

SOFTWARE ARCHITECTURE AND DESIGN PATTERNS

UNIT-I

UNIT-I:

Envisioning Architecture: The Architecture Business Cycle, What is Software Architecture, Architectural patterns, reference models, reference architectures, architectural structures and views

Envisioning Architecture:

1. The Architecture Business Cycle

1.1 Where Do Architectures Come From?

Architecture is the result of a set of business and technical decisions. There are many influences at work in its design, and the realization of these influences will change depending on the environment in which the architecture is required to perform.

An architect designing a system for which the real-time deadlines are believed to be tight will make one set of design choices; the same architect, designing a similar system in which the deadlines can be easily satisfied, will make different choices.

ARCHITECTURES ARE INFLUENCED BY SYSTEM STAKEHOLDERS

Many people and organizations are interested in the construction of a software system. We call these stakeholders: The customer, the end users, the developers, the project manager, the maintainers, and even those who market the system are a few examples. Stakeholders have different concerns that they wish the system to guarantee or optimize, including things as diverse as providing a certain behavior at runtime, performing well on a particular piece of hardware, being easy to customize, achieving short time to market or low cost of development, gainfully employing programmers who have a particular specialty, or providing a broad range of functions. Figure.1 shows the architect receiving helpful stakeholder "suggestions."

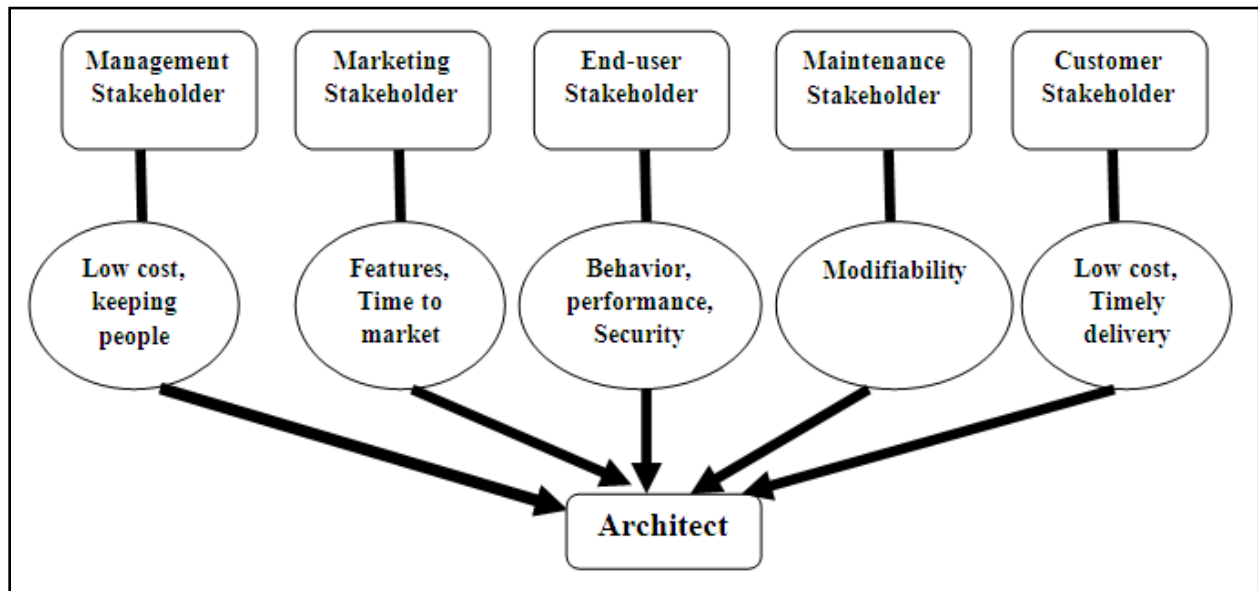


Figure 1. Influence of stakeholders on the architect

SOFTWARE ARCHITECTURE AND DESIGN PATTERNS

UNIT-I

ARCHITECTURES ARE INFLUENCED BY THE DEVELOPING ORGANIZATION

In addition to the organizational goals expressed through requirements, architecture is influenced by the structure or nature of the development organization. For example, if the organization has an abundance of idle programmers skilled in client-server communications, then client-server architecture might be the approach supported by management. If not, it may well be rejected. Staff skills are one additional influence, but so are the development schedule and budget.

There are three classes of influence that come from the developing organization: immediate business, long-term business, and organizational structure.

ARCHITECTURES ARE INFLUENCED BY THE BACKGROUND AND EXPERIENCE OF THE ARCHITECTS

If the architects for a system have had good results using a particular architectural approach, such as distributed objects or implicit invocation, chances are that they will try that same approach on a new development effort. Conversely, if their prior experience with this approach was disastrous, the architects may be reluctant to try it again. Architectural choices may also come from an architect's education and training, exposure to successful architectural patterns, or exposure to systems that have worked particularly poorly or particularly well. The architects may also wish to experiment with an architectural pattern or technique learned from a book (such as this one) or a course.

ARCHITECTURES ARE INFLUENCED BY THE TECHNICAL ENVIRONMENT

A special case of the architect's background and experience is reflected by the technical environment. The environment that is current when architecture is designed will influence that architecture. It might include standard industry practices or software engineering techniques prevalent in the architect's professional community. It is a brave architect who, in today's environment, does not at least consider a Web-based, object-oriented, middleware-supported design for an information system.

RAMIFICATIONS OF INFLUENCES ON ARCHITECTURE

Influences on architecture come from a wide variety of sources. Some are only implied, while others are explicitly in conflict. Almost never are the properties required by the business and organizational goals consciously understood, let alone fully articulated. Indeed, even customer requirements are seldom documented completely, which means that the inevitable conflict among different stakeholders' goals has not been resolved.

A business manages this cycle to handle growth, to expand its enterprise area, and to take advantage of previous investments in architecture and system building. Figure 2 shows the feedback loops. Some of the feedback comes from the architecture itself, and some comes from the system built from it. The architecture affects the structure of the developing organization. An architecture prescribes a structure for a system; as we will see, it particularly 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. Likewise, schedules and budgets allocate resources in chunks corresponding to the units. If a company becomes adept at building families of similar systems, it will tend to invest in each team by nurturing each area of expertise. Teams become embedded in the organization's structure. This is feedback from the architecture to the developing organization.

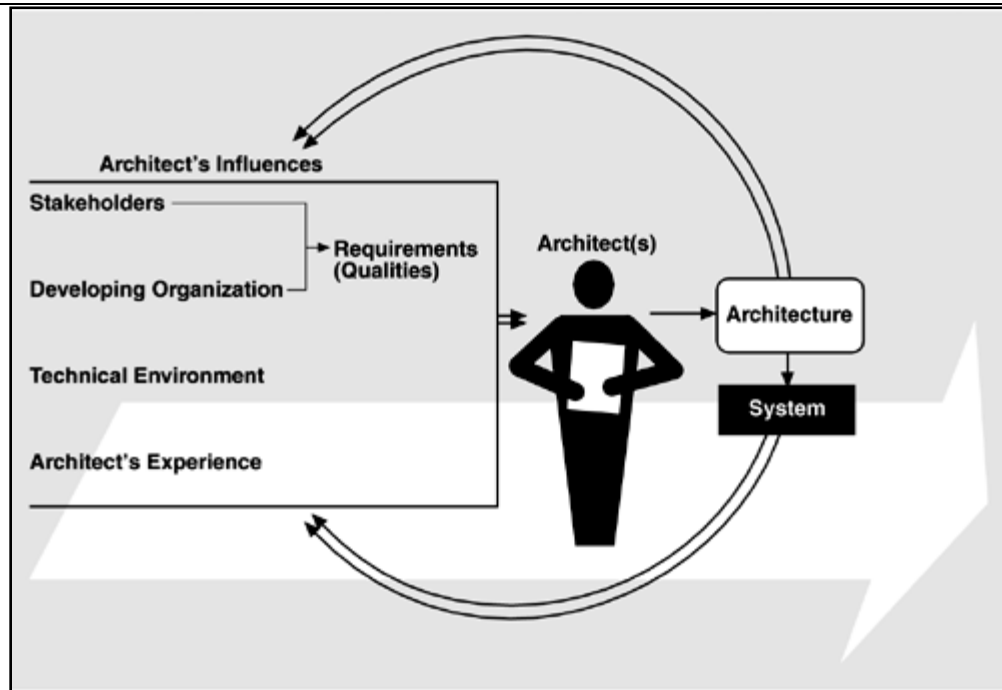


Figure 2. The Architecture Business Cycle

1.2 Software Processes and the Architecture Business Cycle

Software process is the term given to the organization, reutilization, and management of software development activities. What activities are involved in creating software architecture, using that architecture to realize a design, and then implementing or managing the evolution of a target system or application? These activities include the following:

- Creating the business case for the system
- Understanding the requirements
- Creating or selecting the architecture
- Documenting and communicating the architecture
- Analyzing or evaluating the architecture
- Implementing the system based on the architecture
- Ensuring that the implementation conforms to the architecture

ARCHITECTURE ACTIVITIES

As indicated in the structure of the ABC, architecture activities have comprehensive feedback relationships with each other. We will briefly introduce each activity in the following subsections.

Creating the Business Case for the System

Creating a business case: is broader than simply assessing the market need for a system. It is an important step in creating and constraining any future requirements. How much should the product cost? What is its targeted market? What is its targeted time to market? These are all questions that must involve the system's architects.

Understanding the Requirements: There are a variety of techniques for eliciting requirements from the stakeholders. For example, object-oriented analysis uses scenarios, or "use cases" to embody requirements. Safety-critical systems use more rigorous approaches, such as finite-state-machine models or formal specification languages.

SOFTWARE ARCHITECTURE AND DESIGN PATTERNS

UNIT-I

Communicating the Architecture

For the architecture to be effective as the backbone of the project's design, it must be communicated clearly and unambiguously to all of the stakeholders. Developers must understand the work assignments it requires of them, testers must understand the task structure it imposes on them, management must understand the scheduling implications it suggests, and so forth. Toward this end, the architecture's documentation should be informative, unambiguous, and readable by many people with varied backgrounds.

Analyzing or Evaluating the Architecture

In any design process there will be multiple candidate designs considered. Some will be rejected immediately. Others will contend for primacy. Choosing among these competing designs in a rational way is one of the architect's greatest challenges.

1.3 What Makes a 'Good' Architecture?

If it is true that, given the same technical requirements for a system, two different architects in different organizations will produce different architectures, how can we determine if either one of them is the right one? We divide our observations into two clusters: process recommendations and product (or structural) recommendations. Our process recommendations are as follows:

- The architecture should be the product of a single architect or a small group of architects with an identified leader. The architect (or architecture team) should have the functional requirements for the system and an articulated, prioritized list of quality attributes (such as security or modifiability) that the architecture is expected to satisfy.
- The architecture should be well documented, with at least one static view and one dynamic view, using an agreed-on notation that all stakeholders can understand with a minimum of effort.
- The architecture should be circulated to the system's stakeholders, who should be actively involved in its review.
- The architecture should be analyzed for applicable quantitative measures (such as maximum throughput) and formally evaluated for quality attributes before it is too late to make changes to it.
- The architecture should lend itself to incremental implementation via the creation of a "skeletal" system in which the communication paths are exercised but which at first has minimal functionality.

2. What is Software Architecture?

The system consists of four elements, Prop Loss Model (MODP), Reverb Model (MODR), and Noise Model (MODN)? might have more in common with each other than with the fourth model, Control Process (CP). All of the elements apparently have some sort of relationship with each other, since the diagram is fully connected. The models are described in figure 3.

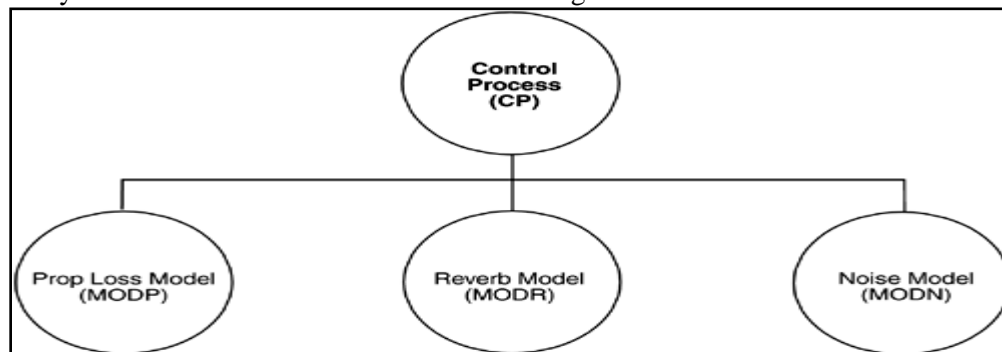


Figure 3: Typical, but uninformative, presentation of software architecture

SOFTWARE ARCHITECTURE AND DESIGN PATTERNS

UNIT-I

3. Architectural patterns, Reference models and Reference architectures

An architectural pattern is a description of element and relation types together with a set of constraints on how they may be used. A pattern can be thought of as a set of constraints on architecture? On the element types and their patterns of interaction? And these constraints define a set or family of architectures that satisfy them. For example, client-server is a common architectural pattern. Client and server are two element types, and their coordination is described in terms of the protocol that the server uses to communicate with each of its clients. Use of the term client-server implies only that multiple clients exist; the clients themselves are not identified, and there is no discussion of what functionality, other than implementation of the protocols, has been assigned to any of the clients or to the server.

A reference model is a division of functionality together with data flow between the pieces. A reference model is a standard decomposition of a known problem into parts that cooperatively solve the problem.

Reference architecture is a reference model mapped onto software elements (that cooperatively implement the functionality defined in the reference model) and the data flows between them. Whereas a reference model divides the functionality, reference architecture is the mapping of that functionality onto system decomposition. The relationship among these design elements is shown in Figure 4.

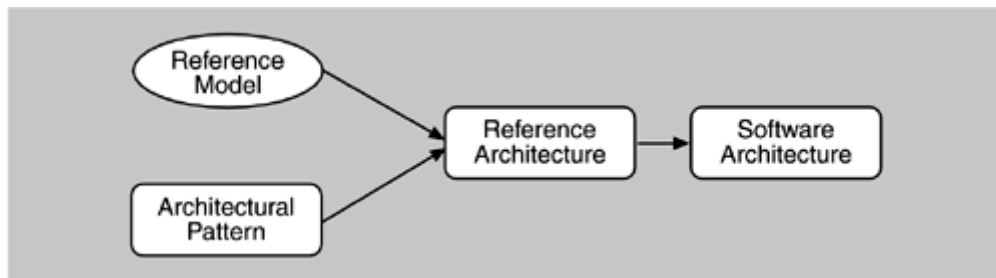


Figure 4: The relationships of design elements

Architectural structures and views

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.

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, as documented by and used by some system stakeholders. These terms are often used interchangeably, but we will adhere to these definitions.

Architectural structures can by and large be divided into three groups, depending on the broad nature of the elements they show.

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.

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?

SOFTWARE ARCHITECTURE AND DESIGN PATTERNS

UNIT-I

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? These three structures correspond to the three broad types of decision that architectural design involves:

- 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 non-software structures in its environment (i.e., CPUs, file systems, networks, development teams, etc.)?

SOFTWARE STRUCTURES

Some of the most common and useful software structures are shown in Figure 5. These are described in the following sections.

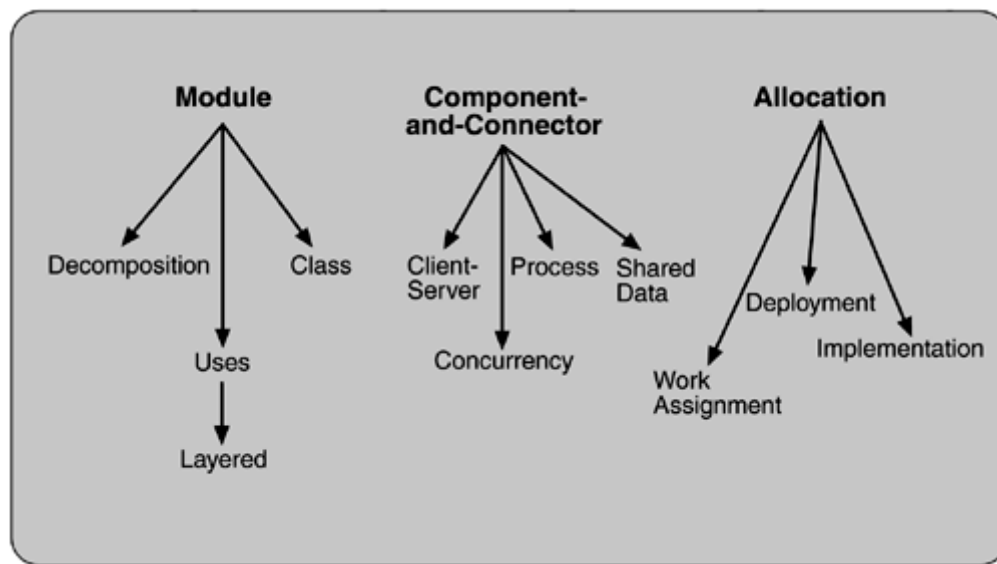


Figure 5: Common software architecture structures

| | | |
|-------------------------|---------------|---|
| Module | Decomposition | The units are modules related to each other by the "is a sub module of" relation, showing how larger modules are decomposed into smaller ones recursively until they are small enough to be easily understood |
| | Uses | The units of this important but overlooked structure are also modules, or procedures or resources on the interfaces of modules. |
| | 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. |
| | Class | The module units in this structure are called classes. The relation is "inherits-from" or "is-an-instance-of." |
| Component-and-Connector | 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. |
| | Concurrency | This component-and-connector structure allows the architect to determine opportunities for parallelism and the locations where resource contention may occur. |
| | Process | Like all component-and-connector structures, this one is orthogonal to the module-based structures and deals with the dynamic aspects of a |

SOFTWARE ARCHITECTURE AND DESIGN PATTERNS
UNIT-I

| | | |
|-------------------|------------------------|---|
| Allocation | | running system. |
| | Shared data | This structure comprises components and connectors that create, store, and access persistent data. |
| | Work assignment | This structure assigns responsibility for implementing and integrating the modules to the appropriate development teams. |
| | Deployment | The deployment structure shows how software is assigned to hardware-processing and communication elements. |
| | 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. |

SOFTWARE ARCHITECTURE AND DESIGN PATTERNS

UNIT-I

Creating Architecture: Quality Attributes, Achieving qualities, Architectural styles and patterns, designing the Architecture, Documenting software architectures, Reconstructing Software Architecture

4. Quality Attributes

Architecture and Quality Attributes

Achieving quality attributes must be considered throughout design, implementation, and deployment. No quality attribute is entirely dependent on design, nor is it entirely dependent on implementation or deployment. Satisfactory results are a matter of getting the big picture (architecture) as well as the details (implementation) correct.

Architecture is critical to the realization of many qualities of interest in a system, and these qualities should be designed in and can be evaluated at the architectural level. Architecture, by itself, is unable to achieve qualities. It provides the foundation for achieving quality, but this foundation will be to no avail if attention is not paid to the details. We will examine the following three classes:

- Qualities of the system. We will focus on **availability, modifiability, performance, security, testability, and usability.**
- Business qualities (such as time to market) that are affected by the architecture.
- Qualities, such as conceptual integrity, that is about the architecture itself although they indirectly affect other qualities, such as modifiability.

Quality Attributes Scenarios

A quality attribute scenario is a quality-attribute-specific requirement. It consists of six parts.

- **Source of stimulus:** This is some entity (a human, a computer system, or any other actuator) that generated the stimulus.
- **Stimulus:** The stimulus is a condition that needs to be considered when it arrives at a system.
- **Environment:** The stimulus occurs within certain conditions. The system may be in an overload condition or may be running when the stimulus occurs, or some other condition may be true.
- **Artifact:** Some artifact is stimulated. This may be the whole system or some pieces of it.
- **Response:** The response is the activity undertaken after the arrival of the stimulus.
- **Response measure:** When the response occurs, it should be measurable in some fashion so that the requirement can be tested.

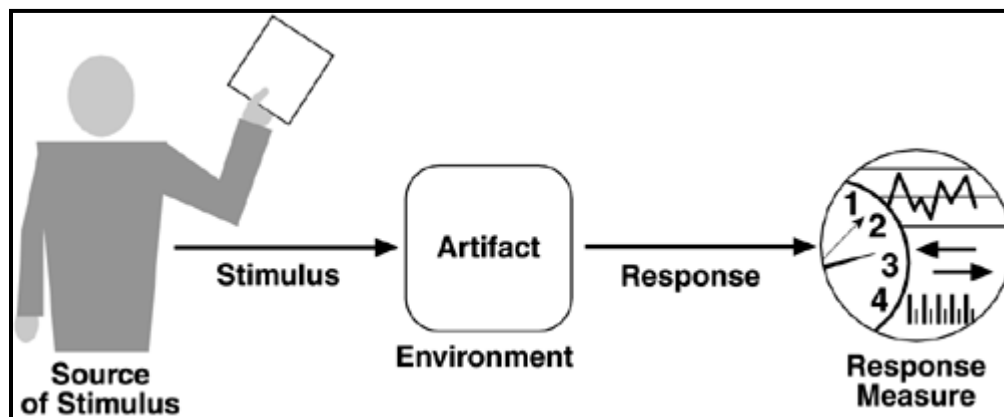


Figure 6: shows the parts of a quality attribute scenario.

Availability Scenario

A general scenario for the quality attribute of availability, for example, is shown in Figure 4.2. Its six parts are shown, indicating the range of values they can take. From this we can derive concrete, system-specific, scenarios. Not every system-specific scenario has all of the six parts. The parts that are necessary are the result of the application of the scenario and the types of testing that will be performed to determine whether the scenario has been achieved.

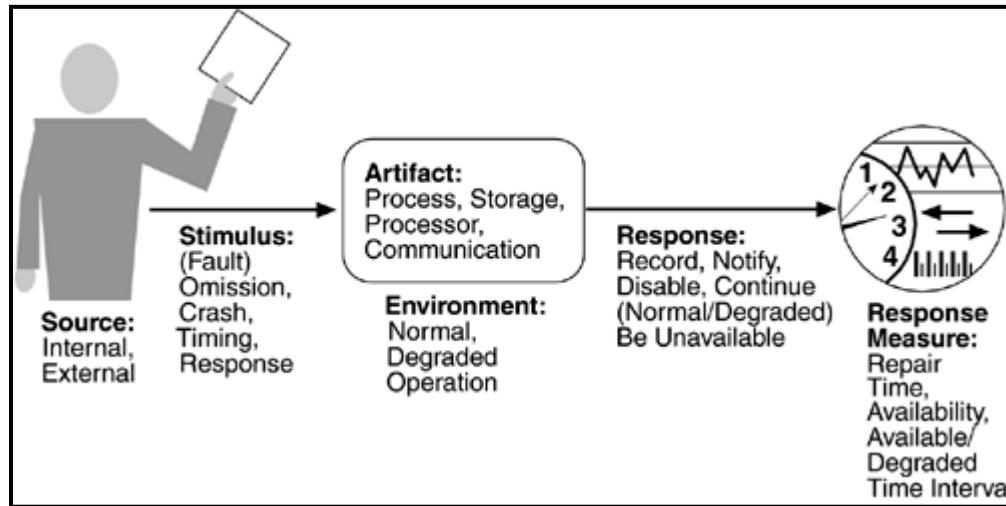


Figure 7: Availability general scenarios

Modifiability Scenario

A sample modifiability scenario is "A developer wishes to change the user interface to make a screen's background color blue. This change will be made to the code at design time. It will take less than three hours to make and test the change and no side effect changes will occur in the behavior." Figure 8 illustrates this sample scenario.

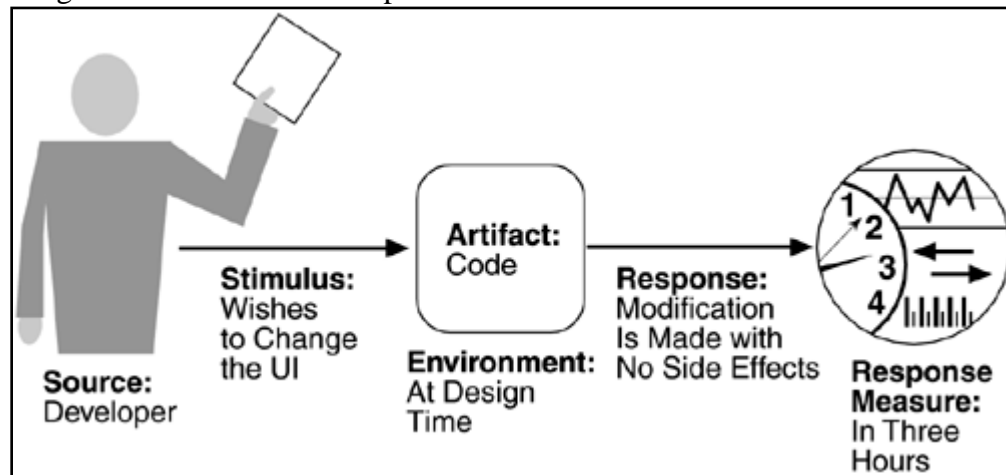


Figure 8: Sample modifiability scenario

5. Achieving qualities

The achievement of these qualities relies on fundamental design decisions. We will examine these design decisions, which we call tactics. A **tactic** is a design decision that influences the control of a quality attribute response. A system design consists of a collection of decisions. Some of these decisions help control the quality attribute responses; others ensure achievement of system functionality.

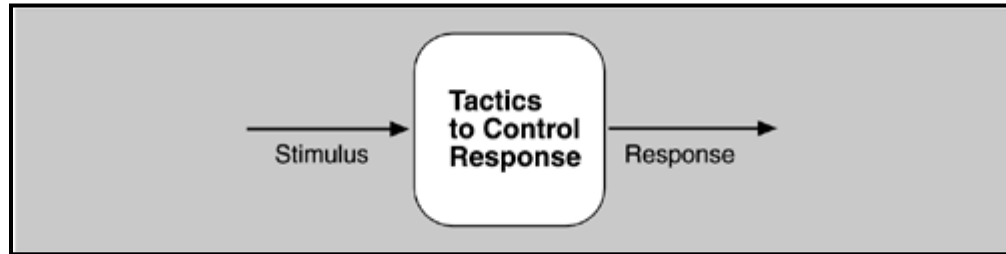


Figure 9: Tactics are intended to control responses to stimuli

We organize the tactics for each system quality attribute as a hierarchy, but it is important to understand that each hierarchy is intended only to demonstrate some of the tactics, and that any list of tactics is necessarily incomplete.

5.1 Availability Tactics: A failure occurs when the system no longer delivers a service that is consistent with its specification; this failure is observable by the system's users. A fault (or combination of faults) has the potential to cause a failure. Recall also that recovery or repair is an important aspect of availability. We first consider fault detection. We then consider fault recovery and finally, briefly, fault prevention.

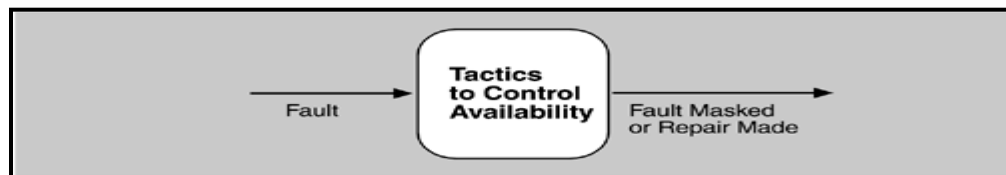


Figure 10: Goal of availability tactics

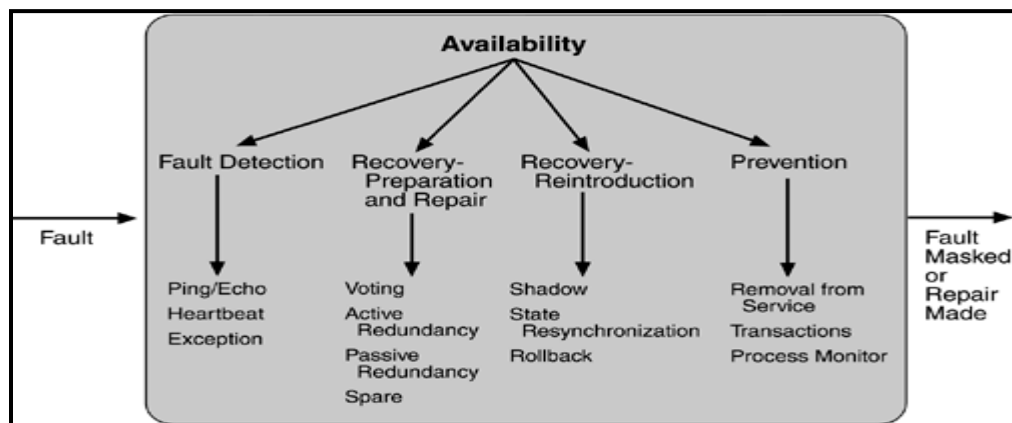


Figure 11: Summary of availability tactics

5.2 Modifiability Tactics: We organize the tactics for modifiability in sets according to their goals. One set has as its goal reducing the number of modules that are directly affected by a change. We call this set **"localize modifications."** A second set has as its goal limiting modifications to the localized modules. We use this set of tactics to **"prevent the ripple effect."** Implicit in this distinction is that there are modules directly affected (those whose responsibilities are adjusted to accomplish the change) and modules indirectly affected by a change (those whose responsibilities remain unchanged but whose implementation must be changed to accommodate the directly affected modules). A third set of tactics has as its goal controlling deployment time and cost. We call this set **"defer binding time."**

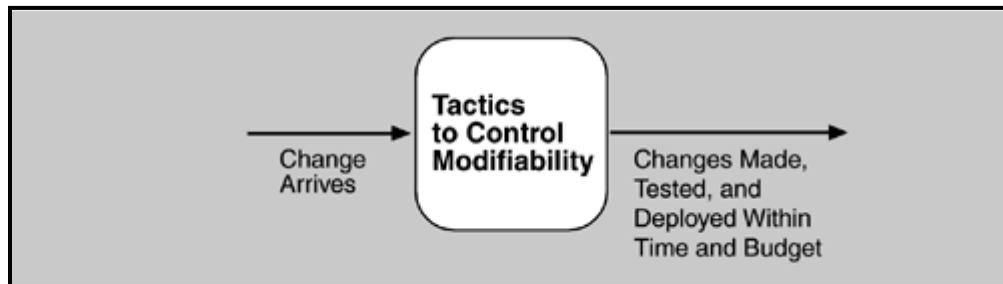


Figure 12: Goal of modifiability tactics

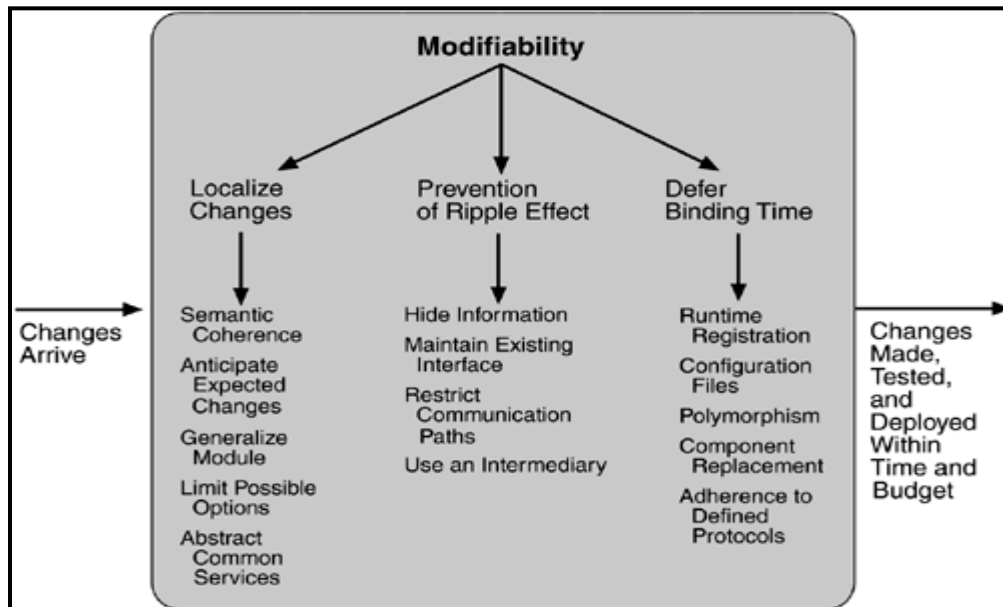


Figure 13: Summary of modifiability tactics

SOFTWARE ARCHITECTURE AND DESIGN PATTERNS

UNIT-I

5.3 Performance Tactics: The event can be single or a stream and is the trigger for a request to perform computation. It can be the arrival of a message, the expiration of a time interval, the detection of a significant change of state in the system's environment, and so forth. The system processes the events and generates a response. Performance tactics control the time within which a response is generated.

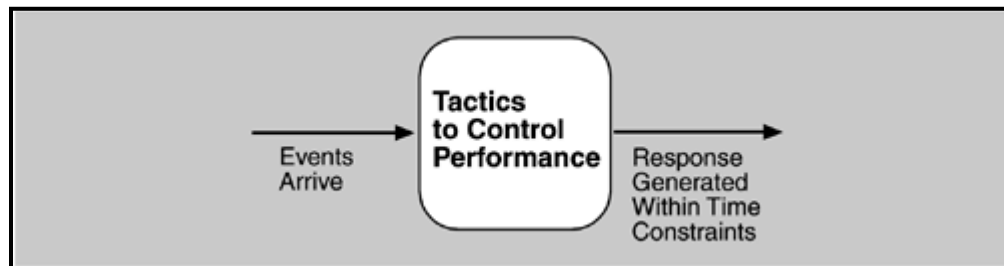


Figure 14: Goal of performance tactics

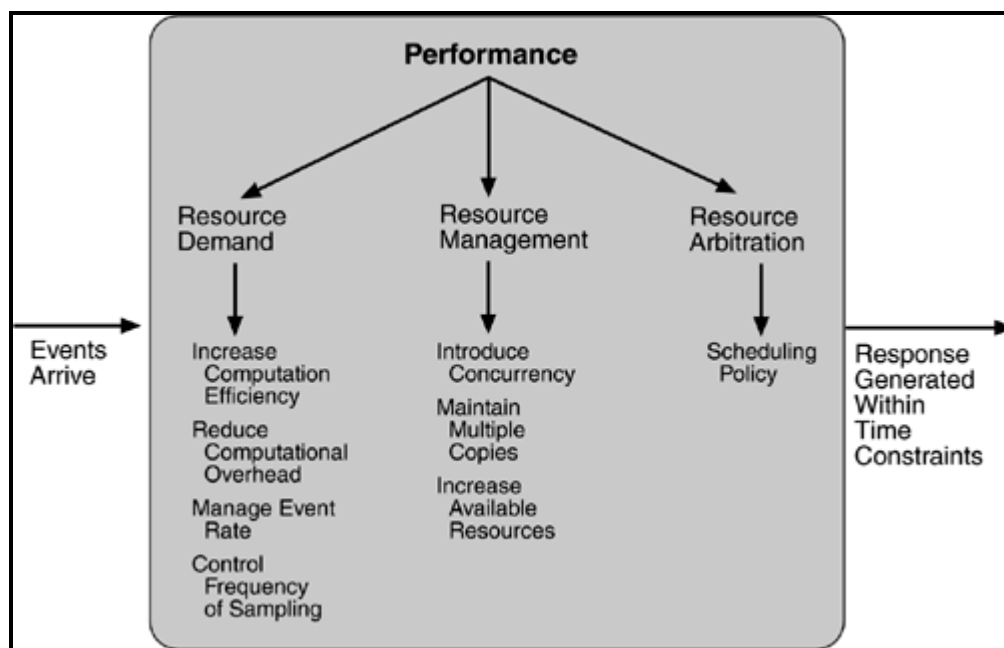


Figure 15: Summary of performance tactics

SOFTWARE ARCHITECTURE AND DESIGN PATTERNS

UNIT-I

5.4 Security Tactics: Tactics for achieving security can be divided into those concerned with resisting attacks, those concerned with detecting attacks, and those concerned with recovering from attacks. All three categories are important. Using a familiar analogy, putting a lock on your door is a form of resisting an attack, having a motion sensor inside of your house is a form of detecting an attack, and having insurance is a form of recovering from an attack.

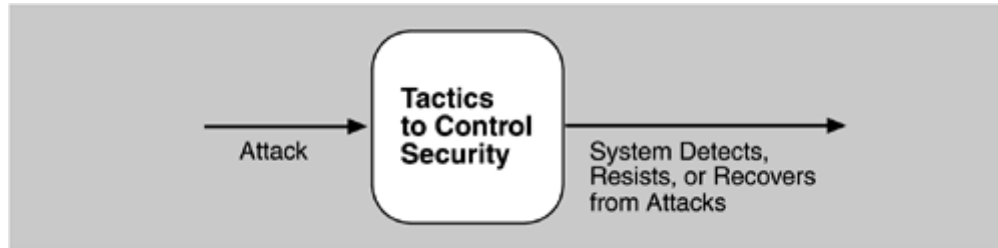


Figure 16: Goal of security tactics

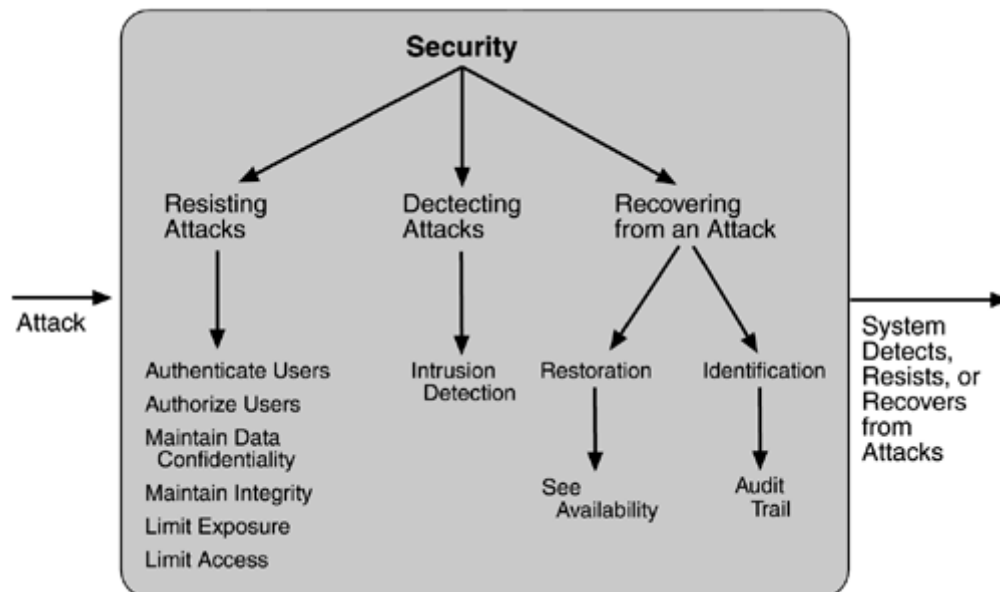


Figure 17: Summary of tactics for security

SOFTWARE ARCHITECTURE AND DESIGN PATTERNS

UNIT-I

5.5 Testability Tactics: The goal of tactics for testability is to allow for easier testing when an increment of software development is completed.

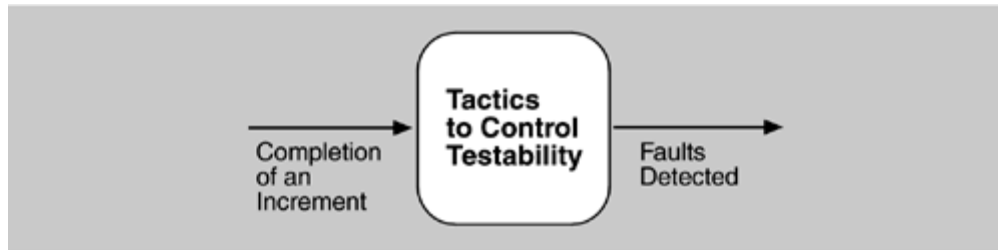


Figure 18: Goal of testability tactics

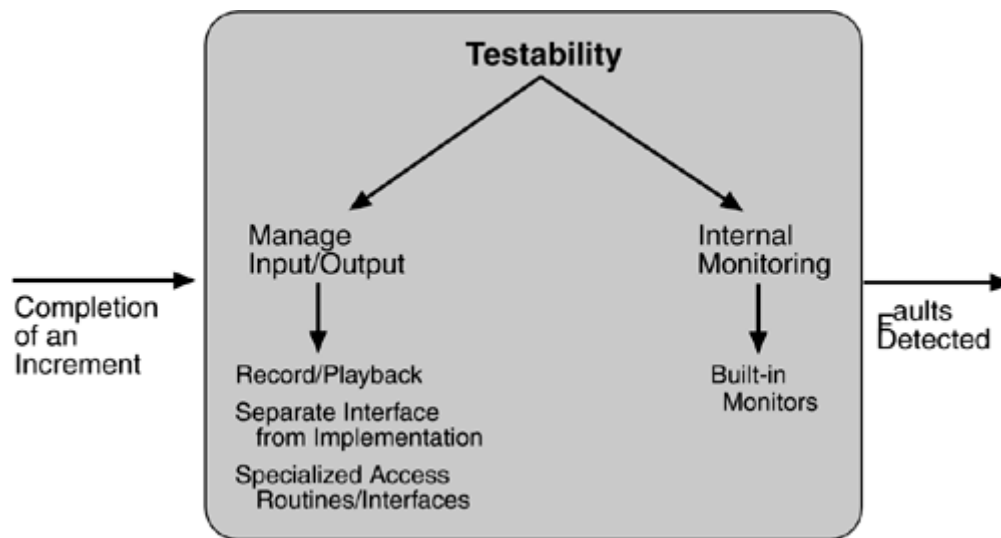


Figure 19: Summary of testability tactics

SOFTWARE ARCHITECTURE AND DESIGN PATTERNS

UNIT-I

5.6 Usability Tactics: Two types of tactics support usability, each intended for two categories of "users." The first category, runtime, includes those that support the user during system execution. The second category is based on the iterative nature of user interface design and supports the interface developer at design time. It is strongly related to the modifiability tactics already presented.

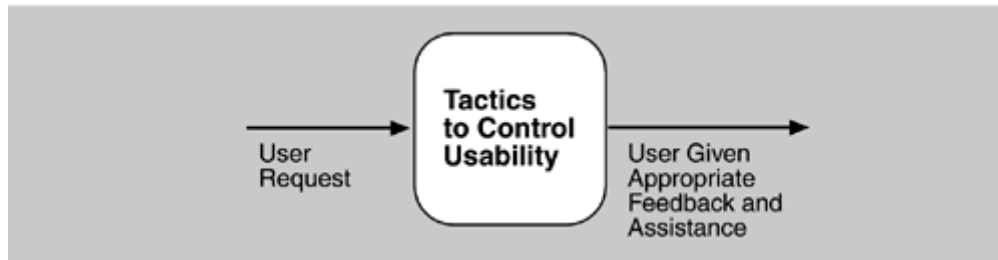


Figure 20: Goal of runtime usability tactics

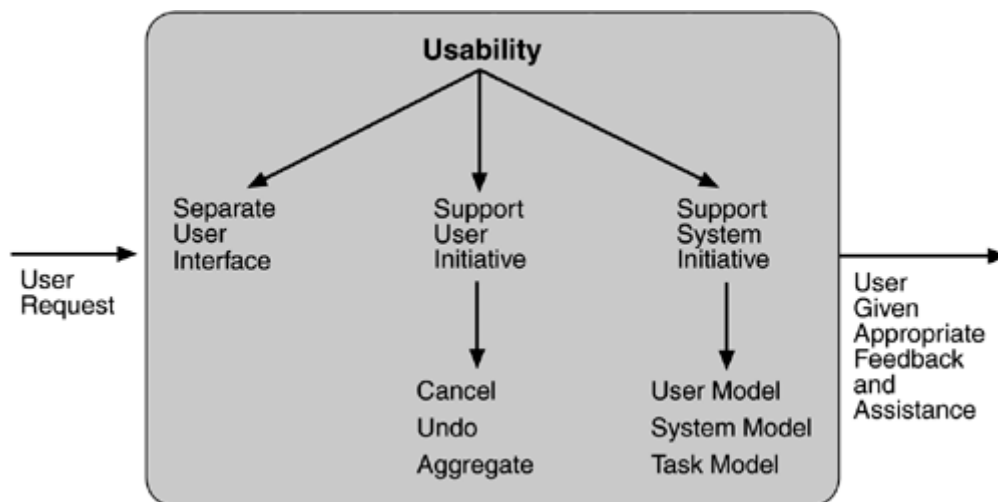


Figure 21: Summary of runtime usability tactics

SOFTWARE ARCHITECTURE AND DESIGN PATTERNS

UNIT-I

6. Architectural styles and patterns,

An architectural pattern in software, also known as an architectural style, is analogous to an architectural style in buildings, such as Gothic or Greek revival or Queen Anne. It consists of a few key features and rules for combining them so that architectural integrity is preserved. An architectural pattern is determined by:

- A set of element types (such as a data repository or a component that computes a mathematical function).
- A topological layout of the elements indicating their interrelation-ships.
- A set of semantic constraints
- A set of interaction mechanisms

In response, a number of recurring architectural patterns, their properties, and their benefits have been cataloged. One such catalog is illustrated in Figure 5.22

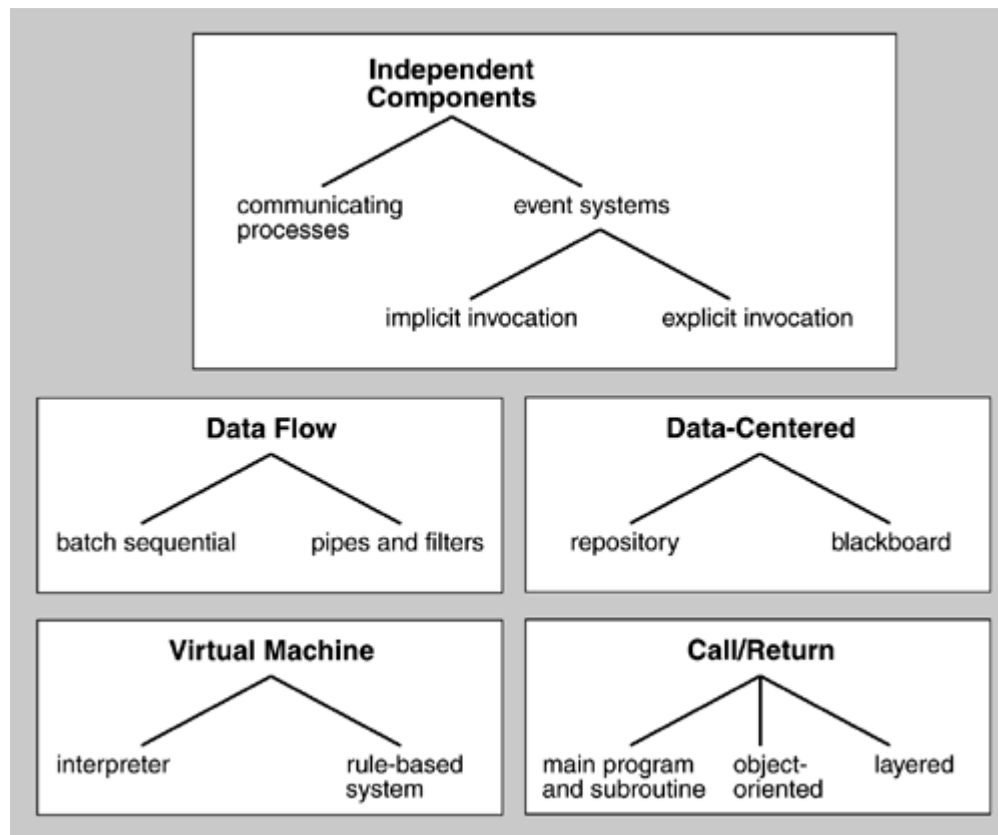


Figure 22: A small catalog of architectural patterns, organized by is-a relations

7. Designing the Architecture,

7.1 Architecture in the Life Cycle: Any organization that embraces architecture as a foundation for its software development processes needs to understand its place in the life cycle. Several life-cycle models exist in the literature, but one that puts architecture squarely in the middle of things is the Evolutionary Delivery Life Cycle model shown in Figure 23. The intent of this model is to get user and customer feedback and iterate through several releases before the final release. The model also allows the adding of functionality with each iteration and the delivery of a limited version once a sufficient set of features has been developed.

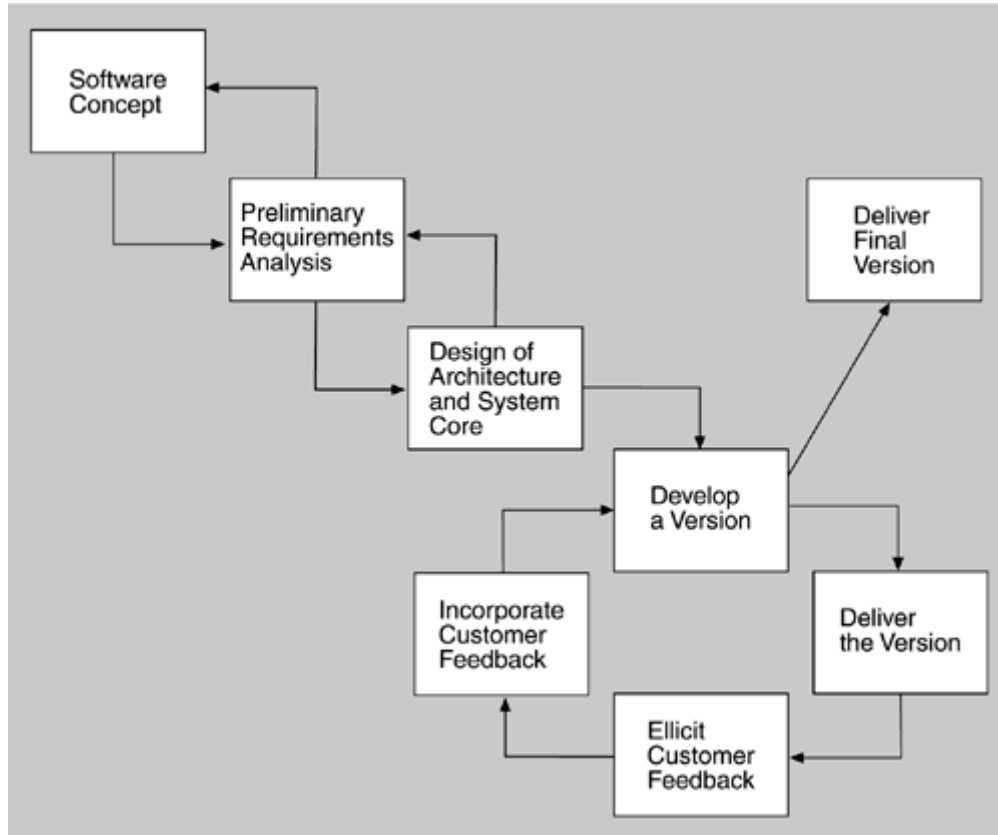


Figure 23: Evolutionary Delivery Life Cycle

7.2 Designing the Architecture: In this section we describe a method for designing architecture to satisfy both quality requirements and functional requirements. We call this method Attribute-Driven Design (ADD). ADD takes as input a set of quality attribute scenarios and employs knowledge about the relation between qualities attribute achievement and architecture in order to design the architecture. The ADD method can be viewed as an extension to most other development methods, such as the Rational Unified Process. The Rational Unified Process has several steps that result in the high-level design of an architecture but then proceeds to detailed design and implementation. Incorporating ADD into it involves modifying the steps dealing with the high-level design of the architecture and then following the process as described by Rational.

ADD is an approach to defining a software architecture that bases the decomposition process on the quality attributes the software has to fulfill. It is a recursive decomposition process where, at each stage, tactics and architectural patterns are chosen to satisfy a set of quality scenarios and then functionality is

SOFTWARE ARCHITECTURE AND DESIGN PATTERNS

UNIT-I

allocated to instantiate the module types provided by the pattern. ADD is positioned in the life cycle after requirements analysis and, as we have said, can begin when the architectural drivers are known with some confidence.

The output of ADD is the first several levels of a module decomposition view of architecture and other views as appropriate. Not all details of the views result from an application of ADD; the system is described as a set of containers for functionality and the interactions among them. This is the first articulation of architecture during the design process and is therefore necessarily coarse grained. Nevertheless, it is critical for achieving the desired qualities, and it provides a framework for achieving the functionality. The difference between an architecture resulting from ADD and one ready for implementation rests in the more detailed design decisions that need to be made. These could be, for example, the decision to use specific object-oriented design patterns or a specific piece of middleware that brings with it many architectural constraints. The architecture designed by ADD may have intentionally deferred this decision to be more flexible.

8. Documenting software architectures: Documenting the architecture is the crowning step to crafting it. Even a perfect architecture is useless if no one understands it or (perhaps worse) if key stakeholders misunderstand it. If you go to the trouble of creating a strong architecture, you must describe it in sufficient detail, without ambiguity, and organized in such a way that others can quickly find needed information. Otherwise, your effort will have been wasted because the architecture will be unusable.

This principle is useful because it breaks the problem of architecture documentation into more tractable parts, which provide the structure for the remainder of this chapter:

- Choosing the relevant views
- Documenting a view
- Documenting information that applies to more than one view

Software architecture views are divided views into these three groups: module, component-and-connector (C&C), and allocation. This three-way categorization reflects the fact that architects need to think about their software in at least three ways at once:

- How it is structured as a set of implementation units
- How it is structured as a set of elements that have runtime behavior and interactions
- How it relates to non-software structures in its environment

There is no industry-standard template for documenting a view, but the seven-part standard organization that we suggest in this section has worked well in practice.

1. Primary presentation
2. Element catalog
3. Context diagram
4. Variability guide
5. Architecture background
6. Glossary of terms
7. Other information

SOFTWARE ARCHITECTURE AND DESIGN PATTERNS

UNIT-I

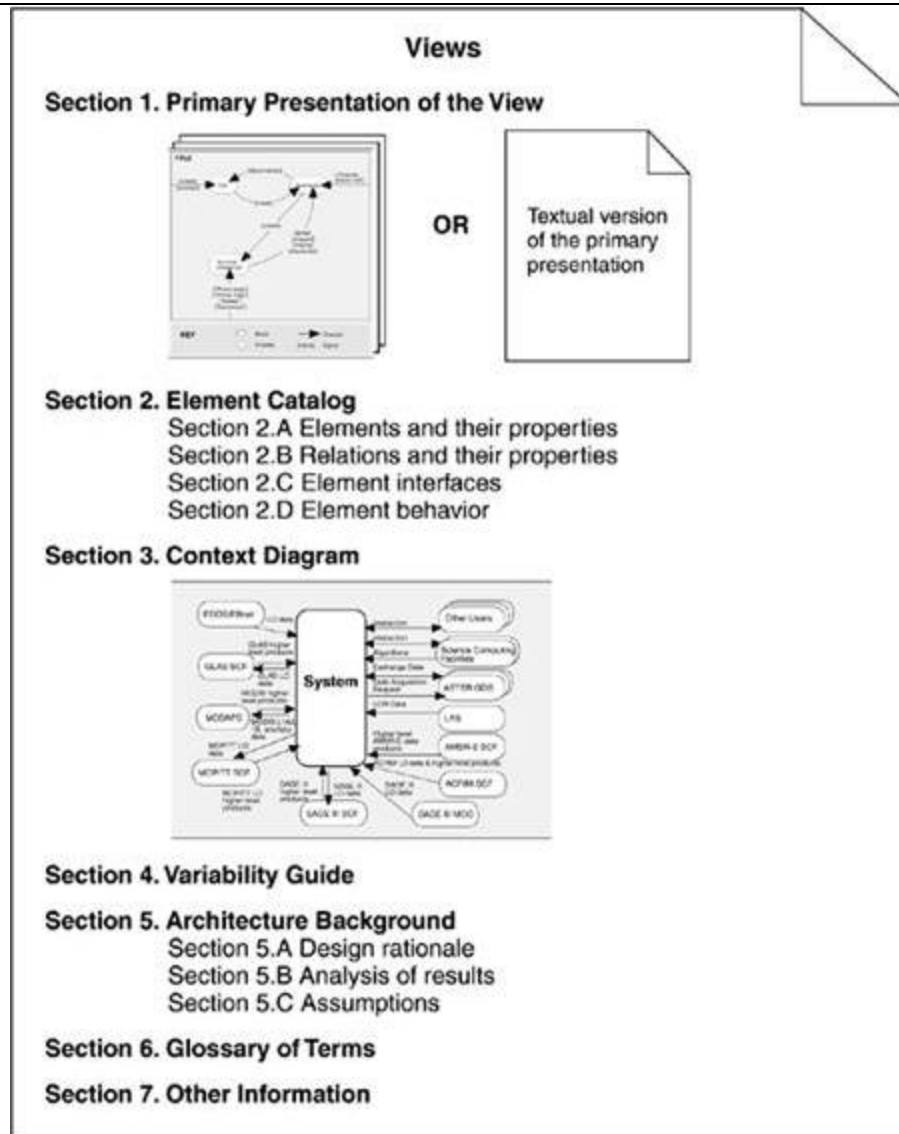


Figure 24: The seven parts of a documented view

SOFTWARE ARCHITECTURE AND DESIGN PATTERNS

UNIT-I

9. Reconstructing Software Architecture: Architecture reconstruction has been used in a variety of projects ranging from MRI scanners to public telephone switches and from helicopter guidance systems to classified NASA systems. It has been used

- To redocument architectures for physics simulation systems.
- To understand architectural dependencies in embedded control software for mining machinery.
- To evaluate the conformance of a satellite ground system's implementation to its reference architecture.
- To understand different systems in the automotive industry.

Reconstruction Activities

Software architecture reconstruction comprises the following activities, carried out iteratively:

- Information extraction. The purpose of this activity is to extract information from various sources.
- Database construction. Database construction involves converting this information into a standard form such as the Rigi Standard Form (a tuple-based data format in the form of relationship <entity1> <entity2>) and an SQL-based database format from which the database is created.
- View fusion. View fusion combines information in the database to produce a coherent view of the architecture.
- Reconstruction. The reconstruction activity is where the main work of building abstractions and various representations of the data to generate an architecture representation takes place.

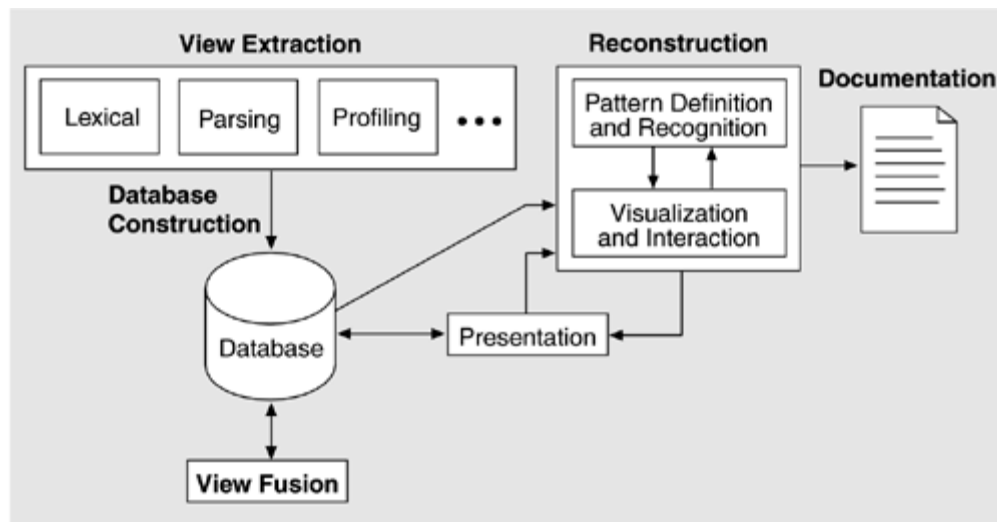


Figure 25: Architecture reconstruction activities.

SOFTWARE ARCHITECTURE AND DESIGN PATTERNS
UNIT-I

Frequently Asked Questions

1. Write a short note on Architecture Business Cycle (ABC)?
2. Explain in detail about software processes used in ABC, and explain what makes a good architecture?
3. Define reference model, reference architecture and architectural patterns, views?
4. Briefly discuss about the list of quality attributes using quality attribute scenarios with examples?
5. Explain how to achieve qualities in attributes using Tactics; explain about any two attributes with examples?
6. Write a short note on architectural styles and patterns?
7. What are the phases of architectures in life cycle?
8. What are the parts of documentation, and how to reconstruct software architecture?

SOFTWARE ARCHITECTURE AND DESIGN PATTERNS

UNIT-II

UNIT-II: Analyzing Architectures: Architecture Evaluation, Architecture design decision making, ATAM, CBAM

Moving from One System to Many: Software Product Lines, Building systems from off the shelf components, Software architecture in future

1. ARCHITECTURE EVALUATION

Architecture evaluation has come to provide relatively a low-cost risk mitigation capability. Making sure the architecture is the right one simply makes good sense. An architecture evaluation should be a standard part of every architecture-based development methodology. Evaluation can also be used to choose between two competing architectures by evaluating both and seeing which one fares better against the criteria for "goodness."

Evaluations can be planned or unplanned.

- **A planned evaluation** is considered a normal part of the project's development cycle. It is scheduled well in advance, built into the project's work plans and budget, and follow-up is expected.
- **An unplanned evaluation** is unexpected and usually the result of a project in serious trouble and taking extreme measures to try to salvage previous effort.

The planned evaluation is ideally considered an asset to the project, at worst a distraction from it. Planned evaluations are pro-active and team-building.

An unplanned evaluation is more of an ordeal for project members, consuming extra project resources and time in the schedule from a project already struggling with both. Unplanned evaluations are reactive, and tend to be tension filled. An evaluation's team leader must take care not to let the activities devolve into finger pointing.

A successful evaluation will have the following properties:

- **Clearly articulated goals and requirements for the architecture.** Architecture is only suitable, or not, in the presence of specific quality attributes. One that delivers breathtaking performance may be totally wrong for an application that needs modifiability.
- **Controlled scope.** In order to focus the evaluation, a small number of explicit goals should be enumerated. The number should be kept to a minimum? Around three to five?.
- **Cost-effectiveness.** Evaluation sponsors should make sure that the benefits of the evaluation are likely to exceed the cost. The types of evaluation we describe are suitable for medium and large-scale projects but may not be cost-effective for small projects.
- **Key personnel availability.** It is imperative to secure the time of the architect or at least someone who can speak authoritatively about the system's architecture and design. This person (or these people) primarily should be able to communicate the facts of the architecture quickly and clearly as well as the motivation behind the architectural decisions.

SOFTWARE ARCHITECTURE AND DESIGN PATTERNS
UNIT-II

- **Competent evaluation team.** Ideally, software architecture evaluation teams are separate entities within a corporation, and must be perceived as impartial, objective, and respected. The team must be seen as being composed of people appropriate to carry out the evaluation, so that the project personnel will not regard the evaluation as a waste of time and so that its conclusions will carry weight.
- **Managed expectations.** Critical to the evaluation's success is a clear, mutual understanding of the expectations of the organization sponsoring it. The evaluation should be clear about what its goals are, what it will produce, what areas it will (and will not) investigate, how much time and resources it will take from the project, and to whom the results will be delivered.

2. Architecture Tradeoff Analysis Method (ATAM)

The ATAM is so named because it reveals how well architecture satisfies particular quality goals, and it provides insight into how quality goals interact? That is, how they trade off. The ATAM is designed to elicit the business goals for the system as well as for the architecture. It is also designed to use those goals and stakeholder participation to focus the attention of the evaluators on the portion of the architecture that is central to the achievement of the goals.

2.1 Participants in the ATAM:

The ATAM requires the participation and mutual cooperation of three groups:

- **The evaluation team:** This group is external to the project whose architecture is being evaluated. It usually consists of three to five people. Each member of the team is assigned a number of specific roles to play during the evaluation.
- **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 will be present (or represented) as well.
- **Architecture stakeholders:** Stakeholders have a vested interest in the architecture performing as advertised. They are the ones whose ability to do their jobs hinges on the architecture promoting modifiability, security, high reliability, or the like. Stakeholders include developers, testers, integrators, maintainers, performance engineers, users, builders of systems interacting with the one under consideration, and others.

| Role | Responsibilities | Desirable characteristics |
|-------------------|---|---|
| Team Leader | Sets up the evaluation; coordinates with client, making sure client's needs are met; establishes evaluation contract; | Well-organized, with managerial skills; good at interacting with client; able to meet deadlines |
| Evaluation Leader | Runs evaluation; facilitates elicitation of scenarios; Administers scenario selection/prioritization process; | Comfortable in front of audience; excellent facilitation skills; good understanding of architectural issues |
| Scenario Scribe | Writes scenarios on flipchart or whiteboard during scenario elicitation; | Good handwriting; stickler about not moving on before an idea |

SOFTWARE ARCHITECTURE AND DESIGN PATTERNS
UNIT-II

| | | |
|--------------------|--|---|
| | captures agreed-on wording of each scenario, halting discussion until exact wording is captured | (scenario) is captured; can absorb and distill the essence of technical discussions |
| Proceedings Scribe | Captures proceedings in electronic form on laptop or workstation, raw scenarios, issue(s) that motivate each scenario | Good, fast typist; well organized for rapid recall of information; good understanding of architectural issues; |
| Timekeeper | Helps evaluation leader stay on schedule; helps control amount of time devoted to each scenario during the evaluation phase | Willing to interrupt discussion to call time |
| Process Observer | Keeps notes on how evaluation process could be improved or deviated from; usually keeps silent but may make discreet process-based suggestions to the evaluation leader during the evaluation; | Thoughtful observer; knowledgeable in the evaluation process; should have previous experience in the architecture evaluation method |
| Process Enforcer | Helps evaluation leader remember and carry out the steps of the evaluation method | Fluent in the steps of the method, and willing and able to provide discreet guidance to the evaluation leader |
| Questioner | Raise issues of architectural interest that stakeholders may not have thought of | Good architectural insights; good insights into needs of stakeholders; experience with systems in similar domains; unafraid to bring up contentious issues and pursue them; familiar with attributes of concern |

2.2 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.
4. A set of risks and non-risks.
 - A risk is defined as an architectural decision that may lead to undesirable consequences in light of quality attribute requirements.
 - A non-risk is an architectural decision that, upon analysis, is deemed safe.
 - The identified risks form the basis for an architectural risk mitigation plan.
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.

SOFTWARE ARCHITECTURE AND DESIGN PATTERNS
UNIT-II

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.

2.3 Phases of the ATAM: Activities in an ATAM-based evaluation are spread out over four phases.

| 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 |

2.4 Steps of the Evaluation Phases

- 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 (0.5-1.5hours):** 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, non-risks, sensitivities, and tradeoffs and discusses whether any new risk themes have arisen.

3. Cost Benefit Analysis Method (CBAM)

The software architect or decision maker wishes to maximize the difference between the benefit derived from the system and the cost of implementing the design. The CBAM begins where the ATAM concludes and, in fact, depends upon the artifacts that the ATAM produces as output. Figure 1 depicts the context for the CBAM.

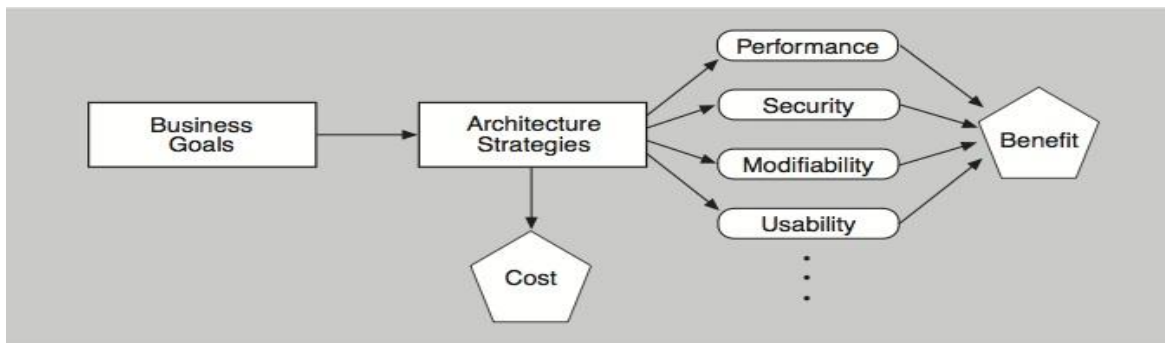


Figure 1: Depicts the context for the CBAM

Implementing the CBAM

A process flow diagram for the CBAM is given in Figure 12.3. The first four steps are annotated with the relative number of scenarios they consider. That number steadily decreases, ensuring that the method concentrates the stakeholders' time on the scenarios believed to be of the greatest potential in terms of ROI.

Step 1: Collate scenarios: Collate the scenarios elicited during the ATAM exercise, and give the stakeholders the chance to contribute new ones. Prioritize these scenarios based on satisfying the business goals of the system and choose the top one-third for further study.

Step 2: Refine scenarios: Refine the scenarios output from step 1, focusing on their stimulus-response measures. Elicit the worst-case, current, desired, and best-case quality attribute response level for each scenario.

Step 3: Prioritize scenarios: Allocate 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% 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, and best-case) for the scenarios from step 3.

SOFTWARE ARCHITECTURE AND DESIGN PATTERNS

UNIT-II

Step 5: Develop architectural strategies for scenarios and determine their expected quality attribute response levels: Develop (or capture already developed) architectural strategies that address the chosen scenarios and determine the "expected" quality attribute response levels that will result from them. Given that an architectural strategy may have effects on multiple scenarios, we must perform this calculation for each scenario affected.

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.

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.

Step 8: Choose architectural strategies based on ROI subject to cost and schedule constraints: Determine the cost and schedule implications of each architectural strategy. Calculate the ROI value for each as a ratio of benefit to cost. Rank-order the architectural strategies according to the ROI value and choose the top ones until the budget or schedule is exhausted.

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.

4. SOFTWARE PRODUCT LINES

Software architecture represents a significant investment of time and effort, usually by senior talent. So it is natural to want to maximize the return on this investment by re-using architecture across multiple systems. Architecturally mature organizations tend to treat their architectures as valuable intellectual property and look for ways in which that property can be leveraged to produce additional revenue and reduce costs. Both are possible with architecture re-use.

Software product lines based on inter-product commonality represent an innovative, growing concept in software engineering. Every customer has its own requirements, which demand flexibility on the part of the manufacturers. Software product lines simplify the creation of systems built specifically for particular customers or customer groups.

The improvements in cost, time to market, and productivity that come with a successful product line can be breathtaking. Consider:

- Nokia is able to produce 25 to 30 different phone models per year (up from 4 per year) because of the product line approach.
- Cummins, Inc., was able to reduce the time it takes to produce the software for a diesel engine from about a year to about a week.

SOFTWARE ARCHITECTURE AND DESIGN PATTERNS

UNIT-II

- Motorola observed a 400% productivity improvement in a family of one-way pagers.
- Hewlett-Packard reported a time to market reduced by a factor of seven and a productivity increase of a factor of six in a family of printer systems.

Creating a successful product line depends on a coordinated strategy involving software engineering, technical management, and organization management.

Architectures for Product Lines: A product line architect needs to consider three things:

1. **Identifying variation points:** Identifying variation is an ongoing activity. Because of the many ways a product can vary, variants can be identified at virtually any time during the development process. Some variations are identified during product line requirements elicitation; others, during architecture design; and still others, during implementation. Variations may also be identified during implementation of the second (and subsequent) products as well.
2. **Supporting variation points:** In a conventional architecture, the mechanism for achieving different instances almost always comes down to modifying the code. But in a software product line, architectural support for variation can take many forms:
 - Inclusion or omission of elements:** This decision can be reflected in the build procedures for different products, or the implementation of an element can be conditionally compiled based on some parameter indicating its presence or absence.
 - Inclusion of a different number of replicated elements:** For instance, high-capacity variants might be produced by adding more servers? The actual number should be unspecified, as a point of variation. Again, a build file would select the number appropriate for a particular product.
3. **Evaluating the architecture for product line suitability:** Like any other, the architecture for a software product line should be evaluated for fitness of purpose. In fact, given the number of systems that will rely on it, evaluation takes on an even more important role for product line architecture.

The good news is that the evaluation techniques described earlier in this book work well for product line architectures. The architecture should be evaluated for its robustness and generality, to make sure it can serve as the basis for products in the product line's envisioned scope. It should also be evaluated to make sure it meets the specific behavioral and quality requirements of the product at hand. We begin by focusing on the and how of the evaluation and then turn to when it should take place.

What and How to Evaluate: The evaluation will have to focus on the variation points to make sure they are appropriate, that they offer sufficient flexibility to cover the product line's intended scope, that they allow products to be built quickly, and that they do not

SOFTWARE ARCHITECTURE AND DESIGN PATTERNS

UNIT-II

impose unacceptable runtime performance costs. If your evaluation is scenario based, expect to elicit scenarios that involve instantiating the architecture to support different products in the family. Also, different products in the product line may have different quality attribute requirements, and the architecture will have to be evaluated for its ability to provide all required combinations. Here again, try to elicit scenarios that capture the quality attributes required of family members.

When to Evaluate: An evaluation should be performed on an instance or variation of the architecture that will be used to build one or more products in the product line. The extent to which this is a separate, dedicated evaluation depends on the extent to which the product architecture differs in quality-attribute-affecting ways from the product line architecture. If it does not differ, the product line architecture evaluation can be abbreviated; since many of the issues normally be raised in a single product evaluation will have been dealt with in the product line evaluation. In fact, just as the product architecture is a variation of the product line architecture, the product architecture evaluation is a variation of the product line architecture evaluation. Therefore, depending on the evaluation method used, the evaluation artifacts (scenarios, checklists, etc.) will have re-use potential, and you should create them with that in mind. The results of evaluation of product architectures often provide useful feedback to the product line architects and fuel architectural improvements.

5. BUILDING SYSTEMS FROM OFF THE SHELF COMPONENTS

Consider the following situation. You are producing software to control a chemical plant. Within chemical plants, specialized displays keep the operator informed as to the state of the reactions being controlled. A large portion of the software you are constructing is used to draw those displays. A vendor sells user interface controls that produce them. Because it is easier to buy than build, you decide to purchase the controls? Which, by the way, are only available for Visual Basic?

What impact does this decision have on your architecture? Either the whole system must be written in Visual Basic with its built-in callback-centered style or the operator portion must be isolated from the rest of the system in some fashion. This is a fundamental structural decision, driven by the choice of a single component for a single portion of the system.

The use of off-the-shelf components in software development, while essential in many cases, also introduces new challenges. In particular, component capabilities and liabilities are a principle architectural constraint.

All but the simplest components have a presumed architectural pattern that is difficult to violate. For example, an HTTP server assumes a client-server architectural pattern with defined interfaces and mechanisms for integrating back-end functionality. If the architecture you design conflicts with the architecture assumed by an HTTP server component, you may find yourself with an exceptionally difficult integration task.

A prototype situated in a specific design context is called a model solution. A model problem may have any number of model solutions, depending on the severity of risk inherent in the design context and on the success of the model solutions in addressing it.

Model problems are normally used by design teams. Optimally, the design team consists of an architect who is the technical lead on the project and makes the principal design decisions, as well as a number of designers/engineers who may implement a model solution for the model problem.

An illustration of the model problem work flow is shown in Figure 2. The process consists of the following six steps that can be executed in sequence:

- The architect and the engineers identify a design question. The design question initiates the model problem, referring to an unknown that is expressed as a hypothesis.
- The architect and the engineers define the starting evaluation criteria. These criteria describe how the model solution will support or contradict the hypothesis.
- The architect and the engineers define the implementation constraints. The implementation constraints specify the fixed (inflexible) part of the design context that governs the implementation of the model solution. These constraints might include such things as platform requirements, component versions, and business rules.

SOFTWARE ARCHITECTURE AND DESIGN PATTERNS

UNIT-II

- The engineers produce a model solution situated in the design context. The model solution is a minimal application that uses only the features of a component (or components) necessary to support or contradict the hypothesis.
- The engineers identify ending evaluation criteria. Ending evaluation criteria include the starting set plus criteria that are discovered as a by-product of implementing the model solution.
- The architect performs an evaluation of the model solution against the ending criteria. The evaluation may result in the design solution being rejected or adopted, but often leads to new design questions that must be resolved in similar fashion.

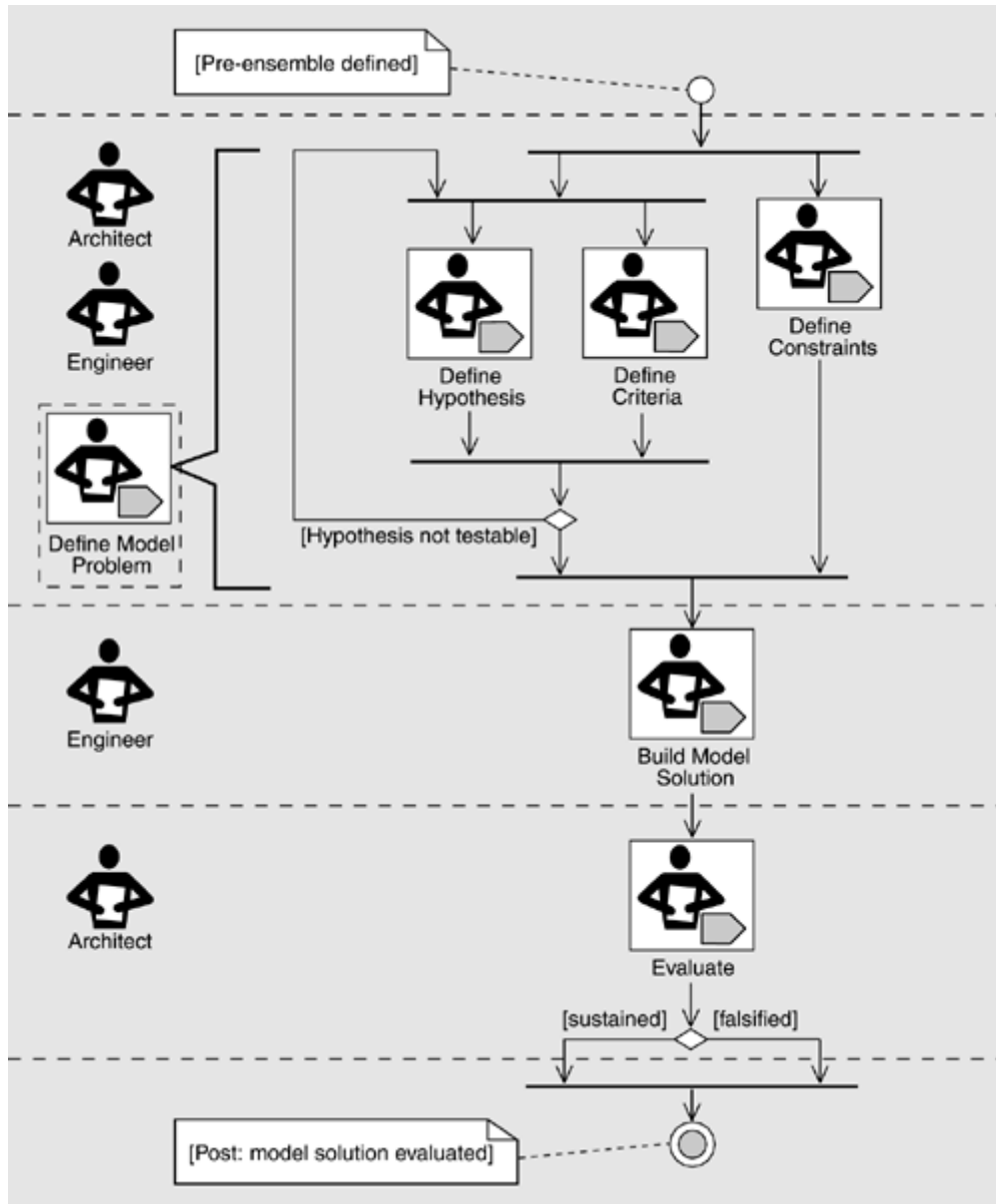


Figure 2: Model problem work flow

SOFTWARE ARCHITECTURE AND DESIGN PATTERNS

UNIT-II

6. SOFTWARE ARCHITECTURE IN FUTURE

The history of programming can be viewed as a succession of ever-increasing facilities for expressing complex functionality. In the beginning, assembly language offered the most elementary of abstractions: exactly where in physical memory things resided (relative to the address in some base register) and the machine code necessary to perform primitive arithmetic and move operations.

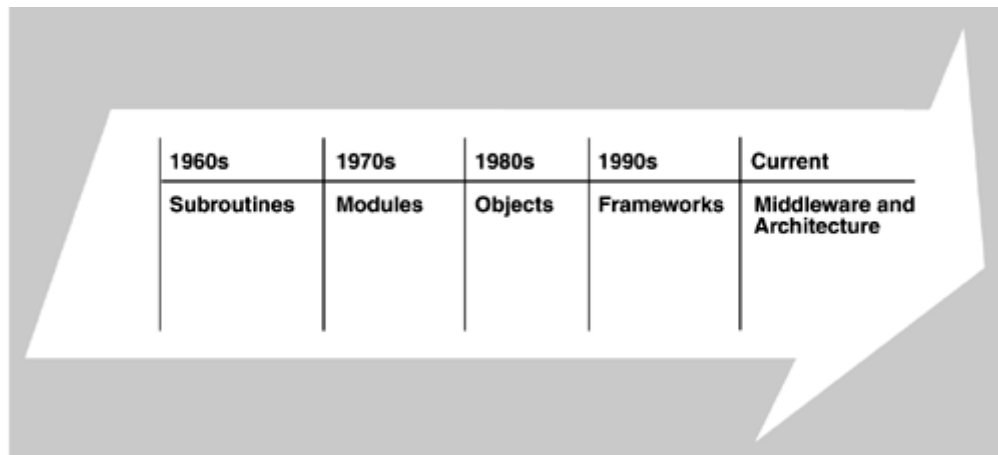
The 1970s saw a concern with the structuring of programs to achieve qualities beyond correct function. Data-flow analysis, entity-relation diagrams, information hiding, and other principles or techniques formed the bases of myriad design methodologies, each of which led to the creation of subroutines or collections of them whose functionality could be rationalized in terms of developmental qualities. These elements were usually called modules.

In the 1980s, module-based programming languages, information hiding, and associated methodologies crystallized into the concept of objects. Objects became the components du jour, with inheritance adding a new kind of (non-runtime) connector.

In the 1990s, standard object-based architectures, in the form of frameworks, started appearing. Objects have given us a standard vocabulary for elements and have led to new infrastructures for wiring collections of elements together. Abstractions have grown more powerful along the way; we now have computing platforms in our homes that let us treat complex entities, such as spreadsheets, documents, graphical images, audio clips, and databases, as interchangeable black-box objects that can be blithely inserted into instances of each other.

Architecture places the emphasis above individual elements and on the arrangement of the elements and their interaction. It is this kind of abstraction, away from the focus on individual elements that makes such breathtaking interoperability possible.

In the current decade, we see the rise of middleware and IT architecture as a standard platform. Purchased elements have security, reliability, and performance support services that a decade ago had to be added by individual project developers. We summarize this discussion in Figure 3.



| 1960s | 1970s | 1980s | 1990s | Current |
|-------------|---------|---------|------------|-----------------------------|
| Subroutines | Modules | Objects | Frameworks | Middleware and Architecture |

Figure 3: Growth in the types of abstraction available over time

Frequently Asked questions

1. Explain about various architecture Evaluation approaches?
2. Explain the participant of ATAM with their roles, responsibilities and characteristics?
3. Write a short note on outputs of ATAM and phases of ATAM?
4. Explain the evaluation steps of ATAM?
5. Briefly explain about Cost Benefit Analysis Method (CBAM) and evaluation steps?
6. Explain about Software Product Lines with suitable examples?
7. Explain in detail about building systems from off the shelf components with examples?
8. Write about Software architecture in future?

UNIT-III:

Patterns: Pattern Description, Organizing catalogs, role in solving design problems, Selection and usage. **Creational Patterns:** Abstract factory, Builder, Factory method, Prototype, Singleton

1. Introduction to design patterns or what is a design pattern (DP)?

Design patterns are repeatable / reusable solutions to commonly occurring problems in a certain context in software design. There are four essential elements of a pattern,

- Pattern name
- Problem
- Solution
- Consequences

Need for design patterns

- Designing reusable object-oriented software (API's) is hard.
- Experienced designers Vs novice designers.
- Patterns make object-oriented software flexible, elegant and reusable.
- Solved a problem previously but don't remember when or how?

Use of design patterns

- Make it easier to reuse successful designs and architectures.
- Make it easy for the new developers to design software.
- Allows choosing different design alternatives to make the software reusable
- Helps in documenting and maintaining the software

Why should we use DP's?

- These are already tested and proven solutions used by many experienced designers.

MVC Architecture

Model View Controller or MVC as it is popularly called, is a software design pattern for developing web applications. A Model View Controller pattern is made up of the following three parts.

Model – the lowest level of the pattern which is responsible for maintaining data

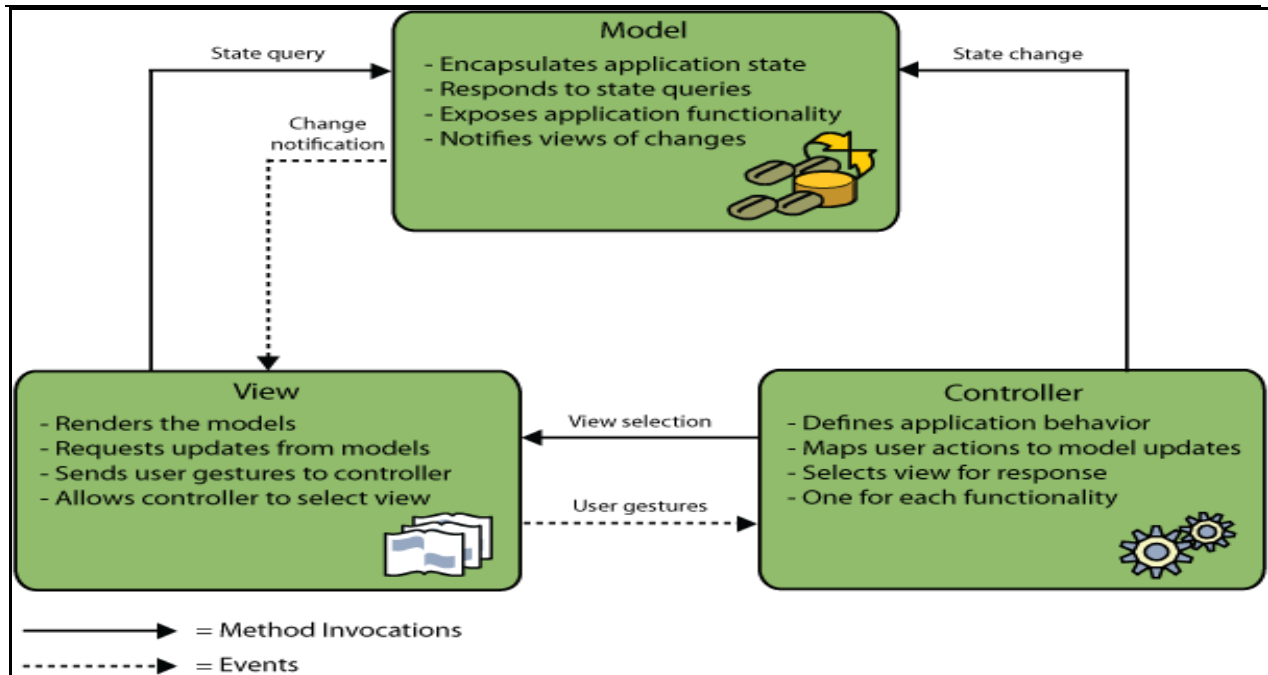
View – this is responsible for displaying all or a portion of the data to the user

Controller – Software Code that controls the interactions between the Model and View

MVC is popular as it isolates the application logic from the user interface layer and supports separation of concerns. Here the Controller receives all requests for the application and then works with the Model to prepare any data needed by the View. The View then uses the data prepared by the Controller to generate a final presentable response. The MVC abstraction can be graphically represented as follows.

SOFTWARE ARCHITECTURE AND DESIGN PATTERNS

UNIT-III



Describing design patterns

1. Pattern name and classification
2. Intent
3. Also known as
4. Motivation
5. Applicability
6. Structure
7. Participants
8. Collaborations
9. Consequences
10. Implementation
11. Sample code
12. Known uses
13. Related patterns

Organizing catalogs

| | | Purpose | | |
|-------|--------|---|--|---|
| | | Creational | Structural | Behavioral |
| Scope | Class | Factory Method | Adapter (class) | Interpreter Template Method |
| | Object | Abstract Factory Builder Prototype Singleton | Adapter (object) Bridge Composite Decorator Facade Flyweight Proxy | Chain of Responsibility Command Iterator Mediator Memento Observer State Strategy Visitor |

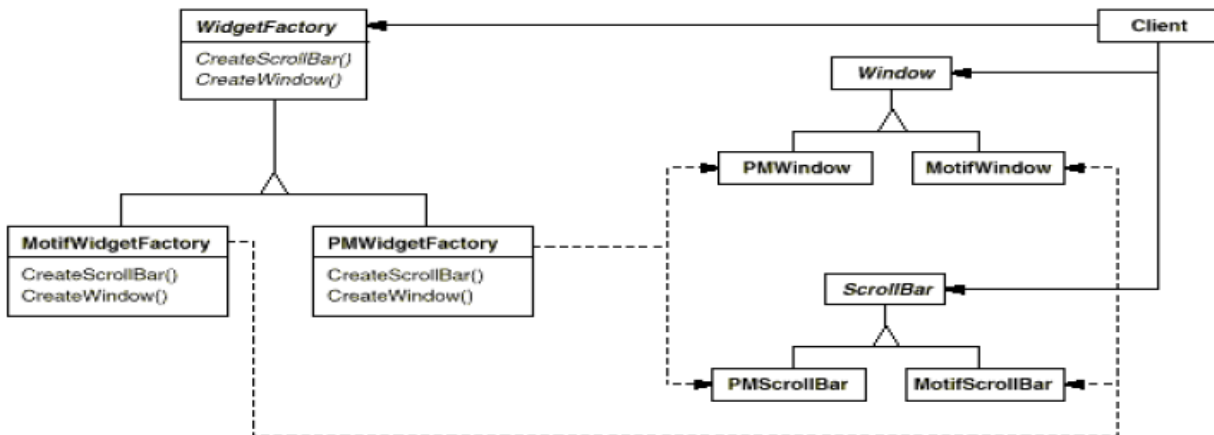
2. ABSTRACT FACTORY: Provides an interface to create a family of related objects, without explicitly specifying their concrete class.

Pattern name and classification

Intent: Provide an interface for creating families of related or dependent objects without specifying their concrete classes.

Also known as: Kit

Motivation: Creating interface components for different look and feels.

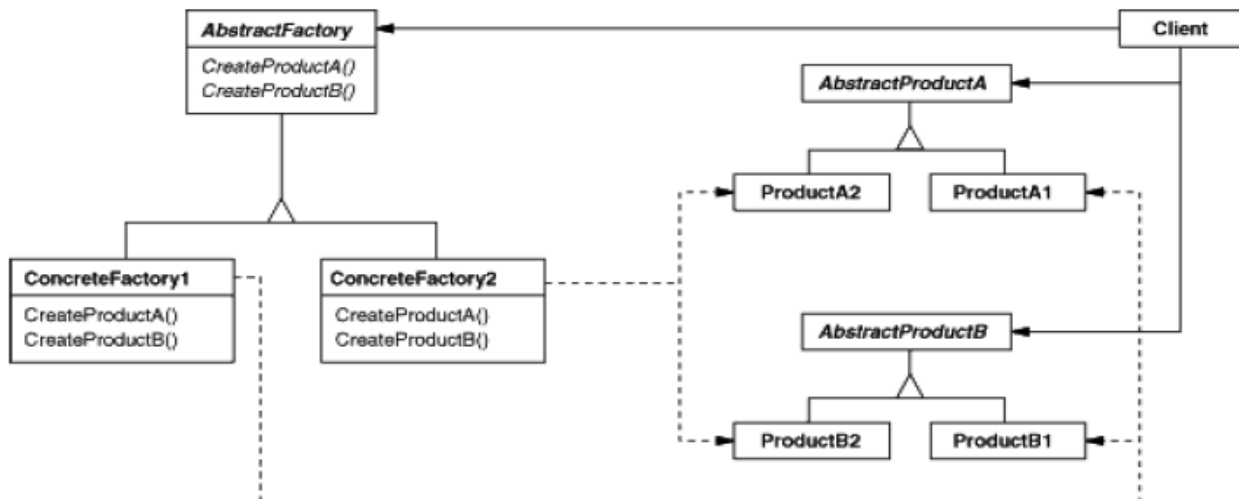


Applicability

We should use Abstract Factory design pattern when:

- The system needs to be independent of the products it works with are created.
- The system should be configured to work with multiple families of products.
- A family of products is designed to be used together and this constraint is needed to be enforced.
- A library of products is to be provided and only their interfaces are to be revealed but not their implementations.

Structure



Participants: The classes that participate in the Abstract Factory are: **AbstractFactory**, **ConcreteFactory**, **AbstractProduct**, **Product**, **Client**

SOFTWARE ARCHITECTURE AND DESIGN PATTERNS

UNIT-III

Collaborations

- The ConcreteFactory class creates products objects having a particular implementation. To create different product, the client should use a different concrete factory.
- AbstractFactory defers creation of product objects to its ConcreteFactory subclass.

Consequences

- It isolates concrete classes.
- It makes exchanging product families easy.
- It creates consistency among products.

Implementation

```
abstract class AbstractProductA
{
    public abstract void operationA1();
    public abstract void operationA2();
}
```

Sample code

```
public interface Window
{
    public void setTitle(String text);
    public void repaint();
}
```

Known uses

ET++ [WGM88] uses the Abstract Factory pattern to achieve portability across different window systems (X Windows and SunView, for example).

Related patterns

- AbstractFactory classes are often implemented with factory methods but they can also be implemented using Prototype.
- A concrete factory is often a Singleton.

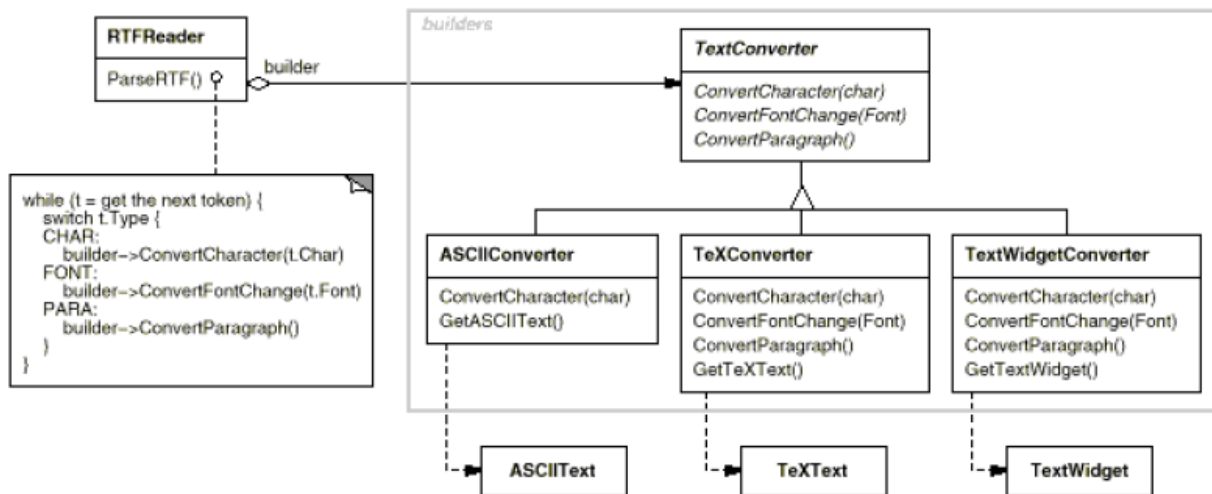
3. BUILDER

Pattern name and classification: Builder and Creational

Intent: Separate the construction of a complex object from its representation so that the same construction process can create different representations.

Also known as

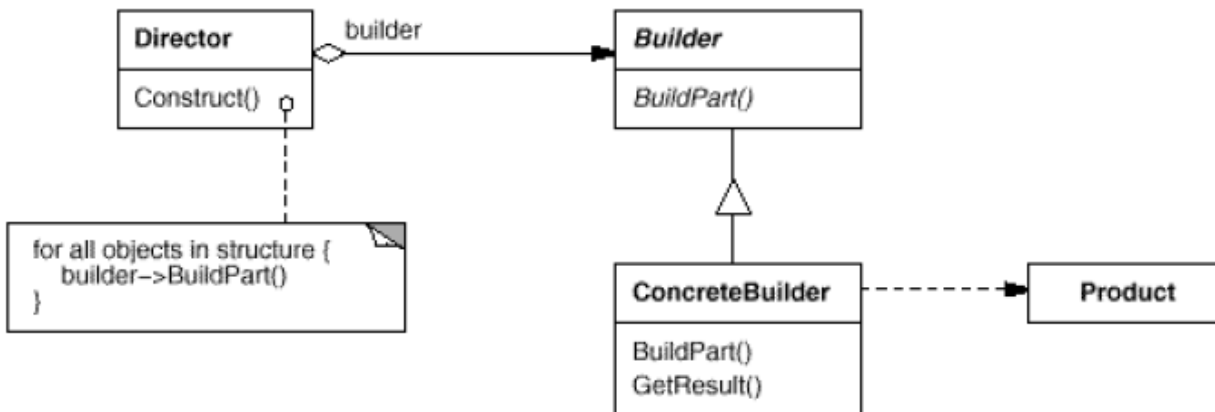
Motivation: Text Converter



Applicability

- The algorithm for creating a complex object should be independent of the parts that make up the object and how they are assembled.
- The construction process must allow different representations for the object that is constructed.

Structure



Participants: Builder, ConcreteBuilder, Director, Product

Collaborations

- The client creates the Director Object and configures it with the desired Builder object.
- Director notifies the builder whenever a part of the product should be built.
- Builder handles requests from the director and adds parts to the product.
- The client retrieves the product from the builder.

Consequences: The benefits and pitfalls of Builder pattern are:

- It lets you vary the product's internal representation.
- It isolates code for construction and representation.
- It gives you finer control over the construction process.

Implementation

//Abstract Builder

```
class abstract class TextConverter
{
    abstract void convertCharacter(char c);
    abstract void convertParagraph();
}
```

// Product

```
class ASCIIText
{
    public void append(char c)
    { //Implement the code here }
}
```

Sample code

```
abstract class AbstractProduct implements Cloneable
{
    public static AbstractProduct thePrototype;
    public static AbstractProduct makeProduct()
    {
        try
        {
            return (AbstractProduct) thePrototype.clone();
        }
        catch(CloneNotSupportedException e)
        {
            return null;
        }
    }
}
```

Known uses

The RTF converter application is from ET++ [WGM88]. Its text building block uses a builder to process text stored in the RTF format.

Related patterns

- Abstract Factory (87) is similar to Builder in that it too may construct complex objects. The primary difference is that the Builder pattern focuses on constructing a complex object step by step.
- A Composite (1 i6s3) what the builder often builds.

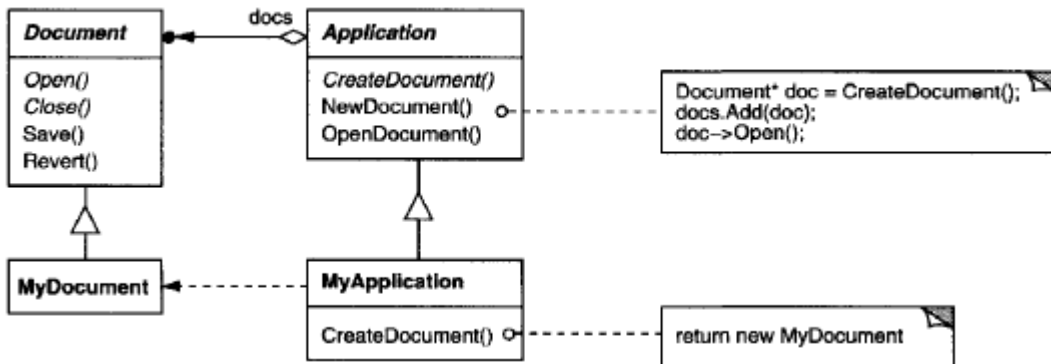
4. FACTORY METHOD

Pattern name and classification: Factory method and Creational

Intent: Define an interface for creating an object, but let subclasses decide which

Also known as: Virtual Constructor

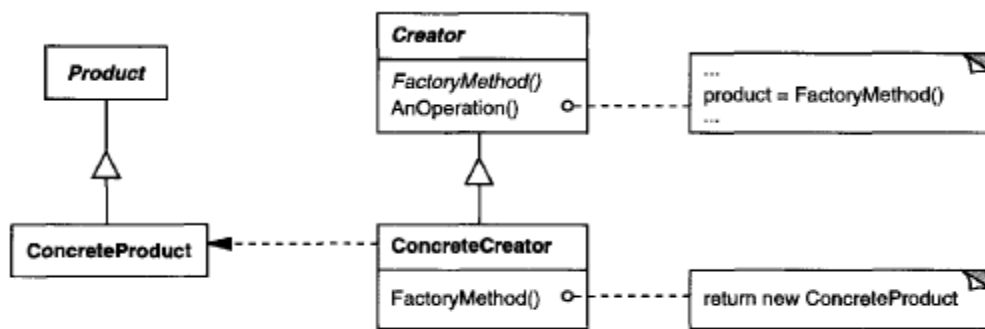
Motivation: Frameworks use abstract classes to define and maintain relationships between objects. A framework is often responsible for creating these objects as well.



Applicability: Use the Factory Method pattern when

- A class can't anticipate the class of objects it must create.
- A class wants its subclasses to specify the objects it creates.
- Classes delegate responsibility to one of several helper subclasses, and you want to localize the knowledge of which helper subclass is the delegate.

Structure



Participants: Product, ConcreteProduct, Creator, ConcreteCreator

Collaborations: Creator relies on its subclasses to define the factory method so that it returns an instance of the appropriate ConcreteProduct.

Consequences: Here are two additional consequences of the Factory Method pattern

- Provides hooks for subclasses
- Connects parallel class hierarchies

Implementation

```
abstract class AbstractProduct implements Cloneable
{
    public static AbstractProduct thePrototype;
    public static AbstractProduct makeProduct()
    {
```



```
        {
            try
            {
                return (AbstractProduct) thePrototype.clone();
            }
            catch(CloneNotSupportedException e)
            {
                return null;
            }
        }
    }
}
```

Sample code

```
class MazeGame { public:
    Maze* CreateMaze();
    // factory methods:
    virtual Maze* MakeMaze() const
        { return new Maze; }
    virtual Room* MakeRoom(int n) const
        { return new Room(n); }
    virtual Wall* MakeWall() const
        { return new Wall; }
    virtual Door* MakeDoor(Room* r1, Room* r2) const
        { return new Door(r1, r2); } };
```

Known uses

Factory methods pervade toolkits and frameworks. The preceding document example is a typical use in MacApp and ET++. The manipulator example is from Unidraw.

Related patterns

- Abstract Factory is often implemented with factory methods, as well.
- Factory methods are usually called within Template Methods
- Prototypes don't require subclassing Creator.

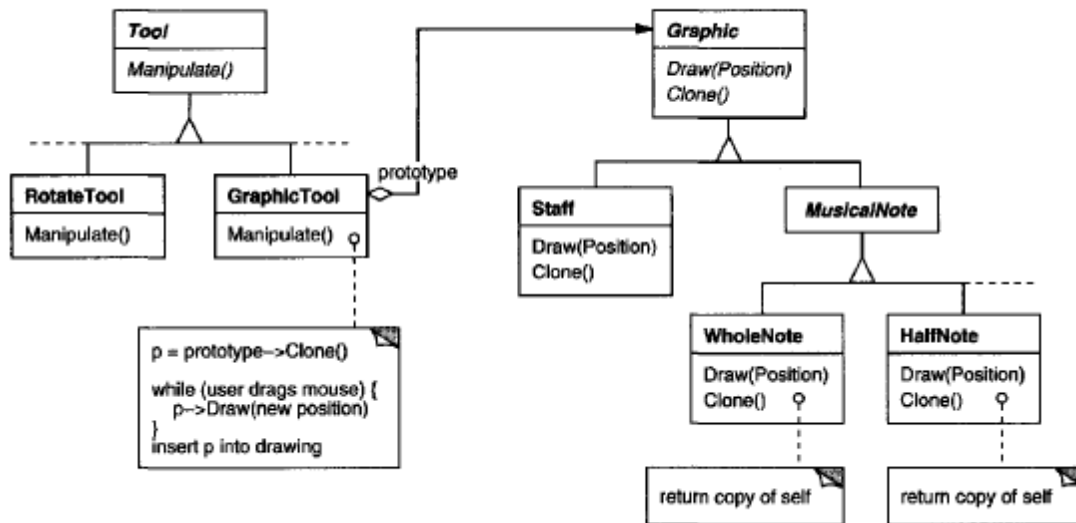
5. PROTOTYPE

Pattern name and classification: Prototype and Creational

Intent: Specify the kinds of objects to create using a prototypical instance, and create new objects by copying this prototype.

Also known as:

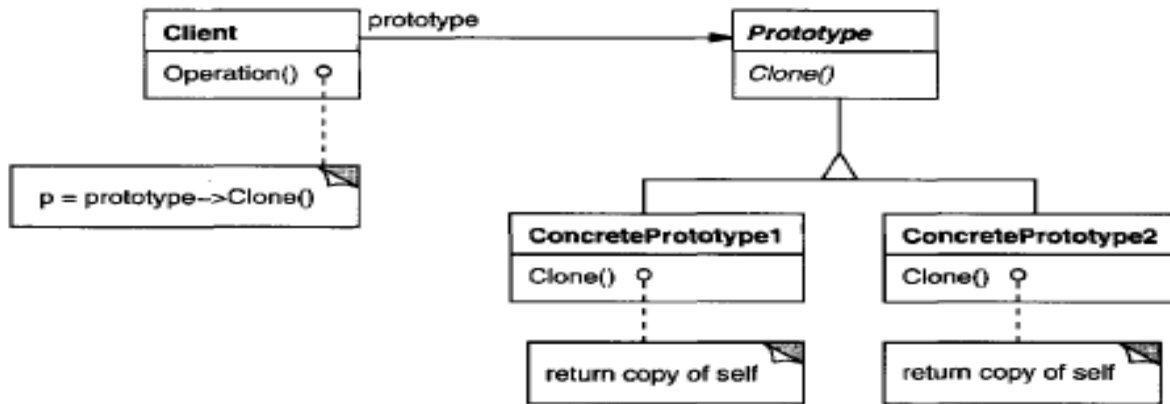
Motivation: You could build an editor for music scores by customizing a general framework for graphical editors and adding new objects that represent notes, rests, and staves.



Applicability: Use the Prototype pattern when a system should be independent of how its products are created, composed, and represented; and

- when the classes to instantiate are specified at run-time, for example, by dynamic loading ; or
- to avoid building a class hierarchy of factories that parallels the class hierarchy of products; or
- When instances of a class can have one of only a few different combinations of state. It may be more convenient to install a corresponding number of prototypes and clone them rather than instantiating the class manually, each time with the appropriate state.

Structure



Participants: Prototype, ConcretePrototype, Client

Collaborations: A client asks a prototype to clone itself

Consequences: Additional benefits of the Prototype pattern are listed below.

- Adding and removing products at run-time
- Specifying new objects by varying values
- Specifying new objects by varying structure
- Reduced sub classing
- Configuring an application with classes dynamically

Implementation: Configuring an application with classes dynamically

- Using a prototype manager
- Implementing the Clone operation
- Initializing clones

Sample code

```
class MazePrototypeFactory : public MazeFactory
{
public:
    MazePrototypeFactory (Maze*, Wall*, Room*, Door*),-

    virtual Maze* MakeMaze() const;
    virtual Room* MakeRoom(int) const;
    virtual Wall* MakeWall() const;
    virtual Door* MakeDoor(Room*, Room*) const;
private:
    Maze* _prototypeMaze;
    Room* _prototypeRoom;
    Wall* _prototypeWall;
    Door* _prototypeDoor;
};
```

Known uses

- The first widely known application of the pattern in an objectoriented language was in ThingLab, where users could form a composite object and then promote it to a prototype by installing it in a library of reusable objects

Related patterns

SOFTWARE ARCHITECTURE AND DESIGN PATTERNS
UNIT-III

- Prototype and Abstract Factory are competing patterns in some ways
- Designs that make heavy use of the Composite and Decorator patterns often can benefit from Prototype as well.

6. SINGLETON

Pattern name and classification

Intent: Ensure a class only has one instance, and provide a global point of access to it

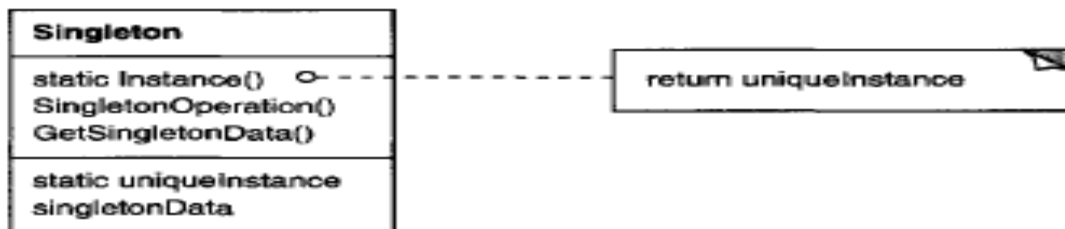
Also known as

Motivation: It's important for some classes to have exactly one instance. Although there can be many printers in a system, there should be only one printer spooler. There should be only one file system and one window manager. A digital filter will have one A/D converter. An accounting system will be dedicated to serving one company.

Applicability: Use the Singleton pattern when

- There must be exactly one instance of a class, and it must be accessible to clients from a well-known access point.
- When the sole instance should be extensible by sub classing, and clients should be able to use an extended instance without modifying their code.

Structure



Participants

- Defines an Instance operation that lets clients access its unique instance. Instance is a class operation (that is, a class method in Smalltalk and a static member function in C++).
- May be responsible for creating its own unique instance

Collaborations: Clients access a Singleton instance solely through Singleton's Instance operation.

Consequences: The Singleton pattern has several benefits

- Controlled access to sole instance
- Reduced name space
- Permits refinement of operations and representation
- Permits a variable number of instances
- More flexible than class operations

Implementation: The Singleton class is declared as

```
class Singleton
{
public:
    static Singleton* Instance();
protected:
    Singleton();
private:
```

```
static Singleton* _instance;  
};
```

The corresponding implementation is

```
Singleton* Singleton::_instance = 0;  
Singleton* Singleton::Instance () {  
    if (_instance == 0) {  
        _instance = new Singleton;  
    }  
    return _instance;  
}
```

Sample code

```
class MazeFactory  
{  
public:  
    static MazeFactory* Instance();  
    // existing interface goes here  
protected:  
    MazeFactory();  
private:  
    static MazeFactory* _instance;  
};
```

The corresponding implementation is

```
MazeFactory* MazeFactory::_instance = 0;  
MazeFactory* MazeFactory::Instance ()  
{  
    if (_instance == 0) {  
        _instance = new MazeFactory;  
    } return _  
instance;  
}
```

Known uses

- An example of the Singleton pattern in Smalltalk-80
- The Interviews user interface toolkit uses the Singleton pattern to access the unique instance of its Session and WidgetKit classes, among others.

Related patterns

- Many patterns can be implemented using the Singleton pattern. See Abstract Factory, Builder, and Prototype.

Frequently Asked Questions

1. Define Design Patterns and explain about MVC architecture?
2. Give detailed information of design patterns catalog?
3. Describe about the Abstract Factory design pattern?
4. Write a short note on Builder design pattern?
5. Explain the working of Factory design pattern?
6. Briefly explain about Prototype design pattern?
7. Describe in detail about Singleton design pattern?

UNIT-IV:

Structural Patterns: Adapter, Bridge, Composite, Decorator, Façade, Flyweight, PROXY.

Structural patterns are concerned with how classes and objects are composed to form larger structures. Structural class patterns use inheritance to compose interfaces or implementations. As a simple example, consider how multiple inheritances mix two or more classes into one. The result is a class that combines the properties of its parent classes. This pattern is particularly useful for making independently developed class libraries work together.

- **Adapter:** interface converter
- **Bridge:** decouple abstraction from its implementation
- **Composite:** compose objects into tree structures, treating all nodes uniformly
- **Decorator:** attach additional responsibilities dynamically
- **Façade:** provide a unified interface to a subsystem
- **Flyweight:** using sharing to support a large number of fine-grained objects efficiently
- **Proxy:** provide a surrogate for another object to control access

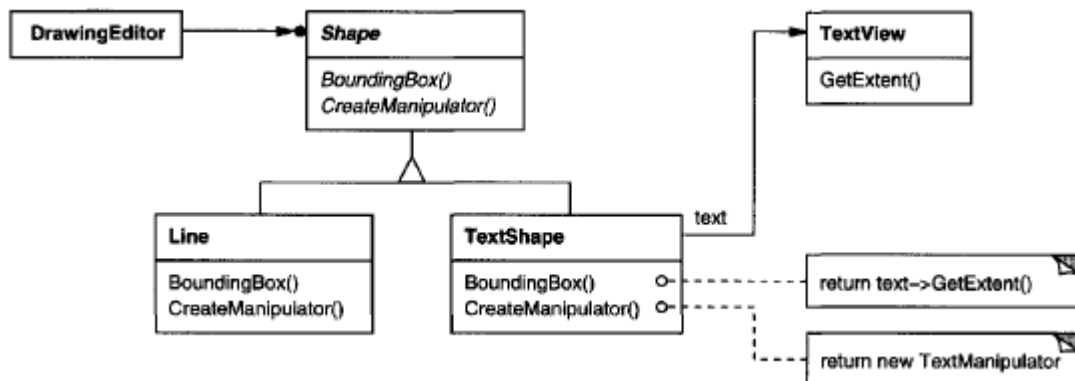
1. Adapter

Intent: Convert the interface of a class into another interface clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces.

Also Known As: Wrapper

Motivation: Sometimes a toolkit class that's designed for reuse isn't reusable only because its interface doesn't match the domain-specific interface an application requires.

Consider for example a drawing editor that lets users draw and arrange graphical elements (lines, polygons, text, etc.) into pictures and diagrams. The drawing editor's key abstraction is the graphical object, which has an editable shape and can draw itself. The interface for graphical objects is defined by an abstract class called Shape. The editor defines a subclass of Shape for each kind of graphical object: a LineShape class for lines, a PolygonShape class for polygons, and so forth.



Applicability

Use the Adapter pattern when

- You want to use an existing class, and its interface does not match the one you need.

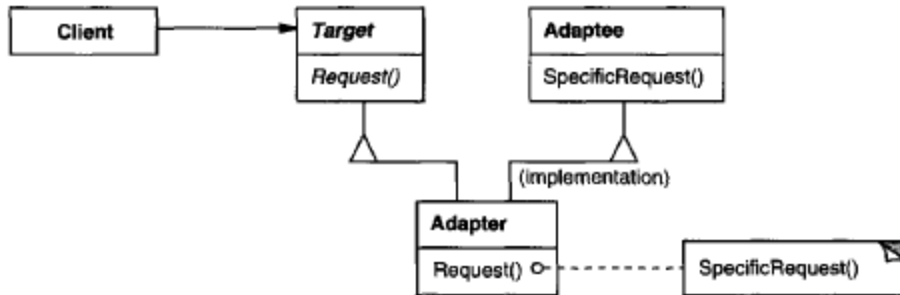
SOFTWARE ARCHITECTURE AND DESIGN PATTERNS

UNIT-IV

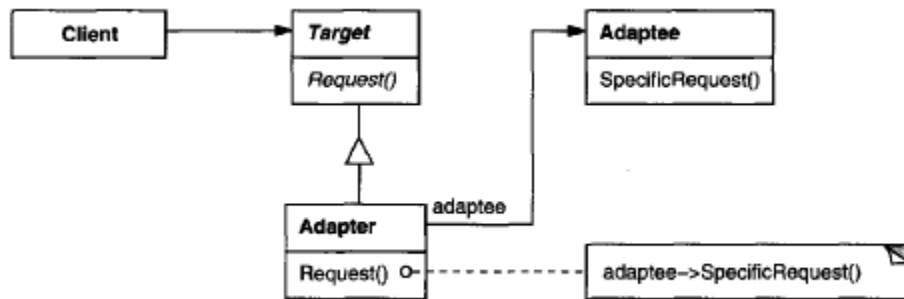
- You want to create a reusable class that cooperates with unrelated or unforeseen classes, that is, classes that don't necessarily have compatible interfaces.

Structure

A class adapter uses multiple inheritances to adapt one interface to another:



An object adapter relies on object composition:



Participants

- Target (Shape): defines the domain-specific interface that Client uses.
- Client (DrawingEditor): collaborates with objects conforming to the Target interface.
- Adaptec (TextView): Defines an existing interface that needs adapting.
- Adapter (TextShape): Adapts the interface of Adaptec to the Target interface.

Collaborations

- Clients call operations on an Adapter instance. In turn, the adapter calls Adaptec operations that carry out the request.

Consequences: Class and object adapters have different trade-offs. A class adapter

- adapts Adaptee to Target by committing to a concrete Adaptee class. As a consequence, a class adapter won't work when we want to adapt a class and all its subclasses.
- Lets Adapter override some of Adaptee's behavior, since Adapter is a subclass of Adaptee.

Implementation: Although the implementation of Adapter is usually straightforward, here are some issues to keep in mind:

- Implementing class adapters in C++. In a C++ implementation of a class adapter, Adapter would inherit publicly from Target and privately from Adaptec. Thus Adapter would be a subtype of Target but not of Adaptec.
- Pluggable adapters. Let's look at three ways to implement pluggable adapters for the TreeDisplay widget described earlier, which can lay out and display a hierarchical structure automatically.

Sample Code

We'll give a brief sketch of the implementation of class and object adapters for the Motivation example beginning with the classes Shape and TextView.

```
Class Shape {
public:
    Shape ();
    virtual void BoundingBox(
        Point& bottomLeft, Point& topRight
    ) const;
    virtual Manipulator* CreateManipulator () const;
};
class TextView {
public:
    TextView();
    void GetOrigin(Coord& x, Coord& y) const;
    void GetExtent(Coord& width, Coord& height) const;
    virtual bool IsEmpty() const;
};
```

Known Uses

The Motivation example comes from ET++Draw, a drawing application based on ET++.

Related Patterns

- Bridge has a structure similar to an object adapter, but Bridge has a different intent:
- Decorator enhances another object without changing its interface. A decorator is thus more transparent to the application than an adapter is.

2. Bridge

Intent

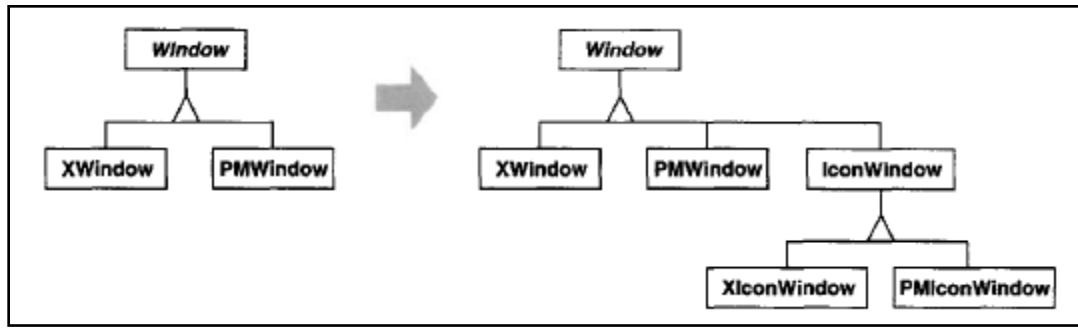
Decouple an abstraction from its implementation so that the two can vary independently.

Also Known As

Handle/Body

Motivation

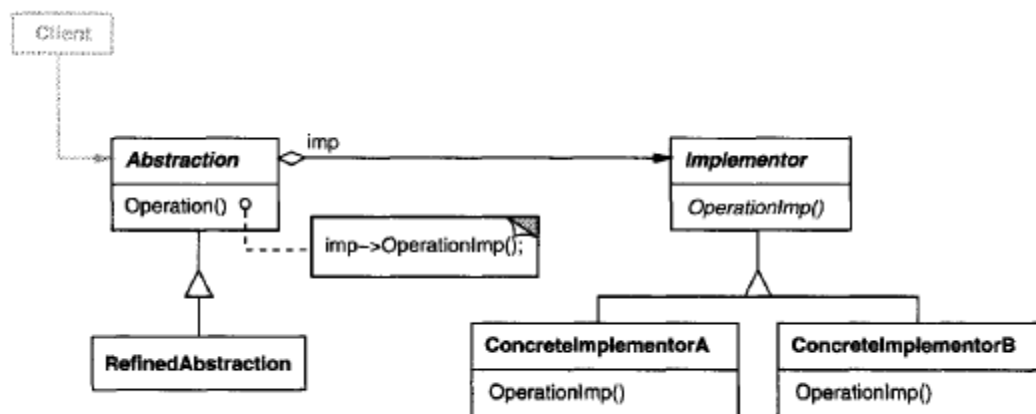
When an abstraction can have one of several possible implementations, the usual way to accommodate them is to use inheritance.



Applicability: Use the Bridge pattern when

- You want to avoid a permanent binding between an abstraction and its implementation. This might be the case, for example, when the implementation must be selected or switched at run-time.
- Both the abstractions and their implementations should be extensible by sub classing. In this case, the Bridge pattern lets you combine the different abstractions and implementations and extend them independently.
- Changes in the implementation of an abstraction should have no impact on clients; that is, their code should not have to be recompiled.

Structure



Participants

SOFTWARE ARCHITECTURE AND DESIGN PATTERNS

UNIT-IV

- Abstraction (Window): Defines the abstraction's interface. Maintains a reference to an object of type Implementor.
- RefinedAbstraction (IconWindow): Extends the interface defined by Abstraction.
- Implementor (WindowImp): Defines the interface for implementation classes. This interface doesn't have to correspond exactly to Abstraction's interface; in fact the two interfaces can be quite different. Typically the Implementor interface provides only primitive operations, and Abstraction defines higher-level operations based on these primitives.
- ConcreteImplementor (XWindowImp, PMWindowImp)
 - implements the Implementor interface and defines its concrete implementation.

Collaborations

- Abstraction forwards client requests to its Implementor object.

Consequences: The Bridge pattern has the following consequences:

1. Decoupling interface and implementation. An implementation is not bound permanently to an interface. The implementation of an abstraction can be configured at run-time. It's even possible for an object to change its implementation at run-time.
2. Improved extensibility. You can extend the Abstraction and Implementor hierarchies independently.

Sample Code: The following C++ code implements the Window/WindowImp example from the Motivation section. The Window class defines the window abstraction for client applications:

```
Class Window {
public:
    Window(View* contents); // requests handled by window
    virtual void DrawContents();
    virtual void Open();
    virtual void Close();
    virtual void Iconify();
    virtual void Deiconify();
    // requests forwarded to implementation
    virtual void SetOrigin(const Point& at);
    virtual void SetExtent(const Point& extent);
    virtual void Raise();
    virtual void Lower();
    virtual void DrawLine(const Point&, const Point&);
    virtual void DrawRect(const Point&, const Point&);
    virtual void DrawPolygon(const Point[], int n);
    virtual void DrawText(const char*, const Point&);
protected:
    WindowImp* GetWindowImp();
```

```
View* GetView();
private:
WindowImp* _imp;
View* _contents; // the window's contents
};
```

Related Patterns

- An Abstract Factory can create and configure a particular Bridge.
- The Adapter pattern is geared toward making unrelated classes work together.

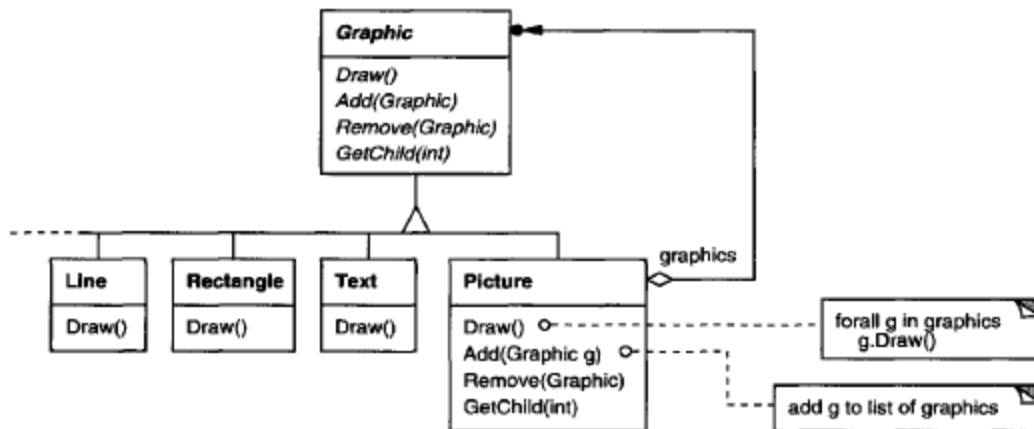
3. Composite

Intent

Compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly.

Motivation

Graphics applications like drawing editors and schematic capture systems let users build complex diagrams out of simple components.



Applicability

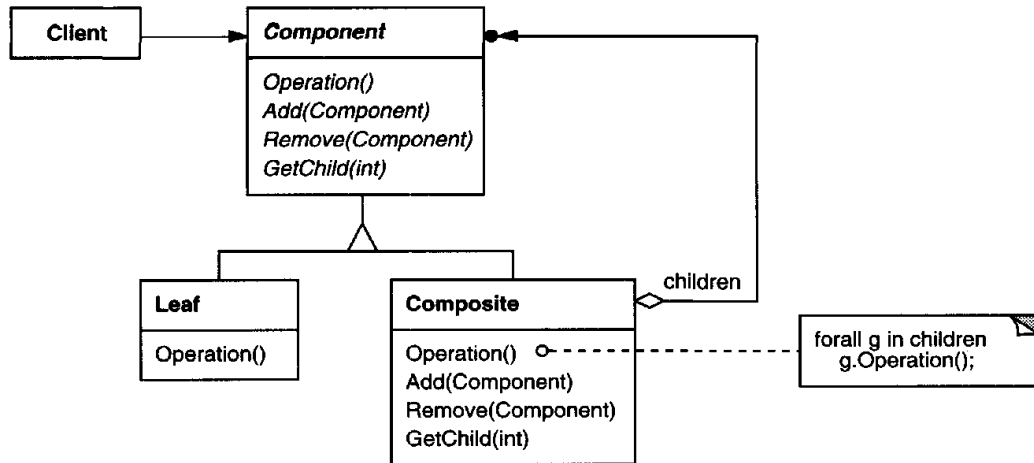
Use the Composite pattern when

- You want to represent part-whole hierarchies of objects
- You want clients to be able to ignore the difference between compositions of objects and individual objects. Clients will treat all objects in the composite structure uniformly.

Structure

SOFTWARE ARCHITECTURE AND DESIGN PATTERNS

UNIT-IV



Participants

- Component (Graphic)
- Leaf (Rectangle, Line)
- Composite (Picture)
- Client

Collaborations

- Clients use the Component class interface to interact with objects in the composite structure. If the recipient is a Leaf, then the request is handled directly.
- If the recipient is a Composite, then it usually forwards requests to its child components, possibly performing additional operations before and/or after forwarding.

Consequences: The Composite pattern

- Defines class hierarchies consisting of primitive objects and composite objects. Primitive objects can be composed into more complex objects, which in turn can be composed, and so on recursively.
- Makes the client simple. Clients can treat composite structures and individual objects uniformly. Clients normally don't know (and shouldn't care) whether they're dealing with a leaf or a composite component.

Implementation: There are many issues to consider when implementing the Composite pattern:

1. Explicit parent references. Maintaining references from child components to their parent can simplify the traversal and management of a composite structure.
2. Sharing components. It's often useful to share components, for example, to reduce storage requirements.

Sample Code

Equipment class defines an interface for all equipment in the part-whole hierarchy.

```
Class Equipment
{
public:
    Virtual "Equipment ()";
    Const char* Name () {return _name; }
```

SOFTWARE ARCHITECTURE AND DESIGN PATTERNS

UNIT-IV

```
Virtual Watt Power ();
Virtual Currency NetPrice ();
Virtual Currency DiscountPrice ();
Virtual void Add (Equipment*);
Virtual void Remove (Equipment*);
Virtual Iterator<Equipment*>* CreateIterator ();
protected:
    Equipment (const char*);
private:
    Const char* _name;
};
```

Related Patterns

- Often the component-parent link is used for a Chain of Responsibility
- Flyweight lets you share components, but they can no longer refer to their parents
- Iterator can be used to traverse composites.

4. Decorator

Intent

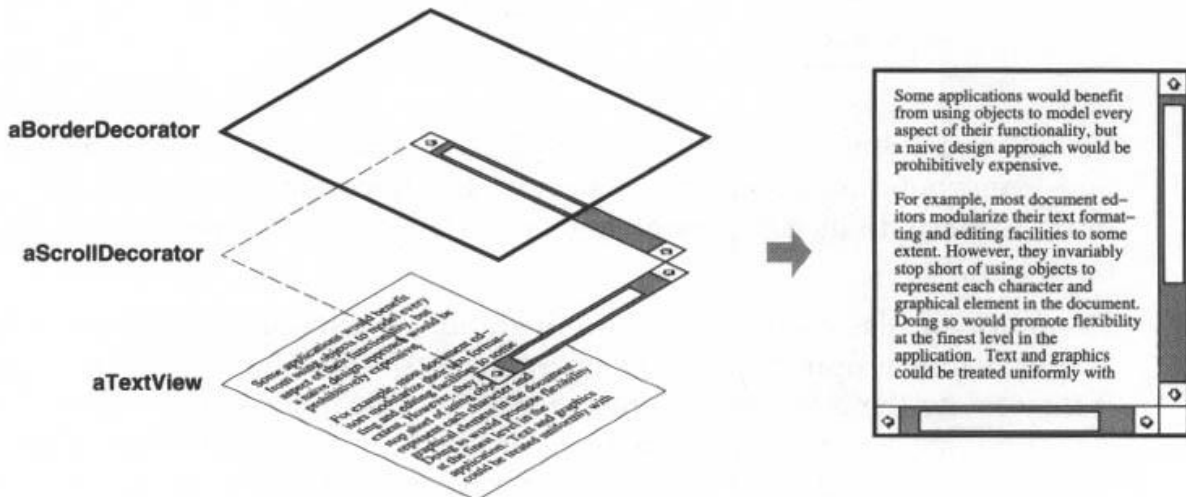
Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to sub classing for extending functionality.

Also Known As

Wrapper

Motivation

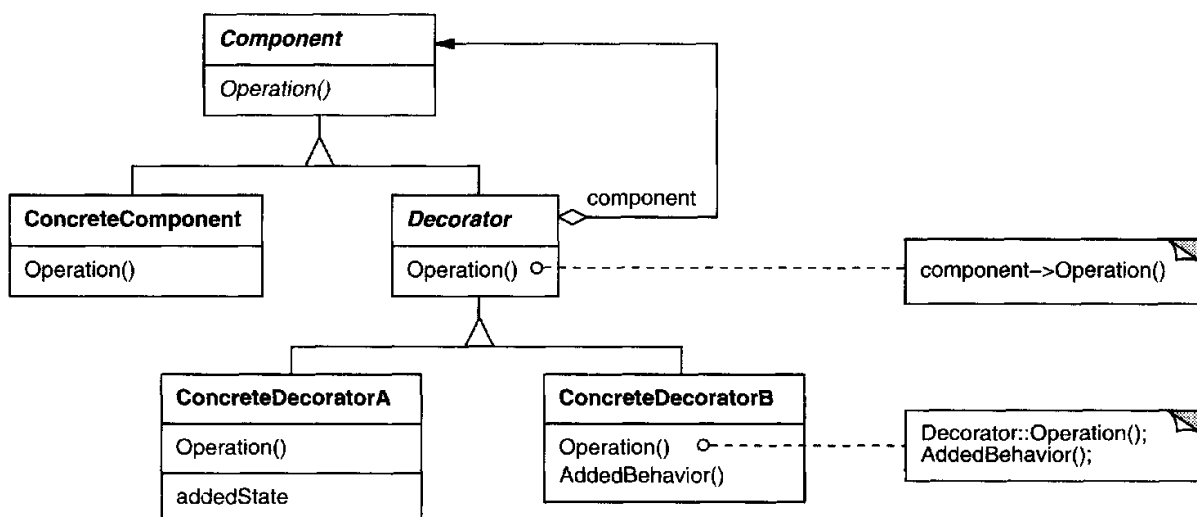
Sometimes we want to add responsibilities to individual objects, not to an entire class. A graphical user interface toolkit, for example, should let you add properties like borders or behaviors like scrolling to any user interface component.



Applicability: Use Decorator

- To add responsibilities to individual objects dynamically and transparently, that is, without affecting other objects.
- For responsibilities that can be withdrawn.

Structure



Participants

- Component
- ConcreteComponent
- Decorator
- ConcreteDecorator

Collaborations

- Decorator forwards requests to its Component object. It may optionally perform additional operations before and after forwarding the request.

Consequences: The Decorator pattern has at least two key benefits and two liabilities:

1. More flexibility than static inheritance.
2. 2. Avoids feature-laden classes high up in the hierarchy.

Implementation: Several issues should be considered when applying the Decorator pattern

1. Interface conformance. A decorator object's interface must conform to the interface of the component it decorates.
2. Omitting the abstract Decorator class

Sample Code

Class VisualComponent

```
{  
public:  
    VisualComponent ();  
    Virtual void Draw ();  
    Virtual void Resize ();  
};
```

Class Decorator: public VisualComponent

```
{  
public:  
    Decorator (VisualComponent*);  
    Virtual void Draw ();  
    Virtual void Resize ();  
private:  
    VisualComponent* _component;  
};
```

Related Patterns

- Adapter a decorator is different from an adapter in that a decorator only changes an object's responsibilities, not its interface; an adapter will give an object a completely new interface.
- Composite: A decorator can be viewed as a degenerate composite with only one component.

5. Façade

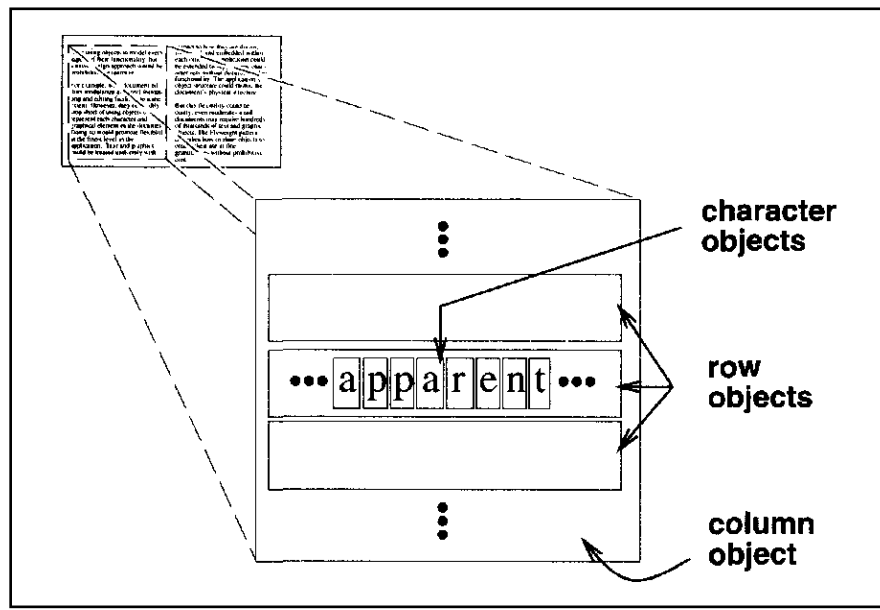
6. Flyweight

Intent

Use sharing to support large numbers of fine-grained objects efficiently.

Motivation

Some applications could benefit from using objects throughout their design, but a naive implementation would be prohibitively expensive. For example, most document editor implementations have text formatting and editing facilities that are modularized to some extent. Object-oriented document editors typically use objects to represent embedded elements like tables and figures.



A **flyweight** is a shared object that can be used in multiple contexts simultaneously. The flyweight acts as an independent object in each context—it's indistinguishable from an instance of the object that's not shared. Flyweights cannot make assumptions about the context in which they operate. The key concept here is the distinction between **intrinsic** and **extrinsic** state. Intrinsic state is stored in the flyweight; it consists of information that's independent of the flyweight's context, thereby making it sharable. Extrinsic state depends on and varies with the flyweight's context and therefore can't be shared. Client objects are responsible for passing extrinsic state to the flyweight when it needs it.

Applicability

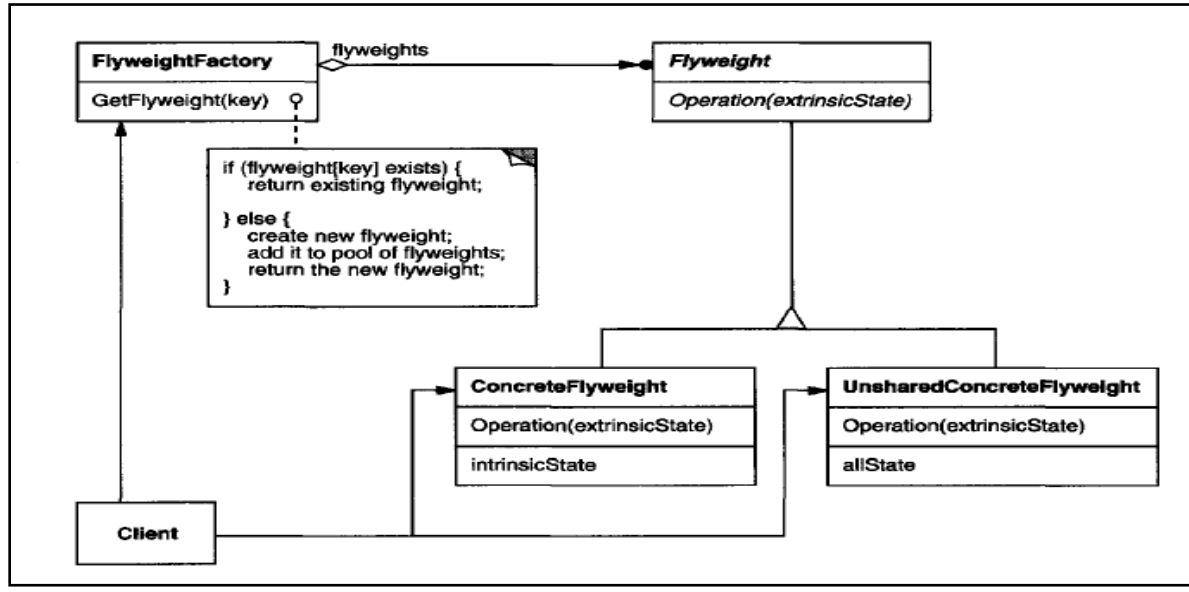
The Flyweight pattern's effectiveness depends heavily on how and where it's used. Apply the Flyweight pattern when *all* of the following are true:

- An application uses a large number of objects.
- Storage costs are high because of the sheer quantity of objects
- Most object state can be made extrinsic.
- Many groups of objects may be replaced by relatively few shared objects once extrinsic state is removed.

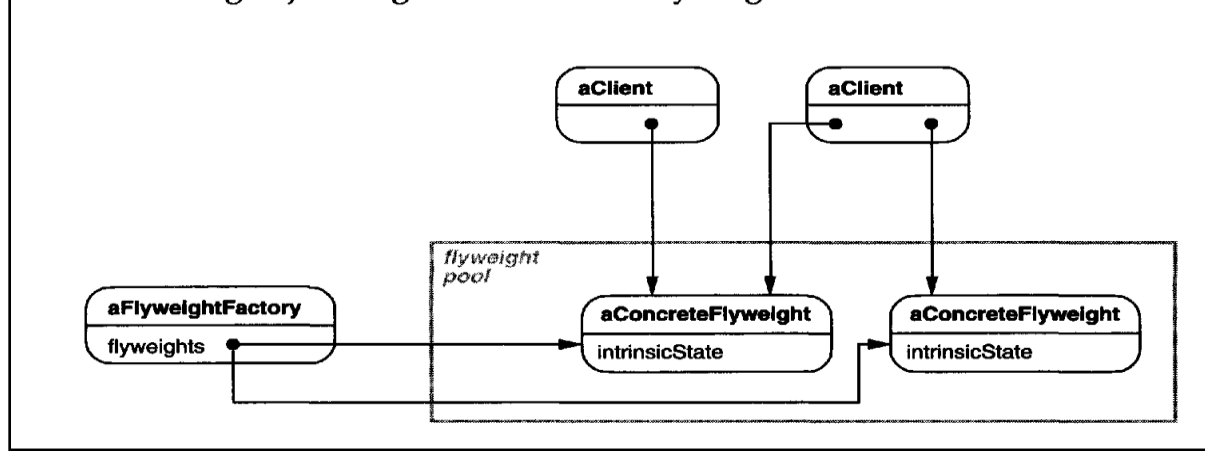
SOFTWARE ARCHITECTURE AND DESIGN PATTERNS

UNIT-IV

Structure



The following object diagram shows how flyweights are shared:



Participants

- Flyweight, ConcreteFlyweight , UnsharedConcreteFlyweight, FlyweightFactory, Client

Collaborations

- State that a flyweight needs to function must be characterized as either intrinsic or extrinsic. Intrinsic state is stored in the ConcreteFlyweight object; extrinsic state is stored or computed by Client objects. Clients pass this state to the flyweight when they invoke its operations.
- Clients should not instantiate ConcreteFlyweights directly. Clients must obtain ConcreteFlyweight objects exclusively from the FlyweightFactory object to ensure they are shared properly.

Consequences

- The reduction in the total number of instances that comes from sharing
- The amount of intrinsic state per object
- Whether extrinsic state is computed or stored

Implementation: Consider the following issues when implementing the Flyweight pattern:

- Removing extrinsic state
- Managing shared objects

Sample Code

Returning to our document formatter example, we can define a Glyph base class for flyweight graphical objects. Logically, glyphs are Composites that have graphic al attributes and ca n draw themselves. Here we focus on just the font attribute, but the same approach can be used for any other graphical attributes a glyph might have.

```
class Glyph {
public:
    virtual ~Glyph();

    virtual void Draw(Window*, GlyphContext&);

    virtual void SetFont(Font*, GlyphContext&);
    virtual Font* GetFont (GlyphContext&);

    virtual void First(GlyphContext&);
    virtual void Next (GlyphContext&);
    virtual bool IsDone(GlyphContext&);
    virtual Glyph* Current(GlyphContext&);

    virtual void Insert(Glyph*, GlyphContext&);
    virtual void Remove(GlyphContext&);
protected:
    Glyph();
};
```

Related Patterns

- The Flyweight pattern is often combined with the Composite pattern to implement a logically hierarchical structure in terms of a directed-acyclic graph with shared leaf nodes
- It's often best to implement State and Strategy objects as flyweights

7. PROXY

Intent

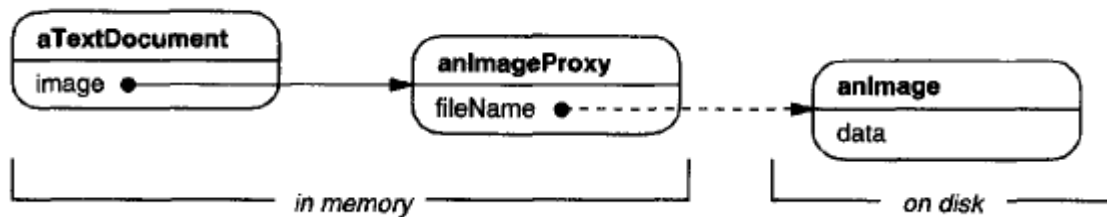
Provide a surrogate or placeholder for another object to control access to it.

Also Known As

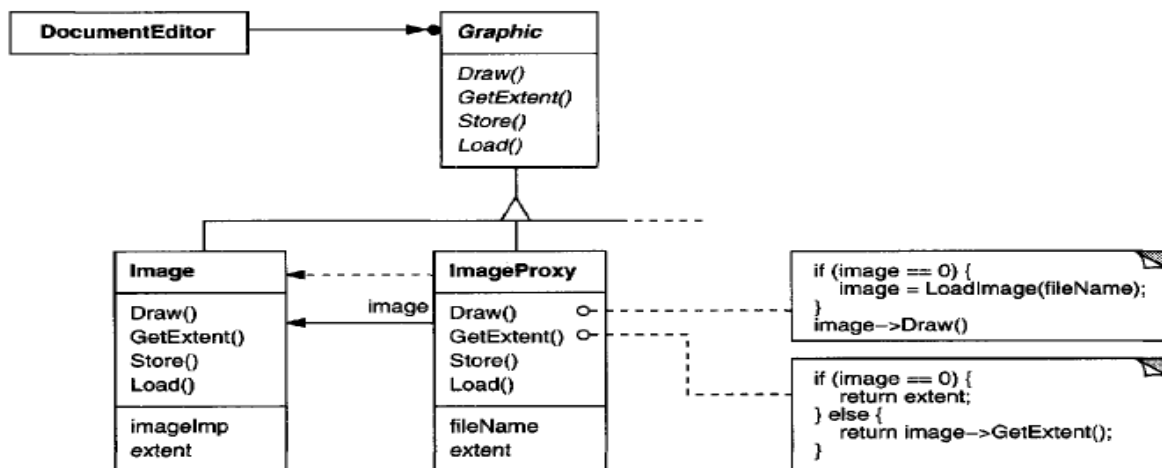
Surrogate

Motivation

One reason for controlling access to an object is to defer the full cost of its creation and initialization until we actually need to use it. Consider a document editor that can embed graphical objects in a document. Some graphical objects, like large raster images, can be expensive to create. But opening a document should be fast, so we should avoid creating all the expensive objects at once when the document is opened. This isn't necessary anyway, because not all of these objects will be visible in the document at the same time.



The following class diagram illustrates this example in more detail.

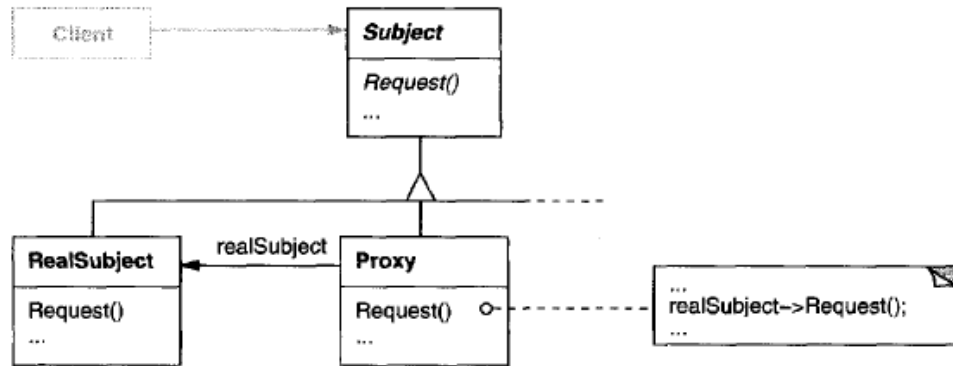


Applicability

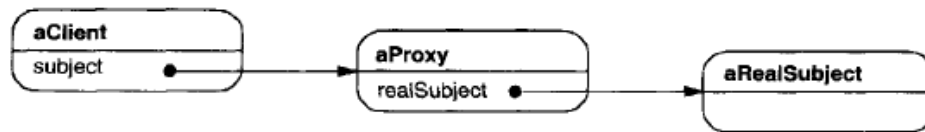
Proxy is applicable whenever there is a need for a more versatile or sophisticated reference to an object than a simple pointer. Here are several common situations in which the Proxy pattern is applicable:

- A remote proxy provides a local representative for an object in a different address space.
- A virtual proxy creates expensive objects on demand. The ImageProxy described in the Motivation is an example of such a proxy.
- A protection proxy controls access to the original object. Protection proxies are useful when objects should have different access rights

Structure



Here's a possible object diagram of a proxy structure at run-time:



Participants

- Proxy, Subject, RealSubject

Collaborations

- Proxy forwards requests to RealSubject when appropriate, depending on the kind of proxy

Consequences: The Proxy pattern introduces a level of indirection when accessing an object. The additional indirection has many uses, depending on the kind of proxy:

- A remote proxy can hide the fact that an object resides in a different address space
- A virtual proxy can perform optimizations such as creating an object on demand
- Both protection proxies and smart references allow additional housekeeping tasks when an object is accessed

Implementation: The Proxy pattern can exploit the following language features:

- *Overloading the member access operator in C++.* C++ supports overloading operator->, the member access operator. Overloading this operator lets you perform additional work whenever an object is dereferenced.

Sample Code: The following code implements two kinds of proxy: the virtual proxy described in the Motivation section, and a proxy implemented with does Not Understand

1. *A virtual proxy.* The Graphic class defines the interface for graphical objects:

```
class Graphic {
public:
    virtual ~Graphic();

    virtual void Draw(const Point& at) = 0;
    virtual void HandleMouse(Event& event) = 0;

    virtual const Point& GetExtent() = 0;

    virtual void Load(istream& from) = 0;
    virtual void Save(ostream& to) = 0;
protected:
    Graphic();
};
```

Related Patterns

- Adapter: An adapter provides a different interface to the object it adapts
- Decorator: Although decorators can have similar implementations as proxies, decorators have a different purpose. A decorator adds one or more responsibilities to an object, whereas a proxy controls access to an object

Frequently Asked Question

1. Write a short note on Adapter design pattern?
2. Briefly discuss about Bridge design pattern?
3. Explain the working of Composite design pattern?
4. Explain in detail about Decorator design pattern?
5. Describe the working of Façade design pattern?
6. Write about Flyweight design pattern?
7. Briefly mention about the working of PROXY design pattern?

UNIT-V:

Behavioral Patterns: Chain of responsibility, command, Interpreter, iterator, mediator, memento, observer, state strategy, template method, visitor.

Behavioral patterns are concerned with algorithms and the assignment of responsibilities between objects. Behavioral patterns describe not just patterns of objects or classes but also the patterns of communication between them. These patterns characterize complex control flow that's difficult to follow at run-time.

1. Chain of responsibility

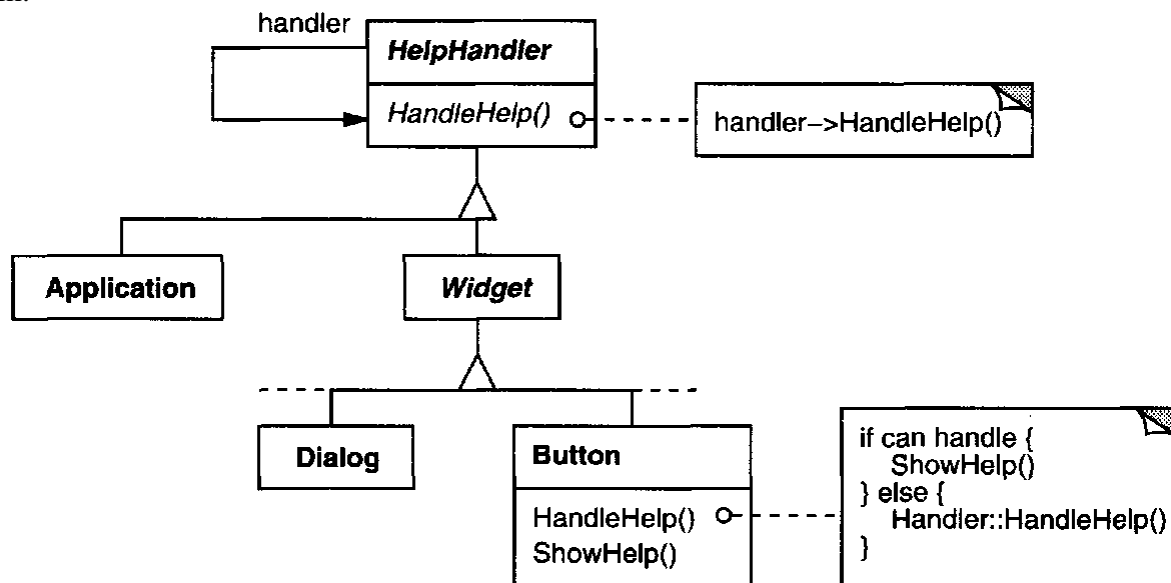
Intent

Avoid coupling the sender of a request to its receiver by giving more than one object a chance to handle the request. Chain the receiving objects and pass the request along the chain until an object handles it.

Motivation

Consider a context-sensitive help facility for a graphical user interface. The user can obtain help information on any part of the interface just by clicking on it. The help that's provided depends on the part of the interface that's selected and its context; **for example**, a button widget in a dialog box might have different help information than a similar button in the main window.

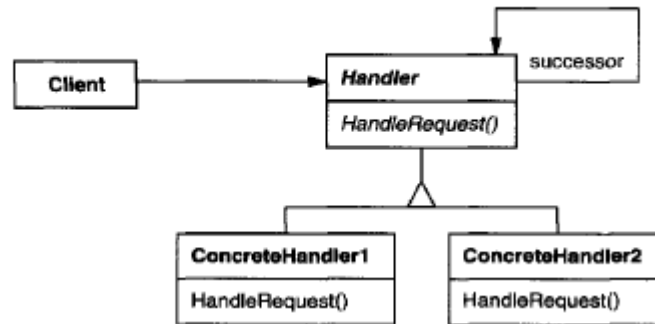
To forward the request along the chain, and to ensure receivers remain implicit, each object on the chain shares a common interface for handling requests and for accessing its successor on the chain.



Applicability: Use Chain of Responsibility when

- More than one object may handle a request, and the handler isn't known a priori. The handler should be ascertained automatically.
- You want to issue a request to one of several objects without specifying the receiver explicitly.
- The set of objects that can handle a request should be specified dynamically.

Structure



A typical object structure might look like this:



Participants

- Handler
- ConcreteHandler
- Client

Collaborations

- When a client issues a request, the request propagates along the chain until a ConcreteHandler object takes responsibility for handling it.

Consequences: Chain of Responsibility has the following benefits and liabilities:

- Reduced coupling
- Added flexibility in assigning responsibilities to objects
- Receipt isn't guaranteed

Implementation: Here are implementation issues to consider in Chain of Responsibility:

1. Implementing the successor chain. There are two possible ways to implement the successor chain:

- (a) Define new links (usually in the Handler, but ConcreteHandlers could define them instead).
- (b) Use existing links.

SOFTWARE ARCHITECTURE AND DESIGN PATTERNS

UNIT-V

```
class HelpHandler {
public:
    HelpHandler(HelpHandler* s) : _successor(s) { }
    virtual void HandleHelp();
private:
    HelpHandler* _successor;
};

void HelpHandler::HandleHelp () {
    if (_successor) {
        _successor->HandleHelp();
    }
}
```

Sample Code

The following example illustrates how a chain of responsibility can handle requests for an on-line help system like the one described earlier. The help request is an explicit operation.

```
typedef int Topic;
const Topic NO_HELP_TOPIC = -1;

class HelpHandler {
public:
    HelpHandler(HelpHandler* = 0, Topic = NO_HELP_TOPIC);
    virtual bool HasHelp();
    virtual void SetHandler(HelpHandler*, Topic);
    virtual void HandleHelp();
private:
    HelpHandler* _successor;
    Topic _topic;
};

HelpHandler::HelpHandler (
    HelpHandler* h, Topic t
) : _successor(h), _topic(t) { }

bool HelpHandler::HasHelp () {
    return _topic != NO_HELP_TOPIC;
}
```

Related Patterns

- Chain of Responsibility is often applied in conjunction with Composite.
- There, a component's parent can act as its successor.

2. COMMAND

Intent

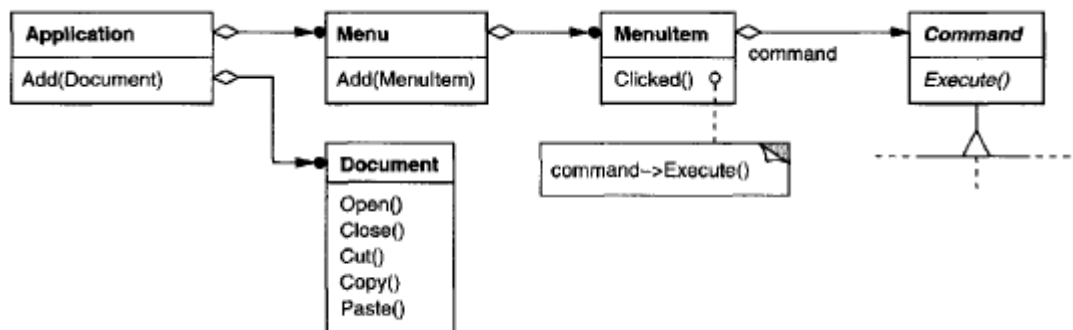
Encapsulate a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operations.

Also Known As

Action, Transaction

Motivation

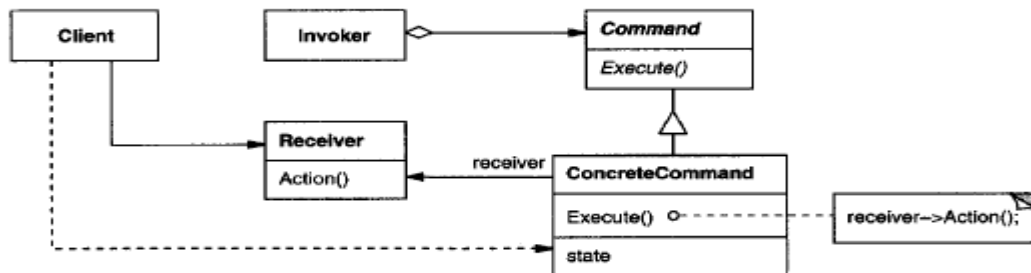
Sometimes it's necessary to issue requests to objects without knowing anything about the operation being requested or the receiver of the request. For example, user interface toolkits include objects like buttons and menus that carry out a request in response to user input.



Applicability: Use the Command pattern when you want to

- Parameterize objects by an action to perform, as Menu item objects did above. You can express such parameterization in a procedural language with a **callback** function,
- Specify, queue, and execute requests at different times. A Command object can have a lifetime independent of the original request.
- Support undo. The Command's Execute operation can store state for reversing its effects in the command itself.

Structure



Participants

- Command, ConcreteCommand, Client, Invoker, Receiver

Collaborations

- The client creates a ConcreteCommand object and specifies its receiver.
- An Invoker object stores the ConcreteCommand object.

Consequences: The Command pattern has the following consequences:

- Command decouples the object that invokes the operation from the one that knows how to perform it.
- Commands are first-class objects. They can be manipulated and extended like any other object.

Implementation: Consider the following issues when implementing the Command pattern:

- How intelligent should a command be? A command can have a wide range of abilities. At one extreme it merely defines a binding between a receiver and the actions that carry out the request.
- Supporting undo and redo. Commands can support undo and redo capabilities if they provide a way to reverse their execution

Sample Code

We'll define OpenCommand, PasteCommand, and MacroCommand. First the abstract Command class:

```
class Command {
public:
    virtual ~Command();

    virtual void Execute() = 0;
protected:
    Command();
};

class OpenCommand : public Command {
public:
    OpenCommand(Application*);

    virtual void Execute();
protected:
    virtual const char* AskUser();
private:
    Application* _application;
    char* _response;
};

OpenCommand::OpenCommand (Application* a) {
    _application = a;
}
```

Related Patterns

- Composite can be used to implement MacroCommands
- Memento can keep state the command requires to undo its effect
- command that must be copied before being placed on the history list acts as a Prototype

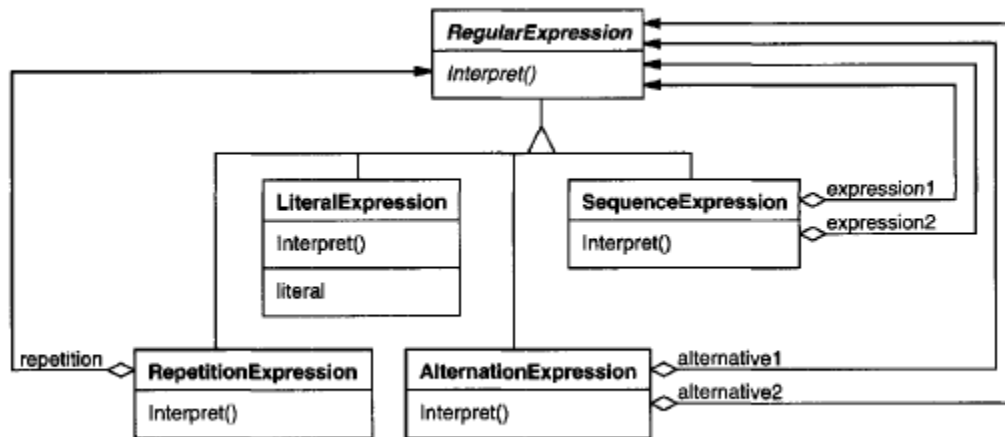
3. INTERPRETER

Intent

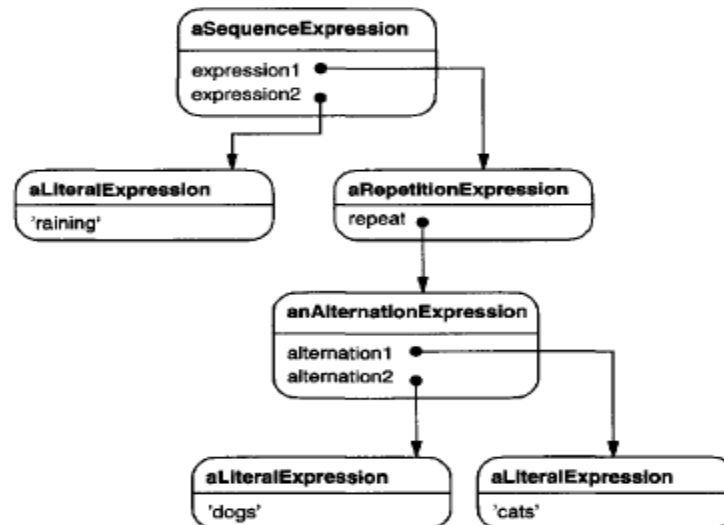
Given a language, define a representation of or its grammar along with an interpreter that uses the representation to interpret sentences in the language.

Motivation

If a particular kind of problem occurs often enough, then it might be worthwhile to express instances of the problem as sentences in a simple language. Then you can build an interpreter that solves the problem by interpreting these sentences.

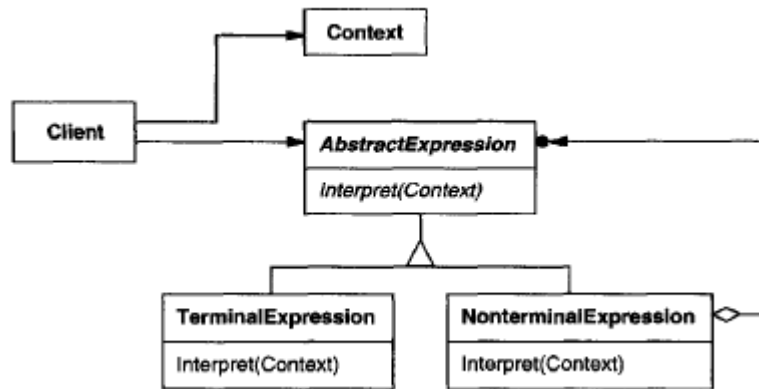


Every regular expression defined by this grammar is represented by an abstract syntax tree made up of instances of these classes. For example, the abstract syntax tree



Applicability: Use the Interpreter pattern when there is a language to interpret, and you can represent statements in the language as abstract syntax trees. The Interpreter pattern works best when the grammar is simple and efficiency is not a critical concern.

Structure



Participants

- AbstractExpression, TerminalExpression, NonterminalExpression, Context, Client

Collaborations

- The client builds (or is given) the sentence as an abstract syntax tree of NonterminalExpression and TerminalExpression instances. Then the client initializes the context and invokes the Interpret operation.
- Each NonterminalExpression node defines Interpret in terms of Interpret on each subexpression. The Interpret operation of each TerminalExpression defines the base case in the recursion.

Consequences: The Interpreter pattern has the following benefits and liabilities:

- It's easy to change and extend the grammar
- Implementing the grammar is easy, too
- Complex grammars are hard to maintain
- Adding new ways to interpret expressions

Implementation: The Interpreter and Composite patterns share many implementation issues. The following issues are specific to Interpreter:

- Creating the abstract syntax tree
- Defining the Interpret operation
- Sharing terminal symbols with the Flyweight pattern

Sample Code

Here are two examples. The first is a complete example in Smalltalk for checking whether a sequence matches a regular expression. The second is a C++ program for evaluating Boolean expressions. The regular expression matcher tests whether a string is in the language defined by the regular expression. The regular expression is defined by the following grammar:

SOFTWARE ARCHITECTURE AND DESIGN PATTERNS

UNIT-V

```
expression ::= literal | alternation | sequence | repetition |  
            '(' expression ')'  
alternation ::= expression '|' expression  
sequence ::= expression '&' expression  
repetition ::= expression 'repeat'  
literal ::= 'a' | 'b' | 'c' | ... { 'a' | 'b' | 'c' | ... }*
```

Related Patterns

- Composite :The abstract syntax tree is an instance of the Composite pattern
- Flyweight shows how to share terminal symbols within the abstract syntax tree
- Iterator: The interpreter can use an Iterator to traverse the structure
- Visitor can be used to maintain the behavior in each node in the abstract syntax tree in one class

4. ITERATOR

Intent

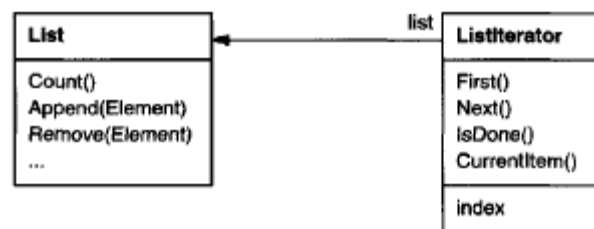
Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation.

Also Known As

Cursor

Motivation

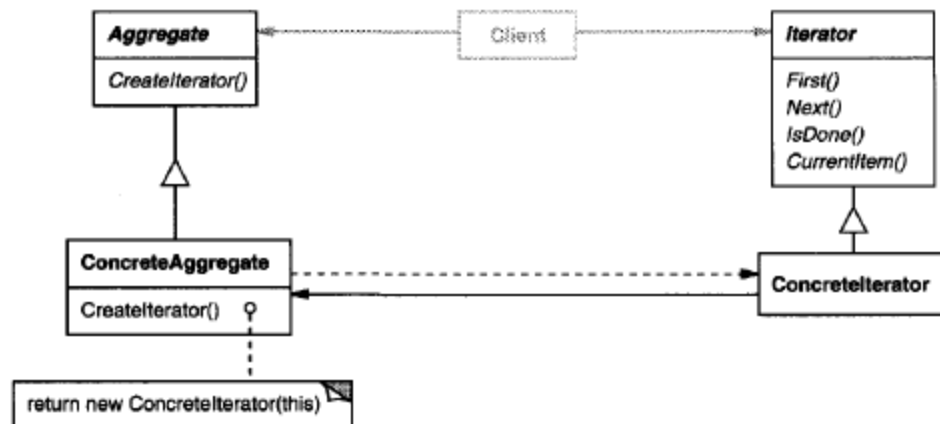
An aggregate object such as a list should give you a way to access its elements without exposing its internal structure. For example, a List class would call for a ListIterator with the following relationship between them:



Applicability: Use the Iterator pattern

- To access an aggregate object's contents without exposing its internal representation.
- To support multiple traversals of aggregate objects.
- To provide a uniform interface for traversing different aggregate structures

Structure



Participants

- Iterator, ConcreteIterator, Aggregate, ConcreteAggregate

Collaborations

- A **ConcreteIterator** keeps track of the current object in the aggregate and can compute the succeeding object in the traversal.

Consequences: The Iterator pattern has three important consequences:

- It supports variations in the traversal of an aggregate
- Iterators simplify the Aggregate interface.
- More than one traversal can be pending on an aggregate

Implementation

Iterator has many implementation variants and alternatives. Some important ones follow. The trade-offs often depend on the control structures your language provides.

Sample Code: We'll look at the implementation of a simple List class, which is part of our foundation library.

```
template <class Item>
class List {
public:
    List(long size = DEFAULT_LIST_CAPACITY);

    long Count() const;
    Item& Get(long index) const;
    // ...
};

template <class Item>
class Iterator {
public:
    virtual void First() = 0;
    virtual void Next() = 0;
    virtual bool IsDone() const = 0;
    virtual Item CurrentItem() const = 0;
protected:
    Iterator();
};
```

Related Patterns

- Composite: Iterators are often applied to recursive structures such as Composites
- Factory Method: Polymorphic iterators rely on factory methods to instantiate the appropriate Iterator subclass.

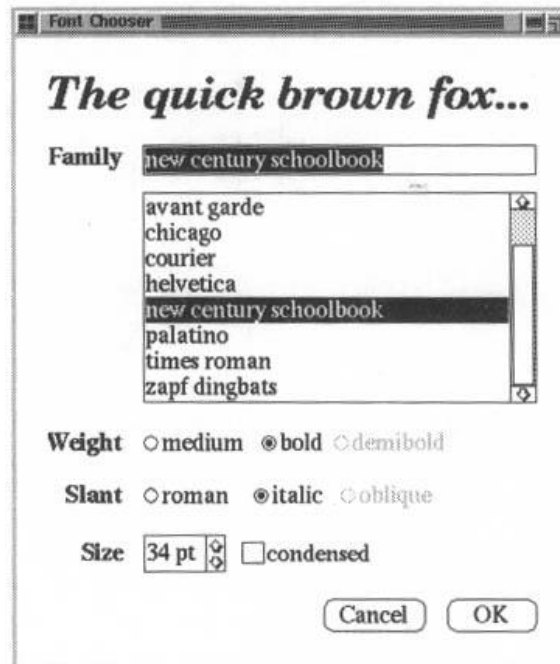
5. MEDIATOR

Intent

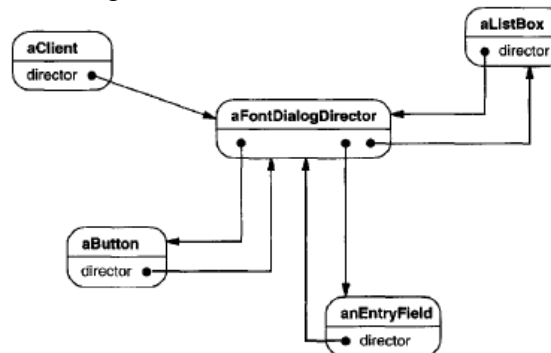
Define an object that encapsulates how a set of objects interact. Mediator promotes loose coupling by keeping objects from referring to each other explicitly, and it lets you vary their interaction independently.

Motivation

Object-oriented design encourages the distribution of behavior among objects. Such distribution can result in an object structure with many connections between objects; in the worst case, every object ends up knowing about every other.



For example, FontDialogDirector can be the mediator between the widgets in a dialog box. A FontDialogDirector object knows the widgets in a dialog and coordinates their interaction. It acts as a hub of communication for widgets:



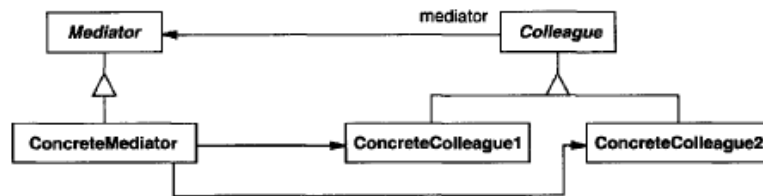
SOFTWARE ARCHITECTURE AND DESIGN PATTERNS

UNIT-V

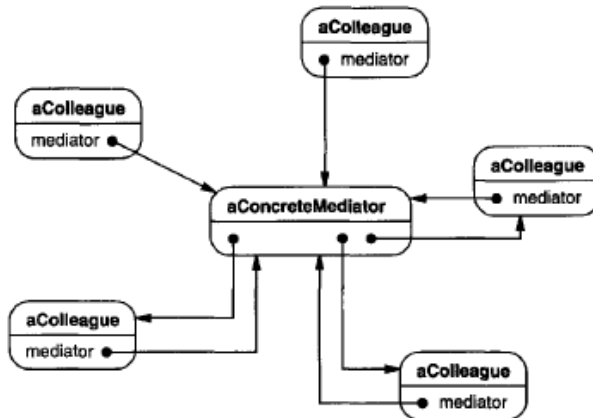
Applicability: Use the Mediator pattern when

- A set of objects communicate in well-defined but complex ways. The resulting interdependencies are unstructured and difficult to understand.
- Reusing an object is difficult because it refers to and communicates with many other objects.
- A behavior that's distributed between several classes should be customizable without a lot of sub classing.

Structure



A typical object structure might look like this:



Participants

- Mediator, ConcreteMediator, Colleague classes

Collaborations

- Colleagues send and receive requests from a Mediator object. The mediator implements the cooperative behavior by routing requests between the appropriate colleague

Consequences: The Mediator pattern has the following benefits and drawbacks:

- It limits subclassing
- It decouples colleagues
- It simplifies object protocols

Implementation: The following implementation issues are relevant to the Mediator pattern:

- Omitting the abstract Mediator class. There's no need to define an abstract Mediator class when colleagues work with only one mediator. The abstract coupling that the Mediator class provides lets colleagues work with different Mediator subclasses, and vice versa.

Sample Code

We'll use a DialogDirector to implement the font dialog box shown in the Motivation. The abstract class DialogDirector defines the interface for directors.

```
class DialogDirector {
public:
    virtual ~DialogDirector();

    virtual void ShowDialog();
    virtual void WidgetChanged(Widget*) = 0;

protected:
    DialogDirector();
    virtual void CreateWidgets() = 0;
};
```

Widget is the abstract base class for widgets. A widget knows its director.

```
class Widget {
public:
    Widget(DialogDirector*);
    virtual void Changed();

    virtual void HandleMouse(MouseEvent& event);
    // ...
private:
    DialogDirector* _director;
};
```

Related Patterns

- Facade differs from Mediator in that it abstracts a subsystem of objects to provide a more convenient interface
- Colleagues can communicate with the mediator using the Observer (293) pattern

6. MEMENTO

Intent

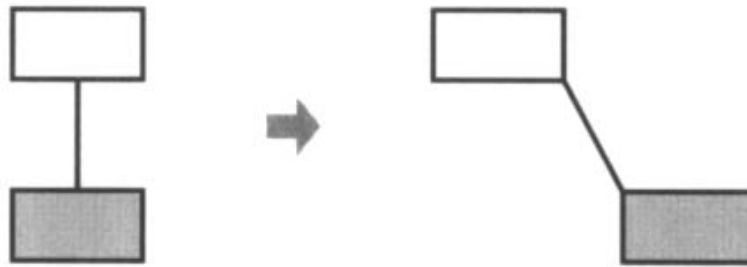
Without violating encapsulation, capture and externalize an object's internal state so that the object can be restored to this state later.

Also Known As

Token

Motivation

Sometimes it's necessary to record the internal state of an object. This is required when implementing checkpoints and undo mechanisms that let users back out of tentative operations or recover from errors.

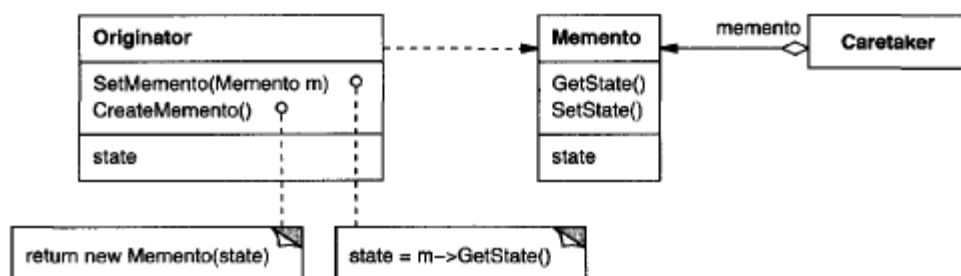


A well-known way to maintain connectivity relationships between objects are with a constraint-solving system. We can encapsulate this functionality in a Constraint- Solver object.

Applicability: Use the Memento pattern when

- A snapshot of (some portion of) an object's state must be saved so that it can be restored to that state later, and
- A direct interface to obtaining the state would expose implementation details and break the object's encapsulation

Structure

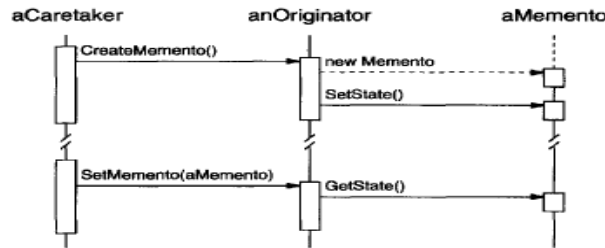


Participants

- Memento, Originator, Caretaker

Collaborations

- A caretaker requests a memento from an originator, holds it for a time, and passes its back to the originator, as the following interaction diagram illustrates:



Consequences: The Memento pattern has several consequences:

- Preserving encapsulation boundaries
- It simplifies Originator
- Using mementos might be expensive

Implementation: Here are two issues to consider when implementing the Memento pattern:

- Language support. Mementos have two interfaces: a wide one for originators and a narrow one for other objects.

```
class State;

class Originator {
public:
    Memento* CreateMemento();
    void SetMemento(const Memento*);
    // ...
private:
    State* _state;      // internal data structures
    // ...
};

class Memento {
public:
    // narrow public interface
    virtual ~Memento();
private:
    // private members accessible only to Originator
    friend class Originator;
    Memento();

    void SetState(State*);
    State* GetState();
    // ...
private:
    State* _state;
    // ...
};
```

Related Patterns

- Command: Commands can use mementos to maintain state for undoable operations
- Iterator: Mementos can be used for iteration as described earlier

7. OBSERVER

Intent

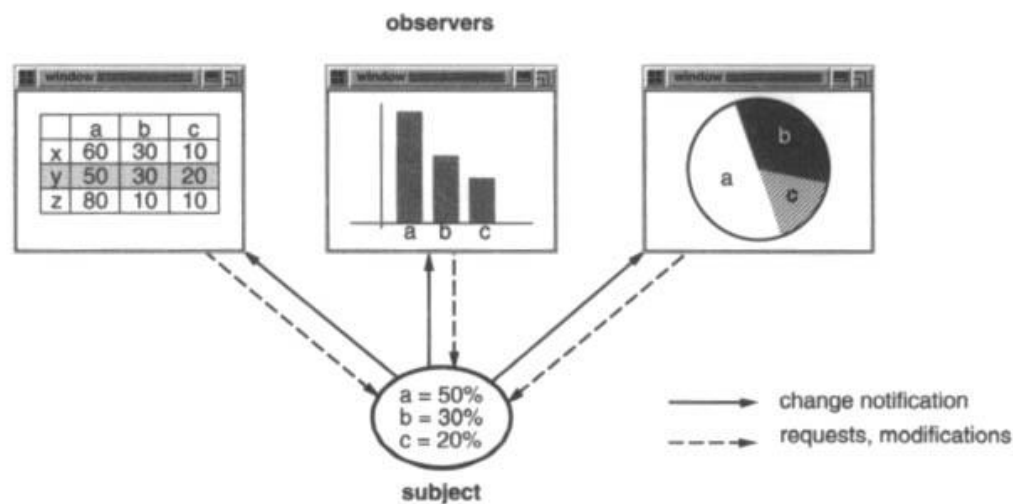
Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.

Also Known As

Dependents, Publish-Subscribe

Motivation

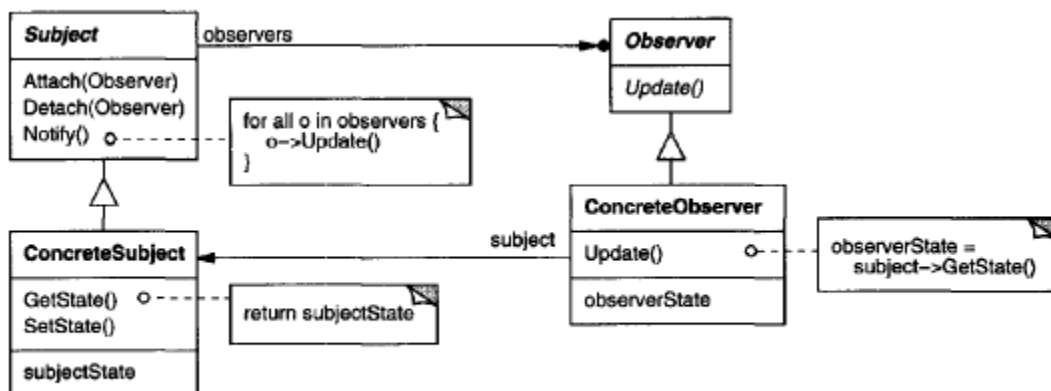
A common side-effect of partitioning a system into a collection of cooperating classes is the need to maintain consistency between related objects. You don't want to achieve consistency by making the classes tightly coupled, because that reduces their reusability.



Applicability: Use the Observer pattern in any of the following situations:

- When an abstraction has two aspects, one dependent on the other. Encapsulating these aspects in separate objects lets you vary and reuse them independently
- When a change to one object requires changing others, and you don't know how many objects need to be changed

Structure

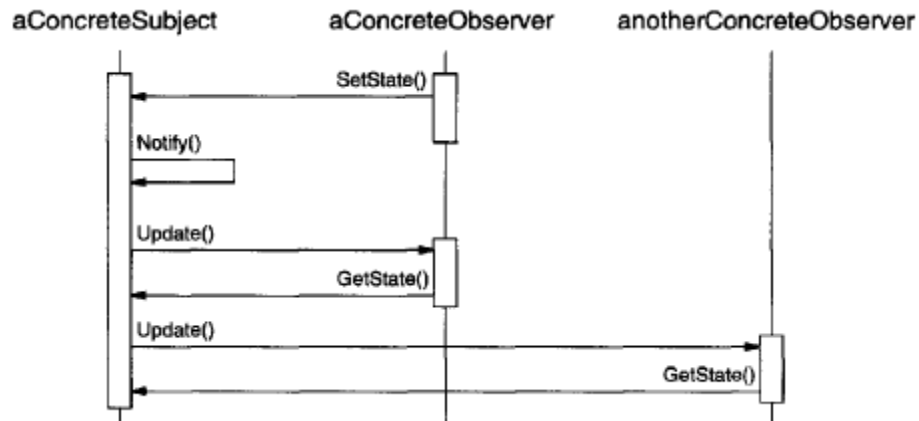


Participants

- Subject, Observer, ConcreteSubject, ConcreteObserver

Collaborations

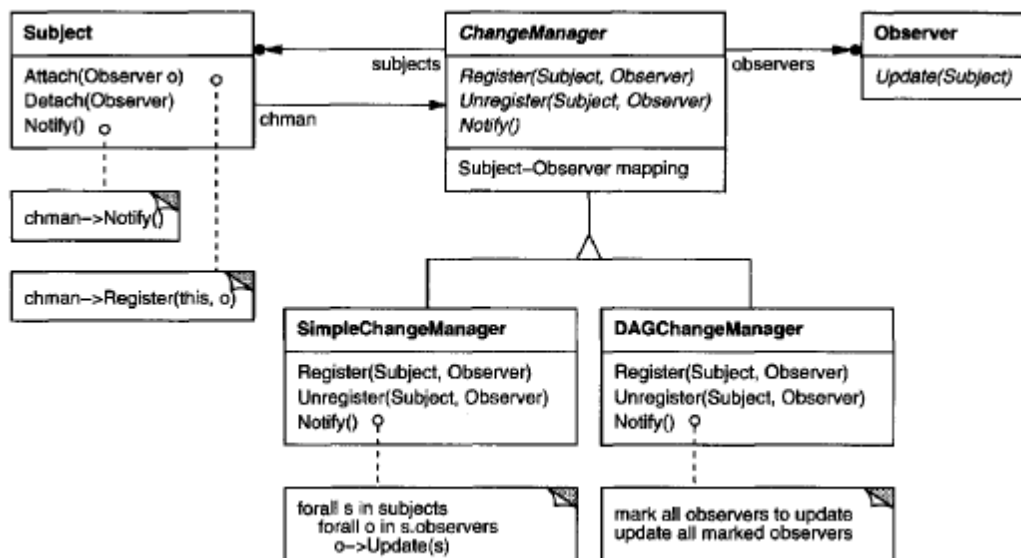
- ConcreteSubject notifies its observers whenever a change occurs that could make its observers' state inconsistent with its own.
- After being informed of a change in the concrete subject, a ConcreteObserver object may query the subject for information



Consequences: The Observer pattern lets you vary subjects and observers independently

- Abstract coupling between Subject and Observer
- Support for broadcast communication
- Unexpected updates

Implementation: Several issues related to the implementation of the dependency mechanism are discussed in this section.



Sample Code: An abstract class defines the Observer interface

```
class Subject;

class Observer {
public:
    virtual ~Observer();
    virtual void Update(Subject* theChangedSubject) = 0;
protected:
    Observer();
};

class Subject {
public:
    virtual ~Subject();

    virtual void Attach(Observer*);
    virtual void Detach(Observer*);
    virtual void Notify();
protected:
    Subject();
private:
    List<Observer*> *_observers;
};

void Subject::Attach (Observer* o) {
    _observers->Append(o);
}

void Subject::Detach (Observer* o) {
    _observers->Remove(o);
}

void Subject::Notify () {
    ListIterator<Observer*> i(_observers);

    for (i.First(); !i.IsDone(); i.Next()) {
        i.CurrentItem()->Update(this);
    }
}
```

Related Patterns

- Mediator: By encapsulating complex update semantics, the ChangeManager acts as mediator between subjects and observers.
- Singleton: The ChangeManager may use the Singleton pattern to make it unique and globally accessible.

8. STATE

Intent

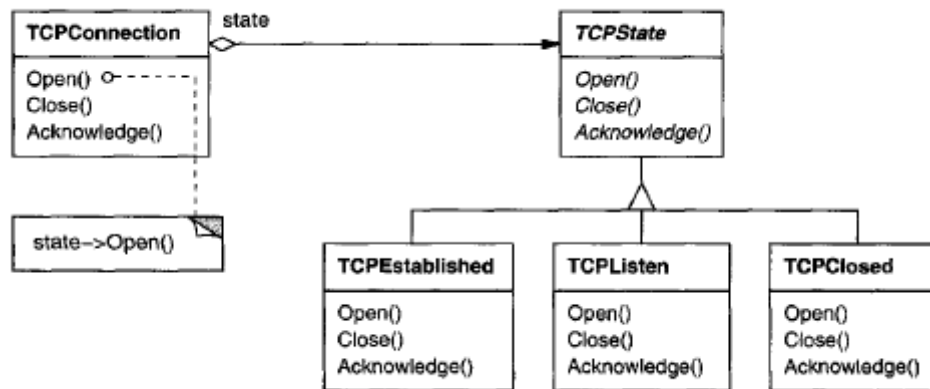
Allow an object to alter its behavior when its internal state changes. The object will appear to change its class.

Also Known As

Objects for States

Motivation

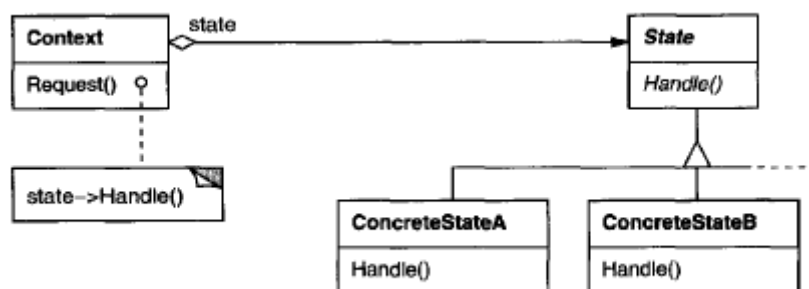
Consider a class TCP Connection that represents a network connection. A TCP Connection object can be in one of several different states: Established, Listening Closed. When a TCP Connection object receives requests from other objects, it responds differently depending on its current state. For example, the effect of an Open request depends on whether the connection is in its closed state or its Established state. The State pattern describes how TCP Connection can exhibit different behavior in each state.



Applicability: Use the State pattern in either of the following cases:

- An object's behavior depends on its state, and it must change its behavior at run-time depending on that state.
- Operations have large, multipart conditional statements that depend on the object's state

Structure



Participants

- Context (TCPConnection)
- State (TCPState)
- ConcreteState subclasses (TCPEstablished, TCPListen, TCPClosed)

Collaborations

- Context delegates state-specific requests to the current ConcreteState object.
- A context may pass itself as an argument to the State object handling the request. This lets the State object access the context if necessary.

Consequences: The State pattern has the following consequences

- It localizes state-specific behavior and partitions behavior for different states
- It makes state transitions explicit
- State objects can be shared

Sample Code: The following example gives the C++ code for the TCP connection example described in the Motivation section.

```
class TCPOctetStream;
class TCPState;

class TCPConnection {
public:
    TCPConnection();

    void ActiveOpen();
    void PassiveOpen();
    void Close();

    void Send();
    void Acknowledge();
    void Synchronize();

    void ProcessOctet(TCPOctetStream*);
private:
    friend class TCPState;
    void ChangeState(TCPState*);
private:
    TCPState* _state;
};
```

Related Patterns

- The Flyweight pattern explains when and how State objects can be shared.
- State objects are often Singletons

9. STRATEGY

Intent

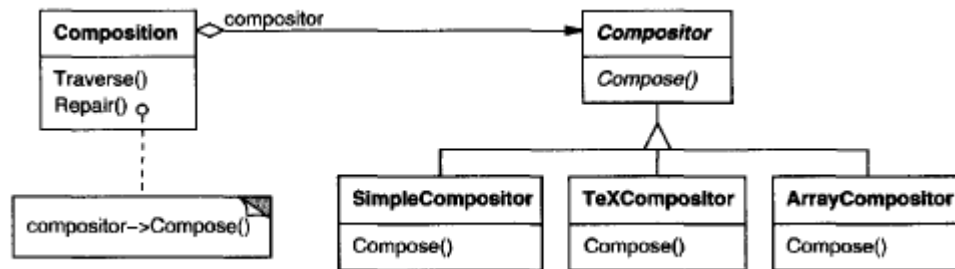
Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it.

Also Known As

Policy

Motivation

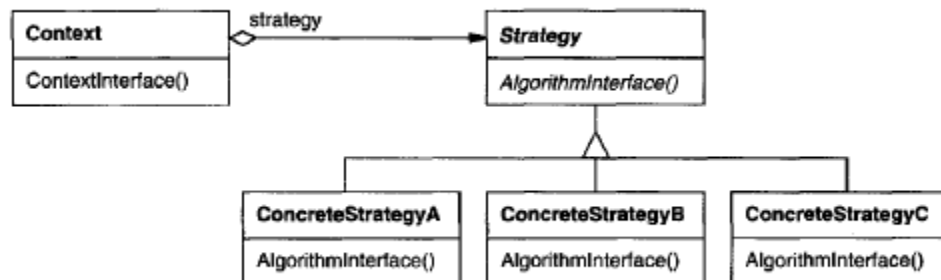
Many algorithms exist for breaking a stream of text into lines. Hard-wiring all such algorithms into the classes that require them aren't desirable for several reasons: We can avoid these problems by defining classes that encapsulate different line breaking algorithms. An algorithm that's encapsulated in this way is called a strategy.



Applicability: Use the Strategy pattern when

- Many related classes differ only in their behavior. Strategies provide a way to configure a class with one of many behaviors.
- You need different variants of an algorithm. For example, you might define algorithms reflecting different space/time trade-offs. Strategies can be used when these variants are implemented as a class hierarchy of algorithms

Structure



Participants

- Strategy, ConcreteStrategy, Context

Collaborations

- Strategy and Context interact to implement the chosen algorithm
- A context forwards requests from its clients to its strategy

Consequences: The Strategy pattern has the following benefits and drawbacks:

- Families of related algorithms
- An alternative to subclassing
- Strategies eliminate conditional statements

Sample Code: We'll give the high-level code for the Motivation example, which is based on the implementation of Composition

```
class Composition {
public:
    Composition(Compositor*);
    void Repair();
private:
    Compositor* _compositor;
    Component* _components;    // the list of components
    int _componentCount;      // the number of components
    int _lineWidth;           // the Composition's line width
    int* _lineBreaks;         // the position of linebreaks
                                // in components
    int _lineCount;           // the number of lines
};

void Composition::Repair () {
    Coord* natural;
    Coord* stretchability;
    Coord* shrinkability;
    int componentCount;
    int* breaks;

    // prepare the arrays with the desired component sizes
    // ...

    // determine where the breaks are:
    int breakCount;
    breakCount = _compositor->Compose(
        natural, stretchability, shrinkability,
        componentCount, _lineWidth, breaks
    );

    // lay out components according to breaks
    // ...
}
```

Related Patterns

- Flyweight (195): Strategy objects often make good flyweights

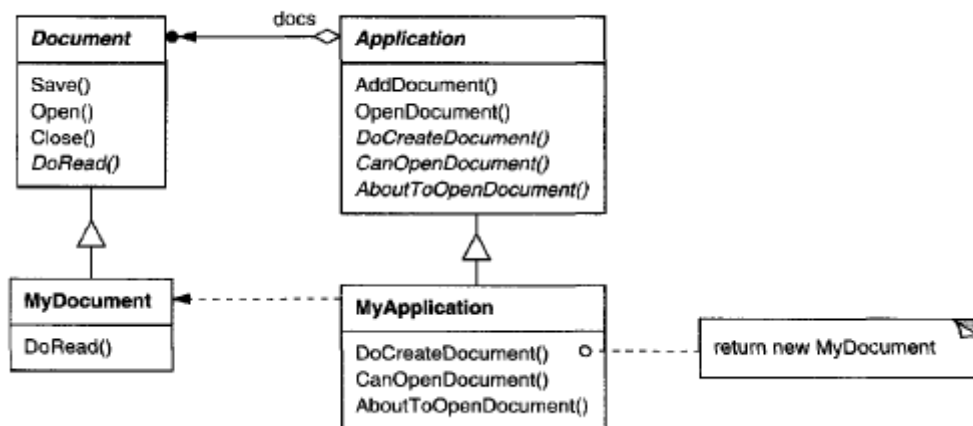
10. TEMPLATE METHOD

Intent

Define the skeleton of an algorithm in an operation, deferring some steps to subclasses. Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure.

Motivation

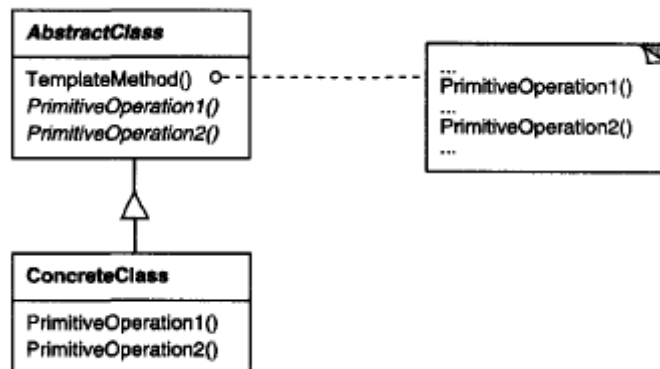
Consider an application framework that provides Application and Document classes. The Application class is responsible for opening existing documents stored in an external format, such as a file. A Document object represents the information in a document once it's read from the file.



Applicability: The Template Method pattern should be used

- To implement the invariant parts of an algorithm once and leave it up to subclasses to implement the behavior that can vary.
- When common behavior among subclasses should be factored and localized in a common class to avoid code duplication.

Structure



Participants

- AbstractClass, ConcreteClass

Collaborations

- ConcreteClass relies on AbstractClass to implement the invariant steps of the algorithm.

Consequences

Template methods are a fundamental technique for code reuse. They are particularly important in class libraries, because they are the means for factoring out common behavior in library classes.

Sample Code

The following C++ example shows how a parent class can enforce an invariant for its subclasses.

```
void View::Display () {  
    SetFocus();  
    DoDisplay();  
    ResetFocus();  
}
```

To maintain the invariant, the View's clients always call Display, and View subclasses always override DoDisplay.

DoDisplay does nothing in View:

```
void View::DoDisplay () { }
```

Subclasses override it to add their specific drawing behavior:

```
void MyView::DoDisplay () {  
    // render the view's contents  
}
```

Known Uses

- Template methods are so fundamental that they can be found in almost every abstract class

Related Patterns

- Factory Methods are often called by template methods.
- Strategy: Template methods use inheritance to vary part of an algorithm.
- Strategies use delegation to vary the entire algorithm.

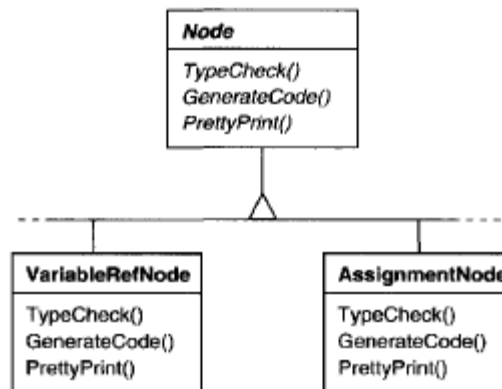
11. VISITOR

Intent

Represent an operation to be performed on the elements of an object structure. Visitor lets you define a new operation without changing the classes of the elements on which it operates.

Motivation

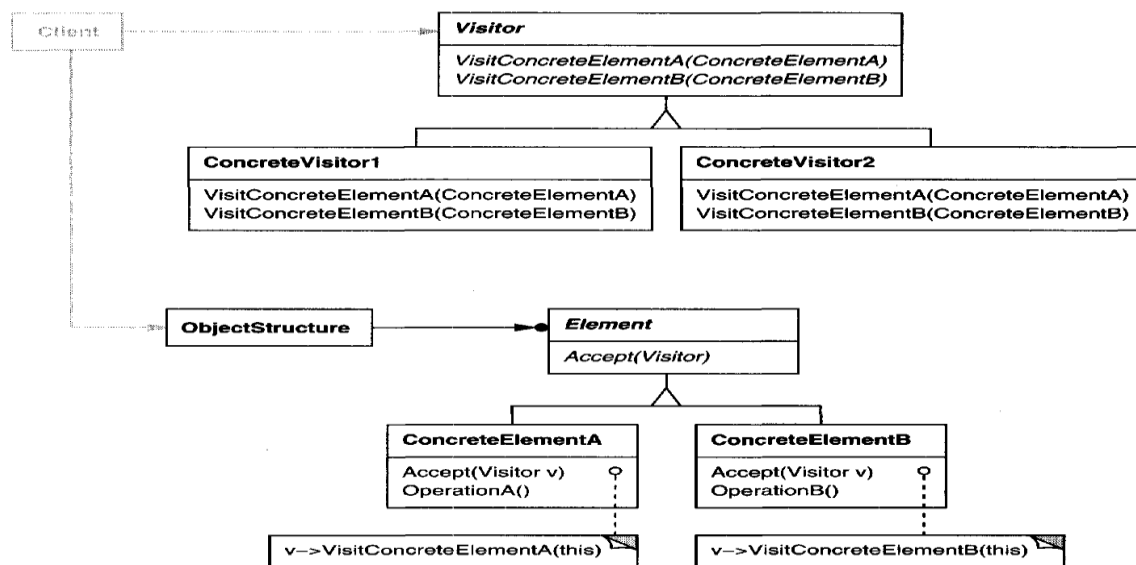
Consider a compiler that represents programs as abstract syntax trees. It will need to perform operations on abstract syntax trees for "static semantic" analyses like checking that all variables are defined,



Applicability: Use the Visitor pattern when

- An object structure contains many classes' of objects with differing interfaces, and you want to perform operations on these objects that depend on their concrete classes.
- Many distinct and unrelated operations need to be performed on objects in an object structure, and you want to avoid "polluting" their classes with these operations.

Structure



Participants

- Visitor, ConcreteVisitor, Element, ConcreteElement, ObjectStructure

Collaborations

- A client that uses the Visit or pattern must create a ConcreteVisitor object and then traverse the object structure, visiting each element with the visitor.
- When an element is visited, it calls the Visitor operation that corresponds to its class.

Consequences: Some of the benefits and liabilities of the Visit or pattern are as follows:

- Visitor makes adding new operations easy
- A visitor gathers related operations and separates unrelated ones
- Adding new ConcreteElement classes is hard

Implementation

Each object structure will have an associated Visitor class. This abstract visitor class declares a VisitConcreteElement operation for each class of ConcreteElement defining the object structure. Each Visit operation on the Visitor declares its argument to be a particular ConcreteElement, allowing the Visitor to access the interface of the ConcreteElement directly. ConcreteVisitor classes override each Visit operation to implement visitor-specific behavior for the corresponding ConcreteElement class.

Sample Code: Because visitors are usually associated with composites, we'll use the Equipment classes defined in the Sample Code of Composite (163) to illustrate the Visitor pattern.

```
class Equipment {
public:
    virtual ~Equipment();

    const char* Name() { return _name; }

    virtual Watt Power();
    virtual Currency NetPrice();
    virtual Currency DiscountPrice();

    virtual void Accept(EquipmentVisitor&);
protected:
    Equipment(const char*);
private:
    const char* _name;
};
```

Related Patterns

- Composite: Visitors can be used to apply an operation over an object structure defined by the Composite pattern
- Interpreter: Visitor may be applied to do the interpretation

SOFTWARE ARCHITECTURE AND DESIGN PATTERNS
UNIT-V

Frequently Asked Questions

1. Discuss in detail about Chain of responsibility design pattern?
2. Write a short note on Command design pattern?
3. Briefly explain the working of Interpreter design pattern?
4. Explain the working of Iterator design pattern?
5. Describe the working of Mediator design pattern?
6. Discuss in detail about Memento design pattern?
7. Write a short note on Observer design pattern?
8. Briefly explain the working of state strategy design pattern?
9. Explain the working of template method design pattern?
10. Describe the working of Visitor design pattern?

SOFTWARE ARCHITECTURE AND DESIGN PATTERNS

UNIT-VI

UNIT-VI:Case Studies

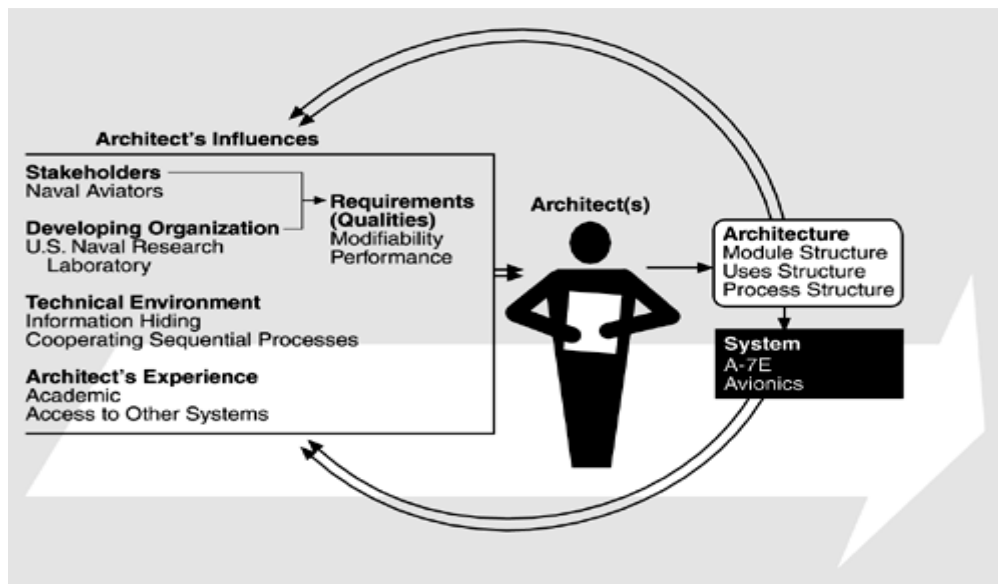
A-7E Avionics System: A Case Study in Utilizing Architectural Structures, Air Traffic Control: A Case Study in Designing for High Availability, Flight Simulation: A Case Study in an Architecture for Integrability, CelsiusTech: A Case Study in Product Line Development, J2EE/EJB: A Case Study of an Industry-Standard Computing Infrastructure, The Luther Architecture: A Case Study in Mobile Applications Using J2EE, The World Wide Web'A Case Study in Interoperability, The Nightingale System: A Case Study in Applying the ATAM, The NASA ECS Project:A Case Study in Applying the CBAM

1. A-7E – A case study in utilizing architectural structures

This case study of an architecture designed by engineering and specifying three specific architectural structures: module decomposition, uses, and process. We will see how these structures complement each other to provide a complete picture of how the system works, and we will see how certain qualities of the system are affected by each one. Table summarizes the three structures we will discuss.

| Structure | Elements | Relation among Elements | Has Influence Over |
|----------------------|---------------------------------|--|---|
| Module Decomposition | Modules (implementation units) | Is a submodule of; shares a secret with | Ease of change |
| Uses | Procedures | Requires the correct presence of | Ability to field subsets and develop incrementally |
| Process | Processes; thread of procedures | Synchronizes with; shares CPU with; excludes | Schedulability; achieving performance goals through parallelism |

The system was constructed beginning in 1977 for the naval aviators who flew the A-7E aircraft and was paid for by the U.S. Navy. The developing organization was the software engineering group at the U.S. Naval Research Laboratory. The developers were creating the software to test their belief that certain software engineering strategies (in this case, information hiding and cooperating sequential processes) were appropriate for high-performance embedded real-time systems.



The architects included one of the authors of this book and one of the leaders in the development of software engineering principles, but the architects had little experience in the avionics domain, although

SOFTWARE ARCHITECTURE AND DESIGN PATTERNS

UNIT-VI

they did have access to other avionics systems and to experts in avionics. There was no compiler available for the target platform.

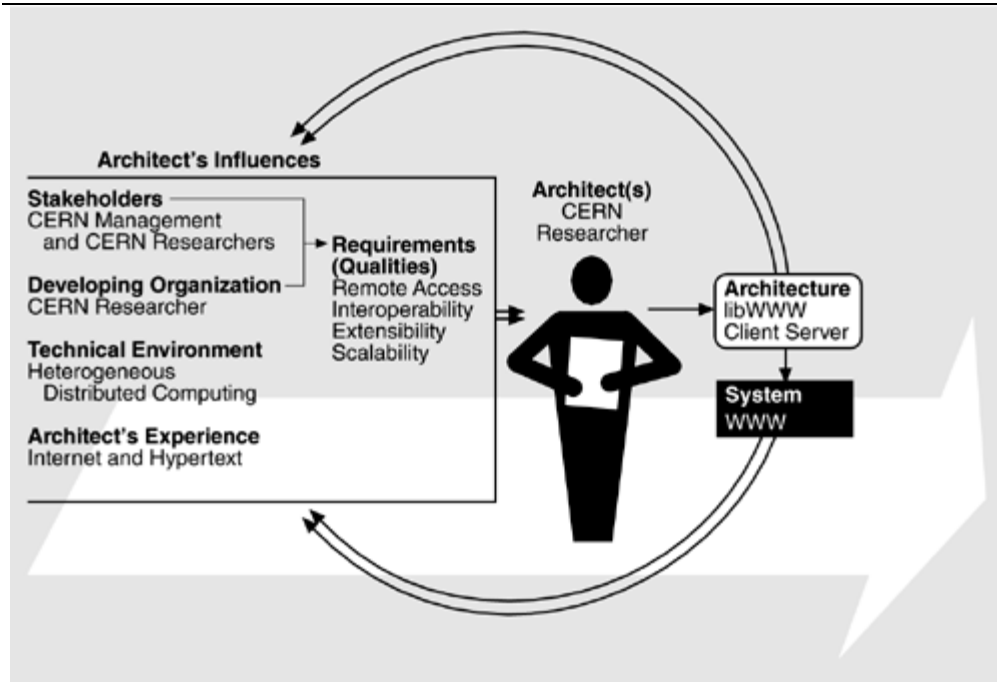
The following are the primary sensors the software reads and manages:

- An air probe that measures barometric pressure and air speed.
- Forward-looking radar that can be aimed in azimuth and elevation and returns the straight-line range to the point on the ground at which it is pointed.
- A Doppler radar that reports ground speed and drift angle (the difference between the direction in which the aircraft's nose is pointed and the direction in which it is moving over the ground).
- An inertial measurement set (IMS) that reports accelerations along each of three orthogonal axes. The software must read these accelerations in a timely manner and integrate them over time to derive velocities, and it must integrate the velocities over time to derive the aircraft's current position in the physical world. It also must manage the alignment and compensate for the drift of the axes to keep them pointed north, east, and vertical, respectively, so that the measurements accurately correspond to the aircraft's frame of reference.
- An interface to the aircraft carrier's inertial measurement system, through which the aircraft can compute its current position while on board a ship.
- Sensors that report which of the A-7E's six underwing bomb racks hold weapons and which of more than 100 kinds of weapons in the aircraft's repertoire they are. The software stores large tables of the parameters for each weapon type, which let it compute how that weapon moves through the atmosphere in a free-fall ballistic trajectory.
- A radar altimeter that measures the distance to the ground.

2. The World Wide Web - a case study in Interoperability

The original proposal for the Web came from Tim Berners-Lee, a researcher with the European Laboratory for Particle Physics (CERN), who observed that the several thousand researchers at CERN formed an evolving human "web." People came and went, developed new research associations, lost old ones, shared papers, chatted in the hallways, and so on, and Berners-Lee wanted to support this informal web with a similar web of electronic information. In 1989, he created and circulated throughout CERN a document entitled Information Management: A Proposal. By October of 1990 a reformulated version of the project proposal was approved by management, the name World Wide Web was chosen, and development began.

Figure shows the elements of the ABC as they applied to the initial proposal approved by CERN management. The system was intended to promote interaction among CERN researchers (the end users) within the constraints of a heterogeneous computing environment. The customer was CERN management, and the developing organization was a lone CERN researcher. The business case made by Berners-Lee was that the proposed system would increase communication among CERN staff. This was a very limited proposal with very limited (and speculative) objectives. There was no way of knowing whether such a system would, in fact, increase communication. On the other hand, the investment required by CERN to generate and test the system was also very limited: one researcher's time for a few months.



The technical environment was familiar to those in the research community, for which the Internet had been a mainstay since its introduction in the early 1970s. The net had weak notions of central control (volunteer committees whose responsibilities were to set protocols for communication among different nodes on the Internet and to charter new newsgroups) and an unregulated, "wild-west" style of interaction, primarily through specialized newsgroups.

Hypertext systems had had an even longer history, beginning with the vision of Vannevar Bush in the 1940s. Bush's vision had been explored throughout the 1960s and 1970s and into the 1980s, with hypertext conferences held regularly to bring researchers together. However, Bush's vision had not been achieved on a large scale by the 1980s: The uses of hypertext were primarily limited to small-scale documentation systems. That was to change.

The World Wide Web, as conceived and initially implemented at CERN, had several desirable qualities. It was portable, able to interoperate with other types of computers running the same software, and was scalable and extensible.

3. Air Traffic Control: A Case Study in Designing for High Availability

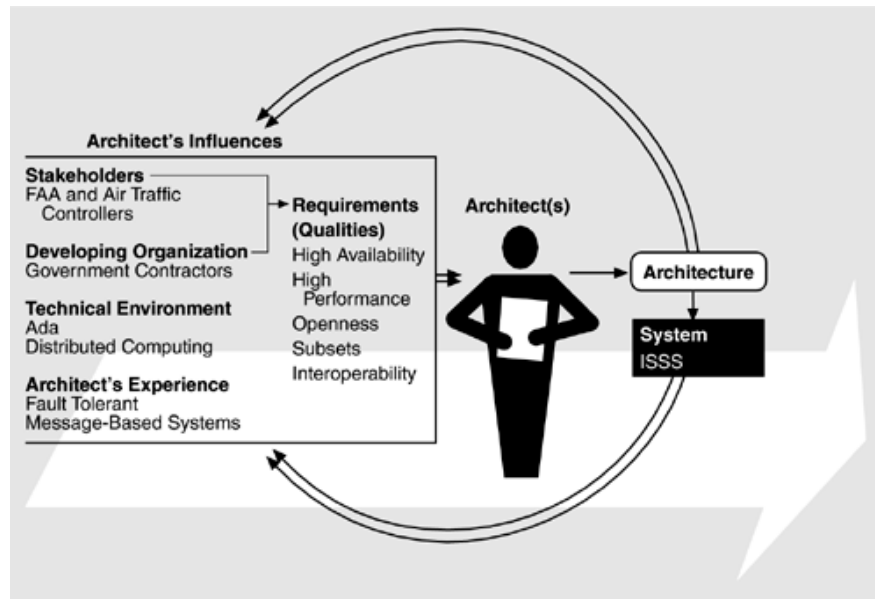
Air traffic control (ATC) is among the most demanding of all software applications. It is hard real time, meaning that timing deadlines must be met absolutely; it is safety critical, meaning that human lives may be lost if the system does not perform correctly; and it is highly distributed, requiring dozens of controllers to work cooperatively to guide aircraft through the airways system. In the United States, whose skies are filled with more commercial, private, and military aircraft than any other part of the world, ATC is an area of intense public scrutiny. Aside from the obvious safety issues, building and maintaining a safe, reliable airways system requires enormous expenditures of public money. ATC is a multibillion-dollar undertaking.

The following Figure shows how the air traffic control system relates to the Architecture Business Cycle (ABC). The end users are federal air traffic controllers; the customer is the Federal Aviation Administration; and the developing organization is a large corporation that supplies many other important

SOFTWARE ARCHITECTURE AND DESIGN PATTERNS

UNIT-VI

software-intensive systems to the U.S. government. Factors in the technical environment include the mandated use of Ada as the language of implementation for large government software systems and the emergence of distributed computing as a routine way to build systems and approach fault tolerance.



Given that air traffic control is highly visible, with huge amounts of commercial, government, and civilian interest, and given that it involves the potential loss of human life if it fails, its two most important quality requirements are as follows:

1. Ultrahigh availability, meaning that the system is absolutely prohibited from being inoperative for longer than very short periods. The actual availability requirement for ISSS is targeted at 0.99999, meaning that the system should be unavailable for less than 5 minutes a year. (However, if the system is able to recover from a failure and resume operating within 10 seconds, that failure is not counted as unavailable time.)
2. High performance, meaning that the system has to be able to process large numbers of aircraft as many as 2,440 without "losing" any of them. Networks have to be able to carry the communication loads, and the software has to be able to perform its computations quickly and predictably.

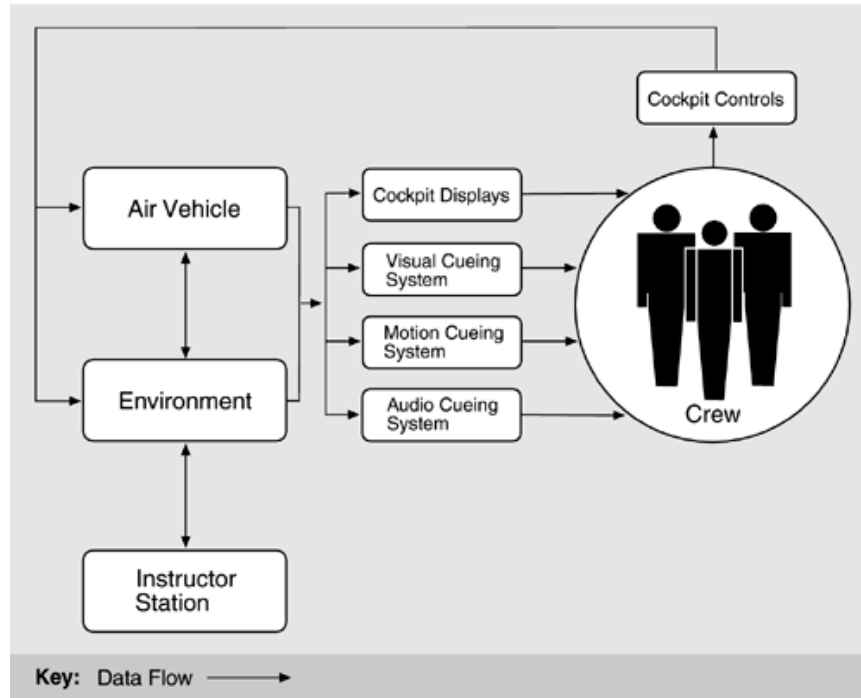
The audit assessed the architecture's ability to deliver the required performance and availability and included modifiability exercises that walked through several change scenarios, including the following:

- Making major modifications to the M&C position's human-computer interface
- Importing third-party-developed air traffic applications into the ISSS system
- Adding new ATC views to the system
- Replacing the RS/6000 processors with a chip of similar capability
- Deleting electronic flight strips from the requirements
- Increasing the system's maximum capacity of flight tracks by 50 percent

In every case, the audit found that the ISSS software architecture had been designed so that the modifications would be straightforward and, in some cases, almost trivial. This is a tribute to its careful design and its explicit consideration of quality attributes and the architectural tactics to achieve them.

4. Flight Simulation: A Case Study in Architecture for Integrability

The following Figure shows a reference model for a flight simulator. The three roles we identified earlier (air vehicle, environment, and instructor) are shown interacting with the crew and the various cueing systems. Typically, the instructor is hosted on a different hardware platform from the air vehicle model. The environment model may be hosted either on a separate hardware platform or with the instructor station.



There are two fundamentally different ways of managing time in a flight simulator?periodic and event-based?and both of these are used. Periodic time management is used in portions that must maintain real-time performance (such as the air vehicle), and event-based time management is used in portions where real-time performance is not critical (such as the instructor station).

Periodic Time Management

A periodic time-management scheme has a fixed (simulated) time quantum based on the frame rate. That is the basis of scheduling the system processes. This scheme typically uses a non-pre-emptive cyclic scheduling discipline, which proceeds by iterating through the following loop:

- Set initial simulated time.
- Iterate the next two steps until the session is complete.
 1. Invoke each of the processes for a fixed (real) quantum. Each process calculates its internal state based on the current simulated time and reports it based on the next period of simulated time. It guarantees to complete its computation within its real-time quantum.
 2. Increment simulated time by quantum.

A simulation based on the periodic management of time will be able to keep simulated time and real time in synchronization as long as each process is able to advance its state to the next period within the time quantum allocated to it.

Typically, this is managed by adjusting the responsibilities of the individual processes so that they are small enough to be computed in the allocated quantum. It is the designer's responsibility to provide the number of processors needed to ensure sufficient computational power to enable all processes to receive their quantum of computation.

Event-Based Time Management

SOFTWARE ARCHITECTURE AND DESIGN PATTERNS

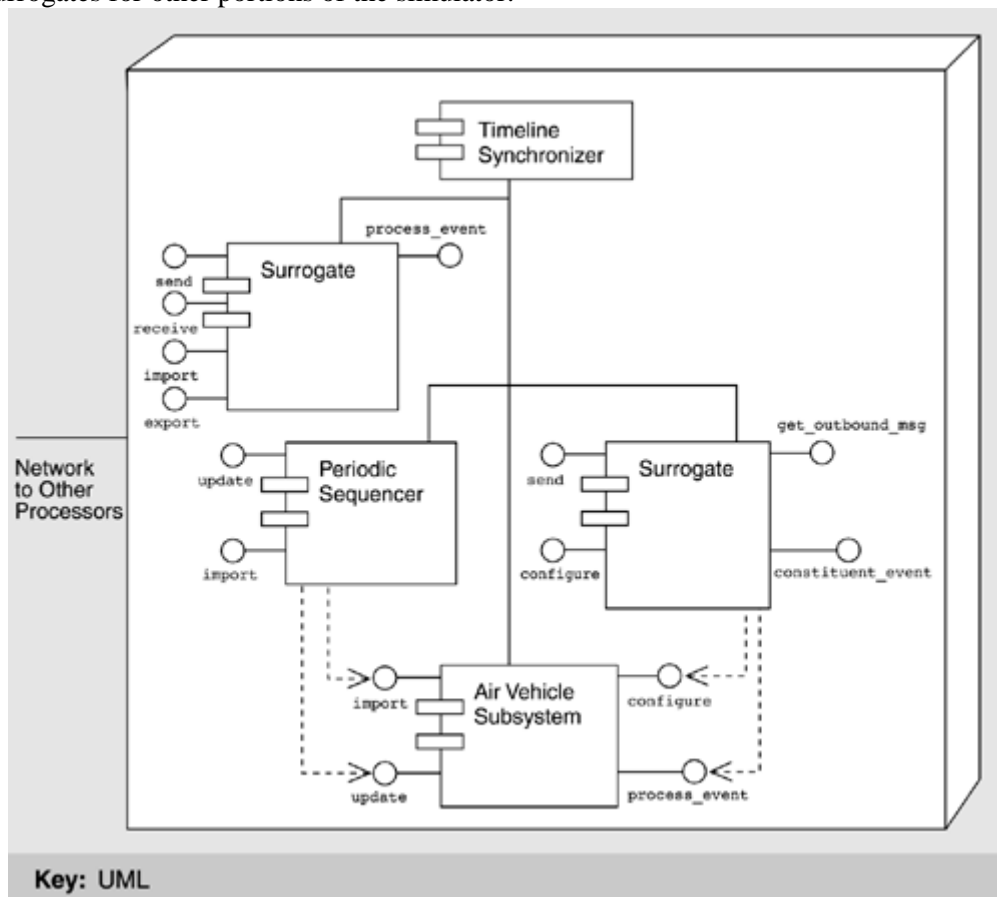
UNIT-VI

An event-based time-management scheme is similar to the interrupt-based scheduling used in many operating systems. The schedule proceeds by iterating through the following loop:

- Add a simulated event to the event queue.
- While there are events remaining in the event queue,
 - choose the event with the smallest (i.e., soonest) simulated time.
 - set the current simulated time to the time of the chosen event.
 - invoke a process for the chosen event. This process may add events to the event queue.

In this case, simulated time advances by the invoked processes placing events on the event queue and the scheduler choosing the next event to process. In pure event-based simulations, simulated time may progress much faster (as in a war game simulation) or much slower (as in an engineering simulation) than real time.

The following Figure shows the air vehicle structural model with the executive pattern given in detail. The modules in the executive are the Timeline Synchronizer, the Periodic Sequencer, the Event Handler, and the Surrogates for other portions of the simulator.



At this point we have identified the division of functionality, its allocation to subsystems and subsystem controllers, and the connections among subsystems. To complete the architecture, we need to do the following:

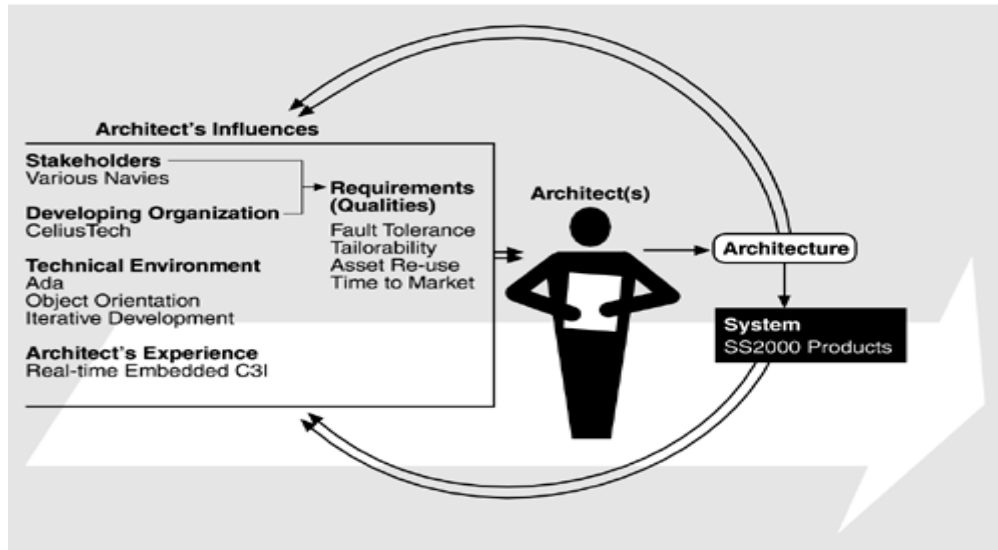
- Identify the controller children instances for the propulsion subsystem.
- Similarly decompose the other groups, their systems, and their subsystems.

To summarize, we decomposed the air vehicle into four groups: kinetics, aircraft systems, avionics, and environment. We then decomposed the kinetics group into four systems: airframe, propulsion, landing gear, and flight controls. Finally, we presented a decomposition of the propulsion system into a collection of subsystems.

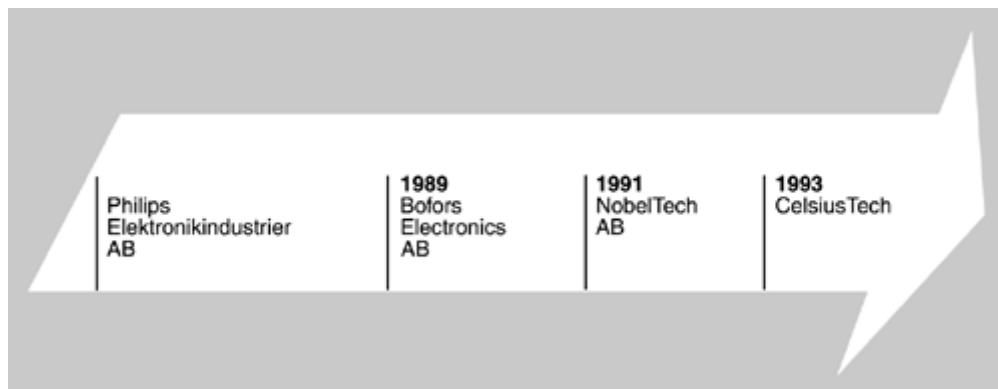
5. CelsiusTech: A Case Study in Product Line Development

This case study relates the experience of CelsiusTech AB, a Swedish naval defense contractor that successfully adopted a product line approach to building complex software-intensive systems. Called Ship System 2000 (SS2000), their product line consists of shipboard command-and-control systems for Scandinavian, Middle Eastern, and South Pacific navies.

This case study illustrates the entire Architecture Business Cycle (ABC), but especially shows how a product line architecture led CelsiusTech to new business opportunities. Figure shows the roles of the ABC stakeholders in the CelsiusTech experience.



This study focuses on CelsiusTech Systems (CelsiusTech for short), whose focus includes command, control, and communication (C3) systems, fire control systems,[1] and electronic warfare systems for navy, army, and air force applications. The organization has undergone several changes in ownership and name since 1985 (see Figure 15.2). Originally Philips Elektronikindustrier AB, the division was sold to Bofors Electronics AB in 1989 and reorganized into NobelTech AB in 1991. It was purchased by CelsiusTech in 1993. Although senior management changed with each transaction, most of the mid-and lower-level management and the technical staff remained, thus providing continuity and stability.



SOFTWARE ARCHITECTURE AND DESIGN PATTERNS

UNIT-VI

Between 1986 and 1998 CelsiusTech evolved from a defense contractor providing custom-engineered point solutions to essentially a vendor of commercial off-the-shelf naval systems. They found the old ways of organizational structure and management insufficient to support the emerging business model. They also found that achieving and sustaining an effective product line was not simply a matter of the right software and system architecture, development environment, hardware, or network. Organizational structure, management practices, and staffing characteristics were also dramatically affected.

The architecture served as the foundation of the approach, both technically and culturally. In some sense, it became the tangible thing whose creation and instantiation were the ultimate goal. Because of its importance, the architecture was highly visible. A small, elite architecture team had the authority as well as the responsibility for it. As a consequence, the architecture achieved the "conceptual integrity" cited by [Brooks 95] as the key to any quality software venture.

Defining the architecture was only the first step in building a foundation for a long-term development effort. Validation through prototyping and early use was also essential. When deficiencies were uncovered, the architecture had to evolve in a smooth, controlled manner throughout initial development and beyond. To manage this natural evolution, CelsiusTech's integration and architecture teams worked together to prevent any designer or design team from changing critical interfaces without the architecture team's explicit approval.

This approach had the full support of project management, and it worked because of the architecture team's authority. The team was a centralized design authority that could not be circumvented, which meant that conceptual integrity was maintained.

The organization necessary to create a product line is different from that needed to sustain and evolve it. Management needs to plan for changing personnel, management, training, and organizational needs. Architects with extensive domain knowledge and engineering skill are vital to the creation of viable product lines. Domain experts remain in demand as new products are envisioned and product line evolution is managed.

CelsiusTech's turnaround from one-at-a-time systems to a product line involved education and training on the part of management and technicians. All of these are what we mean by the return cycle of the ABC.

6. J2EE/EJB: A Case Study of an Industry-Standard Computing Infrastructure

This case study presents an overview of Sun Microsystems's Java 2 Enterprise Edition (J2EE) architecture specification, as well as an important portion of that specification, Enterprise JavaBeans (EJB). J2EE provides a standard description of how distributed object-oriented programs written in Java should be designed and developed and how the various Java components can communicate and interact. EJB describes a server-side component-based programming model. Taken as a whole, J2EE also describes various enterprise-wide services, including naming, transactions, component life cycle, and persistence, and how these services should be uniformly provided and accessed. Finally, it describes how vendors need to provide infrastructure services for application builders so that, as long as conformance to the standard is achieved, the resultant application will be portable to all J2EE platforms.

J2EE/EJB is one approach to building distributed object-oriented systems. There are, of course, others. People have been building distributed object-oriented systems using the Object Management Group's (OMG) Common Object Request Broker Architecture (CORBA) during the last decade. In the CORBA model, an object request broker (ORB) allows objects to publish their interfaces and allows client programs (and perhaps other objects) to locate these remote objects anywhere on the computer network and to request services from them. Microsoft, too, has a technology, .NET, for building distributed systems. The .NET architecture has similar provisions for building distributed object systems for Windows-based platforms.

SOFTWARE ARCHITECTURE AND DESIGN PATTERNS

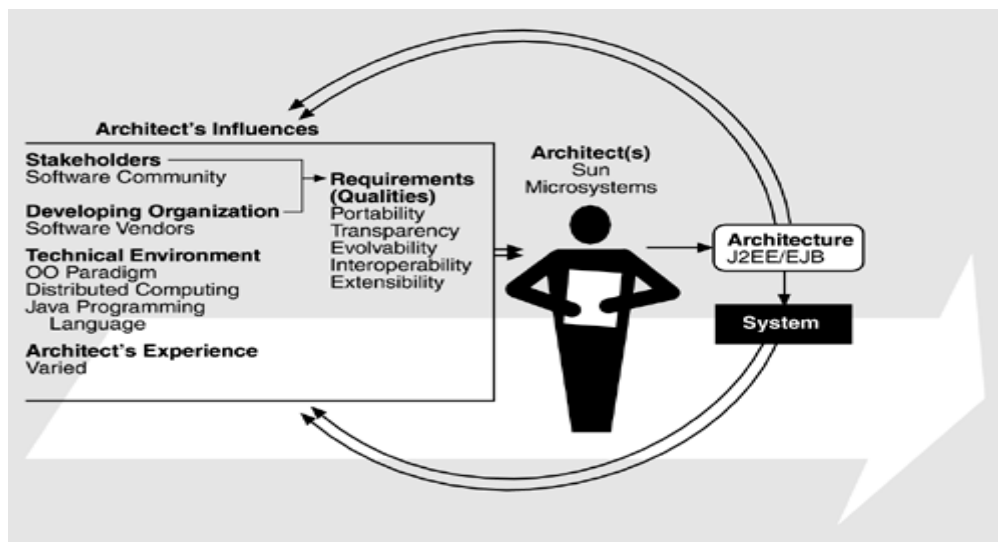
UNIT-VI

We will start the chapter by looking at the business drivers that led to the creation of an industry standard architecture for distributed systems. Then we will discuss how the J2EE/EJB architecture addresses such needs. We will look at the typical quality requirements of Web-based applications and see how the J2EE/EJB architecture fulfills them.

In the 1980s, the price/performance ratio for personal computers was gradually dovetailing with that of high-end workstations and "servers." This newly available computing power and fast network technology enabled the widespread use of distributed computing.

However, rival computer vendors kept producing competing hardware, operating systems, and network protocols. To an end-user organization, such product differentiation presented problems in distributed computing. Typically, organizations invested in a variety of computing platforms and had difficulty building distributed systems on top of such a heterogeneous environment.

The Object Management Group's Common Object Request Broker Architecture was developed in the early 1990s to counter this problem. The CORBA model provided a standard software platform on which distributed objects could communicate and interact with each other seamlessly and transparently. In this case, an ORB allows objects to publish their interfaces, and it allows client programs to locate them anywhere on the computer network and to request services from them. The ABC for J2EE/EJB is shown in Figure.



The creation of the J2EE multi-tier architecture was motivated by the business needs of Sun Microsystems. These business needs were influenced by the lessons of the CORBA model and by the competitive pressures of other proprietary distributed programming models, such as COM+ from Microsoft. J2EE features a server-side component framework for building enterprise-strength server-side Java applications, namely, Enterprise JavaBeans.

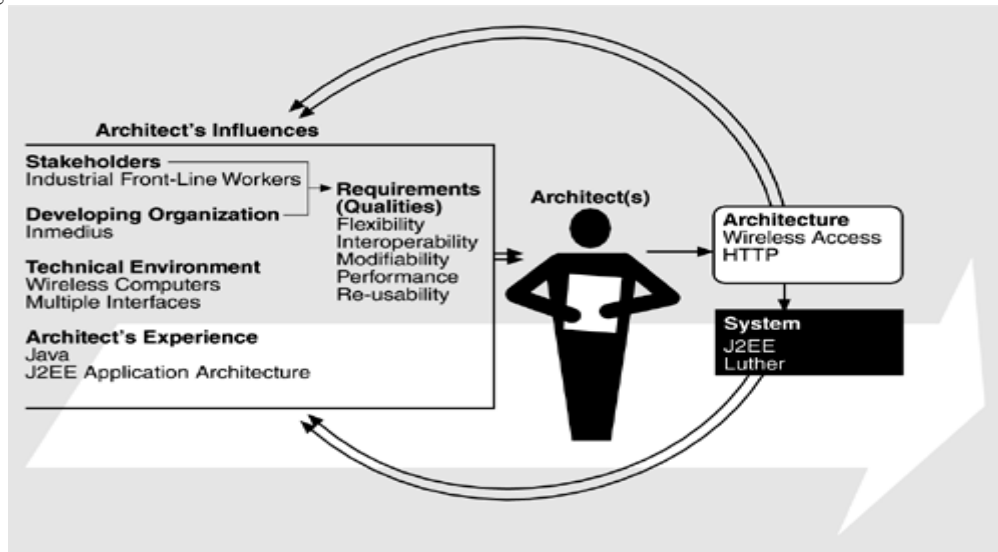
SOFTWARE ARCHITECTURE AND DESIGN PATTERNS

UNIT-VI

7. The Luther Architecture: A Case Study in Mobile Applications Using J2EE

The Luther architecture was designed to provide a general framework within which Inmedius could provide customized solutions for the maintenance problems of its customers. It is based on the Java 2 Enterprise Edition (J2EE) architecture, so becomes an application of the general J2EE/EJB framework (discussed in Chapter 16) to an environment where the end user is connected over a wireless network and has a device with limited input/output capabilities, limited computational capabilities, or both.

The following Figure shows the Architecture Business Cycle (ABC) as it pertains to Inmedius and the Luther architecture. The quality goals of re-usability, performance, modifiability, flexibility of the end user device, and interoperability with standard commercial infrastructures are driven, as always, by the business goals of the customer and the end user.



The Luther architecture was designed to meet two sets of complementary requirements. The first set governs the applications to be built—namely, enterprise applications for field service workers. These requirements are directly visible to customers, since failure to meet them results in applications that do not perform according to expectations—for instance, an application that may work correctly but perform poorly over a wireless network. The second set of requirements involves introducing a common architecture across products. This reduces integration time, brings products to market faster, increases product quality, eases introduction of new technologies, and brings consistency across products.

Overall, the requirements can be separated into six categories:

- Wireless access
- User interface
- Device type
- Existing procedures, business processes, and systems
- Building applications
- Distributed computing

The main architectural decision made in response to requirements was that Luther would be constructed on top of J2EE, which has the following advantages:

- It is commercially available from a variety of vendors. Components, such as work-flow management, that may be useful in Luther are being widely developed.

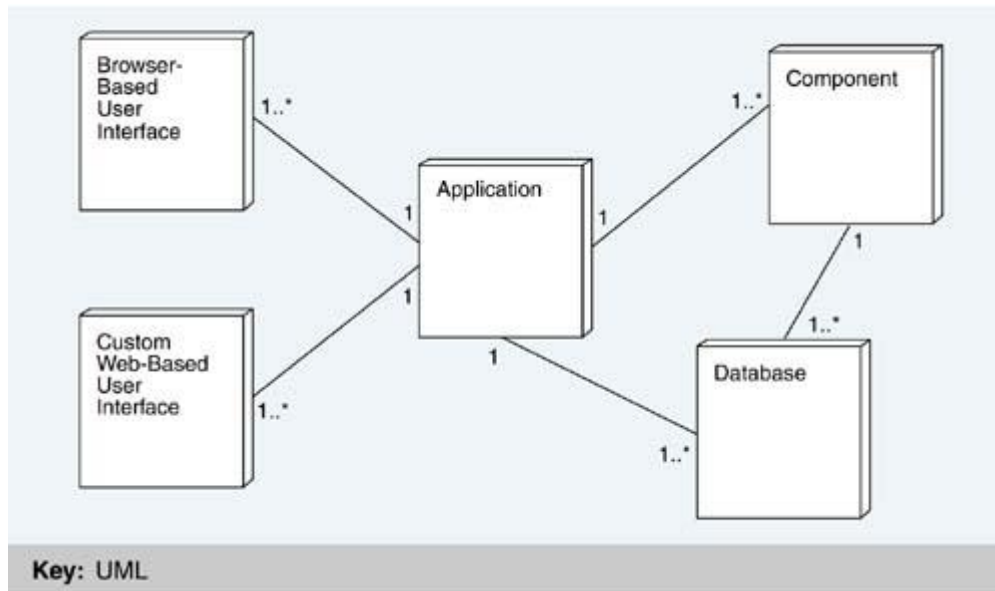
SOFTWARE ARCHITECTURE AND DESIGN PATTERNS

UNIT-VI

- HTTP becomes the basis of communication because it is layered on top of the TCP/IP protocol, which in turn is supported by a variety of commercial wireless standards, such as the IEEE 802.11b. Any Web-based client can be made mobile given the appropriate wireless LAN infrastructure. Most of the devices that must be supported by Luther can support HTTP.

A Luther application is thin; much of its business logic is assembled from existing components, and it is not tied to any specific user interface. Essentially, the application code contains these three things:

- Session state definition and management
- Application-specific (i.e., nonreusable) business logic
- Logic that delegates business requests to an appropriate sequence of component method invocations



Luther is a solution that Inmedius constructed to support the rapid building of customer support systems. It is based on J2EE. A great deal of attention has been given to developing re-usable components and frameworks that simplify the addition of various portions, and its user interface is designed to enable customer- as well as browser-based solutions.

Reliance on J2EE furthered the business goals of Inmedius but also introduced the necessity for additional design decisions in terms of what was packaged as which kind of bean (or not). This is an example of the backward flow of the ABC, emphasizing the movement away from stovepipe solutions toward common solutions.

SOFTWARE ARCHITECTURE AND DESIGN PATTERNS

UNIT-VI

8. The Nightingale System: A Case Study in Applying the ATAM

A major producer of health care systems software, aimed at the hospital, clinic, and HMO markets. The system under consideration was called Nightingale. We learned that it was a large system expected to comprise several million lines of code and that it was well into implementation. Nightingale already had its first customer, a hospital chain with forty-some hospitals throughout the southwestern United States.

The system would serve as the information backbone for the health care institutions in which it was installed. It would provide data about patients' treatment history as well as track their insurance and other payments. And it would provide a data-warehousing capability to help spot trends (such as predictors for relapses of certain diseases). The system would produce a large number of on- demand and periodic reports, each tailored to the institution's specific needs. For those patients making payments on their own, it would manage the work flow associated with initiating and servicing what amounts to a loan throughout its entire life. Further, since the system would either run (or at least be accessible) at all of the health care institution's facilities, it had to be able to respond to a specific office's configuration needs. Different offices might run different hardware configurations, for instance, or require different reports. A user might travel from one site to another, and the system would have to recognize that user and his or her specific information needs, no matter the location. Negotiations to sign a statement of work took about a month?par for the course when legalities between two large organizations are involved? and when it was complete we formed an evaluation team of six people, assigning roles as shown in Table.

| Member | Role |
|--------|--|
| 1 | Team leader, evaluation leader, questioner |
| 2 | Evaluation leader, questioner |
| 3 | Timekeeper, questioner |
| 4 | Scenario scribe, questioner, data gatherer |
| 5 | Questioner, process enforcer |
| 6 | Proceedings scribe, process observer |

PHASE 1: EVALUATION

As called for in phase 1, the evaluation team met with the project's decision makers. In addition to those who had attended the kickoff meeting (the project manager, the lead architect, and the project manager for Nightingale's kickoff customer), two lead designers participated.

Step 1: Present ATAM

The evaluation leader used our organization's standard viewgraph package that explains the method. The hour-long presentation lays out the method's steps and phases, describes the conceptual foundations underlying the ATAM (such as scenarios, architectural approaches, sensitivity points, and the like), and lists the outputs that will be produced by the end of the exercise.

The decision makers were already largely familiar with ATAM, having heard it described during the phase 0 discussions, so this step proceeded without a hitch.

Step 2: Present Business Drivers

At the evaluation, the project manager for the client organization presented the business objectives for the Nightingale system from the development organization, as well as from organizations they hoped would be customers for the system. For the development organization, Nightingale addressed business requirements that included

SOFTWARE ARCHITECTURE AND DESIGN PATTERNS

UNIT-VI

- support for their kickoff customer's diverse uses (e.g., treatment tracking, payment histories, trend spotting, etc.).
- creation of a new version of the system (e.g., to manage doctors' offices) that the development organization could market to customers other than the kickoff customer.

The second business driver alerted us to the fact that this architecture was intended for an entire software product line (see Chapter 14), not just one system.

For the kickoff customer, Nightingale was to replace the multiple existing legacy systems, which were

- old (one was more than 25 years old).
- n based on aging languages and technology (e.g., COBOL and IBM assembler).
- difficult to maintain.
- unresponsive to the current and projected business needs of the health care sites.

The kickoff customer's business requirements included

- the ability to deal with diverse cultural and regional differences.
- the ability to deal with multiple languages (especially English and Spanish) and currencies (especially the U.S. dollar and Mexican peso).
- a new system at least as fast as any legacy system being replaced.
- a new single system combining distinct legacy financial management systems.

The business constraints for the system included

- a commitment to employees of no lost jobs via retraining of existing employees.
- the adoption of a "buy rather than build" approach to software.
- recognition that the customer's marketplace (i.e., number of competitors) had shrunk.

The technical constraints for the system included

- use of off-the-shelf software components whenever possible.
- a two-year time frame to implement the system with the replacement of physical hardware occurring every 26 weeks.

The following quality attributes were identified as high priority:

- **Performance.** Health care systems require quick response times to be considered useful. The 5-second transaction response time of the legacy system was too slow, as were the legacy response times for online queries and reports. System throughput was also a performance concern.
- **Usability.** There was a high turnover of users of the system, so retraining was an important customer issue. The new system had to be easy to learn and use.
- **Maintainability.** The system had to be maintainable, configurable, and extensible to support new markets (e.g., managing doctors' offices), new customer requirements, changes in state laws and regulations, and the needs of the different regions and cultures.

The manager identified the following quality attributes as important, but of somewhat lower priority:

- **Security.** The system had to provide the normal commercial level of security (e.g., confidentiality and data integrity) required by financial systems.
- **Availability.** The system had to be highly available during normal business hours.
- **Scalability.** The system had to scale up to meet the needs of the largest hospital customers and down to meets the needs of the smallest walk-in clinics.
- **Modularity.** The developing organization was entertaining the possibility of selling not just new versions of Nightingale but individual components of it. Providing this capability required qualities closely related to maintainability and scalability.
- **Testability and supportability.** The system had to be understandable by the customer's technical staff since employee training and retention was an issue.

Step 3: Present Architecture

During the evaluation team's interactions with the architect, before as well as during the evaluation exercise, several views of the architecture and the architectural approaches emerged. Key insights included the following:

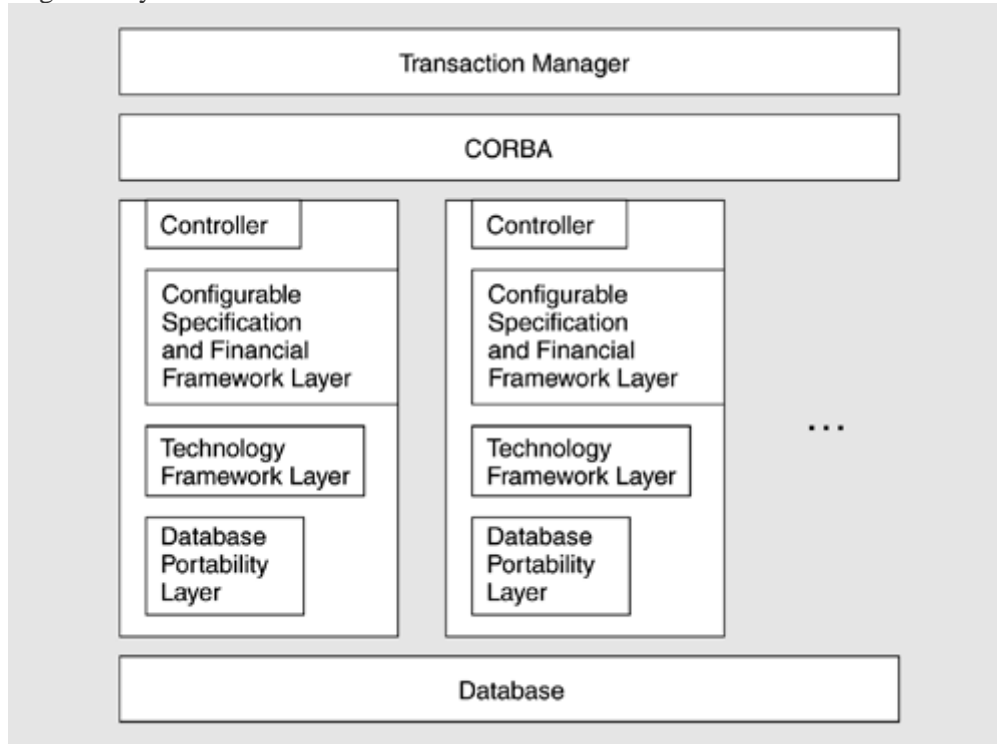
- Nightingale consisted of two major subsystems: OnLine Transaction Manager (OLTM) and Decision Support and Report Generation Manager (DSRGM). OLTM carries interactive

SOFTWARE ARCHITECTURE AND DESIGN PATTERNS

UNIT-VI

performance requirements, whereas DSRGM is more of a batch processing system whose tasks are initiated periodically.

- Nightingale was built to be highly configurable.
- The OnLine Transaction Manager subsystem was strongly layered.
- Nightingale was a repository-based system; a large commercial database lay at its heart.
- Nightingale relied heavily on COTS software, including the central database, a rules engine, a work flow engine, CORBA, a Web engine, a software distribution tool, and many others.
- Nightingale was heavily object oriented, relying on object frameworks to achieve much of its configurability.



- The ATAM is a robust 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 cast as risks or nonrisks to find any trouble spots in the architecture.
- In addition to understanding what the ATAM is, it is also important to understand what it is not.
- The ATAM is not an evaluation of requirements. That is, an ATAM-based evaluation will not tell anyone whether all of the requirements for a system will be met. It will discern whether gross requirements are satisfiable given the current design.
- The ATAM is not a code evaluation. Because it is designed for use early in the life cycle, it makes no assumptions about the existence of code and has no provision for code inspection.
- The ATAM does not include actual system testing. Again because the ATAM is designed for use early in the life cycle, it makes no assumptions of the existence of a system and has no provisions for any type of actual testing.

SOFTWARE ARCHITECTURE AND DESIGN PATTERNS

UNIT-VI

9. The NASA ECS Project: A Case Study in Applying the CBAM

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.

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 requirements. The long-term nature of the project also makes modifiability important. In the execution of the CBAM described next, we concentrated on analyzing the Data Access Working Group (DAWG) portion of the ECS.

STEP 1: COLLATE SCENARIOS

Scenarios from the ATAM were collated with a set of new scenarios elicited from the assembled ECS stakeholders. Because the stakeholders had been through an ATAM exercise, this step was relatively straightforward. A subset of the raw scenarios put forward by the DAWG team were as shown in Table. 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.

| Scenario | Scenario Description |
|----------|--|
| 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 the best-case response goals for each scenario were elicited and recorded, as shown in Table.

SOFTWARE ARCHITECTURE AND DESIGN PATTERNS

UNIT-VI

| Scenario | Response Goals | | | |
|----------|---------------------|---------------------|----------------------|----------------------|
| | 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, as shown in Table 12.3. 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

| Scenario | Response Goals | | | | |
|----------|----------------|---------------------|---------------------|----------------------|----------------------|
| | 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 | 1/1000 granules |
| 10 | 5 | < 50% meet goal | 60% meet goal | 80% meet goal | > 90% meet goal |

The CBAM is an iterative elicitation process combined with a decision analysis framework. It incorporates scenarios to represent the various quality attributes. The stakeholders explore the decision space by eliciting utility-response curves to understand how the system's utility varies with changing attributes. The consensus basis of the method allows for active discussion and clarification amongst the stakeholders. The traceability of the design decision permits updating and continuous improvement of the design process over time.

SOFTWARE ARCHITECTURE AND DESIGN PATTERNS
UNIT-VI

Frequently Asked Questions: Discuss the following case studies of Software Architecture

1. A-7E Avionics System: A Case Study in Utilizing Architectural Structures
2. The World Wide Web: A Case Study in Interoperability
3. Air Traffic Control: A Case Study in Designing for High Availability
4. Flight Simulation: A Case Study in an Architecture for Integrability
5. CelsiusTech: A Case Study in Product Line Development
6. J2EE/EJB: A Case Study of an Industry-Standard Computing Infrastructure
7. The Luther Architecture: A Case Study in Mobile Applications Using J2EE
8. The Nightingale System: A Case Study in Applying the ATAM
9. The NASA ECS Project: A Case Study in Applying the CBAM