Software Engineering Principles

Rigor and formality

- *rigor* means **strict precision**
- *formality* is the **highest degree of rigor**

Separation of concerns

- deal with different aspects of a problem separately => **reduce complexity**
- **separate unrelated concerns**
- consider only the relevant details of a related concern

Modularity

- divide systems into ***modules*** to **reduce complexity**

Cohesion and Coupling

- ***cohesion*** measures interdependence of the elements of one module
- ***coupling*** measures interdependence between different modules
- **maximize cohesion and minimize coupling**

Abstraction

- identify the important aspects and **ignore the details**

Anticipation of change

- **prepare software for changes**
  - modularization - single out elements that are likely to change in the future
  - abstraction - narrow interfaces reduce effects of a change

Generality

- find more **general problem** behind problem at hand**(general solution is more likely to be reusable)**

Incrementality

- characterizes a process which proceeds in a stepwise fashion

Software Development Lifecycle

Object-oriented software development typically includes

- **requirements elicitation**
- **analysis**
- **system design**
- **object design**
- **implementation**
- **testing**

Requirement Elicitation

During **requirements elicitation**, the client and developers define the **purpose of the system**. The result of this activity is a **description of the system in terms of actors and use cases**.

Actors represent the external entities that interact with the system.

Requirement Elicitation Activities

- **identifying actors** - defining the users of the system
- **identifying scenarios**
- **identifying use cases**
- **refining use cases**
- **identifying relationships among use cases**
- **identifying nonfunctional requirements**

Types of requirements

A **functional requirement** is a user **task** that the system must support.

A **non-functional requirement** is a **property** of the system or the domain.

Non-functional requirements

- **quality requirements** - usability, reliability, performance, supportability
- **constraints (pseudo requirements)** - implementation, interface, operations, packaging
- **legal requirements**

Requirements Validation

correctness - completeness - consistency - clarity -realism - traceability

Different Types of Requirements Elicitation

- **Greenfield engineering** - development starts from scratch, **requirements come from users an clients**.
- **Re-engineering** - **redesign** or **reimplement existing system**.
- **Interface engineering** - provide existing services in new environments.

Prioritizing requirements

- **High priority** - addressed during **analysis, design and implementation**.
- **Medium priority** - addressed during **analysis and design**.
- **Low priority** - addressed during **analysis**.

Requirements Specification vs Analysis Model

Both focus on the requirements from the user perspective of the system.

Requirement specification uses **natural language.**

Analysis model uses formal or semi-formal notation(**UML**).

Techniques to elicit requirements

questionnaires - task analysis - **scenarios - use cases**

Scenarios are concrete examples of the future system in use.

Use cases abstractions that describe a class of scenarios.

For each scenario there is a use case and an acceptance test case.

Analysis Model

Model

- **functional** view
  - **use case diagram**
- **object** view
  - **class diagram**
- **dynamic** view
  - **sequence diagram**
  - **state transition diagram**

Analysis object view

Entity objects - the **persistent information** tracked by the system.

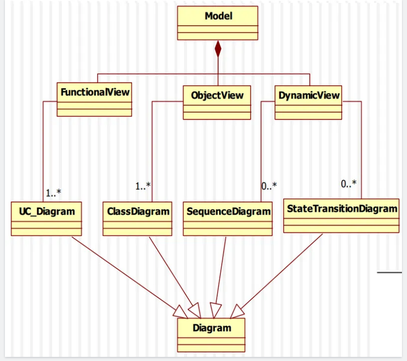
Boundary objects - the **interaction between** the **actors** and the **system**.

Control objects - in charge of **realizing the use cases**, responsible for **coordinating boundary and entity objects**.

In UML these objects are represented by adding <<object_type>> above the name of the object, or with naming conventions.

Analysis Activities

- **identifying** entity, boundary, control **objects**
- **mapping use cases to objects with sequence diagram**
- **identifying associations, aggregates and attributes**
- **modeling state** dependent behavior of individual objects
- **modeling inheritance** relationships
- **review** the analysis model



System Design

System design is the transformation of an analysis model into a system design model.

System Design Activities

- **identifying design goals**
- **decomposing the system into smaller subsystems**
- selecting strategies for building the system
- refining the subsystem decomposition to address the design goals

System design concepts

A **subsystem** is a replaceable part of the system with well defined interfaces, **collection of classes, associations, operations, events and constraints** that are **closely interrelated** with each other.

A **service** is a **set of related operations** that **share a common purpose**. We define the subsystems in terms of the services they provide.

Layering allows a system to be organized as a hierarchy of subsystems. A **layer only depends on services from lower layers**. A **layer has no knowledge of higher layers**.

Partitioning organizes subsystems as peers that mutually provide different services to each other. **Partitions provide services to other partitions on the same layer**.

How the Analysis Models influence System Design

Non-functional requirements => **design goals**

Functional model => **subsystem decomposition**

Object model => **persistent data management**

Dynamic model => **identification of concurrency**, global resource handling, software control

Services and Subsystem Interfaces

A **subsystem interface** is the set of operations of a subsystem that are available to other subsystems.

Subsystem interfaces can be depicted in UML with **ball-and-socket** (assembly connectors). The **interface** is shown as a **ball icon** (lollipop) and **the one requiring the interface** is shown as a **socket icon**.

Relationships between Subsystems

Layer relationships

- **compile time dependency (associations with solid lines in UML)**
- **run time dependency (associations with dashed lines in UML)**

Partition relationship

- **peer-to-peer** (the subsystems have mutual knowledge about each other, they can call each other).

Virtual machine

The term **virtual machine** and **layer** can be used interchangeably.

Layered Architectures

- **Closed Architecture (Opaque Layering)** - each layer can only call operations from the layer bellow => maintainability and flexibility.
- **Open Architecture (Transparent Layering)** - each layer can call operations from any layer bellow => runtime efficiency.

Object Design

The purpose of **object design** is to **prepare for the implementation** of the system model based on design decisions. It **serves as the basis of implementation**.

Object Design Activities

- **reuse**
- **interface specification** - the subsystem services identified during system design are described in terms of class interfaces, including operations, arguments, type signatures, and exceptions. The subsystem service specification is often called subsystem **API** (Application Programmer Interface).
- **restructuring** - address design goals such as maintainability, readability, and understandability of the system model.
- **optimization** - address **performance requirements** of the system model.

Customization: Build custom objects

Composition - Black Box Reuse - new functionality is obtained by aggregation.

Inheritance - White box Reuse - new functionality is obtained by inheritance.

Delegation

Delegation is the **alternative to** implementation **inheritance** that should be **used when reuse is desired**.

A class is said to delegate to another class if it implements an operation by referring another operation of the other class.

Delegation makes explicit the dependencies between the reused class and the new class.

Object Design Activities

1. Reuse: Identification of existing solutions
   - Use of inheritance
   - Off-the-shelf components and additional solution objects
   - Design patterns
2. Interface specification
   - Describes precisely each class interface

Object Design

3. Object model restructuring
   - Transforms the object design model to improve its understandability and extensibility
4. Object model optimization
   - Transforms the object design model to address performance criteria such as response time or memory utilization.

Mapping Models to Code

Testing

Testing is the **process of finding differences between the expected behavior** specified by system models **and the observed behavior of the implemented system**.

The **goal of testing** is to design tests that **reveal problems**.

Types of Errors

Failure is any deviation of the observed behavior from the specified behavior.

Erroneous state/error means the system is in a state such that further processing by the system will lead to a failure.

Fault/defect/bug is the mechanical or algorithmic cause of an erroneous state.

Testing Activities

Usability testing tries to find faults in the user interface design of the system.

Unit testing finds differences between a specification of an object and its realization as a component. Tests an individual component.

Integration testing is the activity of finding faults by testing individual components in combination. Tests groups of components.

Structural testing is the culmination of integration testing involving all components of the system. It finds differences between the system design model and a subset of integrated subsystems.

System testing tests all the components together, seen as a single system to identify faults with respect to the scenarios from the problem statement and the requirements and design goals identified in the analysis and system design, respectively:

- **Functional testing** finds differences between the use case model and the system.
- **Performance testing** finds differences between nonfunctional requirements and actual system performance.
- **Acceptance testing** and **installation testing** check the system against the project agreement and is done by the client.

Testing Concepts

Test component is a part of the system that can be isolated for testing. A component can be an object, a group of objects, or one or more subsystems.

A **test case** is a set of inputs and expected results that exercises a test component with the purpose of causing failures and detecting faults. It has 5 attributes: name, location, input, oracle and log.

A **test stub** is a partial implementation of components on which the tested component depends.

A **test driver** is a partial implementation of a component that depends on the test component.

Test stubs and drivers enable components to be isolated from the rest of the system for testing.

White and Black box testing

Black-box tests focus on the input/output behavior of the component. Tests the functionality of the component.

White-box tests focus on the internal structure of the component. A white-box test makes sure that all branches and statements are tested.

Dealing with Faults

testing - debugging - monitoring

running time + exception

Observations

It is impossible to completely test any nontrivial module or system. Testing can only show the presence of bugs, not their absence.

Testing Order

1. Requirement elicitation and analysis => System testing
2. System design => Integration testing
3. Object design => Unit testing

Integration testing strategies

Big bang - Bottom up - Top down - Sandwich - Modified sandwich - Continuous testing

Diagrams

A **diagram** is a set of graphical rules for depicting views and formal specification.

Use Case Diagram (UCD)

Use case diagrams (UCD) - represent the **functionality of the system from user's point of view**. Use cases **can also be described textually** which consists of **6 parts**: *unique name, participating actors, entry conditions, exit conditions, flow of events, special requirements.*

Use case associations

Use **include relation**:

- for behavior common to 2 or more use cases
- when a function is too complex
- when there is a need for reusing a use case

Use **extend relation** for **exceptions, optional, or rare behavior**.

Use **generalization** when we want to factor out common but not identical behavior.

Class Diagram

Class diagrams - represent the **structure of the system**.

Sequence Diagram

Sequence diagrams - describe the **behavior as interactions between objects**. Used for mapping use cases to objects.

Guide for sequence diagrams

- **first** column should be the **actor** who initiated the use case.
- **second** column should be the **boundary** object.
- **third** column should be the **control** object that manages the UC.
- control objects are created by boundary objects initiating UCs.
- boundary objects are created by control objects.
- **entity objects** are **accessed by control and boundary** objects, and **not the other way**.

State Transition Diagram

State transitions diagrams - describe **behavior by means of states and transitions**.

Components:

- **pseudo states, composite states, simple states**
- **transitions - events, activities, guards**

Object Diagram Snapshot

Object diagram snapshot - represents an **instantiation** of a class diagram.

Collaboration diagrams

Collaboration diagram - an instance diagram that **visualizes the interactions between objects as a flow of messages**. **Messages** can be **events** or **calls to operations**.

Communication diagrams **describe the static structure as well as the dynamic behavior of a system**.

Messages between objects are **labeled** with a **chronological number** and placed near the link the message is sent over.

Activity diagrams

Components - activities, forks, completion transition, join

Component diagrams

Deployment diagram

System Architecture diagram

Architectural Styles

Architectural style is a **pattern for a subsystem** decomposition and a **software architecture** is an **instance of an architectural style**.

Client/Server

One or many **servers provide services to instances of subsystems**, called **clients**.

The **clients know the interface** of the server. The **server does not need to know the interface** of the client.

Design goals: service portability, location-transparency, high performance, scalability, flexibility, reliability.

Peer-to-Peer

Specialization of Client/Server architectural style. **Clients can be servers and servers can be clients (peer is client and/or server)**.

Repository

Subsystems access and modify data from a single data structure called the repository.

Model-View-Controller

Subsystems are classified into 3 different types

- **model** - responsible for application domain knowledge
- **view** - responsible for displaying application domain objects
- **controller** - responsible for sequence of interactions with the user and notifying views of changes in the model

Three-Tier

Consists of **3** hierarchically ordered subsystems: **user interface(presentation layer), middleware(business logic) and a database system**.

Four-Tier

Consists of **4** hierarchically ordered subsystems: **web browser, web server, application server, database**.

Pipes and Filters

A **pipeline** consists of a chain of processing elements (processes, threads, etc.), arranged so that the output of one element is the input to the next element.

An architectural style that consists of two subsystems called pipes and filters.

Filter: A subsystem that does a processing step

Pipe: A Pipe is a connection between two processing steps

Each filter has an input pipe and an output pipe. The data from the input pipe are processed by the filter and then moved to the output pipe.

Design Patterns

Adapter

Adapter pattern works as a **bridge between two incompatible interfaces**.

Proxy

Proxy pattern lets you **provide a substitute or placeholder for another object**. A proxy **controls access to the original object**, allowing you to perform something either before or after the request gets through to the original object.

The ProxyObject class acts on behalf of a RealObject class. Both classes implement the same interface. The ProxyObject stores a subset of the attributes of the RealObject.

Bridge

Bridge pattern is used when we need to decouple an abstraction from its implementation so that the two can vary independently.

This pattern involves an interface which acts as a bridge which makes the functionality of concrete classes independent from interface implementer classes. Both types of classes can be altered structurally without affecting each other concrete classes.

Composite

Composite pattern lets you **compose objects** into **tree structures** and then work with these structures as if they were individual objects.

Command

Command pattern turns a request into a stand-alone object that contains all information about the request. This transformation lets you pass requests as a method arguments, delay or queue a request's execution, and support undoable operations.

Observer

Observer pattern lets you define a subscription mechanism to notify multiple objects about any events that happen to the object they're observing.

Strategy

Strategy pattern lets you define a family of algorithms, put each of them into a separate class, and make their objects interchangeable.