Object-Oriented Analysis and Design (OOAD) and Unified Modeling Language (UML) are fundamental methodologies and tools in software engineering. Here are some key reasons for using OOAD and UML:

**Why Use OOAD**

1. **Modularity**:
   * **Encapsulation**: OOAD promotes encapsulation, where an object's data is protected from outside interference and misuse.
   * **Separation of Concerns**: It helps in separating concerns by dividing the system into manageable, reusable modules or classes.
2. **Reusability**:
   * **Inheritance**: OOAD leverages inheritance to promote reuse of code.
   * **Polymorphism**: It allows objects to be treated as instances of their parent class rather than their actual class, facilitating reuse and flexibility.
3. **Maintainability**:
   * **Clear Structure**: OOAD helps in creating a clear, hierarchical structure of classes and objects, making the system easier to understand and maintain.
   * **Extensibility**: Adding new features or classes to the system is easier without affecting existing code.
4. **Abstraction**:
   * **Simplification**: OOAD allows developers to focus on higher-level system functionalities by abstracting complex details.
   * **Focus on Real-World Entities**: It models real-world entities and their interactions, making it more intuitive.
5. **Analysis and Design**:
   * **Requirement Analysis**: OOAD provides a systematic approach to analyze and capture requirements.
   * **System Design**: It helps in designing a robust architecture that aligns with the analyzed requirements.

**Why Use UML**

1. **Standardization**:
   * **Industry Standard**: UML is an industry-standard modeling language, ensuring consistency and understanding across different teams and organizations.
   * **Wide Adoption**: It is widely adopted and supported by various modeling tools and platforms.
2. **Visualization**:
   * **Graphical Representation**: UML provides graphical representations of the system, making it easier to understand and communicate complex system designs.
   * **Different Diagrams**: UML offers various types of diagrams (e.g., class diagrams, sequence diagrams, use case diagrams) to visualize different aspects of the system.
3. **Communication**:
   * **Common Language**: UML serves as a common language for developers, architects, and stakeholders, facilitating better communication and understanding.
   * **Documentation**: It provides comprehensive documentation of the system's design and architecture.
4. **Analysis and Design Tool**:
   * **Requirement Modeling**: UML helps in capturing and modeling requirements using use case diagrams.
   * **System Design**: Class diagrams, sequence diagrams, and other UML diagrams assist in designing the system's structure and behavior.
5. **Consistency and Completeness**:
   * **Ensuring Consistency**: UML helps ensure that the system design is consistent across different parts of the system.
   * **Validation and Verification**: It aids in validating and verifying the system's design against the requirements.
6. **Flexibility and Scalability**:
   * **Adaptability**: UML can be adapted to model systems of varying complexity, from small applications to large enterprise systems.
   * **Extensibility**: It allows for extensions and customizations to meet specific project needs.

**Example in HVAC Domain**

**OOAD**:

* **Classes and Objects**: Model the HVAC system using classes like **HVACSystem**, **Heater**, **Cooler**, **Thermostat**, and their interactions.
* **Inheritance**: Use inheritance to create specific types of HVAC components, like **ElectricHeater** and **GasHeater** from a base **Heater** class.

**UML**:

* **Use Case Diagram**: Capture and represent the requirements of the HVAC system, such as turning on the heater or setting the temperature.
* **Class Diagram**: Visualize the structure of the HVAC system, showing classes and their relationships.
* **Sequence Diagram**: Model the sequence of interactions between objects when the HVAC system is activated.

**Summary**

Using OOAD and UML together provides a powerful approach to designing and documenting complex software systems. OOAD's principles of modularity, reusability, and maintainability, combined with UML's standardized, graphical, and communicative capabilities, lead to well-structured, understandable, and maintainable systems. In the HVAC domain, these methodologies can significantly enhance the design and communication of system functionalities and interactions.

**One brief example ( we write text and PlantUML will generate diagram, many other options are there, prefer what your company prefers)**

A use case diagram provides a high-level overview of the functionality provided by a system from the perspective of its users. In the context of an HVAC system, here's an example of a use case diagram: (incomplete …

A screenshot of a computer program

Description automatically generated

1. **Control HVAC System**: Users can control the HVAC system, which includes turning it on/off, switching between heating and cooling modes, etc. This use case encompasses various control actions.
2. **Set Temperature**: Users can interact with the thermostat to set the desired temperature for the HVAC system.
3. **View Current Temperature**: Users can view the current temperature detected by the HVAC system.
4. **View HVAC Status**: Users can check the status of the HVAC system, such as whether it's currently heating, cooling, or idle.
5. **View Energy Usage**: Users can view information about the energy usage of the HVAC system, such as energy consumption statistics over time.

**Summary**

This use case diagram provides a high-level overview of the interactions between users and the HVAC system. Each use case represents a specific action that users can perform, contributing to the overall functionality of the system. Use case diagrams like this help in understanding the system's requirements and in defining its behavior from a user's perspective.

**Little More details**

During the **Object-Oriented Analysis (OOA)** and **Object-Oriented Design (OOD)** processes, there are several key activities to perform. Let’s delve into each stage:

1. **Object-Oriented Analysis (OOA)**:
   * OOA is the initial technical activity in object-oriented software engineering. Its purpose is to understand the problem domain and capture system requirements. Here are the essential aspects of OOA:
     + **Modeling the Information Domain**:
       - OOA involves creating a conceptual model of the problem domain. Imagine you’re building a game: OOA helps you identify all the critical elements within the game world—such as characters, their features, and their interactions. It’s like creating a map of everything important.
     + **Representing Behavior**:
       - OOA helps describe how objects (such as game characters) behave. For example, if a character jumps when you press a button, OOA captures that action. It’s akin to writing down a script for each character.
     + **Defining Functions and Tasks**:
       - Every software program has specific tasks or jobs it needs to perform. OOA helps you list and describe these tasks. In our game, these tasks could include moving characters or keeping score. It’s like creating a to-do list for your software.
     + **Dividing Models for Detail**:
       - OOA breaks down the problem into different parts, including data models, functional models, and behavioral models. This division allows for a deeper understanding of the system[1](https://www.geeksforgeeks.org/object-oriented-analysis-and-design/).
2. **Object-Oriented Design (OOD)**:
   * OOD follows OOA and focuses on transforming the requirements captured during analysis into a concrete software design. Here are the key aspects of OOD:
     + **Creating a Blueprint**:
       - OOD involves designing a blueprint for the system. It defines the structure and organization of the objects identified during analysis.
     + **Defining Classes and Objects**:
       - During OOD, you specify the classes, objects, methods, and other implementation details. It’s about determining how the system will achieve what was planned during OOA.
     + **Emphasizing Implementation Details**:
       - OOD shifts the focus to how the system accomplishes its tasks. It includes defining classes, their attributes, methods, and relationships.
     + **Ensuring Maintainability and Scalability**:
       - OOD aims to create a design that is maintainable, scalable, and efficient. It ensures that the software can be implemented using a specific programming language[2](https://www.geeksforgeeks.org/object-oriented-paradigm-object-oriented-analysis-design/).

In summary, OOA helps you understand the problem domain and requirements, while OOD transforms those requirements into a concrete software design. Both processes are crucial for building robust and effective software system

Example in HVAC domain

**Object-Oriented Analysis (OOA)** and **Object-Oriented Design (OOD)** processes in the context of an HVAC (Heating, Ventilation, and Air Conditioning) system. Details never end. .. so what we are explaining just the process

**Object-Oriented Analysis (OOA) in the HVAC Domain:**

1. **Understanding the Problem Domain**:
   * In OOA, we start by understanding the HVAC system’s requirements and the problem domain.
   * For our HVAC system, we identify key elements:
     + **Components**: Air conditioners, heaters, fans, temperature sensors, etc.
     + **Interactions**: How these components communicate, control temperature, and respond to user input.
     + **User Needs**: What users expect from the system (e.g., comfort, energy efficiency).
2. **Modeling the Information Domain**:
   * We create a conceptual model of the HVAC domain. This includes:
     + **Classes**: AirConditioner, Heater, TemperatureSensor, etc.
     + **Attributes**: Properties of each class (e.g., temperature setting, power status).
     + **Relationships**: How these classes interact (e.g., an AirConditioner uses a TemperatureSensor).
3. **Representing Behavior**:
   * Describe how HVAC components behave:
     + An AirConditioner cools the room.
     + A Heater warms the room.
     + A TemperatureSensor measures room temperature.
4. **Defining Functions and Tasks**:
   * Identify tasks related to HVAC components:
     + Turning on/off.
     + Adjusting temperature settings.
     + Reporting temperature readings.

**Object-Oriented Design (OOD) in the HVAC Domain:**

1. **Creating a Blueprint**:
   * OOD involves designing the structure of our HVAC system.
   * We create a blueprint that outlines the classes, their relationships, and how they collaborate.
2. **Defining Classes and Objects**:
   * Define concrete classes based on our analysis:
     + AirConditioner: Represents an air conditioning unit.
     + Heater: Represents a heating unit.
     + TemperatureSensor: Measures room temperature.
   * Specify attributes (e.g., temperature settings) and methods (e.g., turn on/off).
3. **Emphasizing Implementation Details**:
   * For each class:
     + Define attributes (e.g., currentTemperature, isOn).
     + Implement methods (e.g., turnOn(), adjustTemperature()).
     + Consider inheritance (e.g., a common base class for all devices).
4. **Ensuring Maintainability and Scalability**:
   * Design for maintainability:
     + Keep classes modular and loosely coupled.
     + Use interfaces or abstract classes for common behavior.
   * Consider scalability:
     + Can we easily add new device types (e.g., humidifiers, dehumidifiers)?
     + Is the design extensible?

**Example Class Diagram (Simplified):**

In this simplified class diagram, we have the base class HVACDevice with common attributes and methods. Concrete classes (AirConditioner, Heater, TemperatureSensor) inherit from it and add specific functionality. The TemperatureSensor measures room temperature.

Remember that real-world systems would be more complex, but this example demonstrates the OOA and OOD process in the HVAC domain.

The below is an example of using PlantUML.. you can consider other tools also.

Syntaxes and way we draw from tool to tool can change. Think about it.

@startuml

class HVACDevice {

- isOn: bool

- currentTemp: float

+ turnOn()

+ turnOff()

+ adjustTemp(temp: float)

}

class AirConditioner {

- coolingPower: int

+ cool()

}

class Heater {

- heatingPower: int

+ heat()

}

class TemperatureSensor {

+ measureTemperature(): float

}

HVACDevice <|-- AirConditioner

HVACDevice <|-- Heater

HVACDevice <|-- TemperatureSensor

@enduml

In this diagram:

* HVACDevice is the base class representing common attributes and methods for all HVAC devices.
* AirConditioner and Heater are concrete classes that inherit from HVACDevice.
* TemperatureSensor is another concrete class that also inherits from HVACDevice.
* Arrows (<|--) indicate inheritance relationships.

**Order of using UML diagrams**

When working with UML (Unified Modeling Language) diagrams in a project, the order of usage can vary based on the project’s needs and the specific development process. However, here are some common steps and the typical order in which UML diagrams are used:

1. **Understand Requirements and Gather Information**:
   * Before creating any UML diagrams, it’s crucial to understand the project requirements, user needs, and system functionality.
   * Gather information about the system’s architecture, components, and interactions.
2. **Use Case Diagrams**:
   * Start with **Use Case Diagrams**.
   * Use case diagrams depict the interactions between actors (users, external systems) and the system.
   * Identify use cases, actors, and their relationships.
   * These diagrams help you understand the system’s high-level functionality and user interactions[1](https://www.lucidchart.com/blog/uml-diagram-templates).
3. **Class Diagrams**:
   * Next, create **Class Diagrams**.
   * Class diagrams represent the static structure of the system by showing classes, their attributes, methods, and relationships.
   * Identify classes, associations, inheritance, and dependencies.
   * Class diagrams serve as the foundation for designing the system’s architecture[2](https://www.geeksforgeeks.org/unified-modeling-language-uml-introduction/).
4. **Sequence Diagrams**:
   * Move on to **Sequence Diagrams**.
   * Sequence diagrams illustrate the dynamic behavior of the system by showing interactions between objects over time.
   * Capture the order of method calls, messages, and lifelines (representing objects or instances).
   * Sequence diagrams help you understand how components collaborate during runtime[3](https://www.geeksforgeeks.org/unified-modeling-language-uml-sequence-diagrams/).
5. **Activity Diagrams**:
   * Consider creating **Activity Diagrams**.
   * Activity diagrams model workflows, processes, and business logic.
   * Show activities, decisions, forks, and joins.
   * Use them to visualize complex processes or use case scenarios[1](https://www.lucidchart.com/blog/uml-diagram-templates).
6. **State Machine Diagrams** (if applicable):
   * If your system has complex state transitions (e.g., a finite state machine), create **State Machine Diagrams**.
   * Represent states, transitions, events, and actions.
   * Useful for modeling behavior that changes based on internal or external events.
7. **Component Diagrams** (if applicable):
   * For larger systems, consider **Component Diagrams**.
   * Show the high-level components/modules, their interfaces, and dependencies.
   * Useful for understanding system architecture and deployment.
8. **Deployment Diagrams** (if applicable):
   * If your system involves distributed components, create **Deployment Diagrams**.
   * Depict hardware nodes, software components, and their connections.
   * Useful for system deployment planning.
9. **Collaboration Diagrams** (optional):
   * Collaboration diagrams (also known as communication diagrams) show interactions between objects.
   * They complement sequence diagrams but focus on object relationships and message flows.
10. **Package Diagrams** (optional):
    * Use **Package Diagrams** to organize related classes into packages or namespaces.
    * Show dependencies between packages.

Remember that the order can be flexible, and some diagrams may be created concurrently or iteratively. Adapt the sequence based on your project’s specific needs and development process.

Let us look at more time, OOA

1. **Understanding the Problem Domain**:
   * OOA starts by thoroughly understanding the problem domain. Think of it as exploring the landscape where your software will operate. For instance, if you’re designing an e-commerce platform, you’d consider aspects like products, customers, orders, payments, and inventory.
   * During this phase, you identify the key **concepts** (or objects) relevant to your system. These concepts become the building blocks for your software.
2. **Creating an Object Model**:
   * OOA involves creating an **object model** that represents the system’s structure. This model consists of classes (blueprints for objects) and their relationships.
   * Each class encapsulates data (attributes) and behavior (methods). For example:
     + A Customer class might have attributes like name, email, and address.
     + Methods within the Order class could handle order placement, cancellation, and tracking.
3. **Identifying Use Cases and Scenarios**:
   * OOA helps you define **use cases**—specific interactions between users and the system. These use cases describe how the system responds to user actions.
   * For our e-commerce system, use cases might include:
     + **Place Order**: A customer selects items, adds them to the cart, and completes the checkout process.
     + **Manage Inventory**: An admin updates product availability and stock levels.
4. **Behavioral Modeling**:
   * OOA captures the dynamic behavior of the system. This includes state transitions, events, and interactions.
   * Consider a scenario where a customer places an order:
     + The system transitions from an initial state (empty cart) to a new state (order placed).
     + Events trigger actions, such as deducting inventory and sending order confirmation emails.
5. **Refining the Model**:
   * OOA is iterative. You refine the model based on feedback, requirements, and changing needs.
   * You might discover new classes, relationships, or constraints during this process.
6. **Validation and Verification**:
   * OOA models undergo validation to ensure they accurately represent the problem domain.
   * Verification involves checking whether the model adheres to the system requirements.

In summary, OOA lays the groundwork for designing a robust, maintainable, and scalable software system.

**Object-Oriented Design (OOD)**:

1. **Identifying Classes and Objects**:
   * OOD begins by identifying the key **classes** and **objects** in your system. Think of classes as categories or types, and objects as instances of those types.
   * For example, in a library management system:
     + Book is a class representing all books.
     + An actual book (e.g., “To Kill a Mockingbird”) is an object.
2. **Responsibilities and Collaborations**:
   * Each class has **responsibilities**—things it should do. These responsibilities define the class’s behavior.
   * OOD focuses on how classes collaborate to achieve system goals. For instance:
     + A Library class collaborates with Book objects to handle borrowing and returning.
3. **Inheritance and Reusability**:
   * OOD uses **inheritance** to create a hierarchy of classes. A subclass inherits properties and behaviors from a superclass.
   * Reusable code is a key goal. If you have common behavior (e.g., Book and DVD both have a title), you can define it in a superclass (e.g., Item).
4. **Polymorphism in Action**:
   * Polymorphism allows objects of different classes to be treated uniformly. It enables flexibility and extensibility.
   * Consider a Payment interface with methods like processPayment(). Different payment methods (credit card, PayPal) implement this interface, allowing seamless handling.
5. **Design Patterns**:
   * OOD leverages design patterns—proven solutions to recurring design problems.
   * Examples:
     + **Singleton**: Ensures a class has only one instance (e.g., a global configuration manager).
     + **Observer**: Allows objects to subscribe and receive updates (e.g., notifying subscribers when a stock price changes).
6. **CRC Cards**:
   * Class-Responsibility-Collaborator (CRC) cards are a brainstorming tool in OOD.
   * Write each class on a card, list its responsibilities, and note collaborators (other classes it interacts with).
7. **Refining the Design**:
   * OOD is iterative. You refine the design based on feedback, performance considerations, and changing requirements.
   * You might discover new classes, relationships, or optimizations during this process.

Choosing frameworks

Using Frameworks

One particularly valuable way of leveraging existing components, whether third-party or developed in-house, is to build within a framework. A framework is a generic architecture that forms the basis for specific applications within a domain or technology area.

A framework differs from a class library in that committing to a framework dictates the architecture of anapplication. Whereas user code that uses a class library handles control flow itself, using class library objectsas helpers, frameworks take responsibility for control flow, calling user code (we've already talked about inversion of control and the Hollywood principle ("Don't call me, I'll call you")). This takes the sameapproach as the Template Method design pattern, but applies it on a much larger scale.

Frameworks differ from design patterns in that:

Frameworks are concrete, not abstract. While design patterns are conceptual, you can take an

existing framework and build an application with it by adding additional code. This normally

takes the form of implementing framework interfaces or subclassing framework classes.

Frameworks are higher-level than design patterns. A framework may use several design patterns.

Frameworks are usually domain-specific or technology-specific, whereas design patterns can be

applied to many problems. For example, a framework might handle insurance quotations, or

provide a clean separation of logic from presentation for web applications. Most design

patterns can be used in just about any application.

Adopting a good framework that is a good fit can slash a project's development time. The toughest design

problems may have been solved, based on recognized best practices. Much of the project's implementation

will be devoted to filling in the gaps, which shouldn't involve so many difficult design decisions.

On the other hand, trying to shoehorn a project into using a framework that is a poor fit will cause serious

problems. The problems will be much worse than choosing an unsuitable class library. In that case, the

library can be ignored: application developers will simply have to develop their own, more suitable, library

functionality. A poorly fitting framework will impose an unnatural structure on application code.

The performance and reliability of the resulting application can also be no greater than that of the

framework. Usually, this is not a problem, as an existing framework is likely to have been widely used in

earlier projects and its reliability and performance characteristics are known, but in all cases it justifies a

thorough quality check of a framework before making a commitment.

What Makes a Good Framework?

Good frameworks are simple to use, yet powerful.

The Scylla and Charybdis of framework design are excessive flexibility and irritating rigidity.

In Greek mythology, Scylla was a sea monster that lived on one side of the Strait of Messia, opposite the

whirlpool Charybdis. Sailors had to chart a course between the two.

Excessive flexibility means that the framework contains code that will probably never be used, and may be

confusing to work with (it will also be harder to test, as there are more possibilities to cover). However, if a

framework isn't flexible enough to meet a particular requirement, developers will cheerfully implement their

own way of doing things, so that the framework delivers little benefit in practice.

Good framework code is a little different to good application code. A good framework may contain complex code:

this is justified if it conceals that complexity from code that uses it. A good framework simplifies application code.

Benefits of Using Existing Frameworks

Generally, it's better to avoid building any but simple frameworks in-house. Open source has flowered over

the past few years, especially in Java, and there are many existing frameworks. Developing good frameworks

is harder than developing applications.

The main benefit of adopting an existing framework is the same as that in adopting J2EE itself: it enables an

organization's development team to focus its effort on developing the required product, rather than

concerning itself with the underlying infrastructure. If the third-party framework is popular, there is also a

potential advantage in the availability of skills working with that framework.

As usual, there's a trade-off: the learning curve in adopting the framework, and a continuing dependency on the

framework. The more complex the project, the easier it is to justify the initial investment and ongoing dependency.

Design Techniques and Coding Standards for J2EE Projects

169

Evaluating Existing Frameworks

Adopting a framework is a very important decision. In some cases, it can determine whether a project succeeds or

fails; in many cases, it will determine developer productivity. As with choosing an application server, it's important to

conduct a thorough evaluation before making a commitment. Remember that even if choosing a framework involves

no license costs (in the case of an open source framework) there are many other costs to consider, such as the impact of

a learning curve on developer productivity and the likely cost of dealing with any bugs in the framework.

I apply the following criteria to evaluating existing frameworks. Applying them in this order tends to limit the

amount of time spent evaluating unsuitable products:

❑ What is the quality of the project documentation?

❑ What is the project's status?

❑ Is the design sound?

❑ What is the quality of the code?

❑ Does the release include test cases?

Let's look at each criterion in turn.

What is the Quality of the Project Documentation?

Is there a coherent – and persuasive – overview document that explains the framework's rationale and

design? Are there Javadocs for all the classes, and do they contain meaningful information?

What is the Project's Status?

If the product is commercial, the main considerations will be the status of the vendor, the place of this

product in the vendor's strategy, and the licensing strategy. There is a real danger in adopting a commercial,

closed source, product that the vendor will shut shop or abandon it, leaving users unsupported. Clearly this is

less likely to happen with a large vendor.

However, large companies such as IBM initiate many projects that don't fit into their longer-term strategy

(consider many of the projects on the IBM Alphaworks site). The viability of the vendor is no guarantee that

they will continue to resource and support any individual project. Finally, especially if the product is

commercial but currently free, does the small print in the license agreement imply that the vendor could

begin to charge for it at any time? Is your organization prepared to accept this?

If the product is open source, there are different considerations. How live is the project? How many

developers are working on it? When was the last release, and how frequently have releases been made? Does

the project documentation cite reference sites? If so, how impressive are they? How active are the project

mailing lists? Is there anywhere to go for support? Are the project developers helpful? The ideal is to have

both helpful developers responding to newsgroup questions and the existence of paid consulting.

Sites such as SourceForge (http://www.sourceforge.net) have statistics on project activity. Other indications

are active mailing lists and searching with your favorite search engine for material on the product.

Many managers have reservations about adopting open source products. Although the quality of projects varies

widely, such reservations are becoming less and less rational. After all, Apache is now the most widely deployed web

server, and has proven very reliable. Several open source Java products are very widely used: for example, the Xerces

XML parser and Log4j. We're also seeing interest from major commercial players such as IBM in open source. Xalan

and Eclipse, for example, are two significant open source projects that were initially developed at IBM.

Chapter 4

170

Is the Design Sound?

The project's documentation should describe the design used (for example, the design patterns and architectural

approach). Does this meet your needs? For example, a framework based entirely on concrete inheritance (such

as Struts) may prove inflexible. Not only might this pose a problem for your code, but it might necessitate

radical changes in the framework itself to add new functionality in the future. If your classes are forced to extend

framework classes, this might require significant migration effort for your organization in future.

What is the Quality of the Code?

This may be time-consuming, but is very important, assuming that the source code is available. Assuming

that the product has satisfied the previous criteria, the investment of time is justified.

Spend half a day browsing the code. Apply the same criteria as you would to code written within your

organization, and look at some of the core classes to evaluate the cleanliness, efficiency and correctness of the

implementation. As an incidental benefit, your team will end up understanding a lot more about the technology

in question and, if the framework is well written, may see some useful design and coding techniques.

Does the Release Include Test Cases?

There are challenges developing reliable software with a community of geographically dispersed developers

communicating via e-mail and newsgroups. One of the ways to assure quality is to develop a test suite.

Successful open source products such as JBoss have large test suites. If an open source product doesn't have a

test suite, it's a worrying sign. If you commit to it, you may find that your application breaks with each new

release because of the lack of regression tests.

Implementing your own Framework

The first rule of developing frameworks in-house is: don't. In general it's better to adopt existing solutions.

However, there are situations where we have unusual needs, or where existing frameworks don't meet our

needs. In this case, it will be better to develop a simple framework than to use an unsuitable existing product

or to code haphazardly without any framework.

Even in this case, it's not a good idea to jump in early. Attempt to design a framework only after you

understand the problem, and then try to design the simplest possible framework. Don't expect that your first

design will be perfect: let the design evolve before making too big a commitment.

Learn from Existing Frameworks

As writing frameworks is hard, successful frameworks are among the most valuable examples of real world

design. Take a close look at successful frameworks in your domain and others, the design patterns they use

and how they enable application code to extend them.

Implementing a Framework

When implementing a framework, it's vital to have clear goals up front. It's impossible to foresee every

requirement in the framework's future, but, unless you have a vision of what you want to achieve, you'll be

disappointed with the results.

Probably the most important lesson of scoping a framework is to deliver maximum value with minimum

complexity. Often we find a situation where the framework can solve most, but not all, of the problems in a

domain fairly easily, but that providing a complete solution is hard. In this case, it may be preferable to settle for

a simple solution to 90% of problems, rather than seek to force a generalization that covers the remaining 10%.

Design Techniques and Coding Standards for J2EE Projects

171

Apply the Pareto Principle if designing a framework. If a particular function seems

particularly hard to implement, ask whether it's really necessary, or whether the

framework can deliver most of its value without tackling this issue.

Writing a framework differs from writing application code in several ways:

❑ The XP advice of "Writing the simplest thing that could possibly work" isn't always appropriate

It's impossible to refactor the interfaces exposed by a framework without breaking code that

uses it and severely reducing its usefulness. Even within an organization, the cost of

incompatible changes to a framework can be very large (on the other hand, it is possible to

refactor the internals of a framework). So the framework must be designed upfront to meet

reasonably anticipated needs. However, adding unneeded flexibility increases complexity.

This balance calls for fine judgment.

❑ Provide different levels of complexity

Successful frameworks provide interfaces on several levels. It's easy for developers to become

productive with them without a steep learning curve. Yet it's possible for developers with

more complex requirements to use more features if they desire. The goal is that developers

should need to handle no more complexity than is required for the task in hand.

❑ Distinguish between framework internals and externals

Externals should be simple. Internals may be more complex, but should be encapsulated.

❑ It's even more important than usual to have a comprehensive test suite

The cost of framework bugs is usually much higher than the cost of application bugs, as one

framework bug may cause many flow-on bugs and necessitate costly workarounds.