Implementation

Software implementation is the stage in which a functioning software system is created. Design and implementation are often interleaved.

It is often possible to by off-the-shelf systems that can be adapted to the user's requirements. When developing an application this way, the primary design focus is how to properly configure the system to the user's needs.

Object oriented design involves developing a number of different system models. These models require a lot of effort to develop, and thus may not be cost-effective for some systems. For larger systems however, these models serve as an important mechanism for communicating design ideas.

Common steps in the object oriented design process include:

- Defining context
- Designing system architecture
- Identifying principle system objects
- Developing design models
- Specifying object interfaces

Understanding the relationships between software and its environment is crucial for deciding deciding how to design and implement it. It also allows you to establish the boundaries of the system, which helps in deciding with features get implemented in which system.

A **system context model** is a structural model that demonstrates the other systems that exist in a given environment.

An **interaction model** is a dynamic model that shows how the system interacts with its environment.

Once the interactions between a system and its environment are determined, you can effectively design the architecture.

Identifying object classes is often difficult, as there exists no standard method of identifying relevant objects, instead requiring experience and knowledge of the domain you are working in; the design is rarely correct on the first go.

Several approaches to identifying object classes exist:

- Using natural language in identification
- Identifying based on tangible objects in the domain
- Using a behavioral approach
- Scenario-based analysis

Design models show the relations between the objects and their respective classes. Two types of design models exist:

- Structural models that describe static relationships
- Dynamic models that describe dynamic relationships.

Subsystem Models

This type of model shows how related objects are grouped together. In UML, these groups are represented by encapsulation.

Sequence Models

This type of model shows the interactions that take place between objects over time.

State Diagrams

This type of model represents the possible states that objects can reach in a given system.

Design Patterns

A design pattern is a way of reusing abstract knowledge about a problem and its given solution. They should be abstract enough to be used in different settings.

Patterns are ways to describe best practices and good design principles. A pattern consists of the following:

- A name
- A problem description
- A solution description
- Consequences of applying the pattern

Observer Pattern

This pattern separates the representation of an object's state from the object itself. It is used when multiple different displays of a single object's state are required.

Implementation Issues

Reuse, configuration management, and host-target development are all important considerations when implementing software.

Code reuse wasn't common in the early days of programming, but this approach became nonviable as the scale of software continued to grow. There are different levels of code reuse:

- Abstraction
- Object
- Component
- System

The cost of code reuse includes time spent looking for suitable software and assessing whether it meets your needs. Sometimes the software costs money. Adapting and configuring the software, and integrating multiple pieces of reused software are also considerations.

Configuration management is the process of managing a changing software system. This process aims to allow developers to access the source code in a controlled way. It includes version management, system integration, and problem tracking.

Testing

Testing shows that a program does what it is intended to do, or alternatively, exposes that a program does not do what it is intended to do. Executes using artificial data. Can reveal the presence of errors but not their absence.

The goals of testing include demonstrating that software meets its requirements and to discover incorrect behavior of the software. Respectively, these are known as validation testing and defect testing.

Verification is the process of ensuring the product is right, while validation is the process of ensuring the right product.

Verification and validation establishes confidence that the system is fit for its purpose. This depends on the system's purpose, user expectations, and the system's marketing environment.

Software inspection is static verification, while software testing is dynamic verification. Inspections involve examining source code. Some advantages of inspections include limited interactions between errors, ability to inspect incomplete systems, as well as revealing broader quality issues. Both inspections and tests should be used during V & V.

Stages of testing include development testing, release testing, and user testing.

Development Testing

Includes all testing that is carried out by the development team.

- Unit testing tests individual components in isolation. May test individual functions, entire object classes, or even larger components.
- Object class testing involves testing multiple associated functionalities together.

Testing should be automated whenever possible. This is often accomplished using a testing framework. Components of automated tests include a setup, calling the test, and an assertion that ultimately determines whether the test passes.

Unit tests should show that a component does what it is supposed to. Any defects should be revealed by test cases. One type of unit test reflects normal operation, while the other demonstrates possible problems that could arise.

Testing strategies include partition testing and guideline-based testing.

Partition Testing

Data is grouped into equivalence partitions, where if a test case passes for one element in the partition, then it should pass for all elements in that partition.

Testing guidelines include:

- testing software with single element sequences
- testing software with sequences of different sizes
- ensuring that the first, middle, and last elements of the sequence are accessed
- testing with sequences of length zero

General testing guidelines include choosing inputs that force the system into an error state, e.g. overflowing buffers, generate error messages, etc.

Testing composite components should focus on showing that the component interface behaves according to its specification.

Interface types: Parameter, shared memory, procedural, message passing. Interface errors include misuse, misunderstanding, and timing errors.

When testing, ensure that parameters are at the extreme ends of their ranges, always test pointers as null pointers, design tests that cause the component to fail, stress testing, and vary the order in which components are activated.

System testing integrates components, focusing on their interactions, compatibility, and emergent behavior. During system testing, reused components are integrated with newly developed ones, and then tested.

Use-case testing bases tests on use-cases, forcing interactions to occur.

Inputs should always have acknowledgements as reports.

Test policies bypass the impossibility of testing exhaustively.

Test-driven Development

Interleaves testing and code development. Tests are written before code, and in increments. Originated in agile methods.

Test-driven development involves identifying the functionality, writing a test for it, running the test, implementing functionality, and then moving on.

Benefits of TDD are code coverage, regression testing, simpler debugging, and automatic system documentation.

Regression testing ensures that new changes don't break previous work. This is simple with automated tests.

Release testing focuses on a particular release of a system. Convinces the supplier that the system is good enough to use. Tests are usually black-boxed, being derived from system specifications. This is a form of system testing, but a separate team should be responsible for release testing.

Requirements based testing involves adapting test cases to the requirements.

Performance testing focuses on performance and reliability. Tests should reflect the use profile of the system. Often incrementally overloads the system.

User testing involves real users providing input, and is essential, even if other comprehensive tests have been run.

Alpha testing, beta testing, and acceptance testing are types of user testing.

Stages in acceptance testing include:

- Defining acceptance criteria
- Planning acceptance testing

- Deriving acceptance tests
- Running acceptance tests
- Negotiating results
- Rejecting or accepting the system

In agile methods, the user or customer is part of the development team and plays a role in making decisions. Main issue is whether the embedded user is "typical" and can represent the interests of all stakeholders.

Evolution

Software change is inevitable. Now requirements emerge, business changes, errors need to be fixed, new equipment is added to the system, and performance or reliability may need to be improved. This change is a key problem for all organizations.

Organization have investment in their software, so they must be updated when appropriate. Most of the budget goes towards changing existing software.

Spiral model: Specification, Implementation, Validation, Operation, repeat.

Software evolution is when a software system is deployed and changing to its environment. Software servicing is when changes stop being made but the software is still usable. Eventually, software is retired and changes stop.

Evolution Processes

Evolution depends on the type of software, development processes used, and the experiences of the developers. Change proposals drive evolution, and should be linked with components that are affected by the change.

Change identification process, change proposals, software evolution, new system, repeat.

Change implementation involves designing, implementing, and testing changes. Program understanding involves knowing how the program is structured and how changed might affect it.

Urgent changes may be needed that must skip some stages of the software engineering process. Change request, analyze source code, modify source code, and deliver change.

Agile methods seamlessly transition from development to evolution. Regression testing is also valuable when changes are made to a system. Additional user stories.

Sometimes there are handover issues, e.g. if two teams use different development methods.

Legacy Systems

These systems rely on technology that is no longer used for new projects. May be dependent on older hardware or software.

The components of a legacy system include:

- System hardware
- Support software
- Application software
- Application data
- Business processes
- Business policies and rules

Legacy system replacement is risky and expensive due to a lack of complete specification, tight integration of system and business processes, undocumented business rules embedded in legacy code, and the fact that new software may be late or over budget.

No consistent style, use of obsolete languages, inadequate documentation, structure degradation, obfuscating optimizations, and data errors.

Thus organizations that rely on legacy systems must develop a strategy for their maintenance. Scrapping the system, maintaining the system, transforming the system, or replacing it are all options.

Business value assessment should take into account the end-users, business customers, line managers, IT managers, and senior managers.

Issues in business value assessment include systems that are only occasionally used, whether efficient business processes are supported, system dependability, and system outputs.

Business process assessment, environment assessment, and application assessment.

Business process assessment uses a viewpoint-oriented approach to seek answers from stakeholders.