Nature of software

Software is intangible, hard to understand development effort

Software is easy to reproduce Cost is in its development − in other engineering products, manufacturing is the − in other engineering products, manufacturing is the costly stage, the industry is labour-intensive, Hard to automate

Defining Software

Software is instructions that when executed provide the desired features, function and performance, data structures that enable the programs to adequately manipulate information, descriptive information in both hardcopy and virtual forms

Software has one fundamental characteristic, it does not wear out. IT is not susceptible to environmental maladies .However software deteriorates.

Software Application domains

**System software** – a collection of programs written to service other programs.

**Application software** – Standalone programs that solve a specific business need.

**Engineering and scientific software –** algorithms used in science and engineering studies such as volcanogy or astronomy.

**Embedded software** – resides in a product or system and is used to implement and control features.

**Product line software** - designed to provide a specific capability for use by customers.

**Web/mobile applications** – this network-centric software category spans a wide array of applications that reside on mobile devices.

**Artificial intelligence software** – makes use of non-numerical algorithms to solve complex problems.

Legacy Software

In computing, a legacy system is an old method, technology, computer system, or application program.

* Legacy systems often evolve for one or more of the following reasons:
* The software must be adapted to meet the needs of the new computing environments.
* The software must be enhanced to implement new business requirements
* The software must be extended to make it interoperable with other more modern systems
* The software must be re-architected to make it viable within an evolving computing environment.

Changing nature of software

Four broad categories of software are evolving to dominate the industry.

**WebApps**

**Mobile applications**

**Product-line software**

**Cloud computing** – encompasses an infrastructure or ecosystem that enables any user, anywhere to use a computing device.

Defining software engineering.

The application of a systematic disciplined, quantifiable approach to the development, operation and maintenance of software and the study of the approaches of such.

**Software process layers:**

**A quality focus –** software engineering focuses on producing products that meet a certain required (qualitative) standard.

**Process –** forms a basis for management and control of software projects.

**Methods** – encompass a broad array of tasks that include communication, requirements analysis, design modelling, construction, testing and support.

**Tools** – provide automated or semi-automated support.

The software process

1. **Communication**

In communication phase the major task performed is requirement gathering which helps in finding out exact need of customer. Once all the needs of the customer are gathered the next step is planning.

2. **Planning**

In planning major activities like planning for schedule, keeping tracks on the processes and the estimation related to the project are done. Planning is even used to find the types of risks involved throughout the projects.  Planning describes how technical tasks are going to take place and what resources are needed and how to use them.

3. **Modelling**

This is one the important phases as the architecture of the system is designed in this phase. Analysis is carried out and depending on the analysis a software model is designed. Different models for developing software are created depending on the requirements gathered in the first phase and the planning done in the second phase.

4. **Construction**

The actual coding of the software is done in this phase. This coding is done on the basis of the model designed in the modelling phase. So in this phase software is actually developed and tested.

5. **Deployment**

In this last phase the product is actually rolled out or delivered & installed at customer’s end and support is given if required. A feedback is taken from the customer to ensure the quality of the product.

Umbrella activities

**Typical activities in this category include:**

01. Software project tracking and control:

Tracking and Control is the dual process of detecting when a project is drifting off-plan, and taking corrective action to bring the project back on track. But a successful project manager will also be able to tell when the plan itself is faulty, and even re-plan the project and its goals if necessary.

02. Formal technical reviews:

This includes reviewing the techniques that has been used in the project.

03. Software quality assurance:

This is very important to ensure the quality measurement of each part to ensure them.

04. Software configuration management:

In software engineering, software configuration management (SCM or S/W CM) is the task of tracking and controlling changes in the software, part of the larger cross-disciplinary field of configuration management. SCM practices include revision control and the establishment of baselines.

05. Document preparation and production:

All the project planning and other activities should be hardly copied and the production get started here.

06. Reusability management:

This includes the backing up of each part of the software project they can be corrected or any kind of support can be given to them later to update or upgrade the software at user/time demand.

07. Measurement:

This will include all the measurement of every aspects of the software project.

08. Risk management:   
Risk management is a series of steps that help a software team to understand and manage uncertainty. It’s a really good idea to identify it, assess its probability of occurrence, estimate its impact, and establish a contingency plan that─ ‘should the problem actually occur’.

A linear process flow executes each of the five framework activities in sequence, beginning with communication and culminating with deployment.

Process flows

A **linear process flow** executes each of the five framework activities in sequence, beginning with communication and culminating with deployment.

An **iterative process flow** repeats one or more of the activities before proceeding to the next.

An **evolutionary process flow** executes the activities in a “circular” manner.

A **parallel process flow** executes one or more activities in parallel with other activities.

Prescriptive process models

Incremental build model

The **incremental build model** is a method of [software development](https://en.wikipedia.org/wiki/Software_development) where the product is [designed](https://en.wikipedia.org/wiki/Software_design), implemented and [tested](https://en.wikipedia.org/wiki/Software_testing) incrementally (a little more is added each time) until the product is finished. It involves both development and maintenance

The incremental model can be used when developing web applications where:

The incremental model is used when the requirements are clearly defined and understood.

A new technology is being used

There are some high risk features and goals.

Evolutionary process models

Prototyping

**Software prototyping** is the activity of creating [prototypes](https://en.wikipedia.org/wiki/Prototype) of software applications, i.e., incomplete versions of the [software program](https://en.wikipedia.org/wiki/Software) being developed

It begins with communication, meet with the stakeholders and define the overall objectives of the software.

A prototyping iteration is planned quickly and modelling occurs

A quick design leads to the construction of the prototype.

The prototype is deployed and evaluated by stakeholders

Prototyping is used where the user is unsure of the efficiency of the algorithm. Or the adaptability of the operating system.

Spiral Model

Couples the iterative nature of prototyping with the controlled and systematic aspects of the waterfall model, it uses prototyping as a risk reduction mechanism.

The spiral model is used in the development of software when:

* When project is large
* When releases are required to be frequent
* When creation of a prototype is applicable
* When risk and costs evaluation is important

Concurrent Models

The concurrent process model defines a series of events that will trigger transitions from state to state for each of the software engineering activities.

The concurrent process model is applicable to all types of software development and provides an accurate picture of the current state of a project.

Specialized process models

Component based development

It is evolutionary in nature and incorporates many of the characteristics of the spiral model, it demands an iterative approach to the development of software.

It incorporates the following steps:

* Available component based products are researched for the application domain.
* Component integration issues are considered.
* A software architecture is designed to accommodate the components.
* Components are integrated into the architecture.
* Comprehensive testing is conducted to ensure proper functionality.

Formal methods model

Formal methods are mathematical techniques for developing computer-based software and hardware systems.

They enable one to discover and correct errors that might otherwise go undetected.

* Concern about applicability in the business environment:
* The development of formal models is currently quite time consuming.
* Extensive training is required because few software developers have the necessary background to apply the formal methods model.
* It is difficult use the models as a communication mechanism for technically unsophisticated customers.

Aspect oriented software development

**A***spect-***o***riented* **s***oftware* **d***evelopment (***AOSD***)* is an approach to software development that addresses limitations inherent in other approaches, including object-oriented programming. AOSD aims to address crosscutting concerns by providing means for systematic identification, separation, representation and composition.

Crosscutting concerns are encapsulated in separate modules, known as aspects, so that localization can be promoted. This results in better support for modularization hence reducing development, maintenance and evolution costs.

Unified process model

This is an attempt to draw on the best features and characteristics of traditional software process model. It recognizes the important of customer communication and streamlined methods for describing the customer’s view of a system.

Phases of the unified process model:

Inception – encompasses both customer communication and planning activities, by collaborating with stakeholder’s business requirements for the software are identified.

Elaboration – encompasses the planning and modelling activities of the generic process model, it refines and expands the preliminary use cases of the inception phase.

Construction - develops the software components that will make each use case operational for the end user.

Transition – encompasses the later stages of the generic constriction activity, software is given to end users for beta testing and feedback is given do that necessary changes can be implemented.

Production – coincides with the deployment activity of the generic process

Specialized process models

The personal software process

The personal software process (PSP) emphasizes personal measurement of both the work product that is produced and the resultant quality of the work product.

It defines five framework activities:

Planning: Isolates requirements and develops both size and resource estimates.

High level design: external specifications for each component to be constructed are developed.

High-level design review: formal verification methods are applied to uncover errors in the design.

Development: Code is generated, reviewed, compiled and tested.

Post-mortem: The effectiveness of the process is determined. Measures and metrics should provide guidance for modifying the process to improve effectiveness.

The team software process

The goal of TSP is to build a “self-directed” project team that organizes itself to produce high-quality software.

Objective for TSP:

Build self-directed teams that plan and track their work.

Show managers how to coach and motivate their teams.

Accelerate software process improvement

Facilitate university teaching of industrial-grade team skills.

What is an Agile Process?

An agile software process is a process with a more than effective response to change.

Extreme process

Uses an object oriented approach as its preferred development paradigm, and it encompasses a set of rules and practices that occur within the context of: planning, design, coding and testing.

The planning activity begins with listening, a requirements gathering activity that enables technical members of the XP team to understand the business context of the software.

Design follows the KIS principle. A simple design is best, CRC used as a mechanism for thinking about software, CRC encourages the immediate creation of an operational prototype, if a design problem is encountered.

Coding Unit test are done before moving on to code, once unit tests are done, the developer is better able to focus on what must be done to pass the test. Pair programming is used here.

Testing commences where the complete software is tested to see if it meets the set requirements.

Extreme process is used when the customers do not have a firm idea of what the system should do. It is also used when there is too much project risk involved in the project.

Industrial XP

This is an organic evolution of XP, it has a greater inclusion of management and an expanded role for customers.

IXP incorporates six new practices that are designed to ensure that an XP project works successfully.

* Readiness assessment – ascertains whether all members of the project community are on board and understand the skill levels involved.
* Project community – determines where the right people with the right skills and training have been staged for the project.
* Project chartering – The IXP team assesses the project itself to determine whether an appropriate justification for the project exists.
* Test driven management – An IXP team establishes a series of measurable destinations that assess progress to date.
* Retrospectives – The IXP team conducts a specialized technical review after a software increment is delivered.
* Continuous learning – The IXP team is encouraged to learn new methods and techniques that can to higher quality product.

Industrial Extreme process is also used when the customers do not have a firm idea of what the system should do. It is also used when there is too much project risk involved in the project.

Dynamic systems development method

Is an agile development approach that provides a framework for building and maintaining systems which meet tight time constraints through the use of incremental prototyping.

DSDM then defines three different iterative cycles:

Functional model iteration – provides a set of incremental prototypes that demonstrate functionality for the customer.

Design and build iteration – revisits prototypes built during the functional model iteration.

Implementation – places the latest software increment into the operational environment.

DSD process is also used when the customers do not have a firm idea of what the system should do. It is also used when there is too much project risk involved in the project.

Agile Modelling

|  |
| --- |
| Agile Modelling (AM) is a practice-based methodology for effective modelling and documentation of software-based systems.  Principles that make up AM are:  Model with purpose - a developer who uses AM should have a specific goal.  Use multiple models:  Travel light - As software engineering work proceeds keep only those models that will provide long term value and jettison the rest. |

Content is more important than representation – modelling should impart in formation to its intended audience.

Adapt locally – The modelling should be adapted to the needs of the agile team.

Agile Unified Process

It describes a simple, easy to understand approach to developing business application software using agile techniques.

Each AUP iteration addresses the following activities.

Modelling: UM - representations of the business and problem are created.

Implementation - Models are translated into source code.

Testing – A series of unit tests are designed and executed to uncover errors and ensure that source code meets requirements.

Deployment – this focuses on delivery of the software increment and acquisition of feedback from end users.

Configuration and project management: Configuration management addresses change management, risk management and control of any persistent work products.

Environment management – coordinates a process infrastructure that includes standards, tools, and other support technology available to the team.

The Agile unified process is also used when the customers do not have a firm idea of what the system should do. It is also used when there is too much project risk involved in the project.

Characteristics of a software engineer

Individual responsibility – implies the drive to deliver his/her promises to peers.

Resilience under pressure – a software engineer is able to manage pressure so that his/her performance does not suffer.

Brutally honest – if the software engineer sees flaws in design he/she points them out.

Attention to detail- he/she carefully considers the technical decisions he/she makes on a daily basis.

Heightened sense of fairness – he/she gladly shares fairness with colleagues.

Psychology of software engineering

Software engineering psychology focuses on the recognition of the problem to be solved, the problem-solving skills required to solve it and the motivation to complete the solution.

The following roles can be assigned or can evolve naturally.

Ambassador – m represents the team to outside constituencies.

Scout – crosses the team’s boundary to collect organizational information.

Guard – protects access to the teams work.

Sentry – controls the flow of information that stakeholders and others send to the team.

Coordinator – focuses on communicating horizontally across the team.

Team structures

The best team structure depends on the management style of your organization, the number of people that will populate the team and their skill levels.

Four organizational paradigms are suggested for software engineering teams.

Closed paradigm – structures a team loosely and depends on individual initiative of team members.

Random paradigm – structures a team loosely and depends on individual initiative of the tam structures.

Open paradigm – attempts to structure a team in a manner that achieves some controls associated with the closed paradigm

Synchronous paradigm - relies on the natural compartmentalization of a problem.

Agile Teams

**The generic agile team.**

An agile team consists of a small group of dedicated individuals, who together have skills necessary to, define, build, and test solution value all in a short iteration time box

**The XP team.**

XP teams are *self-organizing* and *cross-functional*. This has two important consequences: first, they're responsible for their own success. This means teams define success (by interviewing stakeholders and sponsors), create plans to achieve success, and execute on those plans without explicit management direction.

Second, XP teams include all the expertise necessary to do so

The Impact of social media

Software developers rely on media to communicate, learn, collaborate, and coordinate with others. Recently, social media has dramatically changed the landscape of software engineering, challenging some old assumptions about how developers learn and work with one another. We see the rise of the social programmer who actively participates in online communities and openly contributes to the creation of a large body of crowdsourced socio-technical content

Software engineering using the cloud

Cloud computing can enable or facilitate software engineering activities through the use of computational, storage and other resources over the network.

Collaboration tools

Tools are essential to collaboration among team members.

**Services designed to enhance collaborative work:**

Namespace – allows the project team to store all work products.

A calendar – for coordinating meetings and other project events.

Templates – enable team members to create work products that have a consistent look and structure.

Metrics support – track each team member’s contribution.

Communication analysis – that tracks communication across the team.

Global Teams

Accentuates

Reduces

Complicates

Accentuates the need for

Coordination

Collaboration

Communication

Distance

Barriers and complexity

Improves

Enhances

The intent of this chapter is to provide an introduction to the design process and to describe fundamental design concepts that are essential to an understanding of any software design method. Basic concepts are introduced and a fundamental design model is discussed. The design model consists of the data design, architectural design, interface design, and component-level design.

The goal of design engineering is to produce a model or representation that exhibits firmness, commodity, and delight.

To accomplish this, a designer must practice diversification and then convergence.

Belady states that “diversification is the acquisition of a repertoire of alternatives, the raw material of design: components, component solutions, and knowledge, all contained in catalogues, textbooks, and the mind.”

Once this diverse set of information is assembled, the designer must pick and choose elements from the repertoire that meet the requirements designed by requirement engineering and the analysis models.

As this occurs, alternatives are considered and rejected, and the design engineer converges on “one particular configuration of components, and thus the creation of the final product.”

9.1 Design within the Context of Software Engineering

Software design is the last software engineering action within the modelling activity and sets the stage for construction (code generation and testing).

The flow of information during software design is illustrated in Figure below. The analysis model, manifested by scenario-based, class-based, flow-oriented and behavioural elements, feed the design task.

The architectural design defines the relationship between more structural elements of the software, the architectural styles and design patterns that can be used to achieve the requirements defined for the system, and the constraints that affect the way in which the architectural design can be implemented.

The architectural design can be derived from the System Specs, the analysis model, and interaction of subsystems defined within the analysis model.



The interface design describes how the software communicates with systems that interpolate with it, and with humans who use it. An interface implies a flow of information (data, and or control) and a specific type of behaviour.

The component-level design transforms structural elements of the software architecture into a procedural description of software components.

The importance of software design can be stated with a single word – quality. Design is the place where quality is fostered in software engineering. Design provides us with representations of software that can be assessed for quality. Design is the only way that we can accurately translate a customer’s requirements into a finished software product or system.

9.2 Design Process and Design Quality

Software design is an iterative process through which requirements are translated into a “blueprint” for constructing the software.

Initially, the blueprint depicts a holistic view of software, i.e. the design is represented at a high-level of abstraction.

Throughout the design process, the quality of the evolving design is assessed with a series of formal technique reviews or design walkthroughs.

Three characteristics serve as a guide for the evaluation of a good design:

The design must implement all of the explicit requirements contained in the analysis model, and it must accommodate all of the implicit requirements desired by the customer.

The design must be a readable, understandable guide for those who generate code and for those who test and subsequently support the software.

The design should provide a complete picture of the software, addressing the data, functional, and behavioural domains from an implementation perspective.

Quality Guidelines

In order to evaluate the quality of a design representation, we must establish technical criteria for good design.

A design should exhibit an architecture that:

(1) Has been created using recognizable architectural styles or patterns,

(2) Is composed of components that exhibit good design characteristics, and

(3) Can be implemented in an evolutionary fashion

For smaller systems, design can sometimes be developed linearly.

A design should be modular; that is, the software should be logically partitioned into elements or subsystems

A design should contain distinct representations of data, architecture, interfaces, and components.

A design should lead to data structures that are appropriate for the classes to be implemented and are drawn from recognizable data patterns.

A design should lead to components that exhibit independent functional characteristics.

A design should lead to interfaces that reduce the complexity of connections between components and with the external environment.

A design should be derived using a repeatable method that is driven by information obtained during software requirements analysis.

A design should be represented using a notation that effectively communicates its meaning.

Quality Attributes

Hewlett-Packard developed a set of software quality attributes that has been given the acronym FURPS. The FURPS quality attributes represent a target for all software design:

Functionality: is assessed by evaluating the features set and capabilities of the program, the generality of the functions that are delivered, and the security of the overall system.

Usability: is assessed by considering human factors, overall aesthetics, consistency, and documentation.

Reliability: is evaluated by measuring the frequency and severity of failure, the accuracy of output results, the mean-time-to-failure, the ability to recover from failure, and the predictability of the program.

Performance: is measured by processing speed, response time, resource consumption, throughput, and efficiency.

Supportability: combines the ability to extend the program extensibility, adaptability, serviceability 🡺 maintainability. In addition, testability, compatibility, configurability, etc.

9.3 Design Concepts

This section discusses many significant design concepts (abstraction, refinement, modularity, architecture, patterns, refactoring, functional independence, information hiding, and OO design concepts).

9.3.1 Abstraction

At the highest level of abstraction, a solution is stated in broad terms using the language of the problem environment. At lower levels of abstraction, a more detailed description of the solution is provided.

As we move through different levels of abstraction, we work to create procedural and data abstractions. A procedural abstraction refers to a sequence of instructions that have a specific and limited function. An example of a procedural abstraction would be the word open for a door.

A data abstraction is a named collection of data that describes a data object. In the context of the procedural abstraction open, we can define a data abstraction called door. Like any data object, the data abstraction for door would encompass a set of attributes that describe the door (e.g. door type, swing direction, weight).

9.3.2 Architecture

Software architecture alludes to the “overall structure of the software and the ways in which the structure provides conceptual integrity for a system.”

In its simplest from, architecture is the structure of organization of program components (modules), the manner in which these components interact, and the structure of data that are used by the components.

The goal of software design is to derive an architectural rendering of a system. This rendering serves as a framework from which detailed design activities are constructed.

A set of architectural patterns enable a software engineer to reuse design-level concepts.

The architectural design can be represented using one or more of a number of different models.

Structural models represent architecture as an organized collection of program components.

Framework models increase the level of design abstraction by attempting to identify repeatable architectural design frameworks that are encountered in similar types of applications.

Dynamic models address the behavioural aspects of the program architecture, indicating how the structure or system configuration may change as a function of external events.

Process models focus on the design of business or technical process that the system must accommodate.

Functional models can be used to represent the functional hierarchy of a system.

Architectural design will be discussed in Chapter 10.

9.3.3 Patterns

A design pattern “conveys the essence of a proven design solution to a recurring problem within a certain context amidst computing concerns.”

A design pattern describes a design structure that solves a particular design problem within a specific context and amid “forces” that may have an impact on the manner in which the pattern is applied and used.

The intent of each design pattern is to provide a description that enables a designer to determine:

whether the pattern is applicable to the current work,

whether the pattern can be reused, and

whether the pattern can serve as a guide for developing a similar, but functionally or structurally different pattern.

9.3.4 Modularity

Software architecture and design patterns embody modularity; that is, software is divided into separately named and addressable components, sometimes called modules that are integrated to satisfy problem requirements.

Monolithic software (large program composed of a single module) cannot be easily grasped by a software engineer. The number of control paths, span of reference, number of variables, and overall complexity would make understanding close to impossible.

It is the compartmentalization of data and function. It is easier to solve a complex problem when you break it into manageable pieces. “Divide-and-conquer”

Don’t over-modularize. The simplicity of each small module will be overshadowed by the complexity of integration “Cost”.

9.3.5 Information Hiding

It is about controlled interfaces. Modules should be specified and design so that information (algorithm and data) contained within a module is inaccessible to other modules that have no need for such information.

Hiding implies that effective modularity can be achieved by defining by a set of independent modules that communicate with one another only that information necessary to achieve software function.

The use of Information Hiding as a design criterion for modular systems provides the greatest benefits when modifications are required during testing and later, during software maintenance. Because most data and procedures are hidden from other parts of the software, inadvertent errors introduced during modifications are less likely to propagate to other location within the software.

9.3.6 Functional Independence

The concept of functional Independence is a direct outgrowth of modularity and the concepts of abstraction and information hiding.

Design software so that each module addresses a specific sub-function of requirements and has a simple interface when viewed from other parts of the program structure.

Functional independence is a key to good design, and design is the key to software quality.

Independence is assessed using two qualitative criteria: cohesion and coupling.

Cohesion is an indication of the relative functional strength of a module.

Coupling is an indication of the relative interdependence among modules.

A cohesive module should do just one thing.

Coupling is a qualitative indication of the degree to which a module is connected to other modules and to the outside world “lowest possible”.

9.3.7 Refinement

It is the elaboration of detail for all abstractions. It is a top down strategy.

A program is developed by successfully refining levels of procedural detail.

A hierarchy is developed by decomposing a macroscopic statement of function (a procedural abstraction) in a stepwise fashion until programming language statements are reached.

We begin with a statement of function or data that is defined at a high level of abstraction.

The statement describes function or information conceptually but provides no information about the internal workings of the function or the internal structure of the data.

Refinement causes the designer to elaborate on the original statement, providing more and more detail as each successive refinement (elaboration) occurs.

Abstraction enables a designer to specify procedure and data and yet suppress low-level details.

Refinement helps the designer to reveal low-level details as design progresses.

Refinement causes the designer to elaborate on the original statement, providing more and more detail as each successive refinement “elaboration” occurs.

9.3.8 Refactoring

It is a reorganization technique that simplifies the design of a component without changing its function or behaviour. When software is re-factored, the existing design is examined for redundancy, unused design elements, inefficient or unnecessary algorithms, poorly constructed data structures, or any other design failures that can be corrected to yield a better design

“The overall structure of the software and the ways in which that structure provides conceptual integrity for a system.” [SHA95a]

Structural properties. This aspect of the architectural design representation defines the components of a system (e.g., modules, objects, filters) and the manner in which those components are packaged and interact with one another. For example, objects are packaged to encapsulate both data and the processing that manipulates the data and interact via the invocation of methods

Extra-functional properties. The architectural design description should address how the design architecture achieves requirements for performance, capacity, reliability, security, adaptability, and other system characteristics.

Families of related systems. The architectural design should draw upon repeatable patterns that are commonly encountered in the design of families of similar systems. In essence, the design should have the ability to reuse architectural building blocks.

Patterns

Design Pattern Template

Pattern name—describes the essence of the pattern in a short but expressive name

Intent—describes the pattern and what it does

Also-known-as—lists any synonyms for the pattern

Motivation—provides an example of the problem

Applicability—notes specific design situations in which the pattern is applicable

Structure—describes the classes that are required to implement the pattern

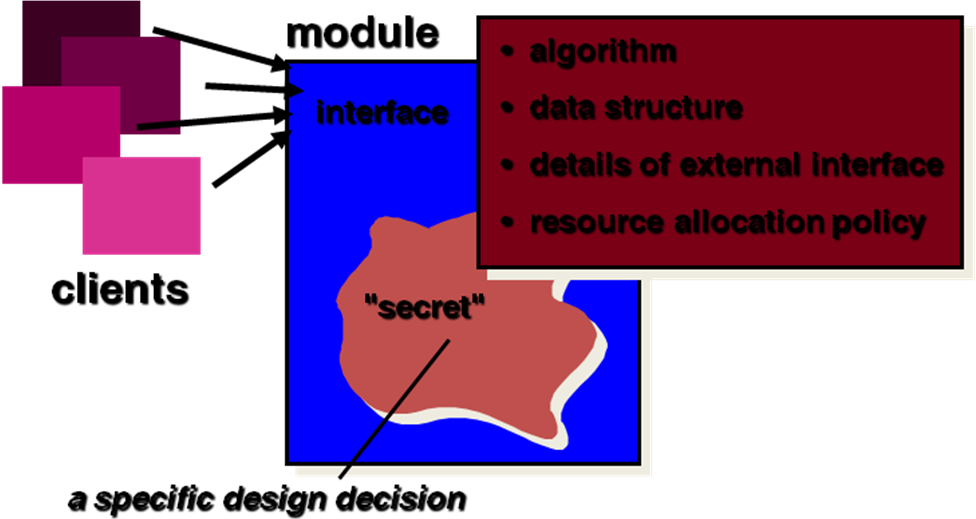
Participants—describes the responsibilities of the classes that are required to implement the pattern

Collaborations—describes how the participants collaborate to carry out their responsibilities

Consequences—describes the “design forces” that affect the pattern and the potential trade-offs that must be considered when the pattern is implemented

Related patterns—cross-references related design patterns

**Information Hiding**



Why Information Hiding?

* Reduces the likelihood of “side effects”
* Limits the global impact of local design decisions
* Emphasizes communication through controlled interfaces
* Discourages the use of global data
* Leads to encapsulation—an attribute of high quality design
* Results in higher quality software

Design classes

As the design model evolves, the software team must define a set of design classes that refines the analysis classes and creates a new set of design classes.

Five different classes’ types are shown below:

* User Interface classes: define all abstractions that are necessary for HCI.
* Business domain classes: are often refinements of the analysis classes defined earlier. The classes identify the attributes and services that are required to implement some element of the business domain.
* Process classes: implement lower-level business abstractions required to fully manage the business domain classes.
* Persistent classes: represent data stores that will persist beyond the execution of the software.
* System classes: implement software management and control functions that enable the system to operate and communicate within its computing environment and with the outside world.

**9.4 The Design Model**



**Centralized Repository**

* Central process acts as control hub
* Central data repository acts as information hub
* Termed *client-server* when based on standalone processes
* *Integrability* allows different clients and servers to interoperate
* Depending on persistency requirements:
  + A *blackboard* can store temporary data acting as a communications hub
  + A *relational database* can provide a persistent data store
* In a *two-tier* architecture the client accesses the database
* In a *three-tier* architecture a separate server offers task-oriented services
* A *transaction monitor* can be used to offer
  + Resiliency
  + Redundancy
  + Load distribution
  + Message sequencing
* *Middleware* allows different parts to communicate with each other
  + CORBA (OMG)
  + .NET (Microsoft)
  + RMI (Java/Sun)

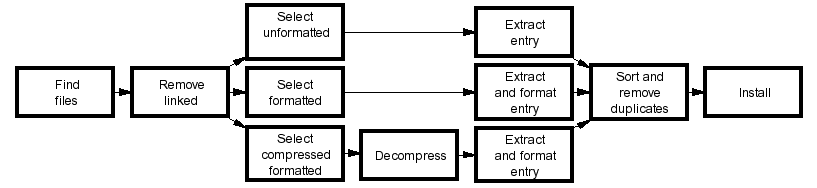
**Examples**

* Window manager
* File server
* Print server
* Collaboration
  + WWW
  + Modern ERP systems
  + Instant messaging
  + Revision control (CVS)

**Data-Flow**

* Also known as *pipes-and-filters* architecture
* Consists of a series of *data transformations*
* Examples:
  + Unix text processing tools (wc, grep, sort, uniq, join)
  + Netpbm graphics processing package

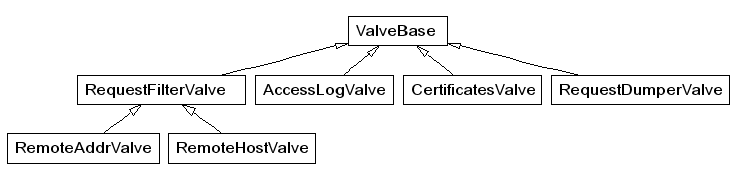
**Create Manual Page Index**



**Object-Oriented**

* Design is based on interacting *objects*
* Each object maintains its own state
* State is manipulated by *methods*
* Objects are typically grouped into *classes*
* Classes are often organised into *generalisation* relationships

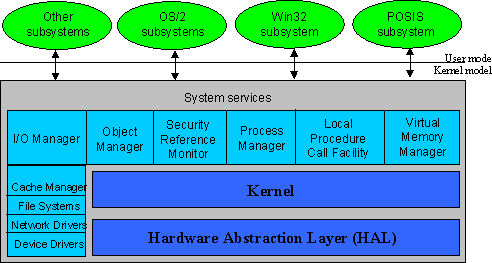
**A generalization relationship in the Tomcat servlet container**



**Layered**

* Can organise multiple peer subsystems
* Each layer:
  + Presents a standard interface to its upper layer
  + Communicates using a different standard interface with its lower layer
* Essentially each layer implements a *virtual machine*
* Lower layers shall not use upper layers
* Examples:
  + Network stacks
  + Operating system structures

**The Windows NT implementation layers**



**Hierarchies**

Hierarchies

* provide structure
* facilitate navigation
* can be used orthogonally with other architectures
* examples
  + directories
  + DNS
  + call graphs
  + identifier namespaces