings in electronic form on laptop or workstation: raw scenarios, issue(s) that motivate each scenario (often lost in the wording of the scenario itself), and resolution of each scenario when applied to architecture(s); also generates a printed list of adopted scenarios for handout to all participants

Questioner

- Raises issues of architectural interest, usually related to the quality attributes in which he or she has expertise

Outputs of the ATAM

1

- A concise presentation of the architecture. The architecture is presented in one hour

2

- Articulation of the business goals. Frequently, the business goals presented in the ATAM are being seen by some of the assembled participants for the first time and these are captured in the outputs.

3

- Prioritized quality attribute requirements expressed as quality attribute scenarios. These quality attribute scenarios take the form described in Chapter 4.

4

- A set of risks and nonrisks.
  - A risk is defined as an architectural decision that may lead to undesirable consequences in light of quality attribute requirements.
  - A nonrisk is an architectural decision that, upon analysis, is deemed safe.
  - The identified risks form the basis for an architectural risk mitigation plan.

Outputs of the ATAM

5.

- A set of risk themes. When the analysis is complete, the evaluation team examines the full set of discovered risks to look for overarching themes that identify systemic weaknesses in the architecture or even in the architecture process and team. If left untreated, these risk themes will threaten the project's business goals.

6.

- Mapping of architectural decisions to quality requirements. For each quality attribute scenario examined during an ATAM, those architectural decisions that help to achieve it are determined and captured.

7.

- A set of identified sensitivity and tradeoff points. These are architectural decisions that have a marked effect on one or more quality attributes.

Intangible Outputs

There are also *intangible* results of an ATAM-based evaluation. These include

- a sense of community on the part of the stakeholders
- open communication channels between the architect and the stakeholders
- a better overall understanding on the part of all participants of the architecture and its strengths and weaknesses.

While these results are hard to measure,

- they are no less important than the others and often are the longest-lasting.

Phases of the ATAM

Phase	Activity	Participants	Typical duration
0	Partnership and preparation: Logistics, planning, stakeholder recruitment, team formation	Evaluation team leadership and key project decision-makers	Proceeds informally as required, perhaps over a few weeks
1	Evaluation: Steps 1-6	Evaluation team and project decision-makers	1-2 days followed by a hiatus of 2-3 weeks
2	Evaluation: Steps 7-9	Evaluation team, project decision makers, stakeholders	2 days
3	Follow-up: Report generation and delivery, process improvement	Evaluation team and evaluation client	1 week

Step 1: Present the ATAM

The evaluation leader presents the ATAM to the assembled project representatives.

This time is used to explain the process that everyone will be following, to answer questions, and to set the context and expectations for the remainder of the activities.

Using a standard presentation, the leader describes the ATAM steps in brief and the outputs of the evaluation.

Step 2: Present Business Drivers

Everyone involved in the evaluation needs to understand the context for the system and the primary business drivers motivating its development.

In this step, a project decision maker (ideally the project manager or the system's customer) presents a system overview from a business perspective.

The presentation should describe the following:

- The system's most important functions
- Any relevant technical, managerial, economic, or political constraints
- The business goals and context as they relate to the project
- The major stakeholders
- The architectural drivers (that is, the architecturally significant requirements)

Step 3: Present the Architecture

The lead architect (or architecture team) makes a presentation describing the architecture.

The architect covers technical constraints such as operating system, hardware, or middleware prescribed for use, and other systems with which the system must interact.

The architect describes the architectural approaches (or patterns, or tactics, if the architect is fluent in that vocabulary) used to meet the requirements.

The architect's presentation should convey the essence of the architecture and not stray into ancillary areas or delve too deeply into the details of just a few aspects.

The architect should present the views that he or she found most important during the creation of the architecture and the views that help to reason about the most important quality attribute concerns of the system.

BITS Pilani, Deemed to be University under Section 3 of UGC Act, 1956

Step 4: Identify Architectural Approaches

The ATAM focuses on analyzing an architecture by understanding its architectural approaches, especially patterns and tactics.

By now, the evaluation team will have a good idea of what patterns and tactics the architect used in designing the system.

- They will have studied the architecture documentation
- They will have heard the architect's presentation in step 3.
- The team should also be adept at spotting approaches not mentioned explicitly

The evaluation team simply catalogs the patterns and tactics that have been identified.

The list is publicly captured and will serve as the basis for later analysis.

Step 5: Generate Utility Tree

The quality attribute goals are articulated in detail via a quality attribute utility tree.

Utility trees serve to make the requirements concrete by defining precisely the relevant quality attribute requirements that the architects were working to provide.

The important quality attribute goals for the architecture under consideration were named in step 2.

In this step, the evaluation team works with the project decision makers to identify, prioritize, and refine the system's most important quality attribute goals.

These are expressed as scenarios, which populate the leaves of the utility tree.

The scenarios are assigned a rank of importance (High, Medium, Low).

Step 6: Analyze Architectural Approaches

The evaluation team examines the highest-ranked scenarios one at a time; the architect is asked to explain how the architecture supports each one.

Evaluation team members—especially the questioners—probe for the architectural approaches that the architect used to carry out the scenario.

Along the way, the evaluation team documents the relevant architectural decisions and identifies and catalogs their risks, nonrisks, sensitivity points, and tradeoffs. Examples:

- Risk: The frequency of heartbeats affects the time in which the system can detect a failed component. Some assignments will result in unacceptable values of this response.
- Sensitivity point: The number of simultaneous database clients will affect the number of transactions that a database can process per second.
- Tradeoff: The heartbeat frequency determines the time for detecting a fault. Higher frequency leads to better availability but consumes more processing time and communication bandwidth (potentially reducing performance).

These, in turn, may catalyze a deeper analysis.

The analysis is not meant to be comprehensive. The key is to elicit sufficient architectural information to establish a link between the architectural decisions made and the quality attribute requirements that need to be satisfied.

BITS Pilani, Deemed to be University under Section 3 of UGC Act, 1956

Example of an Analysis

Scenario #: A12	Scenario: Detect and recover from HW failure of main switch.
Attribute(s)	Availability
Environment	Normal operations
Stimulus	One of the CPUs fails
Response	0.999999 availability of switch

Architectural decisions	Sensitivity	Tradeoff	Risk	Nonrisk
Backup CPU(s)	S2		R8	
No backup data channel	S3	T3	R9	
Watchdog	S4			N12
Heartbeat	S5			N13
Failover routing	S6			N14

Reasoning	Ensures no common mode failure by using different hardware and operating system (see Risk 8) Worst-case rollover is accomplished in 4 seconds as computing state takes that long at worst Guaranteed to detect failure within 2 seconds based on rates of heartbeat and watchdog Watchdog is simple and has proved reliable Availability requirement might be at risk due to lack of backup data channel ... (see Risk 9)
Architecture diagram	

Step 7: Brainstorm and Prioritize Scenarios

The stakeholders brainstorm scenarios that are operationally meaningful with respect to the stakeholders' individual roles.

- A maintainer will likely propose a modifiability scenario
- A user will probably come up with a scenario that expresses useful functionality or ease of operation
- A quality assurance person will propose a scenario about testing the system or being able to replicate the state of the system leading up to a fault.

The purpose of scenario brainstorming is to take the pulse of the larger stakeholder community: to understand what system success means for them.

Once the scenarios have been collected, they are prioritized by voting.

The list of prioritized scenarios is compared with those from the utility tree exercise.

- If they agree, it indicates good alignment between what the architect had in mind and what the stakeholders actually wanted.
- If additional driving scenarios are discovered—and they usually are—this may itself be a risk, if the discrepancy is large. This would indicate that there was some disagreement in the system's important goals between the stakeholders and the architect.

Step 8: Analyze Architectural Approaches

In this step the evaluation team performs the same activities as in step 6, using the highest-ranked, newly generated scenarios.

The evaluation team guides the architect in the process of carrying out the highest ranked new scenarios.

The architect explains how relevant architectural decisions contribute to realizing each one.

This step might cover the top 5-10 scenarios, as time permits.

Step 9: Present Results

The evaluation team confers privately to group risks into risk themes, based on some common underlying concern or systemic deficiency.

- For example, a group of risks about inadequate or out-of-date documentation might be grouped into a risk theme stating that documentation is given insufficient consideration.
- A group of risks about the system's inability to function in the face of various hardware and/or software failures might lead to a risk theme about insufficient attention to backup capability or providing high availability.

For each risk theme, the evaluation team identifies which of the business drivers listed in step 2 are affected.

- This elevates the risks that were uncovered to the attention of management, who cares about the business drivers.

Step 9: Present Results

The collected information from the evaluation is summarized and presented to stakeholders.

The following outputs are presented:

- The architectural approaches documented
- The set of scenarios and their prioritization from the brainstorming
- The utility tree
- The risks discovered
- The nonrisks documented
- The sensitivity points and tradeoff points found
- Risk themes and the business drivers threatened by each one

innovate achieve lead

BITS Pilani

Pilani|Dubai|Goa|Hyderabad

Lightweight Architectural Evaluation

Lightweight Architectural Evaluation

An ATAM is a substantial undertaking.

- It requires some 20 to 30 person-days of effort from an evaluation team, plus even more for the architect and stakeholders.
- Investing this amount of time makes sense on a large and costly project, where the risks of making a major mistake in the architecture are unacceptable.

We have developed a Lightweight Architecture Evaluation method, based on the ATAM, for smaller, less risky projects.

- May take place in a single day, or even a half-day meeting.
- May be carried out entirely by members internal to the organization.
- Of course this lower level of scrutiny and objectivity may not probe the architecture as deeply.

Because the participants are all internal to the organization and fewer in number than for the ATAM, giving everyone their say and achieving a shared understanding takes much less time.

The steps and phases of a Lightweight Architecture Evaluation can be carried out more quickly.

BITS Pilani, Deemed to be University under Section 3 of UGC Act, 1956

Typical Agenda: 4-6 Hours

Step	Time	Notes
1. Present the ATAM	0 hours	Participants already familiar with process.
2. Present business drivers	0.25 hours	The participants are expected to understand the system and its business goals and their priorities. A brief review ensures that these are fresh in everyone's mind and that there are no surprises.
3. Present architecture	0.5 hours	All participants are expected to be familiar with the system. A brief overview of the architecture, using at least module and C&C views, is presented. 1-2 scenarios are traced through these views.
4. Identify architectural approaches	0.25 hours	The architecture approaches for specific quality attribute concerns are identified by the architect. This may be done as a portion of step 3.
5. Generate QA utility tree	0.5- 1.5 hours	Scenarios might exist: part of previous evaluations, part of design, part of requirements elicitation. Put these in a tree. Or, a utility tree may already exist.
6. Analyze architectural approaches	2-3 hours	This step—mapping the highly ranked scenarios onto the architecture—consumes the bulk of the time and can be expanded or contracted as needed.
7. Brainstorm scenarios	0 hours	This step can be omitted as the assembled (internal) stakeholders are expected to contribute scenarios expressing their concerns in step 5.
8. Analyze architectural approaches	0 hours	This step is also omitted, since all analysis is done in step 6.
9. Present results	0.5 hours	At the end of an evaluation, the team reviews the existing and newly discovered risks, nonrisks, sensitivities, and tradeoffs and discusses whether any new risk themes have arisen.

Summary

If a system is important enough for you to explicitly design its architecture, then that architecture should be evaluated.

The number of evaluations and the extent of each evaluation may vary from project to project.

- A designer should perform an evaluation during the process of making an important decision.
- Lightweight evaluations can be performed several times during a project as a peer review exercise.

The ATAM is a comprehensive method for evaluating software architectures. It works by having project decision makers and stakeholders articulate a precise list of quality attribute requirements (in the form of scenarios) and by illuminating the architectural decisions relevant to carrying out each high-priority scenario. The decisions can then be understood in terms of risks or non-risks to find any trouble spots in the architecture.

Lightweight Architecture Evaluation, based on the ATAM, provides an inexpensive, low-ceremony architecture evaluation that can be carried out in an afternoon.

Thank you……..

Credits

- **Chapter Reference from Text T1: 21**
- Slides have been adapted from Authors Slides Software Architecture in Practice – Third Ed.
  - Len Bass
  - Paul Clements
  - Rick Kazman

BITS Pilani

Software Architecture
Module 5
Ensuring conformance to architecture

Harvinder S Jabbal

SEZG651/SSZG653 Software Architectures

Code may drift away from Architecture

Examples of drift:

- Not sticking to the discipline of layers – an object in one layer calling an object located in a layer beyond the adjacent layer
- Accessing data base directly without going through data access layer
- Notifying different modules one by one instead of using ‘publish – subscribe’ model,
- Not making use of a common logging mechanism, etc.

What other architecture violations have you come across?

Techniques to keep the code and architecture consistent

- Embed design concepts in the code (Architecturally evident coding style)
- Use frameworks
- Use code templates
- Update architecture documentation

Embed design in code

Follow ‘Architecturally Evident Coding Style’

Indicate in the code, aspects of architecture being implemented. Example:

- If we are using a layered architecture, indicate to which layer the code belongs
- If we are using ‘Publish – Subscribe’ pattern, indicate in the code whether it belongs to Publisher or Subscriber
- If we are using Message queues (MQ) for communication between components, indicate what the component is doing - inserting message or retrieving message from the MQ.

Use Frameworks

Examples of frameworks:

- Spring
- Hibernate
- Autosar - AUTomotive Open System Architecture

Spring MVC Framework

- Supports Model-View-Controller framework
- Here code belongs to one of the 3 component types - Model, View or Controller
  - **Model** encapsulates the application data
  - **View** renders screens on browser
  - **Controller** processes user requests, interacts with Model components and passes information to View components for rendering

Experience sharing

Can you give examples of other frameworks that you have used?

Other examples of frameworks

- Publish – Subscribe framework (JMS)
- Workflow framework of SalesForce.com
- Rules engine (DROOLS)
- Logging framework - Log4J (for logging events)

Use Code templates

- A template provides a structure for developers to code
- For example a template for developing a code that needs to be fault tolerant will have the following sections in the template

```
Get event

Case (Event type)
    Normal: // received by primary process
        Process X; Send state to backup process;
        Process Y; Send state to backup process;

    Update State data: // received by Backup process
        Update state data;

    Switch over: // received by Backup process
        Notify clients about change in Primary process;

End case;
```

Code needs to be accommodated in this template

Update architecture documentation

- Changes to code should be accompanied with changes to architecture document when applicable
- At least mark parts of the arch document that is no longer applicable as ‘No longer applicable’. This increases trust on remainder of the document
- Technique: At release time, synchronize the arch doc with code.

Exercise

- What other techniques can we think of to ensure that code does not drift away from architecture?

Exercise

What other techniques can we think of to ensure that code does not drift away from architecture?

- Educate new team members about the architecture
- Do Code reviews
- Create folders for each architectural aspect such as layer, service, UI, external interfaces, etc. and follow a discipline of storing the code in respective folders

Appendix

BITS Pilani

Software Architecture
Module 5
Architecture & testing

Harvinder S Jabbal

SEZG651/SSZG653 Software Architectures

Architecture & testing

How does architecture help in testing activities?

Architecture & testing

- Arch helps prioritize test cases based on ASRs
- Architecture can help in creation of integration test plan
- Architecture needs to be designed to support testability requirements

Architecture & testing

- **Arch helps prioritize test cases based on ASRs**

- What work product of architecture design can we use to prioritize the test cases?

What work product of architecture design can we use to prioritize the test cases?

Utility tree can be used to identify high priority scenarios (high on business value and high on architectural impact) which will translate into high priority test cases

Architecture & testing

Architecture can help in creation of integration test plan

How?

Architecture & testing

Architecture can help in creation of integration test plan

How?

- Architecture tells us what modules interact with each other
- Architecture tells us which modules depend on which other modules.
- This helps us identify integration test cases and identify modules needed for integration testing

Architecture & testing

Architecture needs to be designed to support testability requirements such as

- Ability to switch data sources (Test data & Production data)
- Roll back changes made by test cases (to restore system to original state, so that we can execute more test cases later)
- Ability to replace components (simulators of payment gateway, sensors, external systems such as Aadhar system)

Experience sharing

Can you describe how your architect helped the testing team?

Appendix

BITS Pilani

Software Architecture
Module 5
Architecture Reconstruction

Harvinder S Jabbal
SEZG651/SSZG653 Software Architectures

Contents

- Purpose of architecture reconstruction
- Architecture reconstruction technique
- Reconstruction tools

Purpose of architecture reconstruction

- To understand the architecture of a system for which no documentation exists
- To migrate a system from old technology to new. Ex. Mainframe to Web
- To identify reusable components in a system, such as logging component, security component, etc.

Phases of architecture reconstruction

- Identify components and their relationship (using a tool)
- Aggregate components into abstract components (specify grouping to tool)
- Analyse the architecture (tool displays architecture)

Identify components & their relationship

- Extract information from
  - Source code
  - Execution traces
  - Build scripts
  - Etc.

- This gets info such as
  - classes
  - file they use
  - ‘Caller – callee’ relationship
  - global data accessed by different objects

Examples of component extraction

Table 20.1. Examples of Extracted Elements and Relations

Source Element	Relation	Target Element	Description
File	`includes`	File	C preprocessor `#include` of one file by another
File	`contains`	Function	Definition of a function in a file
File	`defines_var`	Variable	Definition of a variable in a file
Directory	`contains`	Directory	Directory contains a subdirectory
Directory	`contains`	File	Directory contains a file
Function	`calls`	Function	Static function call
Function	`access_read`	Variable	Read access on a variable
Function	`access_write`	Variable	Write access on a variable

Ref: Text book: Software Arch in practice by Len Bass and others

Execution trace of method calls

Figure 1.1. Tree representation of a trace of routine calls

Ref: Techniques to Simplify the Analysis of Execution Traces for Program Comprehension by Abdelwahab Hamou-Lhadj

https://pdfs.semanticscholar.org/3db0/dd1980586c0a9d489e4b94c2996f117df2d5.pdf

Case study: 'Vanish' System

Tool used for reconstruction: **ARMIN**

(ARchitecture Reconstruction and MINing)

White-Noise" View Showing All of the Elements and Relations

Ref: https://resources.sei.cmu.edu/asset_files/TechnicalNote/2003_004_001_14141.pdf

Case study: 'Vanish' System

Architecture view after aggregation of components

One can take help of the technical team to get a broad understanding of the system, before starting the aggregation exercise

Case study: 'Vanish' System

Analyse architecture

One can notice that the architecture of 'Vanish' is not strictly layered

Tools

Dali, ARMIN, Lattix, Sonar J, Structure 101

References:

https://resources.sei.cmu.edu/asset_files/TechnicalReport/2003_005_001_14081.pdf

Experience sharing

- Were you involved in architecture reconstruction of an existing system?
- How did you go about it?
- What tools did you use?

Appendix

SAR: Sw Arch Reconstruction

Ref: https://rmod.inria.fr/archives/papers/Duca09c-TSE-SOAArchitectureExtraction.pdf

Phases of architecture reconstruction

- **Raw view extraction** gets information from source code, execution traces and build scripts. It gets info such as classes, file / data they use, caller – callee relationship, global data accessed by different objects
- **DB construction**: Putting extracted data into a common format
- **View fusion**: Combines views of info stored in DB. Source code analysis gives a static view. If some objects are dynamically bound at run time then execution trace will provide this information. Then an expert may group the elements into a layer
- **Arch analysis**: Validate the correctness of architecture elements obtained from view fusion phase. Ex. There could be restriction that a layer calls objects in adjacent layers only. Or all db access should be via an entity bean only
- **Iterate**

Figure 2.1. ISVis scenario view which consists of the information mural view (on the right) and the temporal message-flow diagram (center).

Figure 3.1. The graph representation of a trace is a better way to spot the number of distinct subtrees it contains

Ref: Techniques to Simplify the Analysis of Execution Traces for Program Comprehension by Abdelwahab Hamou-Lhadj

https://pdfs.semanticscholar.org/3db0/dd1980586c0a9d489e4b94c2996f117df2d5.pdf

SonarCube Explores All Execution Paths

https://www.sonarqube.org/features/issues-tracking/

Group extracted components

- Combines views of info stored in DB.

- View 1: Source code analysis gives a static view.
- View 2: If some objects are dynamically bound at run time then execution trace will provide this information.
- View 3: Then an expert may group the elements into a layer

- Combine all these views to form a consolidated view

Sample View fusion using Sonar tool

Figure 20.3. Hypothesized layers and vertical slices

Sonar tool allows definition of layers and vertical slices through the layers

The tool will populate the layers & slices with components / elements

Architecture analysis

- **Check conformance to architecture**

- Ex. There could be restriction that a layer calls objects in adjacent layers only. Or
- Ex. All db access should be via an entity bean only

Example of violation of architecture (detected by Sonar)

Figure 20.5. Highlighting an architecture violation

No portion of the application should depend upon Junit. Based on this specification, Sonar detects the rule violation

Example of architecture violation (detected by Sonar)

Figure 20.6. An architecture violation discovered by dynamic analysis

All database access is supposed to be managed by entity beans. Discovered by tool Disco Tect

BITS Pilani

Pilani | Dubai | Goa | Hyderabad

Module 6 CS 07

Patterns – Part 1 Layers/ Broker/ Pipes and Filters

Harvinder S Jabbal
SSZG653 Software Architectures

Patterns

Outline

What is a Pattern?

Pattern Catalogue

- Module patterns
  - Layered Pattern
- Component and Connector Patterns
  - Broker Pattern
  - MVC Pattern
  - Pipe-and-Filter Pattern
  - Client Server Pattern
  - Peer-to-Peer Pattern
  - SOA Pattern
  - Publish Subscribe
  - Shared Data Pattern
- Allocation Patterns
  - Map-Reduce Pattern
  - Multi-tier Pattern

Relation Between Tactics and Patterns

Using tactics together

Summary

What is a Pattern?

An architectural pattern establishes a relationship between:

A context. A recurring, common situation in the world that gives rise to a problem.

A problem. The problem, appropriately generalized, that arises in the given context.

A solution. A successful architectural resolution to the problem, appropriately abstracted. The solution for a pattern is determined and described by:

- A set of element types (for example, data repositories, processes, and objects)
- A set of interaction mechanisms or connectors (for example, method calls, events, or message bus)
- A topological layout of the components
- A set of semantic constraints covering topology, element behavior, and interaction mechanisms

Layer Pattern

Context: All complex systems experience the need to develop and evolve portions of the system independently. For this reason the developers of the system need a clear and well-documented separation of concerns, so that modules of the system may be independently developed and maintained.

Problem: The software needs to be segmented in such a way that the modules can be developed and evolved separately with little interaction among the parts, supporting portability, modifiability, and reuse.

Solution: To achieve this separation of concerns, the layered pattern divides the software into units called layers. Each layer is a grouping of modules that offers a cohesive set of services. The usage must be unidirectional. Layers completely partition a set of software, and each partition is exposed through a public interface.

Layer Pattern Example

Layer Pattern Solution

Overview: The layered pattern defines layers (groupings of modules that offer a cohesive set of services) and a unidirectional *allowed-to-use* relation among the layers.

Elements: *Layer*, a kind of module. The description of a layer should define what modules the layer contains.

Relations: *Allowed to use.* The design should define what the layer usage rules are and any allowable exceptions.

Constraints:

- Every piece of software is allocated to exactly one layer.
- There are at least two layers (but usually there are three or more).
- The *allowed-to-use* relations should not be circular (i.e., a lower layer cannot use a layer above).

Weaknesses:

- The addition of layers adds up-front cost and complexity to a system.
- Layers contribute a performance penalty.

Broker Pattern

Broker Pattern

Context: Many systems are constructed from a collection of services distributed across multiple servers. Implementing these systems is complex because you need to worry about how the systems will interoperate—how they will connect to each other and how they will exchange information—as well as the availability of the component services.

Problem: How do we structure distributed software so that service users do not need to know the nature and location of service providers, making it easy to dynamically change the bindings between users and providers?

Solution: The broker pattern separates users of services (clients) from providers of services (servers) by inserting an intermediary, called a broker. When a client needs a service, it queries a broker via a service interface. The broker then forwards the client's service request to a server, which processes the request.

Brok

-transfers
*
1
Broker
1
-transfers
*
Client-Side Proxy
+pack_data()
+unpack_data()
+send_request()
+return()
1
+locateServer()
+locateClient()
+registerServer()
+unregisterServer()
1
Server-Side Proxy
+pack_data()
+unpack_data()
+call_service()
+send_response()
-call
1
*
*
0..1
1
-call
1
-call
-uses
Client
+call_server()
+start_task()
+use_Broker_API()
*
Bridge
+pack_data()
+unpack_data()
+forward_message()
+transmit_message()
*
-uses
Server
+initialize()
+enter_main_loop()
+run_service()
+use_Broker_API()

Broker Solution – 1

Overview: The broker pattern defines a runtime component, called a broker, that mediates the communication between a number of clients and servers.

Elements:

- *Client,* a requester of services
- *Server,* a provider of services
- *Broker,* an intermediary that locates an appropriate server to fulfill a client's request, forwards the request to the server, and returns the results to the client
- *Client-side proxy,* an intermediary that manages the actual communication with the broker, including marshaling, sending, and unmarshaling of messages
- *Server-side proxy,* an intermediary that manages the actual communication with the broker, including marshaling, sending, and unmarshaling of messages

Broker Solution - 2

Relations: The *attachment* relation associates clients (and, optionally, client-side proxies) and servers (and, optionally, server-side proxies) with brokers.

Constraints: The client can only attach to a broker (potentially via a client-side proxy). The server can only attach to a broker (potentially via a server-side proxy).

Weaknesses:

- Brokers add a layer of indirection, and hence latency, between clients and servers, and that layer may be a communication bottleneck.
- The broker can be a single point of failure.
- A broker adds up-front complexity.
- A broker may be a target for security attacks.
- A broker may be difficult to test.

Pipe and Filter Pattern

Pipe and Filter Pattern

Context: Many systems are required to transform streams of discrete data items, from input to output. Many types of transformations occur repeatedly in practice, and so it is desirable to create these as independent, reusable parts.

Problem: Such systems need to be divided into reusable, loosely coupled components with simple, generic interaction mechanisms. In this way they can be flexibly combined with each other. The components, being generic and loosely coupled, are easily reused. The components, being independent, can execute in parallel.

Solution: The pattern of interaction in the pipe-and-filter pattern is characterized by successive transformations of streams of data. Data arrives at a filter's input port(s), is transformed, and then is passed via its output port(s) through a pipe to the next filter. A single filter can consume data from, or produce data to, one or more ports.

Pipe and Filter Pattern

Pipe and Filter Solution

Overview: Data is transformed from a system's external inputs to its external outputs through a series of transformations performed by its filters connected by pipes.

Elements:

- *Filter,* which is a component that transforms data read on its input port(s) to data written on its output port(s).
- *Pipe,* which is a connector that conveys data from a filter's output port(s) to another filter's input port(s). A pipe has a single source for its input and a single target for its output. A pipe preserves the sequence of data items, and it does not alter the data passing through.

Relations: The *attachment* relation associates the output of filters with the input of pipes and vice versa.

Constraints:

- Pipes connect filter output ports to filter input ports.
- Connected filters must agree on the type of data being passed along the connecting pipe.

Module 6: CS 07

Patterns – Part 1 Supl 1

Layers

Harvinder S Jabbal
SSZG653 Software Architectures

Introducing Patterns

Patterns

An architectural pattern establishes a relationship between:

- ✓ *A context*.
- ✓ *A problem.*
- ✓ *A solution.*

Context

A context.

A recurring, common situation in the world that gives rise to a problem.

Problem

A problem.

The problem, appropriately generalized, that arises in the given context.

Solution

A solution.

A successful architectural resolution to the problem, appropriately abstracted. The solution for a pattern is determined and described by:

- A set of element types (for example, data repositories, processes, and objects)
- A set of interaction mechanisms or connectors (for example, method calls, events, or message bus)
- A topological layout of the components
- A set of semantic constraints covering topology, element behavior, and interaction mechanisms

Properties of Patterns

- Problem…Context…Solution
- Existing, well-proven design experience
- Identify and Specify Abstractions
- Provide Common Vocabulary
- Means of documenting Software Architecture
- Support construction with defined properties
- Help build complex and heterogenous architectures
- Help to manage software complexity

Problem…Context…Solution

- A pattern addresses a recurring design problem that arises in specific design situations, and presents a solution to it.

- If the problem is supporting variability in user interfaces. This problem may arise when developing software systems with human-computer interaction. You can solve this problem by a strict separation of responsibilities: the core functionality of the application is separated from its user interface.

Existing, well-proven design experience

- Patterns document existing, well-proven design experience.
- They are not invented or created artificially.
- Rather they 'distill and provide a means to reuse the design knowledge gained by experienced practitioners
- Those familiar with an adequate set of patterns 'can apply them immediately to design problems without having to rediscover them'
- Instead of knowledge existing only in the heads of a few experts, patterns make it more generally available.
- You can use such expert knowledge to design high-quality software for a specific task.
- The Model-View-Controller pattern, for example, presents experience gained over many years of developing interactive systems. Many well-known applications already apply the Model-View-Controller pattern-it is the classical architecture for many Smalltalk applications, and under several application frameworks such as MacApp/Windows Apps.

Identify and Specify Abstractions

- Patterns identify and specify abstractions that are above the level of single classes and instances, or of components.
- Typically, a pattern describes several components, classes or objects, and details their responsibilities and relationships, as well as their cooperation.
- All components together solve the problem that the pattern addresses, and usually more effectively than a single component.
- Example, the Model-View-Controller pattern describes a triad of three cooperating components, and each MVC triad also cooperates with other MVC triads of the system.

Provide Common Vocabulary

- Patterns provide a common vocabulary and understanding for design principles.
- Pattern names, if chosen carefully, become part of a widespread design language.
- They facilitate effective discussion of design problems and their solutions.
- They remove the need to explainn a solution to a particular problem with a lengthy and complicated description.
- Instead you can use a pattern name, and explain which parts of a solution correspond to which components of the pattern, or to which relationships between them.
- Example, the name 'Model-View-Controller' and the associated pattern has been to the Smalltalk community since the early '80s, and is used by many software engineers. When we say 'the architecture of the software follows Model-View-Controller', all our colleagues who are familiar with the pattern have an idea of the basic structure and properties of the application immediately.

Means of documenting Software Architecture

- Patterns are a means of documenting software architectures.
- They can describe the vision you have in mind when designing a software system.
- This helps others to avoid violating this vision when extending and modifying the original architecture, or when modifying the system's code.
- Example, if you know that a system is structured according to the Model-View-Controller pattern, you also know how to extend it with a new function: keep core functionality separate from user input and information display.

Support construction with defined properties

- Patterns support the construction of software with defined properties.
- Patterns provide a skeleton of functional behaviour and therefore help to implement the functionality of your application
- Example, patterns exist for maintaining consistency between cooperating components and for providing transparent peer-to-peer inter-process communication. In addition, patterns explicitly address non- functional requirements for software systems, such as changeability, reliability, testability or reusability.
- The Model-View-Controller pattern, for example, supports the changeability of user interfaces and the reusability of core functionality.

Help build complex and heterogenous architectures

- Patterns help you build complex and heterogeneous software architectures.
- Every pattern provides a predefined set of components, roles and relationships between them.
- It can be used to specify particular aspects of concrete software structures.
- Patterns 'act as building-blocks for constructing more complex designs'.
- This method of using predefined design artefacts supports the speed and the quality of your design.
- Understanding and applying well-written patterns saves time when compared to searching for solutions on your own.
- This is not to say that individual patterns will necessarily be better than your own solutions, but, at the very least. a pattern system can help you to evaluate and assess design alternatives.

….cont.

- However, although a pattern determines the basic structure of the solution to a particular design problem, it does not specify a fully detailed solution.
- A pattern provides a scheme for a generic solution to a family of problems, rather than a prefabricated module that can be used 'as is'.
- You must implement this scheme according to the specific needs of the design problem at hand.
- A pattern helps with the creation of similar units.
- These units can be alike in their broad structure, but are frequently quite different in their detailed appearance.
- Patterns help solve problems, but they do not provide complete solutions.

Help to manage software complexity

- Patterns help you to manage software complexity.
- Every pattern describes a proven way to handle the problem it addresses: the kinds of components needed, their roles, the details that should be hidden, the abstractions that should be visible, and how everything works.
- When you encounter a concrete design situation covered by a pattern there is no need to waste time inventing a new solution to your problem.
- If you implement the pattern correctly, you can rely on the solution it provides.
- The Model-View-Controller pattern, for example, helps you to separate the different user interface aspects of a software system and provide appropriate abstractions for them.

Definition: Pattern

A pattern for software architecture describes a particular recurring design problem that arises in specific design contexts, and presents a well-proven generic scheme for its solution. The solution scheme is specified by describing its constituent components, their responsibilities and relationships, and the ways in which they collaborate.

Categories

From Mud to Structure.

- Patterns in this category help you to avoid a 'sea' of components or objects.
- In particular, they support a controlled decomposition of an overall system task into cooperating subtasks.
- The category includes
  - the Layers pattern
  - the Pipes and Filters pattern
  - the Blackboard pattern

Distributed Systems.

- This category includes one pattern.
  - Broker
- and refers to two patterns in other categories,
  - Microkernel
  - Pipes and Filters
- The Broker pattern provides a complete infrastructure for distributed applications.
- The Microkernel and Pipes and Filters patterns only consider distribution as a secondary concern and are therefore listed under their respective primary categories.

Interactive Systems.

- This category comprises two patterns,
  - the Model-View-Controller pattern (well-known from Smalltalk,)
  - the Presentation-Abstraction-Control pattern.
- Both patterns support the structuring of software systems that feature human-computer interaction.

Adaptable Systems.

- This category includes
  - The Reflection pattern
  - the Microkernel pattern
- strongly support extension of applications and their adaptation to evolving technology and changing functional requirements.

From Mud to Structure: Layers

From Mud to Structure: Layers

Context:

All complex systems experience the need to develop and evolve portions of the system independently. For this reason the developers of the system need a clear and well-documented separation of concerns, so that modules of the system may be independently developed and maintained.

From Mud to Structure: Layers

Problem:

The software needs to be segmented in such a way that the modules can be developed and evolved separately with little interaction among the parts, supporting portability, modifiability, and reuse.

From Mud to Structure: Layers

Solution:

To achieve this separation of concerns, the layered pattern divides the software into units called layers. Each layer is a grouping of modules that offers a cohesive set of services. The usage must be unidirectional. Layers completely partition a set of software, and each partition is exposed through a public interface.

Layer: Diagramatical Representation

BITS Pilani, Deemed to be University under Section 3 of UGC Act, 1956

A Typical Solution

Overview:

The layered pattern defines layers (groupings of modules that offer a cohesive set of services) and a unidirectional *allowed-to-use* relation among the layers.

Elements:

Layer, a kind of module. The description of a layer should define what modules the layer contains.

Relations:

Allowed to use. The design should define what the layer usage rules are and any allowable exceptions.

Constraints:

- Every piece of software is allocated to exactly one layer.
- There are at least two layers (but usually there are three or more).
- The *allowed-to-use* relations should not be circular (i.e., a lower layer cannot use a layer above).

Weaknesses:

- The addition of layers adds up-front cost and complexity to a system.
- Layers contribute a performance penalty.

BITS Pilani

Pilani | Dubai | Goa | Hyderabad

Module 6: CS 07

Patterns – Part 1 Suppli 2 Mud to Structures: Layer/Pipes and Filters/BlackBoard

Harvinder S Jabbal
SSZG653 Software Architectures

Categories

From Mud to Structure.

- Patterns in this category help you to avoid a 'sea' of components or objects.
- In particular, they support a controlled decomposition of an overall system task into cooperating subtasks.
- The category includes
  - the Layers pattern
  - the Pipes and Filters pattern
  - the Blackboard pattern

Distributed Systems.

- This category includes one pattern.
  - Broker
- and refers to two patterns in other categories,
  - Microkernel
  - Pipes and Filters
- The Broker pattern provides a complete infrastructure for distributed applications.
- The Microkernel and Pipes and Filters patterns only consider distribution as a secondary concern and are therefore listed under their respective primary categories.

Interactive Systems.

- This category comprises two patterns,
  - the Model-View-Controller pattern (well-known from Smalltalk,)
  - the Presentation-Abstraction-Control pattern.
- Both patterns support the structuring of software systems that feature human-computer interaction.

Adaptable Systems.

- This category includes
  - The Reflection pattern
  - the Microkernel pattern
- strongly support extension of applications and their adaptation to evolving technology and changing functional requirements.

From Mud to Structure

- the Layers pattern
- the Pipes and Filters pattern
- the Blackboard pattern

From Mud to Structure: Layers

Layers- OSI 7 Layer Model

Implementation Steps

1. Define the abstraction criterion for grouping tasks into layers.
2. Determine the number of abstraction levels according to your abstraction criterion.
3. Name the layers and assign tasks to each of them.
4. Specify the services.
5. Refine the layering.
6. Specify an interface for each layer.
7. Structure individual layers.
8. Specify the communication between adjacent layers.
9. Decouple adjacent layers.
10. Design an error-handling strategy.

Variant of OSI: TCP/IP

Variant: Relaxed Layered System

- This is a variant of the Layers pattern that is less restrictive about the relationship between layers.
- In a Relaxed Layered System each layer may use the services of all layers below it, not only of the next lower layer.
- A layer may also be partially opaque- this means that some of its services are only visible to the next higher layer, while others are visible to all higher layers.
- The gain of flexibility and performance in a Relaxed Layered System is paid for by a loss of maintainability.
- This is often a high price to pay, and you should con- sider carefully before giving in to the demands of developers asking for shortcuts.
- We see these shortcuts more often in infrastructure systems, such as the UNIX operating system or the X Window System, than in application software.
- The main reason for this is that infra-structure systems are modified less often than application systems, and their performance is usually more important than their maintainability.

Variant: Layering Through Inheritance.

- This variant can be found in some object-oriented systems.
- In this variant lower layers are implemented as base classes.
- A higher layer requesting services from a lower layer inherits from the lower layer's implementation and hence can issue requests to the base class services.
- An advantage of this scheme is that higher layers can modify lower-layer services according to their needs.
- A drawback is that such an inheritance relationship closely ties the higher layer to the lower layer.
- If for example the data layout of a C++ base class changes, all subclasses must be recompiled.
- Such unintentional dependencies introduced by inheritance are also known as the fragile base class problem.

Known Usages:

Virtual Machines. We can speak of lower levels as a virtual machine that insulates higher levels from low-level details or varying hardware. For example, the Java Virtual Machine (JVM) defines a binary code format. Code written in the Java programming language is translated into a platform-neutral binary code, also called byte- codes, and delivered to the JVM for interpretation. The JVM itself is platform-specific-there are implementations of the JVM for different operating systems and processors. Such a two-step translation process allows platform-neutral source code and the delivery of binary code not readable to humans1, while maintaining platform- independency.

Known Usages:

APIs. An Application Programming Interface is a layer that encapsu- lates lower layers of frequently-used functionality. An API is usually a flat collection of function specifications, such as the UNIX system calls. 'Flat' means here that the system calls for accessing the UNIX file system. These libraries provide the benefit of portabil- ity between different operating systems, and provide additional higher-level services such as output buffering or formatted output. They often carry the liability of lower efficiency2, and perhaps more tightly-prescribed behavior, whereas conventional system calls would give more flexibility-and more opportunities for errors and concep- tual mismatches, mostly due to the wide gap between high-level application abstractions and low-level system calls.

Known Usages:

Information Systems (IS) from the business software domain often use a two-layer architecture. The bottom layer is a database that holds company-specific data. Many applications work concurrently on top of this database to fulfill different tasks. Mainframe interactive systems and the much-extolled Client-Server systems often employ this architecture. Because the tight coupling of user interface and data representation causes its share of problems, a third layer is introduced between them-the domain layer-which models the conceptual structure of the problem domain. As the top level still mixes user interface and application, this level is also split, resulting in a four-layer architecture. These are, from highest to lowest:

Presentation

Application logic

Domain layer

Database

Known Usages:

Windows NT [Cus93]. This operating system is structured according to the Microkernel pattern (171). The NT Executive component corresponds to the microkernel component of the Microkernel pattern. The NT Executive is a Relaxed Layered System, as described in the Variants section. It has the following layers:

System services: the interface layer between the subsystems and the NT Executive.

2. Input/output buffering in Mgher layers is often intended to have the inverse effect-better performance than undisciplined direct use of lower-level system calls.

@ Resource management layer: this contains the modules Object Manager, Security Reference Monitor, Process Manager, 1/0 Manager, Virtual Memory Manager and Local Procedure Calls.

@ Kernel: this takes care of basic functions such as interrupt and ex- ception handling, multiprocessor synchronization, thread schedu- ling and thread dispatching.

@ HAL (Hardware Abstraction Layer): this hides hardware differences between machines of different processor families.

@ Hardware

Windows NT relaxes the principles of the Layers pattern because the Kernel and the 1/0 manager access the underlying hardware directly for reasons of efficiency.

onsequences

Benefits

Benefits

1. Reuse of layers.
2. Support for standardization.
3. Dependencies are kept local

Liabilities

1. Cascades of changing behaviour.
2. Lower efficiency.
3. Unnecessary work.
4. Difiulty of establishing the correct granularity of layers.

Pipes and Filters

Pipes and Filters

The Pipes and Filters architectural pattern provides a structure for systems that process a stream of data. Each processing step is encapsulated in a filter component. Data is passed through pipes between adjacent filters. Recombining filters allows you to build families of related systems.

Pipe and Filter Pattern

Context: Many systems are required to transform streams of discrete data items, from input to output. Many types of transformations occur repeatedly in practice, and so it is desirable to create these as independent, reusable parts.

Problem: Such systems need to be divided into reusable, loosely coupled components with simple, generic interaction mechanisms. In this way they can be flexibly combined with each other. The components, being generic and loosely coupled, are easily reused. The components, being independent, can execute in parallel.

Solution: The pattern of interaction in the pipe-and-filter pattern is characterized by successive transformations of streams of data. Data arrives at a filter's input port(s), is transformed, and then is passed via its output port(s) through a pipe to the next filter. A single filter can consume data from, or produce data to, one or more ports.

Problem

- Imagine you are building a system that must process or transform a stream of input data. Implementing such a system as a single component may not be feasible for several reasons:
  - the system has to be built by several developers,
  - the global system task decomposes naturally into several processing stages, and
  - the requirements are likely to change.
- You therefore plan for future flexibility by exchanging or reordering the processing steps.
- By incorporating such flexibility, it is possible to build a family of systems using existing processing components.

Problem

The design of the system-especially the interconnection of processing steps-has to consider the following forces:

- Future system enhancements should be possible by exchanging processing steps or by recombination of steps, even by users.
- Small processing steps are easier to reuse in different contexts than large components. Non-adjacent processing steps do not share information.
- Different sources of input data exist, such as a network connection or a hardware sensor providing temperature readings, for example. It should be possible to present or store final results in various ways. Explicit storage of intermediate results for further processing in files clutters directories and is error-prone, if done by users.
- You may not want to rule out multi-processing the steps, for example running them in parallel or quasi-parallel.

Pipe and Filter Example

Solution

- The Pipes and Filters architectural pattern divides the task of a system into several sequential processing steps.
- These steps are connected by the data flow through the system-the output data of a step is the input to the subsequent step.
- Each processing step is implemented by a filter component.
- A filter consumes and delivers data incrementally-in contrast to consuming all its input before producing any output-to achieve low latency and enable real parallel processing.
- The input to the system is provided by a data source such as a text file.
- The output flows into a data sink such as a file, terminal, animation program and so on.
- The data source, the filters and the data sink are connected sequentially by pipes.
- Each pipe implements the data flow between adjacent processing steps.
- The sequence of filters combined by pipes is called a processing pipeline.

Pipe and Filter Solution

Overview: Data is transformed from a system's external inputs to its external outputs through a series of transformations performed by its filters connected by pipes.

Elements:

- *Filter,* which is a component that transforms data read on its input port(s) to data written on its output port(s).
- *Pipe,* which is a connector that conveys data from a filter's output port(s) to another filter's input port(s). A pipe has a single source for its input and a single target for its output. A pipe preserves the sequence of data items, and it does not alter the data passing through.

Relations: The *attachment* relation associates the output of filters with the input of pipes and vice versa.

Constraints:

- Pipes connect filter output ports to filter input ports.
- Connected filters must agree on the type of data being passed along the connecting pipe.

Another Example

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 fiters.
5. Design the error handling.
6. Set up the processing pipeline.
   - If your system handles a single task you can use a standardized main program that sets up the pipeline and starts processing. This type of system may benefit from a direct- call pipeline, in which the main program calls the active filter to start processing.
   - You can increase flexibility by providing a shell or other end-user facility to set up various pipelines from your set of filter components.
   - Such a shell can support the incremental development of pipelines by allowing intermediate results to be stored in files, and supporting files as pipeline input.
   - You are not restricted to a text-only shell such as those provided by UNIX, and could even develop a graphical environment for visual creation of pipelines using 'drag and drop' interaction.

Variant: Tee and join pipeline systems.

- The single-input slngle-output filter specification of the Pipes and Filters pattern can be varied to allow filters wlth more than one input and/or more than one output.
- Processing can then be set up as a directed graph that can even contain feedback loops.
- The design of such a system, especially one with feedback loops, requires a solid foundation to explain and understand the complete calculation- a rigorous theoretical analysis and specification using formal methods are appropriate, to prove that the system terminates and produces the desired result.

Known Usage: UNIX

- UNIX popularized the Pipes and Filters paradigm.
- The command shells and the availability of many filter programs made this approach to system development popular.
- As a system for software developers, frequent tasks such as program compilation and documentation creation are done by pipelines on a 'traditional' UNIX system.
- The flexibility of UNIX pipes made the operating system a suitable platform for the binary reuse of filter programs and for application integration.

Known Usage: CMS Pipelines

- CMS Pipelines is an extension to the operating system of IBM mainframes to support Pipes and Filters architectures.
- The implementation of CMS pipelines follows the conventions of CMS, and defines a record as the basic data type that can be passed along pipes, instead of a byte or ASCII character.
- CMS Pipelines provides a reuse and integration platform in the same way as UNIX.
- Because the CMS operating system does not use a uniform I/O-model in the same way as UNIX, CMS Pipelines defines device drivers that act as data sources or sinks, allowing the handling of specific I/O-devices within pipelines.

Known Usage: LASSP Tools

- LASSP Tools is a toolset to support numerical analysis and graphics.
- The toolset consists mainly of filter programs that can be combined using UNIX pipes.
- It contains graphical input devices for analog input of numerical data using knobs or sliders, filters for numerical analysis and data extraction, and data sinks that produce animations from numerical data streams.

Benefits

1. No intermediate files necessary, but possible.
2. Flexibility by filter exchange.
3. Flexibility by recombination.
4. Reuse of filter components.
5. Rapid prototyping of pipelines.
6. Egiciemy by parallel processing.

Liabilities

1. Sharing state information is expensive or inflxible.
2. Efficiency gain by parallel processing is often an illusion.
3. Data transformation overhead.
4. Error handling.

Black Board

Blackboard

- The Blackboard architectural pattern is useful for problems for which no deterministic solution strategies are known.
- In Blackboard several specialized subsystems assemble their knowledge to build a possibly partial or approximate solution.
- **CONTEXT**
- An immature domain in which no closed approach to a solution is known or feasible.

Problem

- The Blackboard pattern tackles problems that do not have a feasible deterministic solution for the transformation of raw data into high-level data structures, such as diagrams, tables or English phrases.
- Vision, image recognition, speech recognition and surveillance are examples of domains in which such problems occur.
- They are characterized by a problem that, when decomposed into sub-problems, spans several fields of expertise.
- The solutions to the partial problems require different representations and paradigms.
- In many cases no predetermined strategy exists for how the 'partial problem solvers' should combine their knowledge.
- This is in contrast to functional de-composition, in which several solution steps are arranged so that the sequence of their activation is hard-coded.

forces that influence solutions to these problems

- A complete search of the solution space is not feasible in a reasonable time.
- Since the domain is immature, you may need to experiment with different algorithms for the same subtask. Individual modules should be easily exchangeable.
- There are different algorithms that solve partial problems. Unrelated logics, representations, algorithms, paradigms or domains may be involved.
- An algorithm usually works on the results of other algorithms.
- Uncertain data and approximate solutions are involved.
- Employing disjoint algorithms induces potential parallelism. If possible you should avoid a strictly sequential solution.

Solution

- The idea behind the Blackboard architecture is a collection of independent programs that work cooperatively on a common data structure.
- Each program is specialized for solving a particular part of the overall task, and all programs work together on the solution.
- These specialized programs are independent of each other.
- They do not call each other, nor is there a predetermined sequence for their activation.
- Instead, the direction taken by the system is mainly determined by the current state of progress.
- A central control component evaluates the current state of processing and coordinates the specialized programs.
- This data-directed control regime is referred to as opportunistic problem solving.
- It makes experimentation with different algorithms possible, and allows experimentally-derived heuristics to control processing.
- During the problem-solving process the system works with partial solutions that are combined, changed or rejected.
- Each of these solutions represents a partial problem and a certain stage of its solution.
- The set of all possible solutions is called the solution space, and is organized into levels of abstraction.
- The lowest level of solution consists of an internal representation of the input.
- Potential solutions of the overall system task are on the highest level.
- The name 'blackboard was chosen because it is reminiscent of the situation in which human experts sit in front of a real blackboard and work together to solve a problem.
- Each expert separately evaluates the current state of the solution, and may go up to the blackboard at any time and add, change or delete information.
- Humans usually decide themselves who has the next access to the blackboard.
- In the pattern we describe, a moderator component decides the order in which programs execute if more than one can make a contribution.

Structure

- Divide your system into a component called blackboard, a collection of knowledge sources, and a control component.
- The blackboard is the central data store. Elements of the solution space and control data are stored here.
- We use the term vocabulary for the set of all data elements that can appear on the blackboard.
- The blackboard provides an interface that enables all knowledge sources to read from and write to it.
- All elements of the solution space can appear on the blackboard.
- For solutions that are constructed during the problem solving process and put on the blackboard, we use the terms hypothesis or blackboard entry.
- Hypotheses rejected later in the process are removed from the blackboard.

Components

- **The blackboard** can be viewed as a three-dimensional problem space with the time line for speech on the X-axis, increasing levels of abstraction on the Y-axis and alternative solutions on the Z-axis
- **Knowledge sources** are separate, independent subsystems that solve specific aspects of the overall problem. Together they model the overall problem domain. None of them can solve the task of the system alone-a solution can only be built by integrating the results of several knowledge sources.
- The **control component** runs a loop that monitors the changes on the blackboard and decides what action to take next. It schedules knowledge source evaluations and activations according to a knowledge application strategy. The basis for this strategy is the data on the blackboard.

Structure

Class	Collaborators
Blackboard	-
Responsibility • Manages central data	

Class	Collaborator
Knowledge Source	• Blackboard
Responsibility • Evaluates its own applicability • Computes a result • Updates Black-board	

Class	Collaborators
Control	• Blackboard • Knowledge Source
Responsibility • Monitors Black-board • Schedules Know-ledge Source acti-vations	

Steps in implementing Blackboard

1. Define the problem.
2. Define the solution spacefor the problem.
3. Divide the solution process into steps.
4. Divide the knowledge into specialized knowledge sources with certain subtasks.
5. Define the vocabulary of the blackboard.
6. Specify the control ofthe system.

Variant: Production System

- This architecture is used in the OPS language.
- In this variant subroutines are represented as condition-action rules, and data is globally available in working memory.
- Condition-action rules consist of a left-hand side that specifies a condition, and a right-hand side that specifies an action.
- The action is executed only if the condition is satisfied and the rule is selected.
- The selection is made by a 'conflict resolution module'.
- A Blackboard system can be regarded as a radical extension of the original production system formalism: arbitrary programs are allowed for both sides of the rules, and the internal complexity of the working memory is increased.
- Complicated scheduling algorithms are used for conflict-resolution.

Variant: Repository.

- This variant is a generalization of the Blackboard pattern.
- The central data structure of this variant is called a repository.
- In a Blackboard architecture the current state of the central data structure, in conjunction with the Control component, finally activates knowledge sources.
- In contrast, the Repository pattern does not specify an internal control.
- A repository architecture may be controlled by user input or by an external program.
- A traditional database, for example, can be considered as a repository.
- Application programs working on the database correspond to the knowledge sources in the Blackboard architecture

Known Usages

HEARSAY-11.

- The first Blackboard system was the HEARSAY-I1 speech recognition system from the early 1970's. It was developed as a natural language interface to a literature database.

HASP/SIAP.

- The HASP system was designed to detect enemy submarines. In this system, hydrophone arrays monitor a sea area by collecting sonar signals.

CRYSALIS.

- This system was designed to infer the three-dimensional structure of protein molecules from X-ray diffraction data

TRICERO.

- This system monitors aircraft activities

SUS: 'Software Understanding System

- In a matching process the system compares patterns from a pattern base to the system under analysis.
- SUS incrementally builds a 'pattern map' of the analyzed software that then can be viewed.

Concusion

Thank You

Credits:

Text Books

BITS Pilani

Pilani | Dubai | Goa | Hyderabad

Module 6 CS07

Patterns – Part 1 Suppl 3

Distributed Systems Brokers/Bridge

Harvinder S Jabbal
SSZG653 Software Architectures

Categories

From Mud to Structure.

- Patterns in this category help you to avoid a 'sea' of components or objects.
- In particular, they support a controlled decomposition of an overall system task into cooperating subtasks.
- The category includes
  - the Layers pattern
  - the Pipes and Filters pattern
  - the Blackboard pattern

Distributed Systems.

- This category includes one pattern.
  - Broker
- and refers to two patterns in other categories,
  - Microkernel
  - Pipes and Filters
- The Broker pattern provides a complete infrastructure for distributed applications.
- The Microkernel and Pipes and Filters patterns only consider distribution as a secondary concern and are therefore listed under their respective primary categories.

Interactive Systems.

- This category comprises two patterns,
  - the Model-View-Controller pattern (well-known from Smalltalk,)
  - the Presentation-Abstraction-Control pattern.
- Both patterns support the structuring of software systems that feature human-computer interaction.

Adaptable Systems.

- This category includes
  - The Reflection pattern
  - the Microkernel pattern
- strongly support extension of applications and their adaptation to evolving technology and changing functional requirements.

Distributed Systems

- Broker
  - The Microkernel and Pipes and Filters patterns only consider distribution as a secondary concern and are therefore listed under their respective primary categories.

Distributed Systems: Broker

Broker

- The Broker architectural pattern can be used to
  - structure distributed software systems
  - with decoupled components
  - that interact by remote service invocations.
- A broker component is responsible
  - for coordinating communication,
  - such as forwarding requests,
  - as well as for transmitting results
  - and exceptions.

Context

- Your environment is a
  - distributed and
  - possibly heterogeneous system with
    - independent
    - cooperating components

Problem

- Building a complex software system as a set of decoupled and inter-operating components, rather than as a monolithic application, results in greater flexibility, maintainability and changeability.
- By partitioning functionality into independent components the system becomes potentially distributable and scalable.
- However, when distributed components communicate with each other, some means of inter-process communication is required.
- If components handle communication themselves, the resulting system faces several dependencies and limitations.

Example

- The system becomes dependent on the communication mechanism used, clients need to know the location of servers, and in many cases the solution is limited to only one programming language.
- Services for adding, removing, exchanging, activating and locating components are also needed.
- Applications that use these services should not depend on system-specific details to guarantee portability and interoperability, even within a heterogeneous network.

developer's viewpoint

- There should essentially be no difference between developing software for centralized systems and developing for distributed ones.
- An application that uses an object should only see the interface offered by the object.
- It should not need to know anything about the implementation details of an object, or about its physical location.

Forces balanced by Broker Architecture

- Components should be able to access services provided by others through remote, location-transparent service invocations.
- You need to exchange, add, or remove components at run-time.
- The architecture should hide system and implementation-specific details from the users of components and services.

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.
- A broker's tasks include locating the appropriate server, forwarding the request to the server and transmitting results and exceptions back to the client.

How an Application accesses distributed services

- By using the Broker pattern, an application can access distributed services simply by sending message calls to the appropriate object, instead of focusing on low-level inter-process communication.
- In addition, the Broker architecture is flexible, in that it allows dynamic change, addition, deletion, and relocation of objects.

How it works

- The Broker pattern reduces the complexity involved in developing distributed applications, because it makes distribution transparent to the developer.
- It achieves this goal by introducing an object model in which distributed services are encapsulated within objects.
- Broker systems therefore offer a path to the integration of two core technologies:
    - distribution and object technology.
- They also extend object models from single applications to distributed applications consisting of decoupled components that can run on heterogeneous machines and that can be written in different programming languages.

Structure

- The Broker architectural pattern comprises six types of participating components:
    - Clients,
    - servers,
    - brokers,
    - bridges,
    - client-side proxies and
    - server-side proxies.

Server

- A Server implements objects that expose their functionality through interfaces that consist of operations and attributes.
- These interfaces are made available either through an interface definition language (IDL) or through a binary standard.
- Interfaces typically group semantically-related functionality.
- There are two kinds of servers:
    - Servers offering common services to many application domains.
    - Servers implementing specific functionality for a single application domain or task.

CRC diagram

Class	Collaborators
Server	• Server-side Proxy • Broker
Responsibility • Implements services. • Registers itself with the local broker. • Sends responses and exceptions back to the client through a server-side proxy.	

Server: Illustration

- WWW servers that provide access to HTML (Hypertext Markup Language) pages.
- WWW servers are implemented as http daemon processes (hypertext transfer protocol daemon) that wait on specific ports for incoming requests.
- When a request arrives at the server, the requested document and any additional data is sent to the client using data streams.
- The HTML pages contain documents as well as CGI (Common Gateway interface) scripts for remotely-executed operations on the network host-the remote machine from which the client received the HTML- page.
- A CGI script may be used to allow the user fill out a form and submit a query, for example a search request for vacant hotel rooms.
- To display animations on the client's browser, Java 'applets' are integrated into the HTML documents.
- For example, one of these Java applets animates the route between one place and another on a city map.
- Java applets run on top of a virtual machine that is part of the WWW browser.
- CGI scripts and Java applets differ from each other: CGI scripts are executed on the server machine, whereas Java applets are transferred to the WWW browser and then executed on the client machine.

Client

- Clients are applications that access the services of at least one server.
- To call remote services, clients forward requests to the broker.
- After an operation has executed they receive responses or exceptions from the broker.
- The interaction between clients and servers is based on a dynamic model, which means that servers may also act as clients.
- This dynamic interaction model differs from the traditional notion of Client-Server computing in that the roles of clients and servers are not statically defined.
- From the viewpoint of an implementation, you can consider clients as applications and servers as libraries-though other implementations are possible.
- Note that clients do not need to know the location of the servers they access.
- This is important, because it allows the addition of new services and the movement of existing services to other locations, even while the system is running.

CRC Diagram

Class	Collaborators
Client	• Client-side Proxy • Broker
Responsibility • Implements user functionality. • Sends requests to servers through a client-side proxy.	

Client: Illustration

- In the context of the Broker pattern, the clients are the available WWW browsers.
- They are not directly connected to the network. Instead, they rely on Internet providers that offer gateways to the Internet, such as vsnl.
- WWW browsers connect to these workstations, using either a modem or a leased line.
- When connected they are able to retrieve data streams from httpd servers, interpret this data and initiate actions such as the display of documents on the screen or the execution of Java applets.

Broker

- A broker is a messenger that is responsible for the transmission of requests from clients to servers, as well as the transmission of responses and exceptions back to the client.
- A broker must have some means of locating the receiver of a request based on its unique system identifier. A broker offers APIs (Application Programming Interfaces) to clients and servers that include operations for registering servers and for invoking server methods.
- When a request arrives for a server that is maintained by the local broker, the broker passes the request directly to the server.
- If the server is currently inactive, the broker activates it.
- All responses and exceptions from a service execution are forwarded by the broker to the client that sent the request.
- If the specified server is hosted by another broker, the local broker finds a route to the remote broker and forwards the request using this route.
- There is therefore a need for brokers to interoperate.
- Depending on the requirements of the whole system, additional services-such as name services or marshalling services may be integrated into the broker.

CRC Diagram

Class	Collaborators
Broker	• Client • Server • Client-side Proxy • Server-side Proxy • Bridge
Responsibility • (Un-)registers servers. • Offers APIs. • Transfers messages. • Error recovery. • Interoperates with other brokers through bridges. • Locates servers.	

Some Definitions

- In this pattern description we distinguish between local and remote brokers.
  - A local broker is running on the machine currently under consideration.
  - A remote broker is running on a remote network node.

- Name services provide associations between names and objects.
  - To resolve a name, a name service determines which server is associated with a given name.
  - In the context of Broker systems, names are only meaningful relative to a name space.

- Marshalling is the semantic-invariant conversion of data into a machine independent format such as
  - ASN.1 (Abstract Syntax Notation or
  - ONC XDR [external Data Representation).
- Marshalling performs the reverse transformation.

Broker: Illustration

- A broker may be a combination of an Internet gateway and the Internet infrastructure itself.
- Every information exchange between a client and a server must pass through the broker.
- A client specifies the information it wants using unique identifiers called URLs (Universal Resource Locators).
- By using these identifiers the broker is able to locate the required services and to route the requests to the appropriate server machines.
- When a new server machine is added, it must be registered with the broker.
- Clients and servers use the gateway of their Internet provider as an interface to the broker.

Client-side proxies

- Client-side proxies represent a layer between clients and the broker.
- This additional layer provides transparency, in that a remote object appears to the client as a local one.
- In detail, the proxies allow the hiding of implementation details from the clients such as:
    - The inter-process communication mechanism used for message transfers between clients and brokers.
    - The creation and deletion of memory blocks.
    - The marshalling of parameters and results.
- In many cases, client-side proxies translate the object model specified as part of the Broker architectural pattern to the object model of the programming language used to implement the client.

CRC Diagram

Class	Collaborators
Client-side Proxy	• Client • Broker
Responsibility • Encapsulates system-specific functionality. • Mediates between the client and the broker.	

Server Side Proxy

- Sewer-side proxies are generally analogous to Client-side proxies.
- The difference is that they are responsible for receiving requests, unpacking incoming messages, marshalling the parameters, and calling the appropriate service.
- They are used in addition for marshalling results and exceptions before sending them to the client.

CRC Diagram

Class	Collaborators
Server-side Proxy	• Server • Broker
Responsibility • Calls services within the server. • Encapsulates system-specific functionality. • Mediates between the server and the broker.	

Bridges

- Bridges are optional components used for hiding implementation details when two brokers interoperate.
- Suppose a Broker system runs on a heterogeneous network.
- If requests are transmitted over the network, different brokers have to communicate independently of the different network and operating systems in use.
- A bridge builds a layer that encapsulates all these system-specific details.

CRC Diagram

Class	Collaborators
Bridge	• Broker • Bridge
Responsibility • Encapsulates network-specific functionality. • Mediates between the local broker and the bridge of a remote broker.	

Broker System

Implementation

1. Define an object model, or use an exlsling model
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. (Note the steps involved)
6. Develop IDL compilers.

Variants

- Direct Communication Broker System
- Message Passing Broker System
- Trader System
- Adapter Broker System
- Callback Broker System

Known Usages

- CORBA
- IBM SOM/DSOM
- Microsoft's OLE 2.x
- World Wide Web
- ATM-P.

Consequences: benefits

- Location Transparency
- Changeability and extensibility of components
- Portability of a Broker system
- Interoperability between dzflerent Broker systems.
- Reusability
- Testing and Debugging

Consequences: Liabilities

- Restricted efficiency
- Lower fault tolerance
- Testing and Debugging

BITS Pilani

Pilani | Dubai | Goa | Hyderabad

Module 6 CS 07

Patterns – Part 2 MVC/SOA

Harvinder S Jabbal
SSZG653 Software Architectures

Model View Controller

Model-View-Controller Pattern

Context: User interface software is typically the most frequently modified portion of an interactive application. Users often wish to look at data from different perspectives, such as a bar graph or a pie chart. These representations should both reflect the current state of the data.

Problem: How can user interface functionality be kept separate from application functionality and yet still be responsive to user input, or to changes in the underlying application's data? And how can multiple views of the user interface be created, maintained, and coordinated when the underlying application data changes?

Solution: The model-view-controller (MVC) pattern separates application functionality into three kinds of components:

- A model, which contains the application's data
- A view, which displays some portion of the underlying data and interacts with the user
- A controller, which mediates between the model and the view and manages the notifications of state changes

MVC Example

Model
• Encapsulates application state
• Responds to state queries
• Exposes application functionality
• Notifies views of changes
State Query
Change Notification
State Change
View
• Renders the models
• Requests updates from models
• Sends user gestures to controller
• Allows controller to select view
View Selection
User Gestures
Controller
• Defines application behavior
• Maps user actions to model updates
• Selects view for response
• One for each functionality
Method Invocations
Events

MVC Solution - 1

Overview: The MVC pattern breaks system functionality into three components: a model, a view, and a controller that mediates between the model and the view.

Elements:

- The *model* is a representation of the application data or state, and it contains (or provides an interface to) application logic.
- The *view* is a user interface component that either produces a representation of the model for the user or allows for some form of user input, or both.
- The *controller* manages the interaction between the model and the view, translating user actions into changes to the model or changes to the view.

MVC Solution - 2

Relations: The *notifies* relation connects instances of model, view, and controller, notifying elements of relevant state changes.

Constraints:

- There must be at least one instance each of model, view, and controller.
- The model component should not interact directly with the controller.

Weaknesses:

- The complexity may not be worth it for simple user interfaces.
- The model, view, and controller abstractions may not be good fits for some user interface toolkits.

Service Oriented Architecture Pattern

Service Oriented Architecture Pattern

- **Context**: A number of services are offered (and described) by service providers and consumed by service consumers. Service consumers need to be able to understand and use these services without any detailed knowledge of their implementation.

Problem: How can we support interoperability of distributed components running on different platforms and written in different implementation languages, provided by different organizations, and distributed across the Internet?

Solution: The service-oriented architecture (SOA) pattern describes a collection of distributed components that provide and/or consume services.

Service Oriented Architecture Example

Service Oriented Architecture Solution - 1

Overview: Computation is achieved by a set of cooperating components that provide and/or consume services over a network.

Elements:

- Components:
  - *Service providers,* which provide one or more services through published interfaces.
  - *Service consumers,* which invoke services directly or through an intermediary.
  - *Service providers* may also be service consumers.
- *ESB,* which is an intermediary element that can route and transform messages between service providers and consumers.
- *Registry of services,* which may be used by providers to register their services and by consumers to discover services at runtime.
- *Orchestration server,* which coordinates the interactions between service consumers and providers based on languages for business processes and workflows.

Service Oriented Architecture Solution - 2

– Connectors:

- *SOAP connector,* which uses the SOAP protocol for synchronous communication between web services, typically over HTTP.
- *REST connector,* which relies on the basic request/reply operations of the HTTP protocol.
- *Asynchronous messaging connector,* which uses a messaging system to offer point-to-point or publish-subscribe asynchronous message exchanges.

Service Oriented Architecture Solution - 3

Relations: Attachment of the different kinds of components available to the respective connectors

Constraints: Service consumers are connected to service providers, but intermediary components (e.g., ESB, registry, orchestration server) may be used.

Weaknesses:

- SOA-based systems are typically complex to build.
- You don’t control the evolution of independent services.
- There is a performance overhead associated with the middleware, and services may be performance bottlenecks, and typically do not provide performance guarantees.

BITS Pilani

Pilani | Dubai | Goa | Hyderabad

Module 6 CS 07

Patterns – Part 2 Suppli

Interactive Systems – MVC/PAC

Harvinder S Jabbal
SSZG653 Software Architectures

Categories

From Mud to Structure.

- Patterns in this category help you to avoid a 'sea' of components or objects.
- In particular, they support a controlled decomposition of an overall system task into cooperating subtasks.
- The category includes
  - the Layers pattern
  - the Pipes and Filters pattern
  - the Blackboard pattern

Distributed Systems.

- This category includes one pattern.
  - Broker
- and refers to two patterns in other categories,
  - Microkernel
  - Pipes and Filters
- The Broker pattern provides a complete infrastructure for distributed applications.
- The Microkernel and Pipes and Filters patterns only consider distribution as a secondary concern and are therefore listed under their respective primary categories.

Interactive Systems.

- This category comprises two patterns,
  - the Model-View-Controller pattern (well-known from Smalltalk,)
  - the Presentation-Abstraction-Control pattern.
- Both patterns support the structuring of software systems that feature human-computer interaction.

Adaptable Systems.

- This category includes
  - The Reflection pattern
  - the Microkernel pattern
- strongly support extension of applications and their adaptation to evolving technology and changing functional requirements.

Interactive Systems

- This category comprises two patterns,
  - the Model-View-Controller pattern

    (well-known from Smalltalk,)
  - the Presentation-Abstraction-Control pattern.
- Both patterns support the structuring of software systems that feature human-computer interaction.

Interactive Systems

User Interaction

- Today's systems allow a high degree of user interaction, mainly achieved with help of graphical user interfaces.
- The objective is to enhance the usability of an application.
- Usable software systems provide convenient access to their services, and therefore allow users to learn the application and produce results quickly.

user interface

- When specifying the architecture of such systems, the challenge is to keep the functional core independent of the user interface.
- The core of interactive systems is based on the functional requirements for the system, and usually remains stable.
- User interfaces, however, are often subject to change and adaptation.
- For example, systems may have to support different user interface standards, customer-specific 'look and feel' metaphors, or interfaces that must be adjusted to fit into a customer's business processes.
- This requires architectures that support the adaptation of user interface parts without causing major effects to application-specific functionality or the data model underlying the software.

Model-View-Controller pattern (MVC)

- MVC divides an interactive application into three components.
    - The model contains the core functionality and data.
    - Views display information to the user.
    - Controllers handle user input.
- Views and controllers together comprise the user interface.
- A change-propagation mechanism ensures consistency between the user interface and the model.

The Presentation-Abstraction-Control pattern (PAC)

- PAC defines a structure for interactive software systems in the form of a hierarchy of cooperating agents.
- Every agent is responsible for a specific aspect of the application's functionality and consists of three components:
    - presentation,
    - abstraction, and
    - control.
- This subdivision separates the human-computer interaction aspects of the agent from its functional core and its communication with other agents.

Model-View-Controller

Model-View-Controller

- The MVC divides an interactive application into three components.
    - The model contains the core functionality and data.
    - Views display information to the user.
    - Controllers handle user input.
- Views and controllers together comprise the user interface.
- A change-propagation mechanism ensures consistency between the user interface and the model.

Context

- Interactive applications with a flexible human-computer interface.

Problem

- User interfaces are especially prone to change requests. When you extend the functionality of an application, you must modify menus to access these new functions.
- A customer may call for a specific user interface adaptation, or a system may need to be ported to another platform with a different 'look and feel' standard.
- Even upgrading to a new release of your windowing system can imply code changes.
- The user interface platform of long-lived systems thus represents a moving target.

Problem

- Different users place conflicting requirements on the user interface.
- A typist enters information into forms via the keyboard.
- A manager wants to use the same system mainly by clicking icons and buttons.
- Consequently, support for several user interface paradigms should be easily incorporated.
- Building a system with the required flexibility is expensive and error-prone if the user interface is tightly interwoven with the functional core.
- This can result in the need to develop and maintain several substantially different software systems, one for each user interface implementation.
- Ensuing changes spread over many modules.

Influences

The following forces influence the solution:

- The same information is presented differently in different windows, for example, in a bar or pie chart.
- The display and behaviour of the application must reflect data manipulations immediately.
- Changes to the user interface should be easy, and even possible at run-time.
- Supporting different 'look and feel' standards or porting the user interface should not affect code in the core of the application.

Solution

- Model-View-Controller (MVC] was first introduced in the Smalltalk-80 programming environment.
- MVC divides an interactive application into the three areas:
    - processing,
    - output, and
    - input.

Model

- The model component encapsulates core data and functionality.
- The model is independent of specific output representations or input behaviour.

View

- View components display information to the user. A view obtains the data from the model.
- There can be multiple views of the model.

Controller

- Each view has an associated controller component.
- Controllers receive input, usually as events that encode mouse movement, activation of mouse buttons, or keyboard input.
- Events are translated to service requests for the model or the view.
- The user interacts with the system solely through controllers.

MVC

- The separation of the model from view and controller components allows multiple views of the same model.
- If the user changes the model via the controller of one view, all other views dependent on this data should reflect the changes.
- The model therefore notifies all views whenever its data changes.
- The views in turn retrieve new data from the model and update the displayed information.
- This change- propagation mechanism is described in the Publisher-Subscriber pattern

Structure: Model

- The model component contains the functional core of the application.
- It encapsulates the appropriate data, and exports procedures that perform application-specific processing.
- Controllers call these procedures on behalf of the user.
- The model also provides functions to access its data that are used by view components to acquire the data to be displayed.
- The change-propagation mechanism maintains a registry of the dependent components within the model.
- All views and also selected controllers register their need to be informed about changes.
- Changes to the state of the model trigger the change-propagation mechanism.
- The change-propagation mechanism is the only link between the model and the views and controllers.

CRC Diagram

Class	Collaborators
Model	• View • Controller
Responsibility • Provides functional core of the application. • Registers dependent views and controllers. • Notifies dependent components about data changes.	

Structure: View

- View components present information to the user.
- Different views present the information of the model in different ways.
- Each view defines an update procedure that is activated by the change- propagation mechanism.
- When the update procedure is called, a view retrieves the current data values to be displayed from the model, and puts them on the screen.
- During initialization all views are associated with the model, and register with the change-propagation mechanism.
- Each view creates a suitable controller.
- There is a one-to-one relationship between views and controllers.
- Views often offer functionality that allows controllers to manipulate the display.
- This is useful for user-triggered operations that do not affect the model, such as scrolling.

CRC Diagram

Class	Collaborators
View	• Controller • Model
Responsibility • Creates and initializes its associated controller. • Displays information to the user. • Implements the update procedure. • Retrieves data from the model.	

Structure: Controller

- The controller components accept user input as events.
- How these events are delivered to a controller depends on the user interface platform.
- For simplicity, let us assume that each controller implements an event-handling procedure that is called for each relevant event.
- Events are translated into requests for the model or the associated view.
- If the behaviour of a controller depends on the state of the model, the controller registers itself with the change-propagation mechanism and implements an update procedure.
- For example, this is necessary when a change to the model enables or disables a menu entry.

CRC Diagram

Class	Collaborators
Controller	• View • Model
Responsibility • Accepts user input as events. • Translates events to service requests for the model or display requests for the view. • Implements the update procedure, if required.	

Popular Structural Variations

- Object Oriented
  - implementation of MVC would define a separate class for each component
- C++
  - view and controller classes share a common parent that defines the update interface.
- Smalltalk.
  - the class Object dehes methods for both sides of the change- propagation mechanism.
  - A separate class Observer is not needed.

Structure: C++

Observer
update
call update
Model
coreData
setOfObservers
attach(Observer)
detach(Observer)
notify
getData
service
attach
getData
View
myModel
myController
initialize(Model)
makeController
activate
display
update
create
manipulate
display
Controller
myModel
myView
initialize(Model,View)
handleEvent
update
attach
call service

Implementation

1. Separate human-computer interactionfrom corefunctionality
2. Implement the change-propagationm echanism
3. Design and implement the views
4. Design and implement the controllers
5. Design and implement the view-controller relationship
6. Implement the set-up of MVC
7. Dynamic view creation
8. 'Pluggable controllers
9. Infrastructure for hierarchical views and controllers
10. Further decoupling from system dependencies

Variants

- Document-View
  - This variant relaxes the separation of view and controller.
  - In several GUI platforms, window display and event handling are closely interwoven.
  - For example, the X Window System reports events relative to a window.
  - You can combine the responsibilities of the view and the controller from MVC in a single component by sacrificing exchangeability of controllers.

Known Usages

- Smalltalk
  - The VisualWorks Smalltalk environment supports different 'look and feel' standards by decoupling view and controllers via display and sensor classes
- MFC (Microsoft Foundation Class Library)
  - The Document-View variant of the Model-View-Controller pattern is integrated in the Visual C++ environment
- ET++
  - Establishes 'look and feel' independence by defining a class Window port that encapsulates the user interface platform dependencies, in the same way as do our display and sensor classes. Uses document-view variant.

Consequences: Benefits

- Multiple views of the same model
- Synchronized views.
- 'Pluggable' views and controllers.
- Exchangeability of 'look and feel'.
- Framework Potential.

Consequences: Liabilities

- Increased complexity.
- Potential for excessive number of updates.
- Intimate connection between view and controller.
- Close coupling of views and controllers to a model.
- Inefficiency of data access in view.
- Inevitability of change to view and controller when porting.
- Difficulty of using MVC with modern user-interface took.

PAC: Presentation-Abstraction-Control

Presentation-Abstraction-Control

- The Presentation-Abstraction-Control architectural pattern (PAC) defines a structure for interactive software systems in the form of a hierarchy of cooperating agents.
- Every agent is responsible for a specific aspect of the application's functionality and consists of three components: presentation, abstraction, and control.
- This subdivision separates the human-computer interaction aspects of the agent from its functional core and its communication with other agents

Context

- Development of an interactive application with the help of agents.
- In the context of this pattern an agent denotes an information-processing component that includes event receivers and transmitters, data structures to maintain stale, and a processor that handles incoming events, updates its own state, and that may produce new events.
- Agents can be as small as a single object, but also as complex as a complete software system.
- We use the terms agent and PAC agent as synonyms in this pattern description

Problem

- Interactive systems can often be viewed as a set of cooperating agents.
- Agents specialized in human-computer interaction accept user input and display data.
- Other agents maintain the data model of the system and offer functionality that operates on this data.
- Additional agents are responsible for diverse tasks such as error handling or communication with other software systems.
- Besides this horizontal decomposition of system functionality, we often encounter a vertical decomposition.

Example

- Production planning systems (PPS) , for example, distinguish between production planning and the execution of a previously specified production plan.
- For each of these tasks separate agents can be defined.
- In such an architecture of cooperating agents, each agent is specialized for a specific task, and all agents together provide the system functionality.
- This architecture also captures both a horizontal and vertical decomposition.

Forces

- Agents often maintain their own state and data.
  - For example, in a PPS system, the production planning and the actual production control may work on different data models, one tuned for planning and simulation and one performance-optimized for efficient production.
  - However, individual agents must effectively cooperate to provide the overall task of the application.
  - To achieve this, they need a mechanism for exchanging, data, messages, and events.
- Interactive agents provide their own user interface, since their respective human-computer interactions often differ widely.
  - For example, entering data into spreadsheets is done using keyboard input, while the manipulation of graphical objects uses a pointing device.
- Systems evolve over time.
  - Their presentation aspect is particularly prone to change.
  - The use of graphics, and more recently, multi-media features, are examples of pervasive changes to user interfaces.
  - Changes to individual agents, or the extension of the system with new agents, should not affect the whole system.

Solution

- Structure the interactive application as a tree-like hierarchy of PAC agents.
- There should be one top-level agent, several intermediate-level agents, and even more bottom-level agents.
- Every agent is responsible for a specific aspect of the application's functionality, and consists of three components: presentation, abstraction, and control.
- The whole hierarchy reflects transitive dependencies between agents.
- Each agent depends on all higher-level agents up the hierarchy to the top-level agent.

Agents

- The agent's presentation component provides the visible behaviour of the PAC agent.
- Its abstraction component maintains the data model that underlies the agent, and provides functionality that operates on this data.
- Its control component connects the presentation and abstraction components, and provides functionality that allows the agent to communicate with other PAC agents.

The top-level PAC agent

- The top-level PAC agent provides the functional core of the system.
- Most other PAC agents depend or operate on this core. Furthermore, the top-level PAC agent includes those parts of the user interface that cannot be assigned to particular subtasks, such as menu bars or a dialog box displaying information about the application.

Bottom-level PAC agents

- Bottom-level PAC agents represent self-contained semantic concepts on which users of the system can act, such as spreadsheets and charts.
- The bottom-level agents present these concepts to the user and support all operations that users can perform on these agents, such as zooming or moving a chart.

Intermediate-level PAC agents

- Intermediate-level PAC agents represent either combinations of, or relationships between, lower-level agents.
    - For example, an intermediate-level agent may maintain several views of the same data, such as a floor plan and an external view of a house in a CAD system for architecture.

Typical Hierarchy of Agents

Structure: Top Level Agent (component)

- The main responsibility of the top-level PAC agent is to provide the global data model of the software.
- This is maintained in the abstraction component of the top-level agent.
- The interface of the abstraction component offers functions to manipulate the data model and to retrieve information about it.
- The representation of data within the abstraction component is media-independent.
- For example, in a CAD system for architecture, walls, doors, and windows are represented in centimetres or inches that reflect their real size, not in pixels for display purposes.
- This media-independency supports adaptation of the PAC agent to different environments without major changes in its abstraction component.

Structure: Top Level Agent (presentation)

- The presentation component of the top-level agent often has few responsibilities.
- It may include user-interface elements common to the whole application.
- In some systems, such as the network traffic manager, there is no top-level presentation component at all.

Structure: Top Level Agent (control)

- The control component of the top-level PAC agent has three responsibilities:

1
    - It allows lower-level agents to make use of the services of the top-level agents, mostly to access and manipulate the global data model.
    - Incoming service requests from lower-level agents are forwarded either to the abstraction component or the presentation component.

2
    - It coordinates the hierarchy of PAC agents.
    - It maintains information about connections between the top-level agent and lower-level agents.
    - The control component uses this information to ensure correct collaboration and data exchange between the top-level agent and lower-level agents.

3
    - It maintains information about the interaction of the user with the system.
    - For example, it may check whether a particular operation can be performed on the data model when triggered by the user.
    - It may also keep track of the functions called to provide history or undo/redo services for operations on the functional core.

Bottom Level Agent

- Bottom-level PAC agents represent a specific semantic concept of the application domain, such as a mailbox in a network traffic management system or a wall in a mobile robot system.
- This semantic concept may be as low-level as a simple graphical object such as a circle, or as complex as a bar chart that summarizes all the data in the system.

Bottom Level Agent (presentation)

- The presentation component of a bottom-level PAC agent presents a specific view of the corresponding semantic concept, and provides access to all the functions users can apply to it.
- Internally, the presentation component also maintains information about the view, such as its position on the screen.

Bottom Level Agent (abstraction)

- The abstraction component of a bottom-level PAC agent has a similar responsibility as the abstraction component of the top-level PAC agent, maintaining agent-specific data.
- In contrast to the abstraction component of the top-level agent, however, no other PAC agents depend on this data.

Bottom Level Agent (control)

- The control component of a bottom-level PAC agent maintains consistency between the abstraction and presentation components, thereby avoiding direct dependencies between them.
- It serves as an adapter and performs both interface and data adaptation.
- The control component of bottom-level PAC agents communicates with higher-level agents to exchange events and data.
- Incoming events-such as a 'close window' request-are forwarded to the presentation component of the bottom-level agent, while incoming data is forwarded to its abstraction component.
- Outgoing events and data, for example error messages, are sent to the associated higher-level agent.

Bottom Level Agent

- Concepts represented by bottom-level PAC agents, such as the bar and pie charts in the example, are atomic in the sense that they are the smallest units a user can manipulate. The users can only operate on the bar chart as a whole, for instance by changing the scaling factor of the y-axis. They cannot, for example, resize an individual bar of a bar chart.
- Bottom-level PAC agents are not restricted to providing semantic concepts of the application domain. You can also specify bottom-level agents that implement system services.
- For example, there may be a communication agent that allows the system to cooperate with other applications and to monitor this cooperation

Intermediate-Level PAC agents (composition)

- Intermediate-Level PAC agents can fulfil two different roles:
- composition and coordination
- When, for example, each object in a complex graphic is represented by a separate PAC agent, an intermediate-level agent groups these objects to form a composite graphical object.
- The intermediate-level agent defines a new abstraction, whose behaviour encompasses both the behaviour of its components and the new characteristics that are added to the composite object.

Intermediate-Level PAC agents (coordination)

- The second role of an intermediate-level agent is to maintain consistency between lower- level agents, for example when coordinating multiple views of the same data.
- The abstraction component maintains the specific data of the intermediate-level PAC agent. The presentation component implements its user interface. The control component has the same responsibilities of the control components of bottom-level PAC agents and of the top-level PAC agent.

CRC Diagrams

Class	Collaborators
Top-level Agent	• Intermediate-level Agent • Bottom-level Agent
Responsibility • Provides the functional core of the system. • Controls the PAC hierarchy.	

Class	Collaborators
Interm. -level Agent	• Top-level Agent • Intermediate-level Agent • Bottom-level Agent
Responsibility • Coordinates lower-level PAC agents. • Composes lower-level PAC agents to a single unit of higher abstraction.	

Class	Collaborators
Bottom-level Agent	• Top-level Agent • Intermediate-level Agent
Responsibility • Provides a specific view of the software or a system service, including its associated human-computer interaction.	

Typical PAC Object Model

Internal Structure of a PAC Agent

Implementation

1. Define a model of the application.
2. Define a general strategy for organizing the PAC hierarchy
3. Specify the top-level PAC agent.
4. Specify the bottom-level PAC agents
5. Specih bottom-level PAC agents for system services.
6. Specify intermediate-level PAC agents to compose lower-level PAC agents
7. Specify intermediate-level PAC agents to coordinate lower-level PAC agents.
8. Separate core functionality from human-computer interaction
9. Provide the external interface.

Variants

- Many large applications-especially interactive ones-are multi-user systems. Multi-tasking is thus a major concern when designing such software systems.

Variants

- PAC agents as active objects

- PAC agents as processes.

Known Usages

- Network Traffic Management.
- Mobile Robot.

Consequences: benefits

- Separation of concerns
- Support for change and extension
- Support for multi-tasking

Consequences: Liabilities

- Increased system complexity.
- Complex control component
- Applicability.

Thank you

The END

BITS Pilani

Pilani | Dubai | Goa | Hyderabad

Module 6 CS 07

Patterns – Part 1 Layers/ Broker/ Pipes and Filters

Harvinder S Jabbal
SSZG653 Software Architectures

Patterns

Outline

What is a Pattern?

Pattern Catalogue

- Module patterns
  - Layered Pattern
- Component and Connector Patterns
  - Broker Pattern
  - MVC Pattern
  - Pipe-and-Filter Pattern
  - Client Server Pattern
  - Peer-to-Peer Pattern
  - SOA Pattern
  - Publish Subscribe
  - Shared Data Pattern
- Allocation Patterns
  - Map-Reduce Pattern
  - Multi-tier Pattern

Relation Between Tactics and Patterns

Using tactics together

Summary

What is a Pattern?

An architectural pattern establishes a relationship between:

A context. A recurring, common situation in the world that gives rise to a problem.

A problem. The problem, appropriately generalized, that arises in the given context.

A solution. A successful architectural resolution to the problem, appropriately abstracted. The solution for a pattern is determined and described by:

- A set of element types (for example, data repositories, processes, and objects)
- A set of interaction mechanisms or connectors (for example, method calls, events, or message bus)
- A topological layout of the components
- A set of semantic constraints covering topology, element behavior, and interaction mechanisms

Layer Pattern

Context: All complex systems experience the need to develop and evolve portions of the system independently. For this reason the developers of the system need a clear and well-documented separation of concerns, so that modules of the system may be independently developed and maintained.

Problem: The software needs to be segmented in such a way that the modules can be developed and evolved separately with little interaction among the parts, supporting portability, modifiability, and reuse.

Solution: To achieve this separation of concerns, the layered pattern divides the software into units called layers. Each layer is a grouping of modules that offers a cohesive set of services. The usage must be unidirectional. Layers completely partition a set of software, and each partition is exposed through a public interface.

Layer Pattern Example

Layer Pattern Solution

Overview: The layered pattern defines layers (groupings of modules that offer a cohesive set of services) and a unidirectional *allowed-to-use* relation among the layers.

Elements: *Layer*, a kind of module. The description of a layer should define what modules the layer contains.

Relations: *Allowed to use.* The design should define what the layer usage rules are and any allowable exceptions.

Constraints:

- Every piece of software is allocated to exactly one layer.
- There are at least two layers (but usually there are three or more).
- The *allowed-to-use* relations should not be circular (i.e., a lower layer cannot use a layer above).

Weaknesses:

- The addition of layers adds up-front cost and complexity to a system.
- Layers contribute a performance penalty.

Broker Pattern

Broker Pattern

Context: Many systems are constructed from a collection of services distributed across multiple servers. Implementing these systems is complex because you need to worry about how the systems will interoperate—how they will connect to each other and how they will exchange information—as well as the availability of the component services.

Problem: How do we structure distributed software so that service users do not need to know the nature and location of service providers, making it easy to dynamically change the bindings between users and providers?

Solution: The broker pattern separates users of services (clients) from providers of services (servers) by inserting an intermediary, called a broker. When a client needs a service, it queries a broker via a service interface. The broker then forwards the client's service request to a server, which processes the request.

Brok

-transfers
*
1
Broker
1
-transfers
*
Client-Side Proxy
+pack_data()
+unpack_data()
+send_request()
+return()
1
+locateServer()
+locateClient()
+registerServer()
+unregisterServer()
1
Server-Side Proxy
+pack_data()
+unpack_data()
+call_service()
+send_response()
-call
1
*
*
0..1
1
-call
1
-call
-uses
Client
+call_server()
+start_task()
+use_Broker_API()
*
Bridge
+pack_data()
+unpack_data()
+forward_message()
+transmit_message()
*
-uses
Server
+initialize()
+enter_main_loop()
+run_service()
+use_Broker_API()

Broker Solution – 1

Overview: The broker pattern defines a runtime component, called a broker, that mediates the communication between a number of clients and servers.

Elements:

- *Client,* a requester of services
- *Server,* a provider of services
- *Broker,* an intermediary that locates an appropriate server to fulfill a client's request, forwards the request to the server, and returns the results to the client
- *Client-side proxy,* an intermediary that manages the actual communication with the broker, including marshaling, sending, and unmarshaling of messages
- *Server-side proxy,* an intermediary that manages the actual communication with the broker, including marshaling, sending, and unmarshaling of messages

Broker Solution - 2

Relations: The *attachment* relation associates clients (and, optionally, client-side proxies) and servers (and, optionally, server-side proxies) with brokers.

Constraints: The client can only attach to a broker (potentially via a client-side proxy). The server can only attach to a broker (potentially via a server-side proxy).

Weaknesses:

- Brokers add a layer of indirection, and hence latency, between clients and servers, and that layer may be a communication bottleneck.
- The broker can be a single point of failure.
- A broker adds up-front complexity.
- A broker may be a target for security attacks.
- A broker may be difficult to test.

Pipe and Filter Pattern

Pipe and Filter Pattern

Context: Many systems are required to transform streams of discrete data items, from input to output. Many types of transformations occur repeatedly in practice, and so it is desirable to create these as independent, reusable parts.

Problem: Such systems need to be divided into reusable, loosely coupled components with simple, generic interaction mechanisms. In this way they can be flexibly combined with each other. The components, being generic and loosely coupled, are easily reused. The components, being independent, can execute in parallel.

Solution: The pattern of interaction in the pipe-and-filter pattern is characterized by successive transformations of streams of data. Data arrives at a filter's input port(s), is transformed, and then is passed via its output port(s) through a pipe to the next filter. A single filter can consume data from, or produce data to, one or more ports.

Pipe and Filter Pattern

Pipe and Filter Solution

Overview: Data is transformed from a system's external inputs to its external outputs through a series of transformations performed by its filters connected by pipes.

Elements:

- *Filter,* which is a component that transforms data read on its input port(s) to data written on its output port(s).
- *Pipe,* which is a connector that conveys data from a filter's output port(s) to another filter's input port(s). A pipe has a single source for its input and a single target for its output. A pipe preserves the sequence of data items, and it does not alter the data passing through.

Relations: The *attachment* relation associates the output of filters with the input of pipes and vice versa.

Constraints:

- Pipes connect filter output ports to filter input ports.
- Connected filters must agree on the type of data being passed along the connecting pipe.

Module 6: CS 07

Patterns – Part 1 Supl 1

Layers

Harvinder S Jabbal
SSZG653 Software Architectures

Introducing Patterns

Patterns

An architectural pattern establishes a relationship between:

- ✓ *A context*.
- ✓ *A problem.*
- ✓ *A solution.*

Context

A context.

A recurring, common situation in the world that gives rise to a problem.

Problem

A problem.

The problem, appropriately generalized, that arises in the given context.

Solution

A solution.

A successful architectural resolution to the problem, appropriately abstracted. The solution for a pattern is determined and described by:

- A set of element types (for example, data repositories, processes, and objects)
- A set of interaction mechanisms or connectors (for example, method calls, events, or message bus)
- A topological layout of the components
- A set of semantic constraints covering topology, element behavior, and interaction mechanisms

Properties of Patterns

- Problem…Context…Solution
- Existing, well-proven design experience
- Identify and Specify Abstractions
- Provide Common Vocabulary
- Means of documenting Software Architecture
- Support construction with defined properties
- Help build complex and heterogenous architectures
- Help to manage software complexity

Problem…Context…Solution

- A pattern addresses a recurring design problem that arises in specific design situations, and presents a solution to it.

- If the problem is supporting variability in user interfaces. This problem may arise when developing software systems with human-computer interaction. You can solve this problem by a strict separation of responsibilities: the core functionality of the application is separated from its user interface.

Existing, well-proven design experience

- Patterns document existing, well-proven design experience.
- They are not invented or created artificially.
- Rather they 'distill and provide a means to reuse the design knowledge gained by experienced practitioners
- Those familiar with an adequate set of patterns 'can apply them immediately to design problems without having to rediscover them'
- Instead of knowledge existing only in the heads of a few experts, patterns make it more generally available.
- You can use such expert knowledge to design high-quality software for a specific task.
- The Model-View-Controller pattern, for example, presents experience gained over many years of developing interactive systems. Many well-known applications already apply the Model-View-Controller pattern-it is the classical architecture for many Smalltalk applications, and under several application frameworks such as MacApp/Windows Apps.

Identify and Specify Abstractions

- Patterns identify and specify abstractions that are above the level of single classes and instances, or of components.
- Typically, a pattern describes several components, classes or objects, and details their responsibilities and relationships, as well as their cooperation.
- All components together solve the problem that the pattern addresses, and usually more effectively than a single component.
- Example, the Model-View-Controller pattern describes a triad of three cooperating components, and each MVC triad also cooperates with other MVC triads of the system.

Provide Common Vocabulary

- Patterns provide a common vocabulary and understanding for design principles.
- Pattern names, if chosen carefully, become part of a widespread design language.
- They facilitate effective discussion of design problems and their solutions.
- They remove the need to explainn a solution to a particular problem with a lengthy and complicated description.
- Instead you can use a pattern name, and explain which parts of a solution correspond to which components of the pattern, or to which relationships between them.
- Example, the name 'Model-View-Controller' and the associated pattern has been to the Smalltalk community since the early '80s, and is used by many software engineers. When we say 'the architecture of the software follows Model-View-Controller', all our colleagues who are familiar with the pattern have an idea of the basic structure and properties of the application immediately.

Means of documenting Software Architecture

- Patterns are a means of documenting software architectures.
- They can describe the vision you have in mind when designing a software system.
- This helps others to avoid violating this vision when extending and modifying the original architecture, or when modifying the system's code.
- Example, if you know that a system is structured according to the Model-View-Controller pattern, you also know how to extend it with a new function: keep core functionality separate from user input and information display.

Support construction with defined properties

- Patterns support the construction of software with defined properties.
- Patterns provide a skeleton of functional behaviour and therefore help to implement the functionality of your application
- Example, patterns exist for maintaining consistency between cooperating components and for providing transparent peer-to-peer inter-process communication. In addition, patterns explicitly address non- functional requirements for software systems, such as changeability, reliability, testability or reusability.
- The Model-View-Controller pattern, for example, supports the changeability of user interfaces and the reusability of core functionality.

Help build complex and heterogenous architectures

- Patterns help you build complex and heterogeneous software architectures.
- Every pattern provides a predefined set of components, roles and relationships between them.
- It can be used to specify particular aspects of concrete software structures.
- Patterns 'act as building-blocks for constructing more complex designs'.
- This method of using predefined design artefacts supports the speed and the quality of your design.
- Understanding and applying well-written patterns saves time when compared to searching for solutions on your own.
- This is not to say that individual patterns will necessarily be better than your own solutions, but, at the very least. a pattern system can help you to evaluate and assess design alternatives.

….cont.

- However, although a pattern determines the basic structure of the solution to a particular design problem, it does not specify a fully detailed solution.
- A pattern provides a scheme for a generic solution to a family of problems, rather than a prefabricated module that can be used 'as is'.
- You must implement this scheme according to the specific needs of the design problem at hand.
- A pattern helps with the creation of similar units.
- These units can be alike in their broad structure, but are frequently quite different in their detailed appearance.
- Patterns help solve problems, but they do not provide complete solutions.

Help to manage software complexity

- Patterns help you to manage software complexity.
- Every pattern describes a proven way to handle the problem it addresses: the kinds of components needed, their roles, the details that should be hidden, the abstractions that should be visible, and how everything works.
- When you encounter a concrete design situation covered by a pattern there is no need to waste time inventing a new solution to your problem.
- If you implement the pattern correctly, you can rely on the solution it provides.
- The Model-View-Controller pattern, for example, helps you to separate the different user interface aspects of a software system and provide appropriate abstractions for them.

Definition: Pattern

A pattern for software architecture describes a particular recurring design problem that arises in specific design contexts, and presents a well-proven generic scheme for its solution. The solution scheme is specified by describing its constituent components, their responsibilities and relationships, and the ways in which they collaborate.

Categories

From Mud to Structure.

- Patterns in this category help you to avoid a 'sea' of components or objects.
- In particular, they support a controlled decomposition of an overall system task into cooperating subtasks.
- The category includes
  - the Layers pattern
  - the Pipes and Filters pattern
  - the Blackboard pattern

Distributed Systems.

- This category includes one pattern.
  - Broker
- and refers to two patterns in other categories,
  - Microkernel
  - Pipes and Filters
- The Broker pattern provides a complete infrastructure for distributed applications.
- The Microkernel and Pipes and Filters patterns only consider distribution as a secondary concern and are therefore listed under their respective primary categories.

Interactive Systems.

- This category comprises two patterns,
  - the Model-View-Controller pattern (well-known from Smalltalk,)
  - the Presentation-Abstraction-Control pattern.
- Both patterns support the structuring of software systems that feature human-computer interaction.

Adaptable Systems.

- This category includes
  - The Reflection pattern
  - the Microkernel pattern
- strongly support extension of applications and their adaptation to evolving technology and changing functional requirements.

From Mud to Structure: Layers

From Mud to Structure: Layers

Context:

All complex systems experience the need to develop and evolve portions of the system independently. For this reason the developers of the system need a clear and well-documented separation of concerns, so that modules of the system may be independently developed and maintained.

From Mud to Structure: Layers

Problem:

The software needs to be segmented in such a way that the modules can be developed and evolved separately with little interaction among the parts, supporting portability, modifiability, and reuse.

From Mud to Structure: Layers

Solution:

To achieve this separation of concerns, the layered pattern divides the software into units called layers. Each layer is a grouping of modules that offers a cohesive set of services. The usage must be unidirectional. Layers completely partition a set of software, and each partition is exposed through a public interface.

Layer: Diagramatical Representation

BITS Pilani, Deemed to be University under Section 3 of UGC Act, 1956

A Typical Solution

Overview:

The layered pattern defines layers (groupings of modules that offer a cohesive set of services) and a unidirectional *allowed-to-use* relation among the layers.

Elements:

Layer, a kind of module. The description of a layer should define what modules the layer contains.

Relations:

Allowed to use. The design should define what the layer usage rules are and any allowable exceptions.

Constraints:

- Every piece of software is allocated to exactly one layer.
- There are at least two layers (but usually there are three or more).
- The *allowed-to-use* relations should not be circular (i.e., a lower layer cannot use a layer above).

Weaknesses:

- The addition of layers adds up-front cost and complexity to a system.
- Layers contribute a performance penalty.

BITS Pilani

Pilani | Dubai | Goa | Hyderabad

Module 6: CS 07

Patterns – Part 1 Suppli 2 Mud to Structures: Layer/Pipes and Filters/BlackBoard

Harvinder S Jabbal
SSZG653 Software Architectures

Categories

From Mud to Structure.

- Patterns in this category help you to avoid a 'sea' of components or objects.
- In particular, they support a controlled decomposition of an overall system task into cooperating subtasks.
- The category includes
  - the Layers pattern
  - the Pipes and Filters pattern
  - the Blackboard pattern

Distributed Systems.

- This category includes one pattern.
  - Broker
- and refers to two patterns in other categories,
  - Microkernel
  - Pipes and Filters
- The Broker pattern provides a complete infrastructure for distributed applications.
- The Microkernel and Pipes and Filters patterns only consider distribution as a secondary concern and are therefore listed under their respective primary categories.

Interactive Systems.

- This category comprises two patterns,
  - the Model-View-Controller pattern (well-known from Smalltalk,)
  - the Presentation-Abstraction-Control pattern.
- Both patterns support the structuring of software systems that feature human-computer interaction.

Adaptable Systems.

- This category includes
  - The Reflection pattern
  - the Microkernel pattern
- strongly support extension of applications and their adaptation to evolving technology and changing functional requirements.

From Mud to Structure

- the Layers pattern
- the Pipes and Filters pattern
- the Blackboard pattern

From Mud to Structure: Layers

Layers- OSI 7 Layer Model

Implementation Steps

1. Define the abstraction criterion for grouping tasks into layers.
2. Determine the number of abstraction levels according to your abstraction criterion.
3. Name the layers and assign tasks to each of them.
4. Specify the services.
5. Refine the layering.
6. Specify an interface for each layer.
7. Structure individual layers.
8. Specify the communication between adjacent layers.
9. Decouple adjacent layers.
10. Design an error-handling strategy.

Variant of OSI: TCP/IP

Variant: Relaxed Layered System

- This is a variant of the Layers pattern that is less restrictive about the relationship between layers.
- In a Relaxed Layered System each layer may use the services of all layers below it, not only of the next lower layer.
- A layer may also be partially opaque- this means that some of its services are only visible to the next higher layer, while others are visible to all higher layers.
- The gain of flexibility and performance in a Relaxed Layered System is paid for by a loss of maintainability.
- This is often a high price to pay, and you should con- sider carefully before giving in to the demands of developers asking for shortcuts.
- We see these shortcuts more often in infrastructure systems, such as the UNIX operating system or the X Window System, than in application software.
- The main reason for this is that infra-structure systems are modified less often than application systems, and their performance is usually more important than their maintainability.

Variant: Layering Through Inheritance.

- This variant can be found in some object-oriented systems.
- In this variant lower layers are implemented as base classes.
- A higher layer requesting services from a lower layer inherits from the lower layer's implementation and hence can issue requests to the base class services.
- An advantage of this scheme is that higher layers can modify lower-layer services according to their needs.
- A drawback is that such an inheritance relationship closely ties the higher layer to the lower layer.
- If for example the data layout of a C++ base class changes, all subclasses must be recompiled.
- Such unintentional dependencies introduced by inheritance are also known as the fragile base class problem.

Known Usages:

Virtual Machines. We can speak of lower levels as a virtual machine that insulates higher levels from low-level details or varying hardware. For example, the Java Virtual Machine (JVM) defines a binary code format. Code written in the Java programming language is translated into a platform-neutral binary code, also called byte- codes, and delivered to the JVM for interpretation. The JVM itself is platform-specific-there are implementations of the JVM for different operating systems and processors. Such a two-step translation process allows platform-neutral source code and the delivery of binary code not readable to humans1, while maintaining platform- independency.

Known Usages:

APIs. An Application Programming Interface is a layer that encapsu- lates lower layers of frequently-used functionality. An API is usually a flat collection of function specifications, such as the UNIX system calls. 'Flat' means here that the system calls for accessing the UNIX file system. These libraries provide the benefit of portabil- ity between different operating systems, and provide additional higher-level services such as output buffering or formatted output. They often carry the liability of lower efficiency2, and perhaps more tightly-prescribed behavior, whereas conventional system calls would give more flexibility-and more opportunities for errors and concep- tual mismatches, mostly due to the wide gap between high-level application abstractions and low-level system calls.

Known Usages:

Information Systems (IS) from the business software domain often use a two-layer architecture. The bottom layer is a database that holds company-specific data. Many applications work concurrently on top of this database to fulfill different tasks. Mainframe interactive systems and the much-extolled Client-Server systems often employ this architecture. Because the tight coupling of user interface and data representation causes its share of problems, a third layer is introduced between them-the domain layer-which models the conceptual structure of the problem domain. As the top level still mixes user interface and application, this level is also split, resulting in a four-layer architecture. These are, from highest to lowest:

Presentation

Application logic

Domain layer

Database

Known Usages:

Windows NT [Cus93]. This operating system is structured according to the Microkernel pattern (171). The NT Executive component corresponds to the microkernel component of the Microkernel pattern. The NT Executive is a Relaxed Layered System, as described in the Variants section. It has the following layers:

System services: the interface layer between the subsystems and the NT Executive.

2. Input/output buffering in Mgher layers is often intended to have the inverse effect-better performance than undisciplined direct use of lower-level system calls.

@ Resource management layer: this contains the modules Object Manager, Security Reference Monitor, Process Manager, 1/0 Manager, Virtual Memory Manager and Local Procedure Calls.

@ Kernel: this takes care of basic functions such as interrupt and ex- ception handling, multiprocessor synchronization, thread schedu- ling and thread dispatching.

@ HAL (Hardware Abstraction Layer): this hides hardware differences between machines of different processor families.

@ Hardware

Windows NT relaxes the principles of the Layers pattern because the Kernel and the 1/0 manager access the underlying hardware directly for reasons of efficiency.

onsequences

Benefits

Benefits

1. Reuse of layers.
2. Support for standardization.
3. Dependencies are kept local

Liabilities

1. Cascades of changing behaviour.
2. Lower efficiency.
3. Unnecessary work.
4. Difiulty of establishing the correct granularity of layers.

Pipes and Filters

Pipes and Filters

The Pipes and Filters architectural pattern provides a structure for systems that process a stream of data. Each processing step is encapsulated in a filter component. Data is passed through pipes between adjacent filters. Recombining filters allows you to build families of related systems.

Pipe and Filter Pattern

Context: Many systems are required to transform streams of discrete data items, from input to output. Many types of transformations occur repeatedly in practice, and so it is desirable to create these as independent, reusable parts.

Problem: Such systems need to be divided into reusable, loosely coupled components with simple, generic interaction mechanisms. In this way they can be flexibly combined with each other. The components, being generic and loosely coupled, are easily reused. The components, being independent, can execute in parallel.

Solution: The pattern of interaction in the pipe-and-filter pattern is characterized by successive transformations of streams of data. Data arrives at a filter's input port(s), is transformed, and then is passed via its output port(s) through a pipe to the next filter. A single filter can consume data from, or produce data to, one or more ports.

Problem

- Imagine you are building a system that must process or transform a stream of input data. Implementing such a system as a single component may not be feasible for several reasons:
  - the system has to be built by several developers,
  - the global system task decomposes naturally into several processing stages, and
  - the requirements are likely to change.
- You therefore plan for future flexibility by exchanging or reordering the processing steps.
- By incorporating such flexibility, it is possible to build a family of systems using existing processing components.

Problem

The design of the system-especially the interconnection of processing steps-has to consider the following forces:

- Future system enhancements should be possible by exchanging processing steps or by recombination of steps, even by users.
- Small processing steps are easier to reuse in different contexts than large components. Non-adjacent processing steps do not share information.
- Different sources of input data exist, such as a network connection or a hardware sensor providing temperature readings, for example. It should be possible to present or store final results in various ways. Explicit storage of intermediate results for further processing in files clutters directories and is error-prone, if done by users.
- You may not want to rule out multi-processing the steps, for example running them in parallel or quasi-parallel.

Pipe and Filter Example

Solution

- The Pipes and Filters architectural pattern divides the task of a system into several sequential processing steps.
- These steps are connected by the data flow through the system-the output data of a step is the input to the subsequent step.
- Each processing step is implemented by a filter component.
- A filter consumes and delivers data incrementally-in contrast to consuming all its input before producing any output-to achieve low latency and enable real parallel processing.
- The input to the system is provided by a data source such as a text file.
- The output flows into a data sink such as a file, terminal, animation program and so on.
- The data source, the filters and the data sink are connected sequentially by pipes.
- Each pipe implements the data flow between adjacent processing steps.
- The sequence of filters combined by pipes is called a processing pipeline.

Pipe and Filter Solution

Overview: Data is transformed from a system's external inputs to its external outputs through a series of transformations performed by its filters connected by pipes.

Elements:

- *Filter,* which is a component that transforms data read on its input port(s) to data written on its output port(s).
- *Pipe,* which is a connector that conveys data from a filter's output port(s) to another filter's input port(s). A pipe has a single source for its input and a single target for its output. A pipe preserves the sequence of data items, and it does not alter the data passing through.

Relations: The *attachment* relation associates the output of filters with the input of pipes and vice versa.

Constraints:

- Pipes connect filter output ports to filter input ports.
- Connected filters must agree on the type of data being passed along the connecting pipe.

Another Example

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 fiters.
5. Design the error handling.
6. Set up the processing pipeline.
   - If your system handles a single task you can use a standardized main program that sets up the pipeline and starts processing. This type of system may benefit from a direct- call pipeline, in which the main program calls the active filter to start processing.
   - You can increase flexibility by providing a shell or other end-user facility to set up various pipelines from your set of filter components.
   - Such a shell can support the incremental development of pipelines by allowing intermediate results to be stored in files, and supporting files as pipeline input.
   - You are not restricted to a text-only shell such as those provided by UNIX, and could even develop a graphical environment for visual creation of pipelines using 'drag and drop' interaction.

Variant: Tee and join pipeline systems.

- The single-input slngle-output filter specification of the Pipes and Filters pattern can be varied to allow filters wlth more than one input and/or more than one output.
- Processing can then be set up as a directed graph that can even contain feedback loops.
- The design of such a system, especially one with feedback loops, requires a solid foundation to explain and understand the complete calculation- a rigorous theoretical analysis and specification using formal methods are appropriate, to prove that the system terminates and produces the desired result.

Known Usage: UNIX

- UNIX popularized the Pipes and Filters paradigm.
- The command shells and the availability of many filter programs made this approach to system development popular.
- As a system for software developers, frequent tasks such as program compilation and documentation creation are done by pipelines on a 'traditional' UNIX system.
- The flexibility of UNIX pipes made the operating system a suitable platform for the binary reuse of filter programs and for application integration.

Known Usage: CMS Pipelines

- CMS Pipelines is an extension to the operating system of IBM mainframes to support Pipes and Filters architectures.
- The implementation of CMS pipelines follows the conventions of CMS, and defines a record as the basic data type that can be passed along pipes, instead of a byte or ASCII character.
- CMS Pipelines provides a reuse and integration platform in the same way as UNIX.
- Because the CMS operating system does not use a uniform I/O-model in the same way as UNIX, CMS Pipelines defines device drivers that act as data sources or sinks, allowing the handling of specific I/O-devices within pipelines.

Known Usage: LASSP Tools

- LASSP Tools is a toolset to support numerical analysis and graphics.
- The toolset consists mainly of filter programs that can be combined using UNIX pipes.
- It contains graphical input devices for analog input of numerical data using knobs or sliders, filters for numerical analysis and data extraction, and data sinks that produce animations from numerical data streams.

Benefits

1. No intermediate files necessary, but possible.
2. Flexibility by filter exchange.
3. Flexibility by recombination.
4. Reuse of filter components.
5. Rapid prototyping of pipelines.
6. Egiciemy by parallel processing.

Liabilities

1. Sharing state information is expensive or inflxible.
2. Efficiency gain by parallel processing is often an illusion.
3. Data transformation overhead.
4. Error handling.

Black Board

Blackboard

- The Blackboard architectural pattern is useful for problems for which no deterministic solution strategies are known.
- In Blackboard several specialized subsystems assemble their knowledge to build a possibly partial or approximate solution.
- **CONTEXT**
- An immature domain in which no closed approach to a solution is known or feasible.

Problem

- The Blackboard pattern tackles problems that do not have a feasible deterministic solution for the transformation of raw data into high-level data structures, such as diagrams, tables or English phrases.
- Vision, image recognition, speech recognition and surveillance are examples of domains in which such problems occur.
- They are characterized by a problem that, when decomposed into sub-problems, spans several fields of expertise.
- The solutions to the partial problems require different representations and paradigms.
- In many cases no predetermined strategy exists for how the 'partial problem solvers' should combine their knowledge.
- This is in contrast to functional de-composition, in which several solution steps are arranged so that the sequence of their activation is hard-coded.

forces that influence solutions to these problems

- A complete search of the solution space is not feasible in a reasonable time.
- Since the domain is immature, you may need to experiment with different algorithms for the same subtask. Individual modules should be easily exchangeable.
- There are different algorithms that solve partial problems. Unrelated logics, representations, algorithms, paradigms or domains may be involved.
- An algorithm usually works on the results of other algorithms.
- Uncertain data and approximate solutions are involved.
- Employing disjoint algorithms induces potential parallelism. If possible you should avoid a strictly sequential solution.

Solution

- The idea behind the Blackboard architecture is a collection of independent programs that work cooperatively on a common data structure.
- Each program is specialized for solving a particular part of the overall task, and all programs work together on the solution.
- These specialized programs are independent of each other.
- They do not call each other, nor is there a predetermined sequence for their activation.
- Instead, the direction taken by the system is mainly determined by the current state of progress.
- A central control component evaluates the current state of processing and coordinates the specialized programs.
- This data-directed control regime is referred to as opportunistic problem solving.
- It makes experimentation with different algorithms possible, and allows experimentally-derived heuristics to control processing.
- During the problem-solving process the system works with partial solutions that are combined, changed or rejected.
- Each of these solutions represents a partial problem and a certain stage of its solution.
- The set of all possible solutions is called the solution space, and is organized into levels of abstraction.
- The lowest level of solution consists of an internal representation of the input.
- Potential solutions of the overall system task are on the highest level.
- The name 'blackboard was chosen because it is reminiscent of the situation in which human experts sit in front of a real blackboard and work together to solve a problem.
- Each expert separately evaluates the current state of the solution, and may go up to the blackboard at any time and add, change or delete information.
- Humans usually decide themselves who has the next access to the blackboard.
- In the pattern we describe, a moderator component decides the order in which programs execute if more than one can make a contribution.

Structure

- Divide your system into a component called blackboard, a collection of knowledge sources, and a control component.
- The blackboard is the central data store. Elements of the solution space and control data are stored here.
- We use the term vocabulary for the set of all data elements that can appear on the blackboard.
- The blackboard provides an interface that enables all knowledge sources to read from and write to it.
- All elements of the solution space can appear on the blackboard.
- For solutions that are constructed during the problem solving process and put on the blackboard, we use the terms hypothesis or blackboard entry.
- Hypotheses rejected later in the process are removed from the blackboard.

Components

- **The blackboard** can be viewed as a three-dimensional problem space with the time line for speech on the X-axis, increasing levels of abstraction on the Y-axis and alternative solutions on the Z-axis
- **Knowledge sources** are separate, independent subsystems that solve specific aspects of the overall problem. Together they model the overall problem domain. None of them can solve the task of the system alone-a solution can only be built by integrating the results of several knowledge sources.
- The **control component** runs a loop that monitors the changes on the blackboard and decides what action to take next. It schedules knowledge source evaluations and activations according to a knowledge application strategy. The basis for this strategy is the data on the blackboard.

Structure

Class	Collaborators
Blackboard	-
Responsibility • Manages central data	

Class	Collaborator
Knowledge Source	• Blackboard
Responsibility • Evaluates its own applicability • Computes a result • Updates Black-board	

Class	Collaborators
Control	• Blackboard • Knowledge Source
Responsibility • Monitors Black-board • Schedules Know-ledge Source acti-vations	

Steps in implementing Blackboard

1. Define the problem.
2. Define the solution spacefor the problem.
3. Divide the solution process into steps.
4. Divide the knowledge into specialized knowledge sources with certain subtasks.
5. Define the vocabulary of the blackboard.
6. Specify the control ofthe system.

Variant: Production System

- This architecture is used in the OPS language.
- In this variant subroutines are represented as condition-action rules, and data is globally available in working memory.
- Condition-action rules consist of a left-hand side that specifies a condition, and a right-hand side that specifies an action.
- The action is executed only if the condition is satisfied and the rule is selected.
- The selection is made by a 'conflict resolution module'.
- A Blackboard system can be regarded as a radical extension of the original production system formalism: arbitrary programs are allowed for both sides of the rules, and the internal complexity of the working memory is increased.
- Complicated scheduling algorithms are used for conflict-resolution.

Variant: Repository.

- This variant is a generalization of the Blackboard pattern.
- The central data structure of this variant is called a repository.
- In a Blackboard architecture the current state of the central data structure, in conjunction with the Control component, finally activates knowledge sources.
- In contrast, the Repository pattern does not specify an internal control.
- A repository architecture may be controlled by user input or by an external program.
- A traditional database, for example, can be considered as a repository.
- Application programs working on the database correspond to the knowledge sources in the Blackboard architecture

Known Usages

HEARSAY-11.

- The first Blackboard system was the HEARSAY-I1 speech recognition system from the early 1970's. It was developed as a natural language interface to a literature database.

HASP/SIAP.

- The HASP system was designed to detect enemy submarines. In this system, hydrophone arrays monitor a sea area by collecting sonar signals.

CRYSALIS.

- This system was designed to infer the three-dimensional structure of protein molecules from X-ray diffraction data

TRICERO.

- This system monitors aircraft activities

SUS: 'Software Understanding System

- In a matching process the system compares patterns from a pattern base to the system under analysis.
- SUS incrementally builds a 'pattern map' of the analyzed software that then can be viewed.

Concusion

Thank You

Credits:

Text Books

BITS Pilani

Pilani | Dubai | Goa | Hyderabad

Module 6 CS07

Patterns – Part 1 Suppl 3

Distributed Systems Brokers/Bridge

Harvinder S Jabbal
SSZG653 Software Architectures

Categories

From Mud to Structure.

- Patterns in this category help you to avoid a 'sea' of components or objects.
- In particular, they support a controlled decomposition of an overall system task into cooperating subtasks.
- The category includes
  - the Layers pattern
  - the Pipes and Filters pattern
  - the Blackboard pattern

Distributed Systems.

- This category includes one pattern.
  - Broker
- and refers to two patterns in other categories,
  - Microkernel
  - Pipes and Filters
- The Broker pattern provides a complete infrastructure for distributed applications.
- The Microkernel and Pipes and Filters patterns only consider distribution as a secondary concern and are therefore listed under their respective primary categories.

Interactive Systems.

- This category comprises two patterns,
  - the Model-View-Controller pattern (well-known from Smalltalk,)
  - the Presentation-Abstraction-Control pattern.
- Both patterns support the structuring of software systems that feature human-computer interaction.

Adaptable Systems.

- This category includes
  - The Reflection pattern
  - the Microkernel pattern
- strongly support extension of applications and their adaptation to evolving technology and changing functional requirements.

Distributed Systems

- Broker
  - The Microkernel and Pipes and Filters patterns only consider distribution as a secondary concern and are therefore listed under their respective primary categories.

Distributed Systems: Broker

Broker

- The Broker architectural pattern can be used to
  - structure distributed software systems
  - with decoupled components
  - that interact by remote service invocations.
- A broker component is responsible
  - for coordinating communication,
  - such as forwarding requests,
  - as well as for transmitting results
  - and exceptions.

Context

- Your environment is a
  - distributed and
  - possibly heterogeneous system with
    - independent
    - cooperating components

Problem

- Building a complex software system as a set of decoupled and inter-operating components, rather than as a monolithic application, results in greater flexibility, maintainability and changeability.
- By partitioning functionality into independent components the system becomes potentially distributable and scalable.
- However, when distributed components communicate with each other, some means of inter-process communication is required.
- If components handle communication themselves, the resulting system faces several dependencies and limitations.

Example

- The system becomes dependent on the communication mechanism used, clients need to know the location of servers, and in many cases the solution is limited to only one programming language.
- Services for adding, removing, exchanging, activating and locating components are also needed.
- Applications that use these services should not depend on system-specific details to guarantee portability and interoperability, even within a heterogeneous network.

developer's viewpoint

- There should essentially be no difference between developing software for centralized systems and developing for distributed ones.
- An application that uses an object should only see the interface offered by the object.
- It should not need to know anything about the implementation details of an object, or about its physical location.

Forces balanced by Broker Architecture

- Components should be able to access services provided by others through remote, location-transparent service invocations.
- You need to exchange, add, or remove components at run-time.
- The architecture should hide system and implementation-specific details from the users of components and services.

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.
- A broker's tasks include locating the appropriate server, forwarding the request to the server and transmitting results and exceptions back to the client.

How an Application accesses distributed services

- By using the Broker pattern, an application can access distributed services simply by sending message calls to the appropriate object, instead of focusing on low-level inter-process communication.
- In addition, the Broker architecture is flexible, in that it allows dynamic change, addition, deletion, and relocation of objects.

How it works

- The Broker pattern reduces the complexity involved in developing distributed applications, because it makes distribution transparent to the developer.
- It achieves this goal by introducing an object model in which distributed services are encapsulated within objects.
- Broker systems therefore offer a path to the integration of two core technologies:
    - distribution and object technology.
- They also extend object models from single applications to distributed applications consisting of decoupled components that can run on heterogeneous machines and that can be written in different programming languages.

Structure

- The Broker architectural pattern comprises six types of participating components:
    - Clients,
    - servers,
    - brokers,
    - bridges,
    - client-side proxies and
    - server-side proxies.

Server

- A Server implements objects that expose their functionality through interfaces that consist of operations and attributes.
- These interfaces are made available either through an interface definition language (IDL) or through a binary standard.
- Interfaces typically group semantically-related functionality.
- There are two kinds of servers:
    - Servers offering common services to many application domains.
    - Servers implementing specific functionality for a single application domain or task.

CRC diagram

Class	Collaborators
Server	• Server-side Proxy • Broker
Responsibility • Implements services. • Registers itself with the local broker. • Sends responses and exceptions back to the client through a server-side proxy.	

Server: Illustration

- WWW servers that provide access to HTML (Hypertext Markup Language) pages.
- WWW servers are implemented as http daemon processes (hypertext transfer protocol daemon) that wait on specific ports for incoming requests.
- When a request arrives at the server, the requested document and any additional data is sent to the client using data streams.
- The HTML pages contain documents as well as CGI (Common Gateway interface) scripts for remotely-executed operations on the network host-the remote machine from which the client received the HTML- page.
- A CGI script may be used to allow the user fill out a form and submit a query, for example a search request for vacant hotel rooms.
- To display animations on the client's browser, Java 'applets' are integrated into the HTML documents.
- For example, one of these Java applets animates the route between one place and another on a city map.
- Java applets run on top of a virtual machine that is part of the WWW browser.
- CGI scripts and Java applets differ from each other: CGI scripts are executed on the server machine, whereas Java applets are transferred to the WWW browser and then executed on the client machine.

Client

- Clients are applications that access the services of at least one server.
- To call remote services, clients forward requests to the broker.
- After an operation has executed they receive responses or exceptions from the broker.
- The interaction between clients and servers is based on a dynamic model, which means that servers may also act as clients.
- This dynamic interaction model differs from the traditional notion of Client-Server computing in that the roles of clients and servers are not statically defined.
- From the viewpoint of an implementation, you can consider clients as applications and servers as libraries-though other implementations are possible.
- Note that clients do not need to know the location of the servers they access.
- This is important, because it allows the addition of new services and the movement of existing services to other locations, even while the system is running.

CRC Diagram

Class	Collaborators
Client	• Client-side Proxy • Broker
Responsibility • Implements user functionality. • Sends requests to servers through a client-side proxy.	

Client: Illustration

- In the context of the Broker pattern, the clients are the available WWW browsers.
- They are not directly connected to the network. Instead, they rely on Internet providers that offer gateways to the Internet, such as vsnl.
- WWW browsers connect to these workstations, using either a modem or a leased line.
- When connected they are able to retrieve data streams from httpd servers, interpret this data and initiate actions such as the display of documents on the screen or the execution of Java applets.

Broker

- A broker is a messenger that is responsible for the transmission of requests from clients to servers, as well as the transmission of responses and exceptions back to the client.
- A broker must have some means of locating the receiver of a request based on its unique system identifier. A broker offers APIs (Application Programming Interfaces) to clients and servers that include operations for registering servers and for invoking server methods.
- When a request arrives for a server that is maintained by the local broker, the broker passes the request directly to the server.
- If the server is currently inactive, the broker activates it.
- All responses and exceptions from a service execution are forwarded by the broker to the client that sent the request.
- If the specified server is hosted by another broker, the local broker finds a route to the remote broker and forwards the request using this route.
- There is therefore a need for brokers to interoperate.
- Depending on the requirements of the whole system, additional services-such as name services or marshalling services may be integrated into the broker.

CRC Diagram

Class	Collaborators
Broker	• Client • Server • Client-side Proxy • Server-side Proxy • Bridge
Responsibility • (Un-)registers servers. • Offers APIs. • Transfers messages. • Error recovery. • Interoperates with other brokers through bridges. • Locates servers.	

Some Definitions

- In this pattern description we distinguish between local and remote brokers.
  - A local broker is running on the machine currently under consideration.
  - A remote broker is running on a remote network node.

- Name services provide associations between names and objects.
  - To resolve a name, a name service determines which server is associated with a given name.
  - In the context of Broker systems, names are only meaningful relative to a name space.

- Marshalling is the semantic-invariant conversion of data into a machine independent format such as
  - ASN.1 (Abstract Syntax Notation or
  - ONC XDR [external Data Representation).
- Marshalling performs the reverse transformation.

Broker: Illustration

- A broker may be a combination of an Internet gateway and the Internet infrastructure itself.
- Every information exchange between a client and a server must pass through the broker.
- A client specifies the information it wants using unique identifiers called URLs (Universal Resource Locators).
- By using these identifiers the broker is able to locate the required services and to route the requests to the appropriate server machines.
- When a new server machine is added, it must be registered with the broker.
- Clients and servers use the gateway of their Internet provider as an interface to the broker.

Client-side proxies

- Client-side proxies represent a layer between clients and the broker.
- This additional layer provides transparency, in that a remote object appears to the client as a local one.
- In detail, the proxies allow the hiding of implementation details from the clients such as:
    - The inter-process communication mechanism used for message transfers between clients and brokers.
    - The creation and deletion of memory blocks.
    - The marshalling of parameters and results.
- In many cases, client-side proxies translate the object model specified as part of the Broker architectural pattern to the object model of the programming language used to implement the client.

CRC Diagram

Class	Collaborators
Client-side Proxy	• Client • Broker
Responsibility • Encapsulates system-specific functionality. • Mediates between the client and the broker.	

Server Side Proxy

- Sewer-side proxies are generally analogous to Client-side proxies.
- The difference is that they are responsible for receiving requests, unpacking incoming messages, marshalling the parameters, and calling the appropriate service.
- They are used in addition for marshalling results and exceptions before sending them to the client.

CRC Diagram

Class	Collaborators
Server-side Proxy	• Server • Broker
Responsibility • Calls services within the server. • Encapsulates system-specific functionality. • Mediates between the server and the broker.	

Bridges

- Bridges are optional components used for hiding implementation details when two brokers interoperate.
- Suppose a Broker system runs on a heterogeneous network.
- If requests are transmitted over the network, different brokers have to communicate independently of the different network and operating systems in use.
- A bridge builds a layer that encapsulates all these system-specific details.

CRC Diagram

Class	Collaborators
Bridge	• Broker • Bridge
Responsibility • Encapsulates network-specific functionality. • Mediates between the local broker and the bridge of a remote broker.	

Broker System

Implementation

1. Define an object model, or use an exlsling model
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. (Note the steps involved)
6. Develop IDL compilers.

Variants

- Direct Communication Broker System
- Message Passing Broker System
- Trader System
- Adapter Broker System
- Callback Broker System

Known Usages

- CORBA
- IBM SOM/DSOM
- Microsoft's OLE 2.x
- World Wide Web
- ATM-P.

Consequences: benefits

- Location Transparency
- Changeability and extensibility of components
- Portability of a Broker system
- Interoperability between dzflerent Broker systems.
- Reusability
- Testing and Debugging

Consequences: Liabilities

- Restricted efficiency
- Lower fault tolerance
- Testing and Debugging

BITS Pilani

Pilani | Dubai | Goa | Hyderabad

Module 6 CS 07

Patterns – Part 2 MVC/SOA

Harvinder S Jabbal
SSZG653 Software Architectures

Model View Controller

Model-View-Controller Pattern

Context: User interface software is typically the most frequently modified portion of an interactive application. Users often wish to look at data from different perspectives, such as a bar graph or a pie chart. These representations should both reflect the current state of the data.

Problem: How can user interface functionality be kept separate from application functionality and yet still be responsive to user input, or to changes in the underlying application's data? And how can multiple views of the user interface be created, maintained, and coordinated when the underlying application data changes?

Solution: The model-view-controller (MVC) pattern separates application functionality into three kinds of components:

- A model, which contains the application's data
- A view, which displays some portion of the underlying data and interacts with the user
- A controller, which mediates between the model and the view and manages the notifications of state changes

MVC Example

Model
• Encapsulates application state
• Responds to state queries
• Exposes application functionality
• Notifies views of changes
State Query
Change Notification
State Change
View
• Renders the models
• Requests updates from models
• Sends user gestures to controller
• Allows controller to select view
View Selection
User Gestures
Controller
• Defines application behavior
• Maps user actions to model updates
• Selects view for response
• One for each functionality
Method Invocations
Events

MVC Solution - 1

Overview: The MVC pattern breaks system functionality into three components: a model, a view, and a controller that mediates between the model and the view.

Elements:

- The *model* is a representation of the application data or state, and it contains (or provides an interface to) application logic.
- The *view* is a user interface component that either produces a representation of the model for the user or allows for some form of user input, or both.
- The *controller* manages the interaction between the model and the view, translating user actions into changes to the model or changes to the view.

MVC Solution - 2

Relations: The *notifies* relation connects instances of model, view, and controller, notifying elements of relevant state changes.

Constraints:

- There must be at least one instance each of model, view, and controller.
- The model component should not interact directly with the controller.

Weaknesses:

- The complexity may not be worth it for simple user interfaces.
- The model, view, and controller abstractions may not be good fits for some user interface toolkits.

Service Oriented Architecture Pattern

Service Oriented Architecture Pattern

- **Context**: A number of services are offered (and described) by service providers and consumed by service consumers. Service consumers need to be able to understand and use these services without any detailed knowledge of their implementation.

Problem: How can we support interoperability of distributed components running on different platforms and written in different implementation languages, provided by different organizations, and distributed across the Internet?

Solution: The service-oriented architecture (SOA) pattern describes a collection of distributed components that provide and/or consume services.

Service Oriented Architecture Example

Service Oriented Architecture Solution - 1

Overview: Computation is achieved by a set of cooperating components that provide and/or consume services over a network.

Elements:

- Components:
  - *Service providers,* which provide one or more services through published interfaces.
  - *Service consumers,* which invoke services directly or through an intermediary.
  - *Service providers* may also be service consumers.
- *ESB,* which is an intermediary element that can route and transform messages between service providers and consumers.
- *Registry of services,* which may be used by providers to register their services and by consumers to discover services at runtime.
- *Orchestration server,* which coordinates the interactions between service consumers and providers based on languages for business processes and workflows.

Service Oriented Architecture Solution - 2

– Connectors:

- *SOAP connector,* which uses the SOAP protocol for synchronous communication between web services, typically over HTTP.
- *REST connector,* which relies on the basic request/reply operations of the HTTP protocol.
- *Asynchronous messaging connector,* which uses a messaging system to offer point-to-point or publish-subscribe asynchronous message exchanges.

Service Oriented Architecture Solution - 3

Relations: Attachment of the different kinds of components available to the respective connectors

Constraints: Service consumers are connected to service providers, but intermediary components (e.g., ESB, registry, orchestration server) may be used.

Weaknesses:

- SOA-based systems are typically complex to build.
- You don’t control the evolution of independent services.
- There is a performance overhead associated with the middleware, and services may be performance bottlenecks, and typically do not provide performance guarantees.

BITS Pilani

Pilani | Dubai | Goa | Hyderabad

Module 6 CS 07

Patterns – Part 2 Suppli

Interactive Systems – MVC/PAC

Harvinder S Jabbal
SSZG653 Software Architectures

Categories

From Mud to Structure.

- Patterns in this category help you to avoid a 'sea' of components or objects.
- In particular, they support a controlled decomposition of an overall system task into cooperating subtasks.
- The category includes
  - the Layers pattern
  - the Pipes and Filters pattern
  - the Blackboard pattern

Distributed Systems.

- This category includes one pattern.
  - Broker
- and refers to two patterns in other categories,
  - Microkernel
  - Pipes and Filters
- The Broker pattern provides a complete infrastructure for distributed applications.
- The Microkernel and Pipes and Filters patterns only consider distribution as a secondary concern and are therefore listed under their respective primary categories.

Interactive Systems.

- This category comprises two patterns,
  - the Model-View-Controller pattern (well-known from Smalltalk,)
  - the Presentation-Abstraction-Control pattern.
- Both patterns support the structuring of software systems that feature human-computer interaction.

Adaptable Systems.

- This category includes
  - The Reflection pattern
  - the Microkernel pattern
- strongly support extension of applications and their adaptation to evolving technology and changing functional requirements.

Interactive Systems

- This category comprises two patterns,
  - the Model-View-Controller pattern

    (well-known from Smalltalk,)
  - the Presentation-Abstraction-Control pattern.
- Both patterns support the structuring of software systems that feature human-computer interaction.

Interactive Systems

User Interaction

- Today's systems allow a high degree of user interaction, mainly achieved with help of graphical user interfaces.
- The objective is to enhance the usability of an application.
- Usable software systems provide convenient access to their services, and therefore allow users to learn the application and produce results quickly.

user interface

- When specifying the architecture of such systems, the challenge is to keep the functional core independent of the user interface.
- The core of interactive systems is based on the functional requirements for the system, and usually remains stable.
- User interfaces, however, are often subject to change and adaptation.
- For example, systems may have to support different user interface standards, customer-specific 'look and feel' metaphors, or interfaces that must be adjusted to fit into a customer's business processes.
- This requires architectures that support the adaptation of user interface parts without causing major effects to application-specific functionality or the data model underlying the software.

Model-View-Controller pattern (MVC)

- MVC divides an interactive application into three components.
    - The model contains the core functionality and data.
    - Views display information to the user.
    - Controllers handle user input.
- Views and controllers together comprise the user interface.
- A change-propagation mechanism ensures consistency between the user interface and the model.

The Presentation-Abstraction-Control pattern (PAC)

- PAC defines a structure for interactive software systems in the form of a hierarchy of cooperating agents.
- Every agent is responsible for a specific aspect of the application's functionality and consists of three components:
    - presentation,
    - abstraction, and
    - control.
- This subdivision separates the human-computer interaction aspects of the agent from its functional core and its communication with other agents.

Model-View-Controller

Model-View-Controller

- The MVC divides an interactive application into three components.
    - The model contains the core functionality and data.
    - Views display information to the user.
    - Controllers handle user input.
- Views and controllers together comprise the user interface.
- A change-propagation mechanism ensures consistency between the user interface and the model.

Context

- Interactive applications with a flexible human-computer interface.

Problem

- User interfaces are especially prone to change requests. When you extend the functionality of an application, you must modify menus to access these new functions.
- A customer may call for a specific user interface adaptation, or a system may need to be ported to another platform with a different 'look and feel' standard.
- Even upgrading to a new release of your windowing system can imply code changes.
- The user interface platform of long-lived systems thus represents a moving target.

Problem

- Different users place conflicting requirements on the user interface.
- A typist enters information into forms via the keyboard.
- A manager wants to use the same system mainly by clicking icons and buttons.
- Consequently, support for several user interface paradigms should be easily incorporated.
- Building a system with the required flexibility is expensive and error-prone if the user interface is tightly interwoven with the functional core.
- This can result in the need to develop and maintain several substantially different software systems, one for each user interface implementation.
- Ensuing changes spread over many modules.

Influences

The following forces influence the solution:

- The same information is presented differently in different windows, for example, in a bar or pie chart.
- The display and behaviour of the application must reflect data manipulations immediately.
- Changes to the user interface should be easy, and even possible at run-time.
- Supporting different 'look and feel' standards or porting the user interface should not affect code in the core of the application.

Solution

- Model-View-Controller (MVC] was first introduced in the Smalltalk-80 programming environment.
- MVC divides an interactive application into the three areas:
    - processing,
    - output, and
    - input.

Model

- The model component encapsulates core data and functionality.
- The model is independent of specific output representations or input behaviour.

View

- View components display information to the user. A view obtains the data from the model.
- There can be multiple views of the model.

Controller

- Each view has an associated controller component.
- Controllers receive input, usually as events that encode mouse movement, activation of mouse buttons, or keyboard input.
- Events are translated to service requests for the model or the view.
- The user interacts with the system solely through controllers.

MVC

- The separation of the model from view and controller components allows multiple views of the same model.
- If the user changes the model via the controller of one view, all other views dependent on this data should reflect the changes.
- The model therefore notifies all views whenever its data changes.
- The views in turn retrieve new data from the model and update the displayed information.
- This change- propagation mechanism is described in the Publisher-Subscriber pattern

Structure: Model

- The model component contains the functional core of the application.
- It encapsulates the appropriate data, and exports procedures that perform application-specific processing.
- Controllers call these procedures on behalf of the user.
- The model also provides functions to access its data that are used by view components to acquire the data to be displayed.
- The change-propagation mechanism maintains a registry of the dependent components within the model.
- All views and also selected controllers register their need to be informed about changes.
- Changes to the state of the model trigger the change-propagation mechanism.
- The change-propagation mechanism is the only link between the model and the views and controllers.

CRC Diagram

Class	Collaborators
Model	• View • Controller
Responsibility • Provides functional core of the application. • Registers dependent views and controllers. • Notifies dependent components about data changes.	

Structure: View

- View components present information to the user.
- Different views present the information of the model in different ways.
- Each view defines an update procedure that is activated by the change- propagation mechanism.
- When the update procedure is called, a view retrieves the current data values to be displayed from the model, and puts them on the screen.
- During initialization all views are associated with the model, and register with the change-propagation mechanism.
- Each view creates a suitable controller.
- There is a one-to-one relationship between views and controllers.
- Views often offer functionality that allows controllers to manipulate the display.
- This is useful for user-triggered operations that do not affect the model, such as scrolling.

CRC Diagram

Class	Collaborators
View	• Controller • Model
Responsibility • Creates and initializes its associated controller. • Displays information to the user. • Implements the update procedure. • Retrieves data from the model.	

Structure: Controller

- The controller components accept user input as events.
- How these events are delivered to a controller depends on the user interface platform.
- For simplicity, let us assume that each controller implements an event-handling procedure that is called for each relevant event.
- Events are translated into requests for the model or the associated view.
- If the behaviour of a controller depends on the state of the model, the controller registers itself with the change-propagation mechanism and implements an update procedure.
- For example, this is necessary when a change to the model enables or disables a menu entry.

CRC Diagram

Class	Collaborators
Controller	• View • Model
Responsibility • Accepts user input as events. • Translates events to service requests for the model or display requests for the view. • Implements the update procedure, if required.	

Popular Structural Variations

- Object Oriented
  - implementation of MVC would define a separate class for each component
- C++
  - view and controller classes share a common parent that defines the update interface.
- Smalltalk.
  - the class Object dehes methods for both sides of the change- propagation mechanism.
  - A separate class Observer is not needed.

Structure: C++

Observer
update
call update
Model
coreData
setOfObservers
attach(Observer)
detach(Observer)
notify
getData
service
attach
getData
View
myModel
myController
initialize(Model)
makeController
activate
display
update
create
manipulate
display
Controller
myModel
myView
initialize(Model,View)
handleEvent
update
attach
call service

Implementation

1. Separate human-computer interactionfrom corefunctionality
2. Implement the change-propagationm echanism
3. Design and implement the views
4. Design and implement the controllers
5. Design and implement the view-controller relationship
6. Implement the set-up of MVC
7. Dynamic view creation
8. 'Pluggable controllers
9. Infrastructure for hierarchical views and controllers
10. Further decoupling from system dependencies

Variants

- Document-View
  - This variant relaxes the separation of view and controller.
  - In several GUI platforms, window display and event handling are closely interwoven.
  - For example, the X Window System reports events relative to a window.
  - You can combine the responsibilities of the view and the controller from MVC in a single component by sacrificing exchangeability of controllers.

Known Usages

- Smalltalk
  - The VisualWorks Smalltalk environment supports different 'look and feel' standards by decoupling view and controllers via display and sensor classes
- MFC (Microsoft Foundation Class Library)
  - The Document-View variant of the Model-View-Controller pattern is integrated in the Visual C++ environment
- ET++
  - Establishes 'look and feel' independence by defining a class Window port that encapsulates the user interface platform dependencies, in the same way as do our display and sensor classes. Uses document-view variant.

Consequences: Benefits

- Multiple views of the same model
- Synchronized views.
- 'Pluggable' views and controllers.
- Exchangeability of 'look and feel'.
- Framework Potential.

Consequences: Liabilities

- Increased complexity.
- Potential for excessive number of updates.
- Intimate connection between view and controller.
- Close coupling of views and controllers to a model.
- Inefficiency of data access in view.
- Inevitability of change to view and controller when porting.
- Difficulty of using MVC with modern user-interface took.

PAC: Presentation-Abstraction-Control

Presentation-Abstraction-Control

- The Presentation-Abstraction-Control architectural pattern (PAC) defines a structure for interactive software systems in the form of a hierarchy of cooperating agents.
- Every agent is responsible for a specific aspect of the application's functionality and consists of three components: presentation, abstraction, and control.
- This subdivision separates the human-computer interaction aspects of the agent from its functional core and its communication with other agents

Context

- Development of an interactive application with the help of agents.
- In the context of this pattern an agent denotes an information-processing component that includes event receivers and transmitters, data structures to maintain stale, and a processor that handles incoming events, updates its own state, and that may produce new events.
- Agents can be as small as a single object, but also as complex as a complete software system.
- We use the terms agent and PAC agent as synonyms in this pattern description

Problem

- Interactive systems can often be viewed as a set of cooperating agents.
- Agents specialized in human-computer interaction accept user input and display data.
- Other agents maintain the data model of the system and offer functionality that operates on this data.
- Additional agents are responsible for diverse tasks such as error handling or communication with other software systems.
- Besides this horizontal decomposition of system functionality, we often encounter a vertical decomposition.

Example

- Production planning systems (PPS) , for example, distinguish between production planning and the execution of a previously specified production plan.
- For each of these tasks separate agents can be defined.
- In such an architecture of cooperating agents, each agent is specialized for a specific task, and all agents together provide the system functionality.
- This architecture also captures both a horizontal and vertical decomposition.

Forces

- Agents often maintain their own state and data.
  - For example, in a PPS system, the production planning and the actual production control may work on different data models, one tuned for planning and simulation and one performance-optimized for efficient production.
  - However, individual agents must effectively cooperate to provide the overall task of the application.
  - To achieve this, they need a mechanism for exchanging, data, messages, and events.
- Interactive agents provide their own user interface, since their respective human-computer interactions often differ widely.
  - For example, entering data into spreadsheets is done using keyboard input, while the manipulation of graphical objects uses a pointing device.
- Systems evolve over time.
  - Their presentation aspect is particularly prone to change.
  - The use of graphics, and more recently, multi-media features, are examples of pervasive changes to user interfaces.
  - Changes to individual agents, or the extension of the system with new agents, should not affect the whole system.

Solution

- Structure the interactive application as a tree-like hierarchy of PAC agents.
- There should be one top-level agent, several intermediate-level agents, and even more bottom-level agents.
- Every agent is responsible for a specific aspect of the application's functionality, and consists of three components: presentation, abstraction, and control.
- The whole hierarchy reflects transitive dependencies between agents.
- Each agent depends on all higher-level agents up the hierarchy to the top-level agent.

Agents

- The agent's presentation component provides the visible behaviour of the PAC agent.
- Its abstraction component maintains the data model that underlies the agent, and provides functionality that operates on this data.
- Its control component connects the presentation and abstraction components, and provides functionality that allows the agent to communicate with other PAC agents.

The top-level PAC agent

- The top-level PAC agent provides the functional core of the system.
- Most other PAC agents depend or operate on this core. Furthermore, the top-level PAC agent includes those parts of the user interface that cannot be assigned to particular subtasks, such as menu bars or a dialog box displaying information about the application.

Bottom-level PAC agents

- Bottom-level PAC agents represent self-contained semantic concepts on which users of the system can act, such as spreadsheets and charts.
- The bottom-level agents present these concepts to the user and support all operations that users can perform on these agents, such as zooming or moving a chart.

Intermediate-level PAC agents

- Intermediate-level PAC agents represent either combinations of, or relationships between, lower-level agents.
    - For example, an intermediate-level agent may maintain several views of the same data, such as a floor plan and an external view of a house in a CAD system for architecture.

Typical Hierarchy of Agents

Structure: Top Level Agent (component)

- The main responsibility of the top-level PAC agent is to provide the global data model of the software.
- This is maintained in the abstraction component of the top-level agent.
- The interface of the abstraction component offers functions to manipulate the data model and to retrieve information about it.
- The representation of data within the abstraction component is media-independent.
- For example, in a CAD system for architecture, walls, doors, and windows are represented in centimetres or inches that reflect their real size, not in pixels for display purposes.
- This media-independency supports adaptation of the PAC agent to different environments without major changes in its abstraction component.

Structure: Top Level Agent (presentation)

- The presentation component of the top-level agent often has few responsibilities.
- It may include user-interface elements common to the whole application.
- In some systems, such as the network traffic manager, there is no top-level presentation component at all.

Structure: Top Level Agent (control)

- The control component of the top-level PAC agent has three responsibilities:

1
    - It allows lower-level agents to make use of the services of the top-level agents, mostly to access and manipulate the global data model.
    - Incoming service requests from lower-level agents are forwarded either to the abstraction component or the presentation component.

2
    - It coordinates the hierarchy of PAC agents.
    - It maintains information about connections between the top-level agent and lower-level agents.
    - The control component uses this information to ensure correct collaboration and data exchange between the top-level agent and lower-level agents.

3
    - It maintains information about the interaction of the user with the system.
    - For example, it may check whether a particular operation can be performed on the data model when triggered by the user.
    - It may also keep track of the functions called to provide history or undo/redo services for operations on the functional core.

Bottom Level Agent

- Bottom-level PAC agents represent a specific semantic concept of the application domain, such as a mailbox in a network traffic management system or a wall in a mobile robot system.
- This semantic concept may be as low-level as a simple graphical object such as a circle, or as complex as a bar chart that summarizes all the data in the system.

Bottom Level Agent (presentation)

- The presentation component of a bottom-level PAC agent presents a specific view of the corresponding semantic concept, and provides access to all the functions users can apply to it.
- Internally, the presentation component also maintains information about the view, such as its position on the screen.

Bottom Level Agent (abstraction)

- The abstraction component of a bottom-level PAC agent has a similar responsibility as the abstraction component of the top-level PAC agent, maintaining agent-specific data.
- In contrast to the abstraction component of the top-level agent, however, no other PAC agents depend on this data.

Bottom Level Agent (control)

- The control component of a bottom-level PAC agent maintains consistency between the abstraction and presentation components, thereby avoiding direct dependencies between them.
- It serves as an adapter and performs both interface and data adaptation.
- The control component of bottom-level PAC agents communicates with higher-level agents to exchange events and data.
- Incoming events-such as a 'close window' request-are forwarded to the presentation component of the bottom-level agent, while incoming data is forwarded to its abstraction component.
- Outgoing events and data, for example error messages, are sent to the associated higher-level agent.

Bottom Level Agent

- Concepts represented by bottom-level PAC agents, such as the bar and pie charts in the example, are atomic in the sense that they are the smallest units a user can manipulate. The users can only operate on the bar chart as a whole, for instance by changing the scaling factor of the y-axis. They cannot, for example, resize an individual bar of a bar chart.
- Bottom-level PAC agents are not restricted to providing semantic concepts of the application domain. You can also specify bottom-level agents that implement system services.
- For example, there may be a communication agent that allows the system to cooperate with other applications and to monitor this cooperation

Intermediate-Level PAC agents (composition)

- Intermediate-Level PAC agents can fulfil two different roles:
- composition and coordination
- When, for example, each object in a complex graphic is represented by a separate PAC agent, an intermediate-level agent groups these objects to form a composite graphical object.
- The intermediate-level agent defines a new abstraction, whose behaviour encompasses both the behaviour of its components and the new characteristics that are added to the composite object.

Intermediate-Level PAC agents (coordination)

- The second role of an intermediate-level agent is to maintain consistency between lower- level agents, for example when coordinating multiple views of the same data.
- The abstraction component maintains the specific data of the intermediate-level PAC agent. The presentation component implements its user interface. The control component has the same responsibilities of the control components of bottom-level PAC agents and of the top-level PAC agent.

CRC Diagrams

Class	Collaborators
Top-level Agent	• Intermediate-level Agent • Bottom-level Agent
Responsibility • Provides the functional core of the system. • Controls the PAC hierarchy.	

Class	Collaborators
Interm. -level Agent	• Top-level Agent • Intermediate-level Agent • Bottom-level Agent
Responsibility • Coordinates lower-level PAC agents. • Composes lower-level PAC agents to a single unit of higher abstraction.	

Class	Collaborators
Bottom-level Agent	• Top-level Agent • Intermediate-level Agent
Responsibility • Provides a specific view of the software or a system service, including its associated human-computer interaction.	

Typical PAC Object Model

Internal Structure of a PAC Agent

Implementation

1. Define a model of the application.
2. Define a general strategy for organizing the PAC hierarchy
3. Specify the top-level PAC agent.
4. Specify the bottom-level PAC agents
5. Specih bottom-level PAC agents for system services.
6. Specify intermediate-level PAC agents to compose lower-level PAC agents
7. Specify intermediate-level PAC agents to coordinate lower-level PAC agents.
8. Separate core functionality from human-computer interaction
9. Provide the external interface.

Variants

- Many large applications-especially interactive ones-are multi-user systems. Multi-tasking is thus a major concern when designing such software systems.

Variants

- PAC agents as active objects

- PAC agents as processes.

Known Usages

- Network Traffic Management.
- Mobile Robot.

Consequences: benefits

- Separation of concerns
- Support for change and extension
- Support for multi-tasking

Consequences: Liabilities

- Increased system complexity.
- Complex control component
- Applicability.

Thank you

The END

BITS Pilani

Pilani | Dubai | Goa | Hyderabad

Module 7

Patterns – Part 3 Suppli

Adaptable Systems.

Harvinder S Jabbal
SSZG653 Software Architectures

Categories

From Mud to Structure.

- Patterns in this category help you to avoid a 'sea' of components or objects.
- In particular, they support a controlled decomposition of an overall system task into cooperating subtasks.
- The category includes
  - the Layers pattern
  - the Pipes and Filters pattern
  - the Blackboard pattern

Distributed Systems.

- This category includes one pattern.
  - Broker
- and refers to two patterns in other categories,
  - Microkernel
  - Pipes and Filters
- The Broker pattern provides a complete infrastructure for distributed applications.
- The Microkernel and Pipes and Filters patterns only consider distribution as a secondary concern and are therefore listed under their respective primary categories.

Interactive Systems.

- This category comprises two patterns,
  - the Model-View-Controller pattern (well-known from Smalltalk,)
  - the Presentation-Abstraction-Control pattern.
- Both patterns support the structuring of software systems that feature human-computer interaction.

Adaptable Systems.

- This category includes
  - The Reflection pattern
  - the Microkernel pattern
- strongly support extension of applications and their adaptation to evolving technology and changing functional requirements.

Adaptable Systems.

- This category includes
  - The Reflection pattern
  - the Microkernel pattern
- strongly support extension of applications and their adaptation to evolving technology and changing functional requirements.

Adaptable

- Systems evolve over time
  - -new functionality is added and existing services are changed.
- They must support new versions of
  - operating systems,
  - user-interface platforms or
  - third-party components and libraries.
- Adaptation to
  - new standards or
  - hardware platforms may also be necessary.
- During system design and implementation,
  - customers may request new features, often urgently and at a late stage. Y
  - You may also need to provide services that differ from customer to customer.

Design for Change

- Design for change is therefore a major concern when specifying the architecture of a software system.
- An application should support its own modification and extension a priori.
- Changes should not affect the core functionality or key design abstractions, otherwise the system will be hard to maintain and expensive to adapt to changing requirements.

Adaptable Systems: Microkernel

Adaptable Systems: Microkernel

- The Microkernel architectural pattern applies to software systems that must be able to adapt to changing system requirements.
- It separates a minimal functional core from extended functionality and customer-specific parts.
- The microkernel also serves as a socket for plugging in these extensions and coordinating their collaboration.

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 is a non-trivial task.
- Well-known examples are application platforms such as operating systems and graphical user interfaces.

FORCES:

- The application platform must cope with continuous hardware and software evolution.
- The application platform should be portable, extensible and adaptable to allow easy integration of emerging technologies
- The applications in your domain need to support different, but similar, application platforms.
- The applications may be categorized into groups that use the same functional core in different ways, requiring the underlying application platform to emulate existing standards.
- The functional core of the application platform should be separated into a component with minimal memory size, and services that consume as little processing power as possible.

Solution

- Encapsulate the fundamental services of your application platform in a microkernel component.
- The microkernel includes functionality that enables other components running in separate processes to communicate with each other.
- It is also responsible for maintaining system- wide resources such as files or processes.
- In addition, it provides interfaces that enable other components to access its functionality.

Solution - details

- Core functionality that cannot be implemented within the microkernel without unnecessarily increasing its size or complexity should be separated in internal servers.
- External servers implement their own view of the underlying microkernel.
- To construct this view, they use the mechanisms available through the interfaces of the microkernel. Every external server is a separate process that itself represents an application platform.
- Hence, a Microkernel system may be viewed as an application platform that integrates other application platforms.
- Clients communicate with external servers by using the communication facilities provided by the microkernel.

Structure

The Microkernel pattern defines five kinds of participating components:

- Internal servers
- External servers
- Adapters
- Clients
- Microkernel

Microkernel

- represents the main component of the pattern.
- It implements central services such as communication facilities or resource handling.
- Other components build on all or some of these basic services.
- They do this indirectly by using one or more interfaces that comprise the functionality exposed by the microkernel.

Microkernel Component

- A microkernel implements **atomic** services, which we refer to as **mechanisms**.
- These **mechanisms** serve as a fundamental base on which more complex functionality, called **policies**, are constructed.
- The microkernel is also responsible for maintaining **system resources** such as **processes** or **files**.
- It controls and coordinates the access to these resources.

Class	***Collaborators***
Microkernel	• Internal Server
Responsibility • Provides core mechanisms. • Offers communication facilities. • Encapsulates system dependencies. • Manages and controls resources.	

BITS Pilani, Deemed to be University under Section 3 of UGC Act, 1956

Internal Server Component

- Also known as a subsystem.
- They are extensions of the microkernel.
- Only accessible by the microkernel component.
- Extends the functionality provided by the microkernel.
- It is a separate component that offers additional functionality.
- The microkernel invokes the functionality of internal servers via service requests.
- Internal servers can therefore encapsulate some dependencies on the underlying hardware or software system.
- For example, device drivers that support specific graphics cards are good candidates for internal servers.

Class	***Collaborators***
Internal Server	• Microkernel
Responsibility • Implements additional services. • Encapsulates some system specifics.	

BITS Pilani, Deemed to be University under Section 3 of UGC Act, 1956

External Server Component

- also known as a personality
- uses the microkernel for implementing its own view of the underlying application domain.
- a view denotes a layer of abstraction built on top of the atomic services provided by the microkernel.
- Different external servers implement different policies for specific application domains.
- External servers expose their functionality by exporting interfaces in the same way as the microkernel itself does.
- Each of these external servers runs in a separate process.
- It receives service requests from client applications, uses the communication facilities provided by the microkernel, interprets these requests, executes the appropriate services and returns results to its clients.
- The implementation of services relies on microkernel mechanisms, so external servers need to access the microkernel's programming interfaces.

Class External Server	***Collaborators*** • Microkernel
Responsibility • Provides programming interfaces for its clients.	

Client & Adapter

If the external server implements an existing application platform, the corresponding adapter mimics the programming interfaces of that platform.

Adapters

- also known as emulators
- represent these interfaces between clients and their external servers
- allow clients to access the services of their external server in a portable way

Class	Collaborators
Client	• Adapter
Responsibility • Represents an application.	

Class	Collaborators
Adapter	• External Server • Microkernel
Responsibility • Hides system dependencies such as communication facilities from the client. • Invokes methods of external servers on behalf of clients.	

OMT Diagram

External Server
receiveRequest
dispatchRequest
executeService
calls
Microkernel
executeMechanism
initCommunication
findReceiver
createHandle
sendMessage
callInternalServer
activates
Internal Server
executeService
receiveRequest
initializes
communication
Adapter
callService
createRequest
sends request
calls service
Client
doTask

Implementation

1. Analyze the application domain.
2. Analyze external servers
3. Categorize the services.
4. Partition the categories.
5. Find a consistent and complete set of operatims and abstractims for every category.
6. Determine strategies for request transmission and retrieval.
7. Structure the microkernel component.
8. specify the programming interface
9. The microkernel is responsible for managing all system resources such as memory blocks, devices or device contexts-a handle to an output area in a graphical user interface implementation.
10. Design and implement the internal ser-uers as separate processes or shared libraries.
11. Implement the external servers
12. Implement the adapters
13. Develop client applications or use existing ones for the ready-to-run Microkernel system

Variant:

Microkernel System with indirect Client-Server connections.

- A client that wants to send a request or message to an external server asks the microkernel for a communication channel.
- After the requested communication path has been established, client and server communicate with each other indirectly using the microkernel as a message backbone.
- Using this variant leads to an architecture in which all requests pass through the microkernel.
- You can apply it, for example, when security requirements force the system to control all communication between participants.

Variant: Distributed Microkernel System

- A microkernel can also act as a message backbone responsible for sending messages to remote machines or receiving messages from them.
- Every machine in a distributed system uses its own microkernel implementation.
- From the user's viewpoint the whole system appears as a single Microkernel system-the distribution remains transparent to the user.
- A distributed Microkernel system allows you to distribute servers and clients across a network of machines or microprocessors.
- To achieve this the micro kernels in a distributed implementation must include additional services for communicating with each other.

Benefits

1. Portability
2. Flexibility and Extensibility
3. Separation of policy and mechanism
4. Scalability
5. Reliability
6. Transparency

liability

1. Performance.
2. Complexity of design and implementation.

Adaptable Systems: Reflection

Adaptable Systems: Reflection

- The Reflection architectural pattern provides a mechanism for changing structure and behaviour of software systems dynamically.
- It supports the modification of fundamental aspects, such as type structures and function call mechanisms.
- In this pattern, an application is split into two parts.
    - A meta level provides information about selected system properties and makes the software self-aware.
    - A base level includes the application logic.
- Its implementation builds on the meta level.
- Changes to information kept in the meta level affect subsequent base-level behaviour.

Context

- Building systems that support their own modification a priori.

Problem

- Software systems evolve over time.
- They must be open to modifications in response to changing technology and requirements.
- Designing a system that meets a wide range of different requirements a priori can be an overwhelming task.
- A better solution is to specify an architecture that is open to modification and extension.
- The resulting system can then be adapted to changing requirements on demand.
- In other words, we want to design for change and evolution.

Forces

- Changing software is tedious, error prone, and often expensive.
- Adaptable software systems usually have a complex inner structure.
- The more techniques that are necessary for keeping a system changeable, such as parameterization, subclassing, mix-ins, or even copy and paste, the more awkward and complex its modification becomes.
- Changes can be of any scale, from providing shortcuts for commonly-used commands to adapting an application framework for a specific customer.
- Even fundamental aspects of software systems can change, for example the communication mechanisms between components.

Solution

- Make the software self-aware, and make selected aspects of its structure and behaviour accessible for adaptation and change.
- This leads to an architecture that is split into two major parts:
    - a meta level and
    - a base level.

Meta Level

- The meta level provides a self-representation of the software to give it knowledge of its own structure and behavior, and consists of so-called rnetaobjects.
- Metaobjects encapsulate and represent information about the software.
- Examples include type structures, algorithms, or even function call mechanisms.

Base Level

- The base level defines the application logic.
- Its implementation uses the metaobjects to remain independent of those aspects that are likely to change.
- For example, base-level components may only communicate with each other via a metaobject that implements a specific user-defined function call mechanism.
- Changing this metaobject changes the way in which base-level components communicate, but without modifing the base-level code.

metaobject protocol (MOP)

- An interface is specified for manipulating the metaobjects.
- It is called the metaobject protocol (MOP), and allows clients to specify particular changes, such as modification of the function call mechanism metaobject mentioned above.
- The metaobject protocol itself is responsible for checking the correctness of the change specification, and for performing the change.
- Every manipulation of metaobjects through the metaobject protocol affects subsequent base-level behavior, as in the function call mechanism example.

Structure

Class	Collaborators
Base Level	• Meta Level
Responsibility • Implements the application logic. • Uses information provided by the meta level.	

Class	Collaborators
Metaobject Protocol	• Meta Level • Base Level
Responsibility • Offers an interface for specifying changes to the meta level. • Performs specified changes	

Class	Collaborators
Meta Level	• Base Level
Responsibility • Encapsulates system internals that may change. • Provides an interface to facilitate modifications to the meta-level.	

- The meta level consists of a set of metaobjects. Each metaobject encapsulates selected information about a single aspect of the structure, behaviour, or state of the base level.
- The base level models and implements the application logic of the software.
- The metaobject protocol (MOP] serves as an external interface to the meta level, and makes the implementation of a reflective system accessible in a defined way.

OMT Diagram

- Since the base-level implementation explicitly builds upon information and services provided by metaobjects, changing them has an immediate effect on the subsequent behaviour of the base level.
- The general structure of a reflective architecture is very much like a Layered system

Implementation

1. Define a model of the application.
2. Identify varying behavior
3. Identify structural aspects of the system, which, when changed, should not affect the implementation of the base level
4. Identify system services that support both the variation of application services and the independence of structural details
5. Define the metaobjects.
6. Define the metaobject protocol.
7. Dew the base level.

Benefits

1. No explicit mod$cation of source code.
2. Changing a software system is easy
3. Support for many kinds of change

liabilities

1. Modifications at the meta level may cause damage
2. Increased number of components
3. Lower efficiency.
4. Not all potential changes to the software are supported.
5. Not all languages support refection.

Thank you

Credits: Material sourced from Text Book…

PATTERN-ORIENTED SOFTWARE ARCHITECTURE

A System of Patterns

- Frank Buschmann
- Regine Meunier
- Hans Rohnert
- Peter Sornmerlad
- Michael Stal

of Siemens AG, Germany

Module 7

Patterns – Part 3

Harvinder S Jabbal
SSZG653 Software Architectures

Client-Server Pattern

Client-Server Pattern

Context: There are shared resources and services that large numbers of distributed clients wish to access, and for which we wish to control access or quality of service.

Problem: By managing a set of shared resources and services, we can promote modifiability and reuse, by factoring out common services and having to modify these in a single location, or a small number of locations. We want to improve scalability and availability by centralizing the control of these resources and services, while distributing the resources themselves across multiple physical servers.

Solution: Clients interact by requesting services of servers, which provide a set of services. Some components may act as both clients and servers. There may be one central server or multiple distributed ones.

Client-Server Example

Client-Server Solution - 1

Overview: Clients initiate interactions with servers, invoking services as needed from those servers and waiting for the results of those requests.

Elements:

- *Client,* a component that invokes services of a server component. Clients have ports that describe the services they require.
- *Server:* a component that provides services to clients. Servers have ports that describe the services they provide.

Request/reply connector: a data connector employing a request/reply protocol, used by a client to invoke services on a server. Important characteristics include whether the calls are local or remote, and whether data is encrypted.

Client-Server Solution- 2

Relations: The *attachment* relation associates clients with servers.

Constraints:

- Clients are connected to servers through request/reply connectors.
- Server components can be clients to other servers.

Weaknesses:

- Server can be a performance bottleneck.
- Server can be a single point of failure.
- Decisions about where to locate functionality (in the client or in the server) are often complex and costly to change after a system has been built.

Peer-to-Peer Pattern

Peer-to-Peer Pattern

Context: Distributed computational entities—each of which is considered equally important in terms of initiating an interaction and each of which provides its own resources—need to cooperate and collaborate to provide a service to a distributed community of users.

Problem: How can a set of "equal" distributed computational entities be connected to each other via a common protocol so that they can organize and share their services with high availability and scalability?

Solution: In the peer-to-peer (P2P) pattern, components directly interact as peers. All peers are "equal" and no peer or group of peers can be critical for the health of the system. Peer-to-peer communication is typically a request/reply interaction without the asymmetry found in the client-server pattern.

Peer-to-Peer Example

Peer-to-Peer Solution - 1

Overview: Computation is achieved by cooperating peers that request service from and provide services to one another across a network.

Elements:

- *Peer,* which is an independent component running on a network node. Special peer components can provide routing, indexing, and peer search capability.
- *Request/reply connector,* which is used to connect to the peer network, search for other peers, and invoke services from other peers. In some cases, the need for a reply is done away with.

Relations: The relation associates peers with their connectors. Attachments may change at runtime.

Peer-to-Peer Solution - 2

Constraints: Restrictions may be placed on the following:

- The number of allowable attachments to any given peer
- The number of hops used for searching for a peer
- Which peers know about which other peers
- Some P2P networks are organized with star topologies, in which peers only connect to supernodes.

Weaknesses:

- Managing security, data consistency, data/service availability, backup, and recovery are all more complex.
- Small peer-to-peer systems may not be able to consistently achieve quality goals such as performance and availability.

Publish-Subscribe Pattern

Publish-Subscribe Pattern

Context: There are a number of independent producers and consumers of data that must interact. The precise number and nature of the data producers and consumers are not predetermined or fixed, nor is the data that they share.

Problem: How can we create integration mechanisms that support the ability to transmit messages among the producers and consumers so they are unaware of each other's identity, or potentially even their existence?

Solution: In the publish-subscribe pattern, components interact via announced messages, or events. Components may subscribe to a set of events. Publisher components place events on the bus by announcing them; the connector then delivers those

BITS Pilani, Deemed to be University under Section 3 of UGC Act, 1956

Publish-Subscribe Example

Publish-Subscribe Solution – 1

Overview: Components publish and subscribe to events. When an event is announced by a component, the connector infrastructure dispatches the event to all registered subscribers.

Elements:

- *Any C&C component* with at least one publish or subscribe port.
- *The publish-subscribe connector*, which will have *announce* and *listen* roles for components that wish to publish and subscribe to events.

Relations: The *attachment* relation associates components with the publish-subscribe connector by prescribing which components announce events and which components are registered to receive events.

Publish-Subscribe Solution - 2

Constraints: All components are connected to an event distributor that may be viewed as either a bus—connector—or a component. Publish ports are attached to announce roles and subscribe ports are attached to listen roles.

Weaknesses:

- Typically increases latency and has a negative effect on scalability and predictability of message delivery time.
- Less control over ordering of messages, and delivery of messages is not guaranteed.

Shared-Data Pattern

Shared-Data Pattern

Context: Various computational components need to share and manipulate large amounts of data. This data does not belong solely to any one of those components.

Problem: How can systems store and manipulate persistent data that is accessed by multiple independent components?

Solution: In the shared-data pattern, interaction is dominated by the exchange of persistent data between multiple *data accessors* and at least one *shared-data store*. Exchange may be initiated by the accessors or the data store. The connector type is *data reading and writing*.

Shared Data Example

Shared Data Solution - 1

Overview: Communication between data accessors is mediated by a shared data store. Control may be initiated by the data accessors or the data store. Data is made persistent by the data store.

Elements:

- *Shared-data store.* Concerns include types of data stored, data performance-oriented properties, data distribution, and number of accessors permitted.
- *Data accessor component*.
- *Data reading and writing connector*.

Shared Data Solution - 2

Relations: *Attachment* relation determines which data accessors are connected to which data stores.

Constraints: Data accessors interact only with the data store(s).

Weaknesses:

- The shared-data store may be a performance bottleneck.
- The shared-data store may be a single point of failure.
- Producers and consumers of data may be tightly coupled.

Thank you

BITS Pilani

Pilani | Dubai | Goa | Hyderabad

Module 7

Patterns – Part 4

Harvinder S Jabbal
SSZG653 Software Architectures

Map-Reduce Pattern

Map-Reduce Pattern

Context: Businesses have a pressing need to quickly analyze enormous volumes of data they generate or access, at petabyte scale.

Problem: For many applications with ultra-large data sets, sorting the data and then analyzing the grouped data is sufficient. The problem the map-reduce pattern solves is to efficiently perform a distributed and parallel sort of a large data set and provide a simple means for the programmer to specify the analysis to be done.

Solution: The map-reduce pattern requires three parts:

- A specialized infrastructure takes care of allocating software to the hardware nodes in a massively parallel computing environment and handles sorting the data as needed.
- A programmer specified component called the map which filters the data to retrieve those items to be combined.
- A programmer specified component called reduce which combines the results of the map

Map-Reduce Example

Map-Reduce Solution - 1

Overview: The map-reduce pattern provides a framework for analyzing a large distributed set of data that will execute in parallel, on a set of processors. This parallelization allows for low latency and high availability. The map performs the extract and transform portions of the analysis and the reduce performs the loading of the results.

Elements:

- Map is a function with multiple instances deployed across multiple processors that performs the extract and transformation portions of the analysis.
- Reduce is a function that may be deployed as a single instance or as multiple instances across processors to perform the load portion of extract-transform-load.
- The infrastructure is the framework responsible for deploying map and reduce instances, shepherding the data between them, and detecting and recovering from failure.

Map-Reduce Solution - 2

Relations:

- Deploy on is the relation between an instance of a map or reduce function and the processor onto which it is installed.
- Instantiate, monitor, and control is the relation between the infrastructure and the instances of map and reduce.

Constraints:

- The data to be analyzed must exist as a set of files.
- Map functions are stateless and do not communicate with each other.
- The only communication between map reduce instances is the data emitted from the map instances as <key, value> pairs.

Weaknesses:

- If you do not have large data sets, the overhead of map-reduce is not justified.
- If you cannot divide your data set into similar sized subsets, the advantages of parallelism are lost.
- Operations that require multiple reduces are complex to orchestrate.

Multi-Tier Pattern

Multi-Tier Pattern

Context: In a distributed deployment, there is often a need to distribute a system's infrastructure into distinct subsets.

Problem: How can we split the system into a number of computationally independent execution structures—groups of software and hardware—connected by some communications media?

Solution: The execution structures of many systems are organized as a set of logical groupings of components. Each grouping is termed a tier.

Multi-Tier Example

Multi-Tier Solution

Overview: The execution structures of many systems are organized as a set of logical groupings of components. Each grouping is termed a *tier*.

Elements:

- *Tier,* which is a logical grouping of software components.

Relations:

- *Is part of,* to group components into tiers.
- *Communicates with,* to show how tiers and the components they contain interact with each other.
- *Allocated to,* in the case that tiers map to computing platforms.

Constraints: A software component belongs to exactly one tier.

Weaknesses: Substantial up-front cost and complexity.

Tactics and Patterns

Relationships Between Tactics and Patterns

Patterns are built from tactics; if a pattern is a molecule, a tactic is an atom.

MVC, for example utilizes the tactics:

- Increase semantic coherence
- Encapsulation
- Use an intermediary
- Use run time binding

Tactics Augment Patterns

Patterns solve a specific problem but are neutral or have weaknesses with respect to other qualities.

Consider the broker pattern

- May have performance bottlenecks
- May have a single point of failure

Using tactics such as

- Increase resources will help performance
- Maintain multiple copies will help availability

Tactics and Interactions

Each tactic/pattern has pluses (its reason for being) and minuses – side effects.

Use of tactics can help alleviate the minuses.

But nothing is free…

Tactics and Interactions - 2

A common tactic for detecting faults is Ping/Echo.

Common side-effects of Ping/Echo are:

security: how to prevent a ping flood attack?

performance: how to ensure that the performance overhead of ping/echo is small?

modifiability: how to add ping/echo to the existing architecture?

Tactics and Interactions - 3

Tactics and Interactions - 4

A tactic to address the performance side-effect is “Increase Available Resources”.

Common side effects of Increase Available Resources are:

cost: increased resources cost more

performance: how to utilize the increase resources efficiently?

Tactics and Interactions - 5

Tactics and Interactions - 6

A tactic to address the efficient use of resources side-effect is “Scheduling Policy”.

Common side effects of Scheduling Policy are:

modifiability: how to add the scheduling policy to the existing architecture

modifiability: how to change the scheduling policy in the future?

Tactics and Interactions - 7

Tactics and Interactions - 8

A tactic to address the addition of the scheduler to the system is “Use an Intermediary”.

Common side effects of Use an Intermediary are:

modifiability: how to ensure that all communication passes through the intermediary?

Tactics and Interactions - 9

Tactics and Interactions – 10.

A tactic to address the concern that all communication passes through the intermediary is “Restrict Communication Paths”.

Common side effects of Restrict Communication Paths are:

performance: how to ensure that the performance overhead of the intermediary are not excessive?

Note: this design problem has now become recursive!

How Does This Process End?

Each use of tactic introduces new concerns.

Each new concern causes new tactics to be added.

Are we in an infinite progression?

No. Eventually the side-effects of each tactic become small enough to ignore.

Summary

An architectural pattern

- is a package of design decisions that is found repeatedly in practice,
- has known properties that permit reuse, and
- describes a *class* of architectures.

Tactics are simpler than patterns

Patterns are underspecified with respect to real systems so they have to be augmented with tactics.

- Augmentation ends when requirements for a specific system are satisfied.

BITS Pilani

Pilani | Dubai | Goa | Hyderabad

Module 8
Part 1

Architectures for the Cloud-1

Harvinder S Jabbal
SE ZG651/ SS ZG653 Software Architectures

Architectures for the Cloud

Chapter Outline

- Basic Cloud Definitions
- Service Models and Deployment Options
- Economic Justification

Basic Cloud Definitions (from NIST)

- *On-demand self-service.* A resource consumer can unilaterally provision computing services, such as server time and network storage, as needed automatically without requiring human interaction with each service's provider.
- *Ubiquitous network access.* Cloud services and resources are available over the network and accessed through standard networking mechanisms that promote use by a heterogeneous collection of clients.
- *Resource pooling.* The cloud provider's computing resources are pooled.
- *Location independence.* The location of the resources need not be of concern to the consumer of the resources.
- *Rapid elasticity.* Capabilities can be rapidly and elastically provisioned.
- *Measured service.* Resource usage can be monitored, controlled, and reported so that consumers of the services are billed only for what they use.
- *Multi-tenancy.* Applications and resources can be shared among multiple consumers who are unaware of each other.

Basic Service Models

- **Software as a Service (SaaS).** The consumer in this case is an end user. The consumer uses applications that happen to be running on a cloud. E.g. mail services or data storage services.
- **Platform as a Service (PaaS).** The consumer in this case is a developer or system administrator. The consumer deploys applications onto the cloud infrastructure using programming languages and tools supported by the provider.
- **Infrastructure as a Service (IaaS).** The consumer in this case is a developer or system administrator. The capability provided to the consumer is to provision processing, storage, networks, and other fundamental computing resources where the consumer is able to deploy and run arbitrary software, which can include operating systems and applications.

Deployment Models

- *Private cloud.* The cloud infrastructure is owned solely by a single organization and operated solely for applications owned by that organization.
- *Public cloud.* The cloud infrastructure is made available to the general public or a large industry group and is owned by an organization selling cloud services.
- *Community cloud.* The cloud infrastructure is shared by several organizations and supports a specific community that has shared concerns.
- *Hybrid cloud.* The cloud infrastructure is a composition of two or more clouds (private, community, or public) that remain unique entities.

Economic Justification

- Economies of scale
- Utilization of equipment
- Multi-tenancy

Economies of Scale

- Large data centers are cheaper to operate (per unit measure) than small data centers.
- *Large* in this context means 100,000+ servers
- *Small* in this context means <10,000 servers.

Reasons for Economies of Scale

- *Cost of power.* The cost of electricity to operate a data center currently is 15 to 20 percent of the total cost of operation.
- Per-server power costs are lower in large data centers
  - Sharing of items such as racks and switches.
  - Negotiated prices. Large power users can negotiate significant discounts.
  - Geographic choice. Large data centers can be located where power costs are lowest.
  - Acquisition of cheaper power sources such as wind farms and rooftop solar energy.
- *Infrastructure labor costs. More efficient utilization of system administrators*
  - Small data center administrators service ~150 servers.
  - Large data center administrators service >1000 servers.

More Reasons for Economies of Scale

- *Security and reliability.* Maintaining a given level of security, redundancy, and disaster recovery essentially requires a fixed level of investment. Larger data centers can amortize that investment over their larger number of servers.
- *Hardware costs.* Operators of large data centers can get discounts on hardware purchases of up to 30 percent over smaller buyers.

Utilization of Equipment

- Use of virtualization technology allows for easy co-location of distinct applications and their associated operating systems on the same server hardware. The effect of this co-location is to increase the utilization of servers.
- Variations in workload can be managed to increase utilization.
  - *Random access.* End users may access applications randomly. The more likely that the randomness of their accesses will end up imposing a uniform load on the server.
- *Time of day.*
  - Co-locate those services that are workplace related with those that are consumer related.
  - Consider time differences among geographically distinct locations.
- *Time of year.* Consider yearly fluctuations in demand.
  - Holidays, tax preparation season
- *Resource usage patterns.* Co-locate heavier CPU services with heavier I/O services.
- *Uncertainty.* Consider spikes in usage
  - news events, marketing events, sporting events

Multi-tenancy

- Some applications such as salesforce.com use a single application for multiple different consumers.
- This reduces costs by reducing costs of
  - Help desk support
  - Upgrade once, simultaneously, for all consumers
  - Single version of the software from a development and maintenance perspective.

Summary

- The cloud provides a new platform for applications with some different characteristics.

BITS Pilani

Pilani | Dubai | Goa | Hyderabad

Module 8
Part 2
Architectures for the Cloud -2

Harvinder S Jabbal
SE ZG651/ SS ZG653 Software Architectures

Architectures for the Cloud - 2

Session Outline

- Base Mechanisms
- Sample Technologies
- Architecting in a Cloud Environment
- Summary

Basic Mechanisms

- Hypervisor
- Virtual Machine
- File system
- Network

Virtual Memory Page Table

Virtual Memory Page Table

Virtual memory for non-virtualized application

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

Hypervisor Manages Virtualization

Virtual Machine

- A virtual machine has an address space isolated from any other virtual machine.
- Looks like a bare metal machine from the application perspective.
- Assigned an IP address and has network capability.
- Can be loaded with any operating system or applications that can execute on the processor of the host machine.

File System

- Each virtual machine has access to a file system.
- We will present HDFS (Hadoop Distributed File System) – a widely used open source cloud file system.
- We describe how HDFS uses redundancy to ensure availability.

HDFS Components

BITS Pilani, Deemed to be University under Section 3 of UGC Act, 1956

HDFS Write – Sunny Day Scenario

- Application writes as to any file system
- Client buffers until it gets 64K block
- Client informs NameNode it wishes to write a new block
- NameNode returns list of three DataNodes to hold block
- Client sends block to first DataNode and informs DataNode of other two replicas.
- First DataNode writes block and sends it to second DataNode. Second DataNode writes block and sends it to last DataNode.
- Each DataNode reports to client when it has completed its write
- Client commits write to NameNode when it has heard from all three DataNodes.

HDFS Write – Failure Cases

- Client fails
  - Application detects and retries
  - Write is not complete until committed by Client
- NameNode fails
  - Backup NameNode takes over
  - Log file maintained to avoid losing information
  - DataNodes maintain true list of which blocks they each have
  - Client detects and retries
- DataNode fails
  - Client (or earlier DataNode in pipeline) detects and asks NameNode for different DataNode.
- Since each block is replicated three times, a failure in a DataNode does not lose any data.

Network

- Every Virtual Machine is assigned an IP address.
- Every message using TCP/IP includes IP address in header.
- Gateway for cloud can adjust IP address for various purposes.

Sample Technologies

- IaaS
- PaaS
- DataBases

IaaS

- An arrangement of servers that manages the base technologies.
  - Servers are arranged in clusters
  - May be thousands of servers in a cluster
  - Some servers are used as the infrastructure of the IaaS
  - Every server has a hypervisor as its base.

IaaS Architecture

IaaS Architecture Components

- Cluster Manager responsible for managing each cluster
- Persistent Object Manager manages persistence
- Virtual Resource Manager manages other resources. It acts as a gateway for messages.
- The File System Manager is similar to HDFS. It manages the network wide file system.

Services Provided by IaaS

- Automatic reallocation of IP addresses in the case of a failure of the underlying virtual machine instance.
- Automatic Scaling. Create or delete new virtual machines depending on load.

PaaS

- Provides an integrated stack for developer.
- E.g. LAMP stack
  - Linux, Apache, MySQL, Python
- The developer writes code in Python and the PaaS manages assignment to underlying layers of the stack.

Databases

- Why relational databases came into question
  - Massive amounts of data are collected from web systems. Much of this data is processed sequentially and so RDBMSs introduce overhead, especially during creation and maintenance.
  - The CAP Theorem shows that it is not possible to simultaneously achieve consistency, availability, and partitioning.
  - The relational model is not the best model for some applications.
- Caused the introduction of new data models
  - Key-value
  - Document centric

Key Value – HBase

- One column designated as a key. The others are all values
- No schema so data can have key + any other values. The values are identified by their variable name.
- Data values are also time stamped
  - Hbase does not support transactions. Time stamps are used to detect collisions after the fact.

Document Centric – MongoDB

Stores objects rather than data

Access data through containing object

Objects can also contain links to other objects

No concept of primary or secondary index. A field is indexed or it is not.

What is Omitted From These DBs

Transactions. No locking is performed. The application must detect interference with other users.

Schemas. No predefined schemas. The application must use correct name.

Consistency. The CAP theorem says something must give. Usually consistency is replaced by “eventual consistency”

Normalization and Joins. Performing a join requires that the join field is indexed. Because there is not a guaranteed index field, joins cannot be performed. This means normalization of tables is not supported.

Architecting in a Cloud Environment

- Quality attributes that are different in a cloud
  - Security
  - Performance
  - Availability

Security

- Multi-tenancy introduces additional concerns over non-cloud environments.
  - Inadvertent information sharing. Possible that information may be shared because of shared use of resources. E.g. information on a disk may remain if the disk is reallocated.
  - A virtual machine “escape”. One user can break the hypervisor. So far, purely academic.
  - Side channel attacks. One user can detect information through monitoring cache, for example. Again, so far, purely academic.
  - Denial of Service attacks. One users can consume resources and deny them to other users.
- Organizations need to consider risks when deciding what applications to host in the cloud.

Performance

- Auto-scaling provides additional performance when load grows.
  - Response time for new resources may not be adequate
  - Architects need to be aware of resource requirements for applications
    - Build that knowledge into the applications
    - May applications self aware so that they can be proactive with respect to resource needs.

Availability

- Failure is a common occurrence in the cloud
  - With 1000s of servers, failure is to be expected
- Cloud providers ensure that the cloud itself will remain available with some notable exceptions.
- Application developers must assume instances will fail and build in detection and correction mechanisms in case of failure.

Summary

- The cloud provides a new platform for applications with some different characteristics.
- Architect needs to know how a cloud cluster works and pay special attention to
  - Security
  - Performance
  - Availability

Module 8
Part 03
Best Practices

Harvinder S Jabbal
SE ZG651/ SS ZG653 Software Architectures

BITS Pilani
Pilani | Dubai | Goa | Hyderabad

Best Practices

AWS Cloud

The Cloud Computing Difference:

- IT Assets become programmable resources
- Global, Availability, and Unlimited Capacity
- Higher Level Managed Services
- Security Built in.

AWS: Design Principles

- Scalability
- Disposable Resources Instead of Fixed Servers
- Automation
- Loose Coupling
- Services, not Server
- Database
- Removing Single Point of Failure
- Optimise for Cost
- Caching
- Security

Multi-tenant Applications for the Cloud: Micro Soft case Study

Windows Azure

- Goals and Requirements
  - The Tenant's Perspective
  - The Provider's Perspective
- Single Tenant vs. Mutliple Tenant
- Multi-Tenancy Architecture in Wondows Azure
- Selecting Single-Tenant or Multi-Tenant Architecture
  - Architectural Consideration
  - Application Life Cycle
  - Customizing the Application
  - Financial Consideration

Other Topics: Microsoft Azure

- Choosing a Multi-Tenant Data Architecture
- Partitioning Multi-Tenant Application
- Maximising Availability, Scalability and Elasticity
- Securing Multi-Tenant Applications
- Managing and Monitoring Multi-Tenant Applications

Microsoft Application Architecture Guide

Software Architecture and Design

- What is Software Architecture?
- Key Principles of Software Architecture
- Architectural Patterns and Styles
- A Technique for Architecture and Design

Microsoft Application Architecture Guide: Design Fundamentals

- Layered Application Guidelines
- Presentation Layer Guidelines
- Business Layer Guidelines
- Data Layer Guidelines
- Service Layer Guidelines
- Component Guidelines
- Designing Presentation Components
- Designing Business Components
- Designing Business Entities
- Designing Workflow Components
- Designing Data Components
- Quality Attributes
- Crosscutting Concerns
- Communication and Messaging
- Physical Tiers and Deployment

Microsoft Application Architecture Guide: Application Archetypes

- Choosing an Application Type
- Designing Web Applications
- Designing Rich Client Applications
- Designing Rich Internet Applications
- Designing Mobile Applications
- Designing Service Applications
- Designing Hosted and Cloud Services
- Designing Office Business Applications
- Designing SharePoint LOB Applications

Thank you

Module 8 Part 4 Review/The Road Ahead

Harvinder S Jabbal
SE ZG651/ SS ZG653 Software Architectures

The Road Ahead

Items for Preview

- SOA
- Cloud
- CAP Theorem

SOA

Service Oriented Architecture Pattern

- **Context**: A number of services are offered (and described) by service providers and consumed by service consumers. Service consumers need to be able to understand and use these services without any detailed knowledge of their implementation.

Problem: How can we support interoperability of distributed components running on different platforms and written in different implementation languages, provided by different organizations, and distributed across the Internet?

Solution: The service-oriented architecture (SOA) pattern describes a collection of distributed components that provide and/or consume services.

Service Oriented Architecture Example

Service Oriented Architecture Solution - 1

Overview: Computation is achieved by a set of cooperating components that provide and/or consume services over a network.

Elements:

- Components:
  - *Service providers,* which provide one or more services through published interfaces.
  - *Service consumers,* which invoke services directly or through an intermediary.
  - *Service providers* may also be service consumers.
- *ESB,* which is an intermediary element that can route and transform messages between service providers and consumers.
- *Registry of services,* which may be used by providers to register their services and by consumers to discover services at runtime.
- *Orchestration server,* which coordinates the interactions between service consumers and providers based on languages for business processes and workflows.

Service Oriented Architecture Solution - 2

– Connectors:

- *SOAP connector,* which uses the SOAP protocol for synchronous communication between web services, typically over HTTP.
- *REST connector,* which relies on the basic request/reply operations of the HTTP protocol.
- *Asynchronous messaging connector,* which uses a messaging system to offer point-to-point or publish-subscribe asynchronous message exchanges.

Service Oriented Architecture Solution - 3

Relations: Attachment of the different kinds of components available to the respective connectors

Constraints: Service consumers are connected to service providers, but intermediary components (e.g., ESB, registry, orchestration server) may be used.

Weaknesses:

- SOA-based systems are typically complex to build.
- You don’t control the evolution of independent services.
- There is a performance overhead associated with the middleware, and services may be performance bottlenecks, and typically do not provide performance guarantees.

Cloud Computing Deployments Should Begin With Service Definition

- Dennis Smith

Cloud Computing Strategies

- Infrastructure and operations leaders often struggle to understand the role of cloud computing and develop strategies that exploit its potential.
- Infrastructure & Operations leaders should complete the prerequisites before making the technology decisions required for successful, service-centered cloud computing strategies.

Key Challenges

- Many enterprises have failed to achieve success with cloud computing, because they failed to develop a cloud strategy rooted in the definition and delivery of IT services linked to business outcomes.
- Many companies are unsure how to initiate their cloud projects, which could cause them to miss chances to capitalize on business opportunities.

Recommendations

- Identify the cloud-computing-related IT services you will offer or procure.
- Document the internal processes that will be affected by the identified cloud services.
- Map applications and workloads to the associated cloud services.

Source: Gartner Report (July 2016)

CAP Theorem

CAP Theorem

- Described the *trade-offs involved in distributed system*

- It is impossible for a web service to provide following *three guarantees at the same time*:
  - **Consistency**
  - **Availability**
  - **Partition-tolerance**

CAP Theorem

Consistency:

- All nodes should see the same data at the same time

Availability:

- Node failures do not prevent survivors from continuing to operate

Partition-tolerance:

- The system continues to operate despite network partitions

- A distributed system can satisfy any two of these guarantees at the same time **but not all three**

Not Consistent

- Data A is being read a node in one partition.
- Data B is being written into a node another partition.

- This assures:

Availability:

  – Node failures do not prevent survivors from continuing to operate

Partition-tolerance:

  – The system continues to operate despite network partitions

Not Available

- Data A in one partition is locked while….
- Data B is being written into a node another partition.

- This assures:

Consistency:

  – All nodes should see the same data at the same time

Partition-tolerance:

  – The system continues to operate despite network partitions

Not Partition Tolerant

- Data A in one partition is being read based on update in …. a node another partition
- Data B is being written into the node in the other partition.

- This assures:

Availability:

– Node failures do not prevent survivors from continuing to operate

Consistency:

– All nodes should see the same data at the same time

Thank you

BITS Pilani

Pilani | Dubai | Goa | Hyderabad

Module 9 Part 1

Management and Governance

Harvinder S Jabbal
SSZG653 Software Architectures

Management and Governance

Outline

Planning

Organizing

Implementing

Measuring

Governance

Summary

Planning

- The planning for a project proceeds over time.
- There is an initial plan that is necessarily top-down to convince upper management to build this system and give them some idea of the cost and schedule.
- This top-down schedule is inherently going to be incorrect, possibly by large amounts.
- Once the system has been given a go-ahead and a budget, the architecture team is formed and produces an initial architecture design.

The Planning Process

Top Down Schedule

- A top down schedule is needed to enable management to decide whether to do the project and to allocate resources.

Example: For a medium size project (~150K SLOC)

- Number of components to be estimated: ~150
- Paper design time per component: ~4 hours
- Time between engineering releases: ~8 weeks
- Overall project development allocation:
  - 40 percent design: 5 percent architectural, 35 percent detailed
  - 20 percent coding
  - 40 percent testing

Remaining Planning Steps

- An architecture team is created and they develop the first level decomposition of the architecture.
- Each member of the architecture team will be the lead architect for each major subsystem.
- A bottom up schedule is created by the architecture team
  - Typically more accurate than the top down schedule
  - The top down and the bottom up schedules must be reconciled to produce final (initial) schedule.
- Software development plan is written that specifies releases dates and features per release. This plan guides the initial activities of the project.

Organizing

- Division of responsibilities between project manager and software architect
- Global Software Development

Project Manager and Software Architect

- This is the most important working relationship on the team.
- The people in each role—PM and SA—must
  - Respect each other
  - Coordinate
  - Stick to their respective spheres.

Project Management Body of Knowledge (PMBOK)

Published by the Project Management Institute

ANSI and IEEE standard

Nine project management areas

1. Integration Management
2. Scope Management
3. Time Management
4. Cost Management
5. Quality Management
6. Human Resource Management
7. Communications Management
8. Risk Management
9. Procurement Management

Integration Management

- Ensuring that the various elements of the project are properly coordinated.
- Developing, overseeing, and updating the project plan. Managing change control process.
  - PM: Organizes project, manages resources, budgets and schedules. Defines metrics and metric collection strategy. Oversees change control process.
  - SA: Creates design and organizes team around design. Manages dependencies. Implements the capture of the metrics. Orchestrates requests for changes. Ensures that appropriate IT infrastructure exists.

Scope Management

- Ensuring that the project includes all of the work required and only the work required.
- Requirements
  - PM: Negotiates project scope with marketing and software architect.
  - SA: Elicits, negotiates, and reviews run time requirements and generate development requirements. Estimates cost, schedule, and risk of meeting requirements.

Time Management

- Ensuring that the project completes in a timely fashion.
- Work breakdown structure and completion tracking. Project network diagram with dates.
  - PM: Oversees progress against schedule. Helps define work breakdown structure. Schedule coarse activities to meet deadlines.
  - SA: Helps define work breakdown structure. Defines tracking measures. Recommends assignment of resources to software development teams.

Cost Management

- Ensuring that the project is completed within the required budget.
- Resource planning, cost estimation, cost budgeting.
  - PM: Calculates cost to completion at various stages, makes decisions regarding build/buy and allocation of resources.
  - SA: Gathers costs from individual teams, makes recommendations regarding build/buy and resource allocations.

Quality Management

- Ensuring that the project will satisfy the needs for which it was undertaken.
- Quality & Metrics
  - PM: Defines productivity, size, and project-level quality measures.
  - SA: Designs for quality and tracks system against design. Defines code-level quality metrics.

Human Resource Management

- Ensuring that the project makes the most effective use of the people involved with the project.
- Managing people and their careers
  - PM: Maps skill sets of people against required skill sets. Ensures that appropriate training is provided. Monitors and mentors career paths of individuals. Authorizes recruitment.
  - SA: Defines required technical skill sets. Mentors developers about career paths. Recommends training. Interviews candidates.

Communications Management

- Ensuring timely and appropriate generation, collection, dissemination, storage, and disposition of project information.
- Communicating
  - PM: Manages communication between team and external entities. Reports to upper management.
  - SA: Ensures communication and coordination among developers. Solicits feedback as to progress, problems, and risks.

Risk Management

- Identify, analyze and respond to project risk.
- Risk Management
  - PM: Prioritizes risks, reports risks to management, takes steps to mitigate risks.
  - SA: Identifies and quantifies risks, adjusts architecture and processes to mitigate risk.

Procurement Management

- Acquire goods and services from outside organization.
- Technology
  - PM: Procures necessary resources. Introduces new technology.
  - SA: Determines technology requirements. Recommends technology, training, and tools.

Global Software Development

- A very common development context.
- Driven by
  - (Labor) costs
  - Skill sets and labor availability.
  - Local knowledge of markets.
- Global development means that coordination among teams is critical.

Team Coordination Induced by Module Interaction

If there is a dependency between two modules, the teams assigned to those modules must coordinate over the shared interfaces.

Coordination

- Local coordination can be informal and spontaneous.
- Remote coordination must be more structured.
- Coordination mechanisms:
  - Documentation
  - Meetings
  - Electronic media

Implementation Issues

- Trade-offs
- Incremental development
- Tracking progress

Trade-offs

- Software architect makes trade-offs among various quality attributes.
- Project manager makes trade-offs among
  - Features
  - Schedule
  - Quality
- Project manager should resist creeping functionality (scope creep)
  - Affects schedule
  - Can use a Change Control Board to manage (typically slow down) the pace of changes

Incremental Development

- A release may be in one of three states
  - Planning
  - Development
  - Test and repair
- All three states can be simultaneously active for different releases.

Tracking Progress

Progress can be tracked through

- Personal contact (doesn't scale)
- Meetings
- Metrics
- Risk management

Meetings

- Expensive use of time
- Either status or working – do not intermix
- One output of status meetings should be risks

Risks have

- Cost if they occur
- Likelihood of their occurrence

Project manager prioritizes risks

Measuring

- Metrics are an important tool for project managers. They enable the manager to have an objective basis both for their own decision making and for reporting to upper management on the progress of the project.
- Metrics can be global—pertaining to the whole project—or they may depend on a particular phase of the project.

Global Metrics

- Global metrics aid the project manager in obtaining an overall sense of the project and tracking its progress over time.
- Some example metrics, that any project should capture:
  - Size
  - Schedule deviation
  - Developer productivity
  - Defects
- Metrics should be tracked both historically for the organization and for the specific project.

Phase Metrics and Cost to Complete

Phase metrics

- Open issues
- Unmitigated risks

Cost to complete

- Bottom up metric
  - Responsibility of lead architect for each subsystem team

Governance

Four responsibilities of a governing board

1. Implementing a system of controls over the creation and monitoring of all architectural components and activities, to ensure the effective introduction, implementation, and evolution of architectures within the organization.
2. Implementing a system to ensure compliance with internal and external standards and regulatory obligations.
3. Establishing processes that support effective management of the above processes within agreed parameters.
4. Developing practices that ensure accountability to a clearly identified stakeholder community, both inside and outside the organization.

Summary

A project must be planned, organized, implemented,, tracked, and governed.

- Top down schedule based on size
- Bottom up schedule based on top level decomposition.
- Reconciliation of two schedules is the basis for the software development plan.

Teams are created based on the software development plan.

The software architect and the project manage must coordinate to oversee the implementation.

Global development creates a need for an explicit coordination strategy.

Management trade offs are between schedule, function, and cost.

Progress must be tracked.

Larger systems require formal governance mechanisms.

BITS Pilani

Module 9 Part 2

Mobile application architecture

Harvinder S Jabbal
SSZG653 Software Architectures

Architecting Mobile applications

Can you give some examples of mobile apps?

Mobile applications

Examples

- Uber
- Swiggy
- Courier delivery
- eCom
- Banking
- Spotify
- Where Is My Train
- …

Mobile Application: Design considerations

- Simple User interface: Easy to type, Large buttons, Minimal features, menu options, actions
- Responsive design: Adapt to different screen sizes & orientations
- Compact code: Less usage of CPU, memory, storage
- Few layers to ensure performance
- Connectivity: Store data locally and synchronize later if connection is poor

Types of Mobile Apps

Ref: buildfire.com

Mobile Native app

- They are built for specific platforms
  - Examples: Google Maps, Facebook, LinkedIn – one version for iOS and one for Android

- Languages used:
  - Native iOS : Swift or Objective-C
  - Native Android: Java or Kotlin

- Integrated Development Environment (IDE) such as Android studio are used for this

Mobile Web apps

- Progressive web apps use modern web technology to deliver app-like experiences to users, right in their browsers.

- Examples:
  - Flipkart
  - BookMyShow
  - MakeMyTrip

- Uses HTML5, CSS3, JavaScript and runs on a browser

Mobile web application

Hybrid app

- A hybrid app combines elements of both native apps and web applications.
  - Examples: Twitter, Uber, Instagram

- Hybrid apps are essentially web apps (HTML, CSS, Javascript) that have been put in a native app shell.

- The shell is able to connect to native capabilities of the mobile platform such as camera, accelerometer, GPS, etc.

- Tools such as Xamarin and React Native allows app to run across platforms

Pros & Cons

App type	Pro	Con
Native	• High performance • Superior user experience • Access to all features of OS	• Runs only on one platform • Need to know special language • Need to update versions
Web App	• Easy to deploy new versions • Common code base	• Little scope to use device hardware • Lower user experience • Need to search for app
Hybrid	• Does not need browser • Single code base • Access to device hardware	• Slower

UI design patterns

- Action bars for quick access to frequently used actions
- Login using Facebook, Google, etc. instead of separate user id / password
- Large buttons for ease of use
- Notifications of recent activity
- Discoverable controls: Controls show up only when an item is selected (ex. In WhatsApp, the Forward button shows up when a message is selected)

Mobile optimized web site

innovate achieve lead

All features

Reduced features

Responsive design

Example: Boston Globe News

This is how the display looks on desktop

A

Responsive design

Example: Boston Globe News

B

C

This is how the display looks on mobile

Adjusted vertically

Responsive design

Single website for laptop & mobile & tablet

Principle: Adapt rendering depending on screen sizes & orientation

Ref: Wikipedia

Responsive design

Ref: Wikipedia

Flexi grids

- Divide a screen into multiple columns
- Assign HTML elements to one or more columns
- Choose a different layout depending on screen size

Responsive design

Technique

- Use CSS and HTML to shrink, hide or move content
- Flexible grids (CSS 3)
  - Use media queries to determine screen size
  - Specify grid width as % of screen size rather than fixed pixels
- Flexible images – Specify image size as % of grid size

Store locally, Sync later
In case of intermittent connectivity

Doctors enter patient data in mobile, which gets synced with server later

Mobile Application Architecture

Android application architecture

4 types of components

- Activity (UI)
- Service (background process) - ex. playing music, download
- Content provider (Storage) - ex. SQLite, files,
- Broadcast receiver (Acts on events received from OS and other apps) - Ex arrival of SMS

Examples of components

Activity & Content providers

Personal Calendar App

(Background) Service

Broadcast receiver (Event handlers)

Patient monitoring system

Android application structure

Exercise: Mobile app components

Give examples of mobile app components in Uber app.

UI

Screen to book a cab

Broadcast receiver

Receive location of cab from backend server and provide to UI for display

Service

Provide cab location to backend server after journey starts

Database

Configuration file containing user data

Exercise

Consider a mobile app carried by the courier delivery boy.

The app should support the following functions:

a) View courier packages to be delivered
b) Mark a package as delivered.
c) Upon this event, the app should send information to central server

Identify the components of a mobile app & its inter-connections and draw an appropriate software architecture diagram

a) View courier packages to be delivered

Courier boy's mobile

b) Mark a package as delivered. Upon this event, the app sends information to central server

Review questions

Mobile apps

1. What is a cross platform mobile app?
2. If connectivity is poor, how do we ensure consistency between data in mobile phone and backend server?
3. What is ‘Broadcast receiver’ in an Android app?

Appendix

Popular services of cloud vendors

Google App Engine	Microsoft Azure	Amazon Web Services
• Python & Java development environment • Auto scaling • Database replication	• .Net environment • Auto scaling • Load balancing • Failure detection & auto replication of services • Database replication	• Java development • Auto scaling • Load balancing • Failure detection & auto replication of services • Database replication • Data caching (Memcached) • Service discovery (SoA) • Notification of events (SNS) • Message queue (SQS) • CDN (Content Delivery Network) – YouTube

Mobile app: Example

Amazon's approach to handling issues in distributed systems

- To scale you have to partition, so you are left with choosing either high consistency or high availability for a particular system. You must find the right overlap of availability and consistency.

- Choose a specific approach based on the needs of the service.

- For the checkout process you always want to honor requests to add items to a shopping cart because it's revenue producing. In this case you choose high availability. Errors are hidden from the customer and sorted out later.

- When a customer submits an order you favor consistency because several services--credit card processing, shipping and handling, reporting--are simultaneously accessing the data.

Issues in Cloud based systems

- Availability
  - Cloud vendors promise high availability ex. 99.95%
  - But still not 100%
  - Need to design for the 0.05%
  - One approach: Store same data in different geographical zones as done by Netflix

Exercise

Scenario

A start-up company is developing a GST tax returns filing system. This software will be deployed on the Cloud and offered to small and medium businesses as a SaaS.

The clients will have to input their data or upload data using an Excel file. They also need to provide other details such as GST #, etc. The software will process the data and file the returns into the Government's GST system on behalf of the client.

After developing the software, what options exist for deploying it in the cloud?

Different ways to deploy applications on the cloud

Scenario

A start-up company – Tailspin - is developing a **customer survey & analysis** application. This software will be deployed on the Cloud and offered to clients as a SaaS.

- The clients (subscriber of the application) can create and launch a survey.
- After this the survey participants will access the application and answer the survey questions.
- After the data has been gathered the application will perform analysis and present the results to the client organization

What options exist for Tailspin to deploy the application on the cloud?

Context diagram

FIGURE 1
The Surveys application

Multi-tenant application Architecture – High level

FIGURE 2
The Surveys application architecture

Architecture Options – High level

Issues in Cloud based systems

- Security issue due to multi-tenancy
  - Poor design leading to inadvertent sharing of information
  - Virtual machine 'Escape'

Other fields of the table					OU_ID
					Org1
					Org1
					Org1
					Org2
					Org2

Same table contains data of different organizations

Virtual machine 'Escape'

How to choose your multi-tenancy degree?

IT workloads will reside in a hybrid environment of public and private clouds. Both will be multi-tenant.

https://www.computerworld.com/article/2517005/data-center/multi-tenancy-in-the-cloud--why-it-matters.html

How to choose your multi-tenancy degree?

The **characteristics of the workload** in question have to be carefully studied first, including the workload's **utilitarian versus strategic value, volatility, security, etc.**

Higher degrees of multi-tenancy are best suited for cross-industry utilitarian workloads such as **e-mail, expense reporting, travel authorization and sales force management.**

These applications can very easily share the **same schema.**

https://www.computerworld.com/article/2517005/data-center/multi-tenancy-in-the-cloud--why-it-matters.html

Degree of multi-tenancy

- Highest degree: IaaS and PaaS are multi-tenant. SaaS is fully multi-tenant also.
- Middle degree: IaaS and PaaS are multi-tenant. Small SaaS clusters are multi-tenant.
- Lowest degree: IaaS and PaaS are multi-tenant. SaaS is single tenant.

‘Cloud Native’ applications

- Cloud-native computing takes advantage of many modern techniques, including
  - PaaS,
  - multicloud,
  - microservices,
  - agile methodology,
  - containers,
  - CI/CD, and
  - devops

What is cloud native?

Reference: https://www.infoworld.com/article/3281046/what-is-cloud-native-the-modern-way-to-develop-software.html

Exercise

When would you use the following options:

a) App 1
Web tier – Multi-tenant
App tier – Single tenant
Data tier – Multi-tenant

b) App 2
Web tier – Single tenant
App tier – Multi-tenant
Data tier – Single tenant

Exercise

When would you use the following options:

a) App 1

Web tier – Multi-tenant

App tier – Single tenant

Data tier – Multi-tenant

Answer: Web tier processing is light, App processing is heavy, data is not confidential

b) App 2

Web tier – Single tenant

App tier – Multi-tenant

Data tier – Single tenant

Answer: Web tier processing is heavy, App processing is light, data is confidential

Mobile Application Architecture

Types of mobile apps	Characteristics
Native app	Makes use of OS and native devices. Ex. Games
Cross platform app	Same code runs on multiple mobile platforms such as Android and iOS
Mobile web application	Has a mobile component which interacts with a server component. Ex. Uber, PayTM, Banking

Tools and technology for mobile app development

Development Approach	Native	Cross-Mobile Platforms	Mobile Web
Definition and Tools	Build the app using native frameworks: - iPhone SDK - Android SDK - Windows Phone SDK	Build once, deploy on multiple platforms as native apps: - RhoMobile - Titanium Appcelerator - PhoneGap - Worklight - Etc.	Build using web technologies: - HTML5 - Sencha - JQuery Mobile - Etc.
Underlying Technology	- iPhone: Objective C - Android: Java - Windows Phone: .NET	- RhoMobile: Ruby on Rails - Appcelerator: Javascript, HTML - PhoneGap: Javascript, HTML - Worklight: Javascript, HTML	- Javascript, HTML

Xamarin – cross platform tool

BITS Pilani

Module 9 Part 3
Big Data technologies

Harvinder S Jabbal
SSZG653 Software Architectures

Contents

1. Big data
2. Hadoop, HDFS, Map-Reduce
3. Analytics & Real time analytics
4. In-Memory database
5. NoSQL databases

Big data & Analytics

Wikipedia defines ***"Big Data"*** as a collection of data sets so large and complex that it becomes difficult to process using on-hand database management tools or traditional data processing applications.

History of Big Data

Lots of data got created due to

- Proliferation of Internet
- Social media
- eCommerce

Before we begin, let us understand the scale of data we deal with

Unit	Power of 10
Mega byte	6
Giga	9
Tera	12
Peta	15

Big Data Statistics

- Volume of data
  - 500+ new websites are created every minute of the day
  - 100 Terabytes of data is uploaded to Facebook every day
  - Twitter generates 12 Terabytes of data every day
  - YouTube users upload 48 hours of new video content **every minute** of the day

- Processing
  - Decoding of the human genome used to take 10 years. Now it can be done in 7 days
  - Facebook Stores, Processes, and Analyzes more than 30 Petabytes of user generated data
  - LinkedIn processes and mines Petabytes of user data to power the "People You May Know" feature

Source: Wikibon - A Comprehensive List of Big Data Statistics

Characteristics of Big data

Surveillance videos, satellites images, cell phone location, health of power station turbines, furnaces & other industrial machinery, weather and meteorological data, click stream

Archived data:
Patient records, Scanned copies of agreements, records of ex-employees/completed projects, banking transactions older than the compliance regulations

Ex. Trading/stock exchange data, tweets on Twitter, status updates/likes/shares on Facebook

Use of big data

- Recommend cancer medication based on what worked well in similar situations for other patients
- Weather prediction for fishermen, farmers
- Predict equipment malfunctioning in large nuclear power plant, chemical plants, etc.
- Credit card fraud detection

Example of handling big data

Google search

Do you know how Google Search works?

3 steps:

- **Catalog & index**: Even before you search, Google goes through various web sites and linked web sites (crawling) and catalogs all the web sites. This runs into several Tera bytes.

- **Understand & enrich your query**: Correct spelling mistakes, consider synonyms, etc.

- **Search:** When the search is requested, it uses this catalog to determine which web sites closely match the requirement. For this it uses a 'Page Rank' (named after Larry Page) algorithm which considers factors such as which page has max number of occurrences of the search string, how many other web sites refer to this web site, what is the reputation of the websites that refer to this website, etc.

Google – Big table

So Google invented a data storage structure called Big Table

Google BigTable

Google BigTable is a wide table containing several attributes of a website;

Here are some attributes:

- The contents of Web page
- Anchor text*
- Websites referencing the page.
- Time stamp when the data was stored

Google BigTable is built on technologies like Google File System (GFS)

Google BigTable is used by applications such as Google Maps, Google Analytics, etc.

*Anchor text is the text that appears in Blue colour in search result. It contains a link to the website

Google Big Table

Features:

- Versioning of data,
- Compression,
- Distribution across servers,
- Fault tolerant,
- Fast access,
- Dynamic addition of servers,
- Load balancing

Google disclosed the design of Big table. Then came Hadoop Distributed File System (Yahoo) and serveral NoSQL databases

Use of Big Data

Banking and Financial Services

- Fraud Detection to detect the possible fraud or suspicious transactions in Accounts, Credit Cards, Debit Cards, and Insurance etc.

Retail

- Targeting customers with different discounts, coupons, and promotions etc. based on demographic data like gender, age group, location, occupation, dietary habits, buying patterns, and other information which can be useful to differentiate/categorize the customers.

Sentiment Analysis

- Organizations use the data from social media sites like Facebook, Twitter etc. to understand what customers are saying about the company, its products, and services.
- Words like “I like this phone”, “This food is too salty”, etc.. indicate sentiments
- This type of analysis is also performed to understand which companies, brands, services, or technologies people are talking about.

Use of Big Data …

Customer Service

- IT Services and BPO companies analyze the call records/logs to gain insights into customer complaints and feedback, call center executive response/ability to resolve the ticket, and to improve the overall quality of service.
- Call center data from telecommunications industries can be used to analyze the call records/logs and optimize the price, and calling plan, messaging plan, and data plans

Industrial equipment monitoring & alerting

- A large power plant or chemical factory has thousands of critical equipment that needs to be monitored
- The equipment data needs to be analysed to detect any malfunctioning or danger of accidents

Weather forecasting

- Satellite data from remote sensing satellites need to be analysed at high speed to warn fishermen, farmers and public about potential cyclones, delayed monsoon, etc.

BigTable can be used to store…

- **Time-series data,** such as CPU and memory usage over time for multiple servers.
- **Marketing data,** such as purchase histories and customer preferences.
- **Financial data,** such as transaction histories, stock prices, and currency exchange rates.
- **Internet of Things data,** such as usage reports from energy meters and home appliances.
- **Graph data,** such as information about how users are connected to one another.

Big data architecture / Eco-system

Hadoop

Hadoop is an open source framework, from the Apache foundation, capable of processing large amounts of heterogeneous data sets in a distributed fashion across clusters of commodity computers and hardware using a simplified programming model.

Hadoop provides a reliable shared storage and analysis system.

Components of Hadoop

HDFS (Hadoop Distributed File System)

- HDFS offers a highly reliable and distributed storage, and ensures reliability, even on a commodity hardware, by replicating the data across multiple nodes.
- Unlike a regular file system, when data is pushed to HDFS, it will automatically split into multiple blocks (configurable parameter) and stores/replicates the data across various data nodes. This ensures high availability and fault tolerance.

MapReduce

- MapReduce offers an analysis system which can perform complex computations on large datasets.
- This component is responsible for performing all the computations and works by breaking down a large complex computation into multiple tasks and assigns those to individual worker/slave nodes and takes care of coordination and consolidation of results

Hadoop - HDFS

Figure 9-1: How a Hadoop cluster is mapped to hardware.

Ref: Big data for Dummies

Hadoop - HDFS

Data

- Large files are broken down into blocks (128 MB usually) and spread across Data nodes.
- Data blocks are replicated and Degree of replication can be adjusted

Meta data

- Name node stores meta data – data about files, distribution of data (which block is in which nodes), etc.
- For good performance, all the metadata is loaded into the physical memory of the NameNode server.

Features

- Data nodes provide Heart beat messages to Name node
- Supports data pipelines. A connection between data nodes to move data from one node to another
- Rebalancer: Balances distribution of data

Map-Reduce pattern

- Used to analyse vast amount of data

- Suppose we keep track of every click of the user on a web site and store these details in a database
- Let us say we want to find out the average time spent by users on each web page of the web site, across thousands of users who visited the web site in the last 30 days
- How can we speed up the analysis?

Map-Reduce pattern: Example

Map-Reduce pattern

- Executes in parallel
- Leads to low latency & high availability
- Map performs extract & transform and produces <Key, Value> instances
- Reduce summarizes transformed data

Map Reduce pattern

Source: https://www.quora.com/What-is-the-relationship-between-MapReduce-and-Hadoop

Map-Reduce pattern

- **Example**: Determine the average time (duration) spent by users on different web pages

- Step 1: **Map** processes data on each node are outputs <Web page, (Avg time, # of users)>

- Step 2: **Reduce** produces <Web page, weighted Avg time>

Map – Reduce pattern

Example: Determine the average duration spent by users on different web pages

Experience Sharing

Have you come across systems that use this pattern?

Hadoop - HDFS

Features

- Can store peta bytes of data
- Distributed
- Replicated
- Fault tolerant (self healing)

Using Hadoop

When to Use Hadoop (Hadoop Use Cases)

Hadoop can be used in various scenarios including some of the following:

- Analytics
- Search
- Data Retention
- Log file processing
- Analysis of Text, Image, Audio, & Video content
- Recommendation systems like in E-Commerce Websites

When Not to Use Hadoop

There are few scenarios in which Hadoop is not the right fit. Following are some of them:

- Low-latency or near real-time data access.
- If you have a large number of small files to be processed. This is due to the way Hadoop works. Namenode holds the file system metadata in memory and as the number of files increases, the amount of memory required to hold the metadata increases.
- Multiple writes scenario or scenarios requiring arbitrary writes or writes between the files.

Hadoop may still be slower for some use cases

- Sometimes we require even faster processing than Map-Reduce, for example in real time fraud detection
- Map Reduce is disk based
- If we can retrieve disk data into memory and then use it for further processing, we can get even better response time

Difference between Hadoop & Spark

Iterative operation using Hadoop using disk

Iterative operation using Spark using memory

Source: https://www.tutorialspoint.com/apache_spark/apache_spark_rdd.htm

Apache Spark

- Apache Spark is open source, general-purpose distributed computing engine used for processing and analyzing a large amount of data
- Main feature: In-memory cluster computing
- Useful for real time computations

Spark architecture

Real time analytics

- Real time analytics lets users see, analyze and understand data as it arrives in a system.

- It can give users insights for making real-time decisions.

- Examples:
  - Real time advertising
  - Identify security breaches
  - Sensor data processing to predict issues in machines

Real time analytics - Fraud detection in bank

Continuous monitoring of client's activity to see if there are any potential issues

Real time analytics at Dream11 – a fantasy sports platform

Objectives:

1. Know the real-time rate of contest joins
2. Know the real-time aggregated status of payment gateways
3. Identify real-time anomalies eg: unusual traffic on the system
4. Realtime aggregated view of outcome of marketing campaigns
5. How customers are using discount coupons once promotion goes live
6. Realtime alerting once Mega contest is above 90%

Real time analytics at Dream11 – a fantasy sports platform

Dream11 case study

Ref: medium.com

IoT: Internet of Things

Inter-connected devices which work together and perform operations with little human intervention

Examples:

- Tracking machine parameters using sensors and controlling for optimum performance
- Tracking goods, real time information exchange about inventory among suppliers and retailers
- Sensing for soil moisture and nutrients, controlling water usage for plant growth

Azure IoT reference architecture

Ref: docs.microsoft.com

Appendix

BITS Pilani

Module 9 Part 4
NoSQL Databases

Harvinder S Jabbal
SSZG653 Software Architectures

NoSQL databases

- While Relational databases (SQL databases) are good for transaction management, not all applications need this feature.
- Many web applications such as marketing applications, IoT applications, need fast processing of data but do not need transaction management
- NoSQL databases came as a response to this need

https://www.dataversity.net/a-brief-history-of-non-relational-databases/#

NoSQL databases

Traditional databases also have limitations on storage

NoSQL databases are

- Scalable (Sharding)
- Fast (In Memory)
- Available (Replication)
- Handle semi-structured and unstructured data
- Rapidly adapt to changing data needs

However most lack

- Transaction support (ACID)
- Join feature

Examples of usage of NoSQL DB

- **Tesco**, Europe's #1 retailer, uses NoSQL to manage millions of products, promotions supply chain, etc.
- **Ryanair**, the world's busiest airline, uses NoSQL to power its mobile app serving over 3 million users
- **Marriott** is deploying NoSQL for its reservation system that books $38 billion annually
- **Gannett** (USA Today) the #1 U.S. newspaper publisher, uses NoSQL for its proprietary content management system, Presto
- **GE** is deploying NoSQL for its Predix platform to help manage the Industrial Internet

NoSQL - Flexibility

In RDBMS, if we want to add a new column, we need to change the schema

This can not be stored

Ref: https://www.couchbase.com/resources/why-nosql

NoSQL - Flexibility

In NoSQL DB, we can easily add new columns.

Ref: https://www.couchbase.com/resources/why-nosql

NoSQL - Simplicity

NoSQL

SQL

NoSQL - Simplicity

NoSQL – Cost effective

Cost: Memory, CPU, storage

NoSQL Database

Types of NoSQL databases:

- Document
- Key Value
- Column stores
- Graph

Document NoSQL database

- The database stores and retrieves documents, which can be XML, JSON, BSON, and so on.
- Some of the popular document databases we have seen are MongoDB, CouchDB , Terrastore, OrientDB , RavenDB, and of course the well-known and often reviled Lotus Notes that uses document storage.

<Key=CustomerID>

```
{
    "customerid": "fc986e48ca6"
    "customer":
    {
    "firstname": "Pramod",
    "lastname": "Sadalage",
    "company": "ThoughtWorks",
    "likes": [ "Biking","Photography" ]
    }
    "billingaddress":
    { "state": "AK",
      "city": "DILLINGHAM",
      "type": "R"
     }
}
```

Key

Example of one record

Document DB vs Relational DB

Good for

- Ecommerce platform
- Content management systems

Features of Mongo DB

- Indexing
- Ad hoc search
- Replication
- Partitioning / Sharding
  - Ex. Partition data by Product or Geography

Key-Value database

- Data model is Key value pair
- Uses hashing for fast access
- The DB does not care what is contained in the value
- Example scenarios: Phone directory, Stock trading

- Some of the popular key-value databases are Riak, Redis (often referred to as Data Structure server), Memcached and its flavors, Berkeley DB, upscaledb (especially suited for embedded use), Amazon DynamoDB (not open-source), Project Voldemort and Couchbase.

Phone Directory

Key	Value
Bob	(123) 456-7890
Jane	(234) 567-8901
Tara	(345) 678-9012
Tiara	(456) 789-0123

Example of key value database

Stock Trading

This example uses a list as the value.

The list contains the stock ticker, whether its a “buy” or “sell” order, the number of shares, and the price.

Key	Value
123456789	APPL, Buy, 100, 84.47
234567890	CERN, Sell, 50, 52.78
345678901	JAZZ, Buy, 235, 145.06
456789012	AVGO, Buy, 300, 124.50

More examples: User profiles, Blog comments

Uses of Redis

- Session cache, with persistence

Column oriented database

This is useful for data analysis scenarios

Example: Calculate the average usage of electricity in 2018 in East Bangalore region

Traditional way: Read all the billing records of 2018

Cust id, Name, Addrs, Region, Month, Year, Usage, Amt,

Record 1: 001, John Mancha, Addr 1, East, Jan, 2018, 100, 600

Record 2: 002, Vivek Kulkarni, Addr 2, East, Jan, 2018, 90, 540

Record 3: 003, Shanti Sharma, Addr 3, West, Jan, 2018, 110, 660

….

….

Column oriented database

Instead if we store the data as follows:

Record 1: 001, John Mancha, Addr 1, East // Customer details
Record 2: 002, Vivek Kulkarni, Addr 2, East

Record A: Jan, 2018, 100, 90, 110, … // Usage – in customer order
Record B: Feb, 2018, 110, 92, 115, …

Only one record 'Record A' is needed to calculate average usage in Jan 2018

Good for summarization

Graph database: Example: Knowledge graph

Entities and their relationships

Graph database

Entities and relationships have properties (attributes)

Ex.

- Eifel tower properties can be height, date of construction
- Visited relationship can have properties such as date of visit

Uses

- Store a large amount of inter-related information and to search for an entity along with its relationships
- Fraud detection
- Social networks

Source: https://opensourceforu.com/2017/05/different-types-nosql-databases/

Can be use to detect hidden patterns

Easy to understand clusters

Graph database

- Graph databases allow you to store entities and relationships between these entities.
- Entities are also known as nodes, which have properties. Think of a node as an instance of an object in the application.
- Relations are known as edges that can have properties. Edges have directional significance; nodes are organized by relationships which allow you to find interesting patterns between the nodes.
- There are many graph databases available, such as Neo4J, Infinite Graph, OrientDB, or FlockDB

In-Memory databases

- An **in-memory database** (**IMDB**; also **main memory database system** or **MMDB** or **memory resident database**) is a database management system that primarily relies on main memory
- Applications where response time is critical, such as those running telecommunications network equipment and mobile advertising networks, often use main-memory databases
- With the introduction of non-volatile random access memory technology (Flash memory), in-memory databases will be able to run at full speed and maintain data in the event of power failure
- Popular In-memory databases are SAP's HANA, IBM DB2 BLU, Oracle
- These databases support OLTP and OLAP (Online Analytical Processing)

Acknowledgement

Sources

Hadoop

https://www.mssqltips.com/sqlserverauthor/77/dattatrey-sindol/

NoSQL Database

https://www.thoughtworks.com/insights/blog/nosql-databases-overview

Appendix

Typical Architecture of Hadoop

Files read from HDFS
Input Format
Split
Split
RR
RR
Map
Map
Film
Film
Partitioner
Sort
Shuffle process
reduce
Output Format
Files written to HDFS
Input Format
Split
Split
RR
Map
Map
Partitioner
Sort
reduce
Output Format
Files read from HDFS
Film
Film
Files written to HDFS

Example of Stream processing in a cab hailing company like Ola / Uber

Analytics

Data Visualization

Multi-dimensional data

Data mining

Examples…

Tools ….

Real-Time Streaming and Data Analytics For IoT

Source: https://www.xenonstack.com/blog/big-data-engineering/real-time-streaming-analytics-tools/

Traditional Data Processing

ODS: Operational Data Store

Modern (Next Generation) Data Processing & Management

Use cases of Stream Processing

Following are some of the use cases.

- Algorithmic Trading, Stock Market Surveillance,
- Smart Patient Care
- Monitoring a production line
- Supply chain optimizations
- Intrusion, Surveillance and Fraud Detection (e.g. Uber)
- Most Smart Device Applications : Smart Car, Smart Home ..
- Smart Grid—(e.g. load prediction and outlier plug detection see Smart grids, 4 Billion events, throughout in range of 100Ks)
- Traffic Monitoring, Geo fencing, Vehicle and Wildlife tracking—e.g. TFL London Transport Management System
- Sport analytics—Augment Sports with realtime analytics (e.g. this is a work we did with a real football game (e.g. Overlaying realtime analytics on Football Broadcasts)
- Context-aware promotions and advertising
- Computer system and network monitoring
- Predictive Maintenance, (e.g. Machine Learning Techniques for Predictive Maintenance)
- Geospatial data processing

neotechnology
graphs are everywhere

Graph database

Difference between Hadoop & Spark…

Interactive operation using Hadoop

Interactive operation using Spark

Source: https://www.tutorialspoint.com/apache_spark/apache_spark_rdd.htm

Real-time analytics

Detecting bank fraud requires real-time analytics as events happen

Such situations demand processing of each event as they happen rather processing a batch of data on disk

This led to tools such as Spark Streams and Storm which support in-memory processing, than disk based processing

Storm for real time computing

Apache Storm is a free and open source **distributed real time computation system**. Storm makes it easy to reliably process unbounded streams of data, doing for realtime processing what Hadoop did for batch processing. Storm is simple, can be used with any programming language, and is a lot of fun to use!

Storm has many **use cases**: **real time analytics, online machine learning, continuous computation**, distributed RPC, ETL, and more. Storm is fast: a benchmark clocked it at over a million tuples processed per second per node. It is scalable, fault-tolerant, guarantees your data will be processed, and is easy to set up and operate.

Storm **integrates with the queueing and database technologies** you already use. A Storm topology consumes streams of data and processes those streams in arbitrarily complex ways, repartitioning the streams between each stage of the computation however needed. Read more in the tutorial.

Source: http://storm.apache.org/

Storm topology

Twitter uses Storm for real-time analytics, personalization, search, revenue optimization
Groupon uses Storm for Real-time data integration systems
Yahoo! Uses Storm for processing user events, content feeds, and application logs.

Architecture of Storm system

Combining Kafka and Storm for real time computing

Kafka is a distributed and robust queue that can handle high volume data and enables you to pass messages from one end-point to another (Publish - Subscribe)

Source: https://hortonworks.com/blog/storm-kafka-together-real-time-data-refinery/

Storm & Kafka

The common flow of these tools (as I know it) goes as follows:

real-time-system --> Kafka --> Storm --> NoSql --> BI(optional)

MapReduce – Word Count Example Flow

BITS Pilani

Module 9 Part 5

Security Technology & Tools

Harvinder S Jabbal
SSZG653 Software Architectures

Contents

- Transport Layer Security (TLS)
- OpenID & OpenAuth
- LDAP
- Identity & Access management
- Firewalls

Introduction

The objective of this session is to provide an introduction to a few important technology topics.

Transport layer security (TLS)

(Older version is SSL)

- Used for secure communication between **client & server** (example between browser and a web site)

- It provides

  - **Privacy:** No intruder can know what communication is going on
  - **Data integrity:** Data being communicated can not be modified by an intruder. If he does, it can be detected

TLS: How does it work?

Client & server do the following:

- Agree on the version of the TLS protocol to use.
- Select cryptographic algorithms to use
- Authenticate each other by validating digital certificates.
- Generate a shared secret key, for the symmetric encryption of messages (This is faster than asymmetric encryption)

Source: https://www.ibm.com/support/knowledgecenter/en/SSFKSJ_7.1.0/com.ibm.mq.doc/sy10660_.htm

Encryption algorithms used

- Key Exchange Algorithms (RSA, DH, ECDH, DHE, ECDHE, PSK)
- Authentication/Digital Signature Algorithm (RSA, ECDSA, DSA)
- Bulk Encryption Algorithms (AES, CHACHA20, Camellia, ARIA)
- Message Authentication Code Algorithms (SHA-256, POLY1305)

TLS / SSL steps

Source: https://www.ibm.com/support/knowledgecenter/en/SSFKSJ_7.1.0/com.ibm.mq.doc/sy10660_.htm

OpenID

- We use several websites.
- One issue we face is, remembering user ids & passwords of several websites
- With OpenID technology, we can use a single account, such as Facebook, Google or Yahoo, to sign-in to thousands of websites

Sample login page: American Cancer Society

Sample login page: Kodak Tips and project Exchange

innovate achieve lead

OpenID: How does it work?

(One typical approach)

- The website redirects the user to an OpenId provider such as Facebook or Google
- The OpenID provider authenticates the user
- The user is redirected back to the website along with the end-user's credentials (such as user id, but not the password)

Reference: https://en.wikipedia.org/wiki/OpenID

OAuth

OpenID is to authenticate users.

OAuth authorizes a client to access your data stored in another website such as Yahoo (data such as your profile, contacts in Yahoo)

Reference: https://tools.ietf.org/html/draft-ietf-oauth-use-cases-01#section-2.1

OAuth: Use cases

Use case 1:

- An application can *use OAuth* to obtain permission from users to store files in their Google Drives.

Use case 2

- A photo printing application (www.printphotos.example.com) can access your photographs stored on a server www.storephotos.example.com and then print them

Reference: https://tools.ietf.org/html/draft-ietf-oauth-use-cases-01#section-2.1

OAuth: How does it work?

- You try to log onto a website and it offers opportunities to log on using Google, Yahoo, etc. Let us say you choose Google.
- You are redirected to Google
- Google authenticates you, and returns a token to the website.
- The token enables the website to login to Google as you, and access resources such as your contacts.

https://www.csoonline.com/article/3216404/what-is-oauth-how-the-open-authorization-framework-works.html

LDAP: Lightweight Directory Access Protocol

Scenario suitable for LDAP

- Imagine you have a website that has a million registered users with thousands of page requests per second.
- By using LDAP, you can easily offload the user validation and gain significant performance improvement.
- Good use cases for LDAP:
  - You need to locate ONE piece of data many times and you want it fast
  - You don’t update, add, or delete the data very often
  - The size of each data entry is small

https://stackoverflow.com/questions/239385/what-is-ldap-used-for

LDAP

- LDAP is an Internet protocol to talk to a Directory service such as Active Directory of Microsoft
- Directory services store information in a tree structure
- LDAP is used in many open source solutions such as Docker, Kubernetes, Jenkins, etc.

Identity & Access management

- Identity and access management (IAM), is a framework for ensuring that **people in an enterprise have the appropriate access** to technology resources.

- IAM solutions are typically used in large organizations to ensure regulatory compliance.

- **Features of IAM**
  - Authentication of a user
  - Authorization to use applications
  - Definition of Roles. A User belonging to a group is authorized to perform certain operations defined for the role. Operations such as create sales order, approve credit card request, etc.
  - Delegation of permission to another user
  - Interchange identify information with trusted entities using OpenID, etc.

Leading Identity & Access management products

- Azure Active Directory
- IBM Security Identity and Access Assurance
- Oracle Identity Cloud Service
- Okta

Firewall

- A firewall is a network security device that **monitors incoming and outgoing network traffic** and decides whether to **allow or block** specific traffic based on a defined set of **security rules.**

De-Militarized Zones (DMZ)

DMZ acts a buffer between Internet and organization network

https://searchsecurity.techtarget.com/definition/DMZ

How DMZs work?

- DMZs are intended to function as a sort of **buffer zone between the public internet and the organizational network.**

- If a better-prepared threat actor is able to get through the first firewall, they must then gain unauthorized access to those services before they can do any damage

Some Firewall features

- Intrusion detection: Identify & block security threats such as malware, spyware, etc.
- Grant access to users based on business need
- Fingerprint applications and track their usage: # users, bandwidth usage
- Allocate bandwidth to applications (ex. Allocate more BW to SalesForce.com and less to YouTube)

https://www.cio.com.au/article/365101/top_seven_firewall_capabilities_effective_application_control/

https://www.fortinet.com/products/next-generation-firewall.html#services

https://www.securedgenetworks.com/blog/11-Features-to-Look-for-in-Your-Next-Generation-Firewall

Firewall techniques

Techniques used:

1. Packet filtering: Looks at IP address and Port # and drops packets coming from or destined to certain IP addresses

2. Circuit level gateways: Detect conversations by looking at end-point pairs

3. Application layer filtering: Detects applications trying to use disallowed protocols or ports

4. Hide addresses and perform network address translation.
   - Hacker can not know the IP address of the server that receives the message. Even if it knows, the firewall will block it.

Appendix

Business Process Management

Business Process Management Tools: **Business Process Management** (**BPM**) **tools** are used for automating, measuring and optimizing business processes.

BPM tools use workflow and collaboration to provide meaningful metrics to business leaders

Example: Appian, Zoho

SSL

1. Browser connects to a web server (website) secured with SSL (https). Browser requests that the server identify itself.
2. Server sends a copy of its SSL Certificate, including the server's public key.
3. Browser checks the certificate root against a list of trusted CAs and that the certificate is unexpired, unrevoked, and that its common name is valid for the website that it is connecting to. If the browser trusts the certificate, it creates, encrypts, and sends back a symmetric session key using the server's public key.
4. Server decrypts the symmetric session key using its private key and sends back an acknowledgement encrypted with the session key to start the encrypted session.
5. Server and Browser now encrypt all transmitted data with the session key.

Logging: Apache Log4j

- Logging is an important component of the software development.
- A well-written logging code offers quick debugging, easy maintenance, and structured storage of an application's runtime information.
- Logging does have its drawbacks also. It can slow down an application.

Asynchronous operation

AJAX Architecture

AJAX stands for **A**synchronous **Ja**vaScript and **X**ML.

AJAX is a new technique for creating better, faster, and more interactive web applications with the help of XML, HTML, CSS, and Java Script.

Some famous web applications that use AJAX:

Google Maps (Drag entire map)
Google Suggest (Google suggests as you type)

Simple Web application architecture

Dynamic web pages – using JSP and Servlet

Dynamic web pages

Web application architecture

Source: Wikipedia

vs.

BITS Pilani

Module 9 Part 6

Machine Learning

Harvinder S Jabbal
SSZG653 Software Architectures

Contents

- Introduction
  - What is Machine Learning (ML)?
  - Applications of ML
- Different types of ML
  - Algorithms used in ML
- Steps to build a ML model
- Architecture of ML system
- Popular tools for ML

Introduction

What is ML?

ML is the ability of machines to detect patterns and perform activities that humans do:

Applications

- Determine treatment for a patient based on their genetic makeup, demographic (age, gender, ethnicity) and psychographic characteristics (lifestyle, attitude, values)
- Pro-active maintenance of industrial equipment by looking at health parameters such as temperature, vibration, oil level
- Credit card Fraud detection by looking at spend frequency, amount, time of day, place of transaction, product purchased, customer profile

Major types of machine learning

Some algorithms used in ML

Regression

Eg. Predict price of house given size, location, etc.

Decision tree

Clustering

Neural networks

Supervised learning

- ***Supervised:*** In this approach, we provide a labelled dataset to the machine.
- Labelled dataset consists of features and result or class.
- Using the dataset, the machine builds a model (an equation or structure) to predict.

Regression

Eg. Predict price of house given size, location, etc.

Decision tree

Eg. Does this person like computer game? (based on age and gender)

Decision tree: Data provided to algorithm

Day	Outlook	Temperature	Humidity	Wind	Play Golf
D1	Sunny	Hot	High	Weak	No
D2	Sunny	Hot	High	Strong	No
D3	Overcast	Hot	High	Weak	Yes
D4	Rain	Mild	High	Weak	Yes
D5	Rain	Cool	Normal	Weak	Yes
D6	Rain	Cool	Normal	Strong	No
D7	Overcast	Cool	Normal	Strong	Yes
D8	Sunny	Mild	High	Weak	No
D9	Sunny	Cool	Normal	Weak	Yes
D10	Rain	Mild	Normal	Weak	Yes
D11	Sunny	Mild	Normal	Strong	Yes
D12	Overcast	Mild	High	Strong	Yes
D13	Overcast	Hot	Normal	Weak	Yes
D14	Rain	Mild	High	Strong	No

Decision tree built by algorithm to predict whether to play or not

Unsupervised Learning

- ***Unsupervised:*** Here we have an unlabelled dataset. We do not know what all features will constitute a class. The machine learns by itself and builds a model.

Circles indicates clusters of crime areas in Delhi

Ref: Crime Prediction using K-means Algorithm: Global Research and Development Journal for Engineering | Volume 2 | Issue 5 | April 2017

Clustering

- Customers can be segmented (clustered) based on Gender, age, annual income, products purchased, etc. (Luxury car buyers)

- We can source potential customer data and determine to which segment they belong to.

- Based on the segment, we can target them and send promotion details for the right product

Clustering

Reinforcement Learning

- ***Reinforcement Learning:*** In this, the machine learns from the environment by interacting with it.
- The machine is provided a set of allowed actions, rules and potential end states.
- By exploring different actions and observing resulting reactions the machine learns to exploit the rules to create a desired outcome.
- Eg. Game of chess, Robotics
- Algorithms used: Neural networks, Learning Automata, Q-Learning, Markov decision process

Deep Learning

- Deep learning: Used to understand and analyse image, sound and video. Uses Neural networks

- Used to understand human language (Natural Language Processing - NLP). Eg. Chatbots

Neural networks

- These networks can learn and model the relationships between inputs and outputs that are complex.

- **Examples**: Detecting rare events such as frauds, help doctors with an opinion

Neural networks - SAS

Neural networks

How does a neural network look like?

Steps to build ML model

1. Identify the problem to be solved (how will it benefit the business?)
2. Identify the features: This needs domain knowledge, creativity and lots of time. Ex: If we want to categorize customers, the features used to categorize can be age, salary, product purchased, where purchased, when, etc.
3. Decide on the model: What algorithm to use – supervised, unsupervised, reinforcement?
4. Train-test-validate the model
5. Experiment: Keep improving the model

Building the model: Fraud detection

Fraud?
- Yes
- No

Architecture of ML system:

Image classification for insurance claims

Ref: https://docs.microsoft.com/en-us/azure/architecture/example-scenario/ai/intelligent-apps-image-processing

Popular tools

- Scikit Learn - It provides models and algorithms for Classification, Regression, Clustering, Dimensional reduction, Model selection
- PyTorch – Neural Networks
- Tensor Flow – Neural networks
- Apache Mahout - Regression, Clustering, Recommenders, and Distributed Linear Algebra.
- Spark MLib –

Experience sharing

What problem did you solve using ML?

What steps did you follow to develop the system?

What were the key challenges you faced?

References

ML primer SAS

Case studies in ML

Workflow of ML project

5 steps to build a ML system

Appendix

Reference Chapter 23
Software Architecture in Practice
Third Edition
Len Bass
Paul Clements
Rick Kazman

BITS Pilani
Pilani|Dubai|Goa|Hyderabad

Module 10

Economic Evaluation through

Cost Benefit Analysis Method

Harvinder S Jabbal
SSZG653 Software Architectures

Chapter Outline

- Decision-Making Context
- The Basis for the Economic Analyses
- Putting Theory into Practice: The CBAM
- Case Study: The NASA ECS Project
- Summary

BITS Pilani, Deemed to be University under Section 3 of UGC Act, 1956

Decision-Making Context

Decision-making Context

The quality attributes achieved by architecture decisions have economic implications in terms of the benefits (*utility*) that can be derived from those decisions.

This interplay between the costs and the benefits of architectural decisions guides (and torments) the architect.

Decision-making Context

Example:

- Using redundant hardware to achieve a desired level of availability has a cost
- Checkpointing to a disk file has a different cost.
- Both of these architectural decisions will result in measurable levels of availability that will have some value to the organization developing the system.

Knowing the costs and benefits associated with particular decisions enables reasoned selection from among competing alternatives.

Economic analysis does not make decisions for the stakeholders. It simply aids in the elicitation and documentation of *value for cost* (VFC).

- VFC: a function of the costs, benefits, and uncertainty of a "portfolio" of architectural investments.
- It gives the stakeholders a framework within which they can apply a rational decision-making process that suits their needs and their risk aversion.

Economic analysis isn't something to apply to every architectural decision, but rather to the most basic ones that put an overarching architectural strategy in place.

BITS Pilani, Deemed to be University under Section 3 of UGC Act, 1956

Basis for Economic Analyses

Basis for Economic Analyses

Begin with a collection of scenarios generated from requirements elicitation, architectural evaluation, or specifically for economic analysis.

Examine how these scenarios differ in the values of their projected responses.

Assign utility to those values.

- The utility is based on the importance of each scenario with respect to its anticipated response value.

Consider the architectural strategies that lead to the various projected responses.

- Each strategy has a cost, and each impacts *multiple* quality attributes.
- That is, an architectural strategy could be implemented to achieve some projected response, but while achieving that response, it also affects some other quality attributes.
- The utility of these "side effects" must be taken into account when considering a strategy's overall utility.

Combine this overall utility with the project cost of an architectural strategy to calculate a final VFC measure.

BITS Pilani, Deemed to be University under Section 3 of UGC Act, 1956

Utility-Response Curves

Our economic analysis uses quality attribute scenarios (from Chapter 4) as the way to concretely express and represent specific quality attributes.

When we vary the values of the responses, and ask what the utility is of each response we get a set of points that we can plot: a *utility-response curve*.

Each scenario's stimulus-response pair provides some utility (value) to the stakeholders, and the utility of different possible values for the response can be compared.

- To help us make major architectural decisions, we might wish to compare the value of high performance against the value of high modifiability against the value of high usability, and so forth. The concept of utility lets us do that.

With prodding, stakeholders can express their needs using concrete response measures, such as "99.999 percent available." But how much would they value slightly less demanding quality attributes, such as "99.99 percent available"? Would that be almost as good?

If so, then the lower cost of achieving that lower value might make that the preferred option,

Capturing the utility of alternative responses of a scenario better enables the architect to make tradeoffs involving that quality attribute.

We can portray each relationship between a set of utility measures and a corresponding set of response measures as a graph—a utility-response curve.

BITS Pilani, Deemed to be University under Section 3 of UGC Act, 1956

Example Utility-Response Curves

Weighting the Scenarios

Different scenarios will have different importance to the stakeholders.

To make a choice of architectural strategies that is best suited to the stakeholders' desires, we must weight the scenarios.

- It does no good to spend a great deal of effort optimizing a scenario in which the stakeholders actually have very little interest.

BITS Pilani, Deemed to be University under Section 3 of UGC Act, 1956

Determining Benefit and Normalization

The overall benefit of an architectural strategy across various quality attribute scenarios is the sum of the utility associated with each scenario, weighted by the importance of the scenario.

For each architectural strategy i, its benefit B_i over *j* scenarios (each with weight W_j) is

- $B_i = \sum_j (b_{i,j} \times W_j)$

Each $b_{i,j}$ is calculated as the change in utility (over whatever architectural strategy is currently in place, or is in competition with the one being considered) brought about by the architectural strategy with respect to this scenario:

- $b_{i,j} = U_{expected} - U_{current}$

This is the utility of the expected value of the architectural strategy minus the utility of the *current* system relative to this scenario.

BITS Pilani, Deemed to be University under Section 3 of UGC Act, 1956

Calculating Value for Cost

The VFC for each architectural strategy is the ratio of the total benefit, B_i, to the cost, C_i, of implementing it:

- $VFC = B_i / C_i$

The cost C_i is estimated using a model appropriate for the system and the environment being developed, such as a cost model that estimates implementation cost by measuring an architecture's interaction complexity.

You can use this VFC score to rank-order the architectural strategies under consideration.

BITS Pilani, Deemed to be University under Section 3 of UGC Act, 1956

Putting Theory into Practice: The CBAM

Practicalities of Utility Curve Determination

To build the utility-response curve, we first determine the quality attribute levels for the best-case and worst-case situations.

The best-case quality attribute level is that above which the stakeholders foresee no further utility.

- For example, a system response to the user of 0.1 second is perceived as instantaneous, so improving it further so that it responds in 0.03 second has no additional utility.

The worst-case quality attribute level is a minimum threshold above which a system must perform; otherwise it is of no use to the stakeholders.

These levels—best-case and worst-case—are assigned utility values of 100 and 0, respectively.

Practicalities of Weighting Determination

One method of weighting the scenarios is to prioritize them and use their priority ranking as the weight.

- For *N* scenarios, the highest priority one is given a weight of 1, the next highest is given a weight of (*N*–1)/*N*, and so on.
- This turns the problem of weighting the scenarios into one of assigning priorities.

The stakeholders can determine the priorities through a variety of voting schemes.

- One simple method is to have each stakeholder prioritize the scenarios (from 1 to *N*) and the total priority of the scenario is the sum of the priorities it receives from all of the stakeholders.
- Voting can be public or secret.

Other schemes are possible. Regardless of the scheme used, it must make sense to the stakeholders and it must suit their culture.

BITS Pilani, Deemed to be University under Section 3 of UGC Act, 1956

Practicalities of Cost Determination

There are very few cost models for various architectural strategies.

There are many software cost models, but they are based on overall system characteristics such as size or function points.	These are inadequate to answer the question of how much does it cost to, for example, use a publish-subscribe pattern in a particular portion of the architecture.	There are cost models that are based on complexity of modules (by function point analysis according to the requirements assigned to each module) and the complexity of module interaction, but these are not widely used in practice.	More widely used in practice are corporate cost models based on previous experience with the same or similar architectures, or the experience and intuition of senior architects.

BITS Pilani, Deemed to be University under Section 3 of UGC Act, 1956

Practicalities of Cost Determination

Architects often turn to cost estimation techniques.

An absolute number for cost isn't necessary to rank candidate architecture strategies.

- You can often say something like "Suppose strategy A costs $x. It looks like strategy B will cost $2x, and strategy C will cost $0.5x." That's enormously helpful.
- A second approach is to use very coarse estimates. Or if you lack confidence for that degree of certainty, you can say something like "Strategy A will cost a lot, strategy B shouldn't cost very much, and strategy C is probably somewhere in the middle."

BITS Pilani, Deemed to be University under Section 3 of UGC Act, 1956

Cost Benefit Analysis Method (CBAM)

The CBAM places the principles just discussed into a set of steps: a method.

CBAM Stakeholders

The stakeholders in a CBAM exercise include people who can authoritatively speak to the utility of various quality attribute responses.

They probably include the same people who were the source of the quality attribute scenarios being used as input.

BITS Pilani, Deemed to be University under Section 3 of UGC Act, 1956

Step 1

Collate scenarios.

- Give the stakeholders the chance to contribute new scenarios.
- Ask the stakeholders to prioritize the scenarios based on satisfying the business goals of the system.
- This can be an informal prioritization using a simple scheme such as “high, medium, low” to rank the scenarios.
- Choose the top one-third for further study.

BITS Pilani, Deemed to be University under Section 3 of UGC Act, 1956

Step 2

Refine scenarios.

- Refine the scenarios chosen in step 1, focusing on their stimulus-response measures.
- Elicit the worst-case, current, desired, and best-case quality attribute response level for each scenario.
- For example, a refined performance scenario might tell us that worst-case performance for our system's response to user input is 12 seconds, the best case is 0.1 seconds, and our desired response is 0.5 seconds. Our current architecture provides a response of 1.5 seconds.

Step 3

Prioritize scenarios.

- Prioritize the refined scenarios, based on stakeholder votes.
- Give 100 votes to each stakeholder and have them distribute the votes among the scenarios, where their voting is based on the desired response value for each scenario.
- Total the votes and choose the top 50 percent of the scenarios for further analysis.
- Assign a weight of 1.0 to the highest-rated scenario; assign the other scenarios a weight relative to the highest rated.
- This becomes the weighting used in the calculation of a strategy's overall benefit.
- Make a list of the quality attributes that concern the stakeholders.

Step 4

Assign utility.

- Determine the utility for each quality attribute response level (worst-case, current, desired, best-case) for the scenarios from step 3.
- You can conveniently capture these utility curves in a table (one row for each scenario, one column for each of the four quality attribute response levels).

Scenario	Worst Case	Current	Desired	Best Case
Scenario #17: Response to user input	12 seconds	1.5 seconds	0.5 seconds	0.1 seconds
	Utility 5	Utility 50	Utility 80	Utility 85

Step 5

Map architectural strategies to scenarios and determine their expected quality attribute response levels.

- For each architectural strategy under consideration, determine the expected quality attribute response levels that will result for each scenario.

BITS Pilani, Deemed to be University under Section 3 of UGC Act, 1956

Step 6

Determine the utility of the expected quality attribute response levels by interpolation.

- Using the elicited utility values (that form a utility curve), determine the utility of the expected quality attribute response level for the architectural strategy.
- Do this for each relevant quality attribute enumerated in step 3.
- For example, if we are considering a new architectural strategy that would result in a response time of 0.7 seconds, we would assign a utility proportionately between 50 (which it exceeds) and 80 (which it doesn't exceed).
- The formula for interpolation between two data points (x_a, y_a) and (x_b, y_b) is:

$$y = y_a + (y_b - y_a)\frac{(x - x_a)}{(x_b - x_a)}$$

- For us, the *x* values are the quality attribute response levels and the *y* values are the utility values. So, employing this formula, the utility value of a 0.7-second response time is 74 .

BITS Pilani, Deemed to be University under Section 3 of UGC Act, 1956

Step 7

Calculate the total benefit obtained from an architectural strategy.

- Subtract the utility value of the "current" level from the expected level and normalize it using the votes elicited in step 3.
- Sum the benefit due to a particular architectural strategy across all scenarios and across all relevant quality attributes.

BITS Pilani, Deemed to be University under Section 3 of UGC Act, 1956

Step 8

Choose architectural strategies based on VFC subject to cost and schedule constraints.

- Determine the cost and schedule implications of each architectural strategy.
- Calculate the VFC value for each as a ratio of benefit to cost. Rank-order the architectural strategies according to the VFC value and choose the top ones until the budget or schedule is exhausted.

BITS Pilani, Deemed to be University under Section 3 of UGC Act, 1956

Step 9

Confirm results with intuition.

- For the chosen architectural strategies, consider whether these seem to align with the organization's business goals.
- If not, consider issues that may have been overlooked while doing this analysis.
- If there are significant issues, perform another iteration of these steps.

BITS Pilani, Deemed to be University under Section 3 of UGC Act, 1956

CASE STUDY:
The NASA ECS Project

Case Study: The NASA ECS Project

The Earth Observing System is a constellation of NASA satellites that gathers data for the U.S. Global Change Research Program and other scientific communities worldwide.

The Earth Observing System Data Information System (EOSDIS) Core System (ECS) collects data from various satellite downlink stations for further processing.

ECS's mission is to process the data into higher-form information and make it available to scientists in searchable form. The goal is to provide both a common way to store (and hence process) data and a public mechanism to introduce new data formats and processing algorithms, thus making the information widely available.

BITS Pilani, Deemed to be University under Section 3 of UGC Act, 1956

Case Study: The NASA ECS Project

The ECS processes an input stream of hundreds of gigabytes of raw environment-related data per day. The computation of 250 standard “products” results in thousands of gigabytes of information that is archived at eight data centers in the United States. The system has important performance and availability and modifiability requirements.

The ECS project manager had a limited annual budget to maintain and enhance his current system. The manager used the CBAM to make a rational decision based on the economic criterion of return on investment.

BITS Pilani, Deemed to be University under Section 3 of UGC Act, 1956

Step 1: Collate Scenarios

IN PRIORITY ORDER

Note that they are not yet well formed and that some of them do not have defined responses. These issues are resolved in step 2, when the number of scenarios is reduced.

1. Reduce data distribution failures that result in hung distribution requests requiring manual intervention.
2. Reduce data distribution failures that result in lost distribution requests.
3. Reduce the number of orders that fail on the order submission process.
4. Reduce order failures that result in hung orders that require manual intervention.
5. Reduce order failures that result in lost orders.
6. There is no good method of tracking ECSGuest failed/canceled orders without much manual intervention (e.g., spreadsheets).
7. Users need more information on why their orders for data failed.
8. Because of limitations, there is a need to artificially limit the size and number of orders.
9. Small orders result in too many notifications to users.
10. The system should process a 50-GB user request in one day, and a 1-TB user request in one week.

Step 2: Refine Scenarios

The scenarios were refined, paying particular attention to precisely specifying their stimulus-response measures.

The worst-case, current-case, desired-case, and best-case response goals for each scenario were elicited and recorded in a table.

BITS Pilani, Deemed to be University under Section 3 of UGC Act, 1956

Step 2: Refine Scenarios

Scenario	Worst	Current	Desired	Best
1	10% hung	5% hung	1% hung	0% hung
2	> 5% lost	< 1% lost	0% lost	0% lost
3	10% fail	5% fail	1% fail	0% fail
4	10% hung	5% hung	1% hung	0% hung
5	10% lost	< 1% lost	0% lost	0% lost
6	50% need help	25% need help	0% need help	0% need help
7	10% get information	50% get information	100% get information	100% get information
8	50% limited	30% limited	0% limited	0% limited
9	1/granule	1/granule	1/100 granules	1/1,000 granules
10	< 50% meet goal	60% meet goal	80% meet goal	> 90% meet goal

Step 3: Prioritize Scenarios

In voting on the refined representation of the scenarios, the close-knit team deviated slightly from the method.

Rather than vote individually, they chose to discuss each scenario and arrived at a determination of its weight via consensus.

The votes allocated to the entire set of scenarios were constrained to 100.

Although the stakeholders were not required to make the votes multiples of 5, they felt that this was a reasonable resolution and that more precision was neither needed nor justified.

BITS Pilani, Deemed to be University under Section 3 of UGC Act, 1956

Step 3: Prioritize Scenarios

Scenario	Votes	Worst	Current	Desired	Best
1	10	10% hung	5% hung	1% hung	0% hung
2	15	> 5% lost	< 1% lost	0% lost	0% lost
3	15	10% fail	5% fail	1% fail	0% fail
4	10	10% hung	5% hung	1% hung	0% hung
5	15	10% lost	< 1% lost	0% lost	0% lost
6	10	50% need help	25% need help	0% need help	0% need help
7	5	10% get information	50% get information	100% get information	100% get information
8	5	50% limited	30% limited	0% limited	0% limited
9	10	1/granule	1/granule	1/100 granules	1/1,000 granules
10	5	< 50% meet goal	60% meet goal	80% meet goal	> 90% meet goal

Step 4: Assign Utility

In this step the utility for each scenario was determined by the stakeholders, again by consensus.

A utility score of 0 represented no utility; a score of 100 represented the most utility possible.

BITS Pilani, Deemed to be University under Section 3 of UGC Act, 1956

Step 4: Assign Utility

		Utility Scores			
Scenario	Votes	Worst	Current	Desired	Best
1	10	10	80	95	100
2	15	0	70	100	100
3	15	25	70	100	100
4	10	10	80	95	100
5	15	0	70	100	100
6	10	0	80	100	100
7	5	10	70	100	100
8	5	0	20	100	100
9	10	50	50	80	90
10	5	50	50	80	90

Step 5: Develop Architectural Strategies and Determine Their Expected Quality Attribute Response Levels

Based on the requirements implied by the preceding scenarios, a set of 10 architectural strategies was developed by the ECS architects.

For each architectural strategy/scenario pair, the response levels expected to be achieved with respect to that scenario are shown (along with the current response, for comparison purposes).

BITS Pilani, Deemed to be University under Section 3 of UGC Act, 1956

Step 5 Results (Scenarios 1-5)

Strategy	Name	Description	Scenarios Affected	Current Response	Expected Response
1	Order persistence on submission	Store an order as soon as it arrives in the system.	3	5% fail	2% Fail
			5	<1% lost	0% lost
			6	25% need help	0% need help
2	Order chunking	Allow operators to partition large orders into multiple small orders.	8	30% limited	15% limited
3	Order bundling	Combine multiple small orders into one large order.	9	1 per granule	1 per 100
			10	60% meet goal	55% meet goal
4	Order segmentation	Allow an operator to skip items that cannot be retrieved due to data quality or availability issues.	4	5% hung	2% hung
5	Order reassignment	Allow an operator to reassign the media type for items in an order.	1	5% hung	2% hung

Step 5 Results (Scenarios 5-10)

6	Order retry	Allow an operator to retry an order or items in an order that may have failed due to temporary system or data problems.	4	5% hung	3% hung
7	Forced order completion	Allow an operator to override an item's unavailability due to data quality constraints.	1	5% hung	3% hung
8	Failed order notification	Ensure that users are notified only when part of their order has truly failed and provide detailed status of each item; user notification occurs only if operator okays notification; the operator may edit notification.	6	25% need help	20% need help
			7	50% get information	90% get information
9	Granule-level order tracking	An operator and user can determine the status for each item in their order.	6	25% need help	10% need help
			7	50% get information	95% get information
10	Links to user information	An operator can quickly locate a user's contact information. Server will access SDSRV information to determine any data restrictions that might apply and will route orders/order segments to appropriate distribution capabilities, including DDIST, PDS, external subsetters and data processing tools, etc.	7	50% get information	60% get information

Step 6: Determine the Utility of the "Expected" Quality Attribute Response Levels by Interpolation

Once the expected response level of every architectural strategy has been characterized with respect to a set of scenarios, their utility can be calculated by consulting the utility scores for each scenario's current and desired responses for all of the affected attributes.

Using these scores, we may calculate, via interpolation, the utility of the expected quality attribute response levels for the architectural strategy/scenario pair.

Step 6 Results

Strategy	Name	Scenarios Affected	Current Utility	Expected Utility
1	Order persistence on submission	3	70	90
		5	70	100
		6	80	100
2	Order chunking	8	20	60
3	Order bundling	9	50	80
		10	70	65
4	Order segmentation	4	80	90
5	Order reassignment	1	80	92
6	Order retry	4	80	85
7	Forced order completion	1	80	87
8	Failed order notification	6	80	85
		7	70	90
9	Granule-level order tracking	6	80	90
		7	70	95
10	Links to user information	7	70	75

Step 7: Calculate the Total Benefit Obtained from an Architectural Strategy

Total benefit of each architectural strategy can now be calculated:

- $B_i = \sum_j (b_{i,j} \times W_j)$

This equation calculates total benefit as the sum of the benefit that accrues to each scenario, normalized by the scenario's relative weight.

Using this formula, the total benefit scores for each architectural strategy are now calculated.

BITS Pilani, Deemed to be University under Section 3 of UGC Act, 1956

Step 7

Strategy	Scenario Affected	Scenario Weight	Raw Architectural Strategy Benefit	Normalized Architectural Strategy Benefit	Total Architectural Strategy Benefit
1	3	15	20	300	
1	5	15	30	450	
1	6	10	20	200	950
2	8	5	40	200	200
3	9	10	30	300	
3	10	5	–5	–25	275
4	4	10	10	100	100
5	1	10	12	120	120
6	4	10	5	50	50
7	1	10	7	70	70
8	6	10	5	50	
8	7	5	20	100	150
9	6	10	10	100	
9	7	5	25	125	225
10	7	5	5	25	25

Step 8: Choose Architectural Strategies Based on VFC Subject to Cost Constraints

To complete the analysis, the team estimated cost for each architectural strategy.

The estimates were based on experience with the system, and a return on investment for each architectural strategy was calculated.

Using the VFC, we were able to rank each strategy.

The ranks roughly follow the ordering in which the strategies were proposed: strategy 1 has the highest rank; strategy 3 the second highest. Strategy 9 has the lowest rank; strategy 8, the second lowest.

This simply validates stakeholders' intuition about which architectural strategies were going to be of the greatest benefit. For the ECS these were the ones proposed first.

Step 8

Strategy	Cost	Total Strategy Benefit	Strategy VFC	Strategy Rank
1	1200	950	0.79	1
2	400	200	0.5	3
3	400	275	0.69	2
4	200	100	0.5	3
5	400	120	0.3	7
6	200	50	0.25	8
7	200	70	0.35	6
8	300	150	0.5	3
9	1000	225	0.22	10
10	100	25	0.25	8

Results of the CBAM Exercise

- The most obvious results of the CBAM are the ordering of architectural strategies based on their predicted VFC.
- However, there are social and cultural benefits as well.
- The CBAM process provides structure to what is always largely unstructured discussions, where requirements and architectural strategies are freely mixed and where stimuli and response goals are not clearly articulated.
- The CBAM process forces the stakeholders to make their scenarios clear in advance, to assign utility levels of specific response goals, and to prioritize these scenarios based on the resulting determination of utility.
- Finally, it produces clarification of both scenarios and requirements, which by itself is a significant benefit.

BITS Pilani, Deemed to be University under Section 3 of UGC Act, 1956

Summary

Architecture-based economic analysis is grounded on understanding the utility-response curve of various scenarios and casting them into a form that makes them comparable.

Once they are in this common form—based on the common coin of utility—the VFC for each architecture improvement, with respect to each relevant scenario, can be calculated and compared.

Applying the theory in practice has a number of practical difficulties, which CBAM solves.

The application of economic techniques is inherently better than the ad hoc decision-making approaches that projects employ today.

Giving people the appropriate tools to frame and structure their discussions and decision making is an enormous benefit to the disciplined development of a complex software system.

BITS Pilani, Deemed to be University under Section 3 of UGC Act, 1956

Thank you……..

Credits

- **Chapter Reference from Text T1: 23**
- Slides have been adapted from Authors Slides Software Architecture in Practice – Third Ed.
    - Len Bass
    - Paul Clements
    - Rick Kazman

BITS Pilani, Deemed to be University under Section 3 of UGC Act, 1956