CSE360 Summer 2021 Notes

John J Li

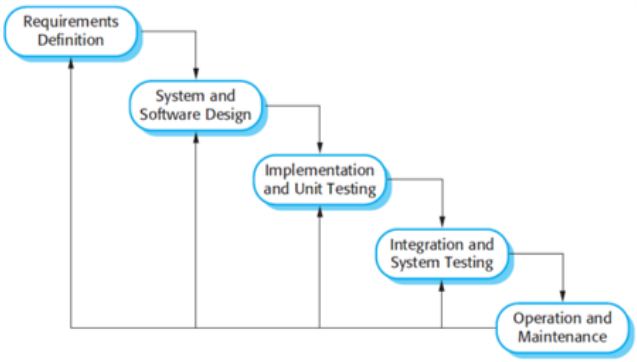
June 10, 2021

Plan-driven Process Models

A plan-driven model plans process activities in advance and the progress is measured against the plan. There are 5 types of plan-driven models: waterfall, v-model, incremental, prototype, and spiral.

Waterfall Model

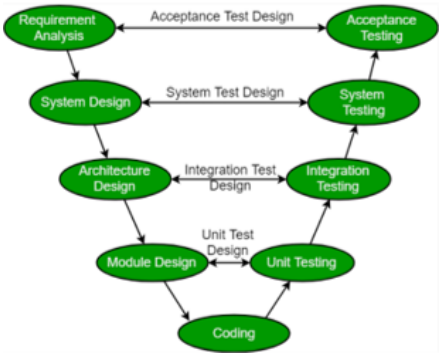
Each fundamental process activity is placed into seperated phases and performed in a linear order. Use this when requirements are well understood and won't likely change, it is a large system, and multiple companies are involved. Commonly used in financial, security, gov/military, embedded systems. Difficult to change once the process starts.



V-Model

Extension of waterfall model – process steps are bent upwards after the coding phase. Each dev phase has a testing phase. Verification (leftside) analyzes that the requirements are met. Validation (right side) tests that the implementation meets the requirements. Useful for maintaining strict deadlines and milestones and detecting errors earlier in the dev process.

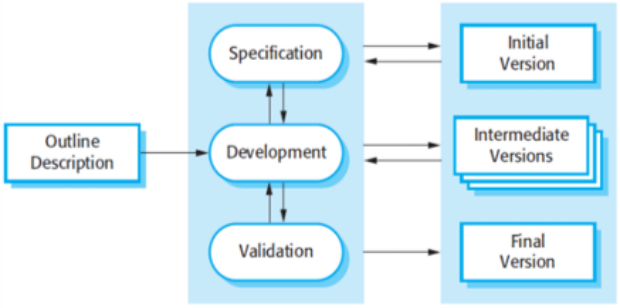
Unit testing eleminates bugs at code or unit level. Integration testing verifies communication between modules. System testing tests the complete application and function and non-function requirements. User acceptance testing verifies applicaiton is ready for the real world.



Incremental Model

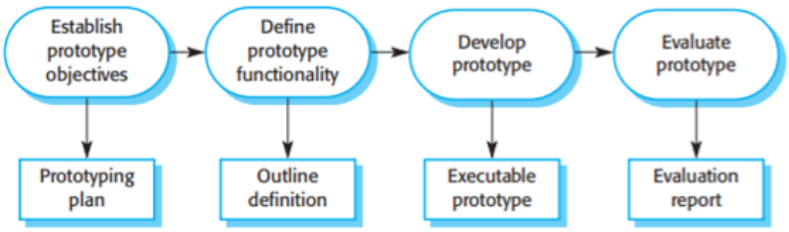
Split activities into pieces. It is an iterative model. Develop an initial implementation which complete a portion of each activity and gets user feedback then evolve the next implementation version and repeat.

There is frequent user feedback and reduce cost of changing requirements plus a rapid delivery of useful software. However it is harder to measure progress and system structure tends to degrade as new increments are added.



Prototyping

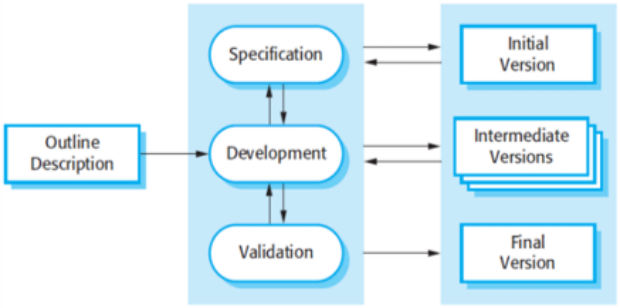
Early version of the system or part of the system that is developed quickly. Customers are involved through this process and it anttcipate changes but is often discarded.



Spiral Model

Similar to the incremental model, but includes risk assessment. Each loop represents a phase. Determine goals in top left, evaluate risks in top right, develop and test in bottom right, and plan in bottom left.

This is useful for large projects, high-risk projects, needing a lot of documentation, and there are significant changes are expected. Doesn't work for small projects, project time estimation is difficult and success is dependent on effective risk assessment.



Agile Process Models

The agile model plans in increments and can change to reflect changes in requirements. There are 2 types of agile models: SCRUM and XP(extreme programming). Emerged in the 1980s and 1990s focused on code and an iterative approach to software development. The aim was the reduce overheads in the software process by limiting documentation and responding quickly to changes.

Manifesto for Agil Software Development

"We are uncovering better ways of developing software by doing it and helping others do it. Through this work, we have come to value:

- Individuals and interations over processes and tools
- Working software over comprehensive documentation
- Customer collaboration over contract negotiation
- Responding to change over following a plan

That is, while there is value in the items on the right, we value the item on the left more."

Agile Definition

- Software development under which requirements and solutions evolve through the collaborative effort of self-organizing cross-functional teams
- Adaptive planning to provide a rapid and flexible response to change
- Incremental lightweight process
- Early delivery
- Feedback-driven empirical approach

Extreme Programming (XP)

Several new versions may be developed by different programmers at the same time. They write tests for each task before writing code. This is difficult to integrate with typical business practices. Requirements expressed as user stories.

User Scenarios and User Stories

User Scenario: Long description for the user of the product. Can include mulitple requirements. User Story: Short description of one requirement such as "As a ¡user¿, I want ¡feature¿ so I can ¡rationale¿.

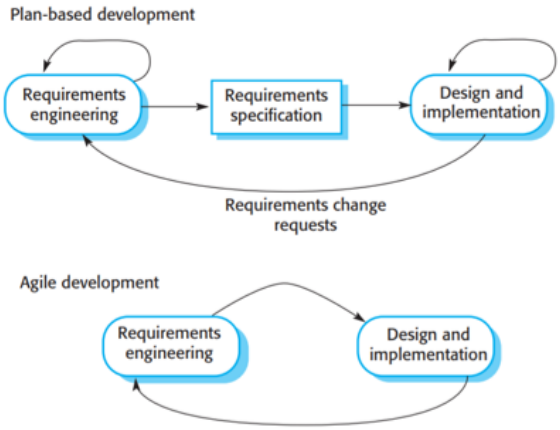
Agile vs Plan-driven Dev

Plan-driven

- Each phase requires formal documentation (i.e. requirements specification) between each major process activity, i.e.
- Requirements engineering: identify all requirements
- Requirements specification: document requirements
- Design and implementation: determine how to satisfy requirements and implement it

Agile

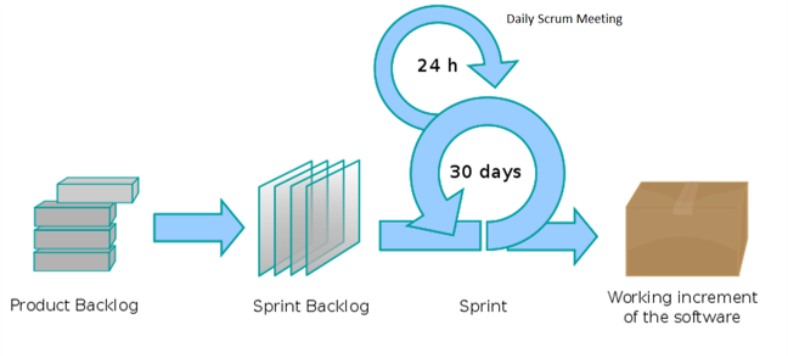
- Each iteration goes through all the activities
- Often requirements and design are developed together rather than separately



Scrum

There is the product owner which communicate vision of the product to the dev team and they represents customer's interests. There is the scrum master who is the leader in the team and acts between product owner and team. Finally, there is the individual team members.

First there is the product backlog which prioritizes list of items that need to be worked on. Then comes sprint backlog which is the set of features that will be worked on in the upcoming sprint. Then the sprint which is the time period to develop the next iteration and includes a daily scrum meeting which is a meeting to ask about progress and issues.



Agile Benefits

- Product is broken down into a set of manageable and understandable chunks
- Unstable requirements do not hold up progress
- The whole team can see what is happening at every step and leads to improving team communication
- Customers see on-time delivery of increments and can get feedback on how well the product works
- Works well on small to medium-sized products

Agile Challenges

- The informality of agile development is incompatible with the legal approach to contract definition (requirement specification) that is commonly used in large companies
- Agile methods are most appropriate for new software development rather than software maintenance In large companies, a lot of work is devoted to maintaining existing software systems
- Agile methods are designed for small co-located team, yet software development now involves worldwide, distributed teams

Requirements Engineering

Types of Requirements

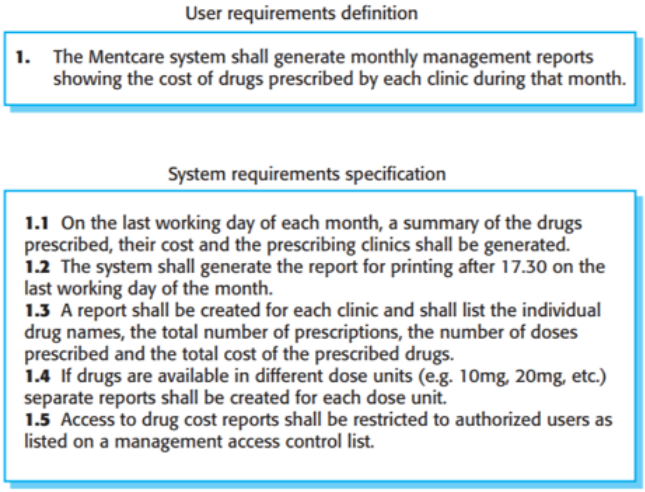
User requirements

- Statements are written in the natural language

- Diagrams of the services the system provides and its operational constraints
- Varies between broad statements of system features to detailed, precise descriptions of system functionality
- Typically, written for customers

System requirements

- Structured document setting out detailed descriptions of the system's functions, services, and operational constraints
- Defines exactly what is to be implemented
- Typically, written for developers

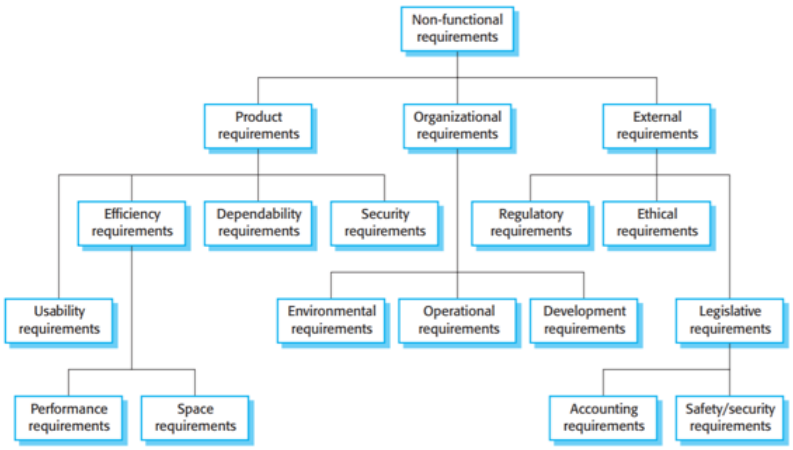


Functional Requirements

Statements of services the system should provide. How should the system react to particular inputs? How should the system behave in particular situations? May state what the system should not do. And is typically written in natural language.

Non-functional Requirements

Requirements are not directly related to specific services and often applies to the system as a whole. Constraints on the services or function offered by the system and may be more critical than functional requirements.



Some problems are: stakeholder propose non-functional requirements as goals, it is impossible to objectively verify the goal as been met so try to write them quantitatively.

Requirements Validation

- Demonstrates that the requirements define the system that the customer really wants
- Fixing requirement errors may cost a lot if it has to be done after the product is delivered
- What to check for:
  - Validity Requirements reflect the real needs of the system users
  - Consistency No requirement contradictions
  - Completeness Requirements define all functions and constraints intended by the system user
  - Realism Requirements can be implemented within the proposed budget and schedule
  - Verifiability System requirements are written in such a way that they can be verified after implementation

Validation techniques:

- Requirement reviews Systematic manual analysis of the requirements Performed by both software engineers and users
- Prototyping Develop an executable model of the system
- Test-case generation Develop tests for requirements to check testability If test is difficult or impossible to design, often means the requirement will be hard to implement and should be reconsidered

System Architecture

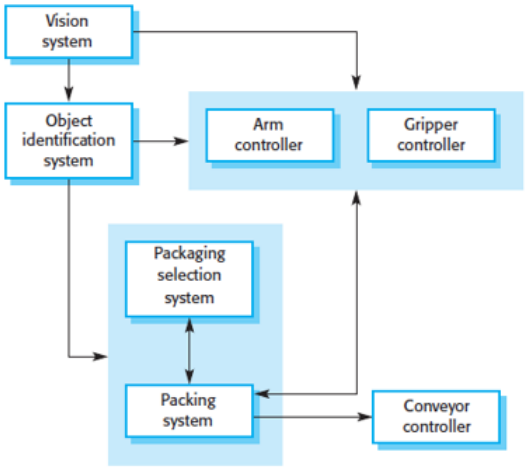
Architecture Design

- Understading how a software system should be organized
- Designing the overall structure of the system

Often represented using simple block diagrams

- High-level picture of the system structure
- Each block represents a component in the system
- Arrows represent data or signals passed from one component to another

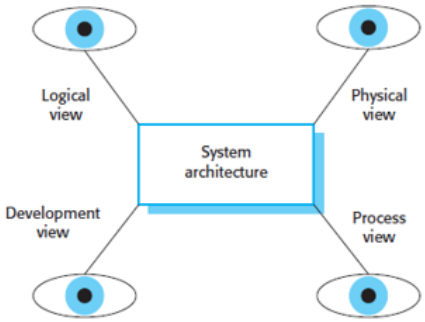
The advantages: easy to understand. Disadvantages: too informal, doesn't show the type of relationships among components or externally visible properties.



4+1 View Model

- Logical view
  - Key abstractions in the system as objects or object classes
  - Relate system requirements to objects
- Process view
  - How the system is composed of interacting processes
  - Usingful for making judgement about non-functional requirements

- Development view
  - How the software is decomposed for development
  - Useful for software managers and programmers
- Physical view
  - How the system hardware and software components are distributed across the processors in the system
  - Useful for system engineers
- +1 link all views through common use cases or scenarios



Architectural Patterns

A pattern is a means for representing, sharing, and reusing knowledge; and it may be represented using tabular and graphical descriptions.

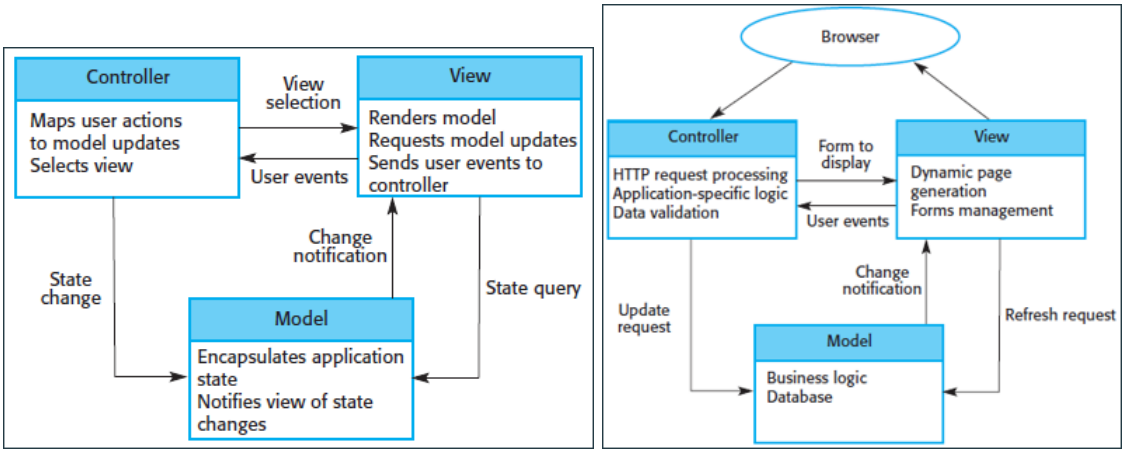
An architectural pattern is a stylized, abstract description of good design practice. It has been tried and tested in diff environments and it should include info about when they are and are not useful and the strengths and weaknesses.

Examples:

- Model-View-Controller (MVC)
- Layered architecture
- repository architecture
- Client-server architecture
- Pipe and filter architecture

Model-View-Controller

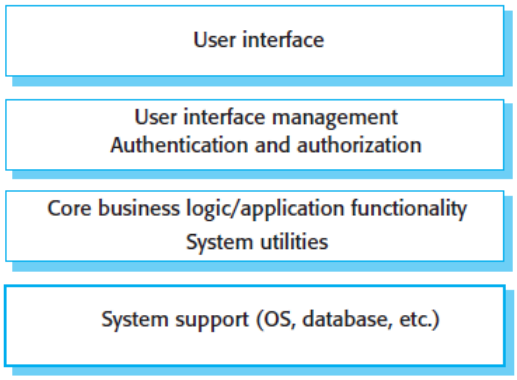
Often used for web applications and it splits the system into three different components: Model - stores and manages data; View - GUI; Controller - converts input from the View into demands to retrieve or update data from the Model and passes info from Model to the View.



Description	Separates presentation and interaction from the system data. The system is structured into three logical components that interact with each other. The Model component manages the system data and associated operations on that data. The View component defines and manages how the data is presented to the user. The Controller component manages user interaction (e.g., key presses, mouse clicks, etc.) and passes these interactions to the View and the Model. See Figure 6.5.
Example	Figure 6.6 shows the architecture of a web-based application system organized using the MVC pattern.
When used	Used when there are multiple ways to view and interact with data. Also used when the future requirements for interaction and presentation of data are unknown.
Advantages	Allows the data to change independently of its representation and vice versa. Supports presentation of the same data in different ways, with changes made in one representation shown in all of them.
Disadvantages	May involve additional code and code complexity when the data model and interactions are simple.

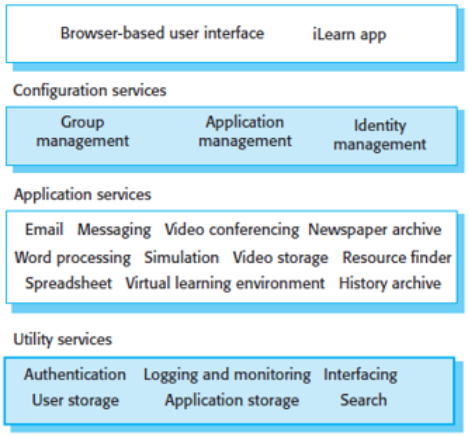
Layered Architecture

Used to model the interfacing of the subsystems and it supports incremental dev of subsystems in diff layers. When a layer interface changes, only the adjacent layers are affected.



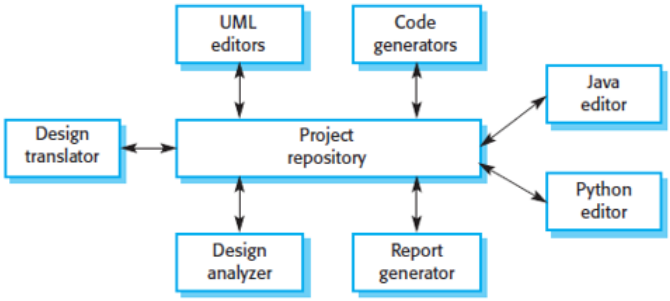
The number of layers is arbitrary.

Description	Organizes the system into layers, with related functionality associated with each layer. A layer provides services to the layer above it, so the lowest level layers represent core services that are likely to be used throughout the system. See Figure 6.8.
Example	A layered model of a digital learning system to support learning of all subjects in schools (Figure 6.9).
When used	Used when building new facilities on top of existing systems; when the development is spread across several teams with each team responsibility for a layer of functionality; when there is a requirement for multilevel security.
Advantages	Allows replacement of entire layers as long as the interface is maintained. Redundant facilities (e.g., authentication) can be provided in each layer to increase the dependability of the system.
Disadvantages	In practice, providing a clean separation between layers is often difficult, and a high-level layer may have to interact directly with lower-level layers rather than through the layer immediately below it. Performance can be a problem because of multiple levels of interpretation of a service request as it is processed at each layer.



Repository Architecture

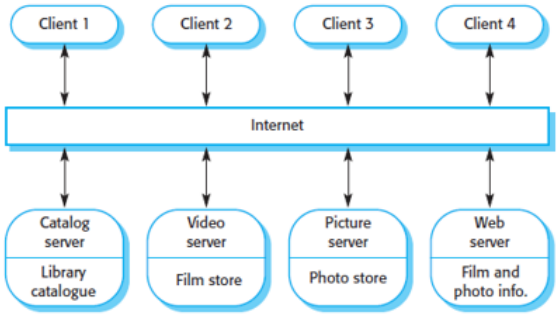
Subsystems must exchange data which can be done by storing datat in central database and which can be accessed by all subsystems or each subsystem maintains its own database and passes data to other subsystems. When large amounts of data need to be shared, the repository model is most commonly used.



Description	All data in a system is managed in a central repository that is accessible to all system components. Components do not interact directly, only through the repository.
Example	Figure 6.11 is an example of an IDE where the components use a repository of system design information. Each software tool generates information, which is then available for use by other tools.
When used	You should use this pattern when you have a system in which large volumes of information are generated that has to be stored for a long time. You may also use it in data-driven systems where the inclusion of data in the repository triggers an action or tool.
Advantages	Components can be independent; they do not need to know of the existence of other components. Changes made by one component can be propagated to all components. All data can be managed consistently (e.g., backups done at the same time) as it is all in one place.
Disadvantages	The repository is a single point of failure so problems in the repository affect the whole system. May be inefficiencies in organizing all communication through the repository. Distributing the repository across several computers may be difficult.

Client-Server Architecture

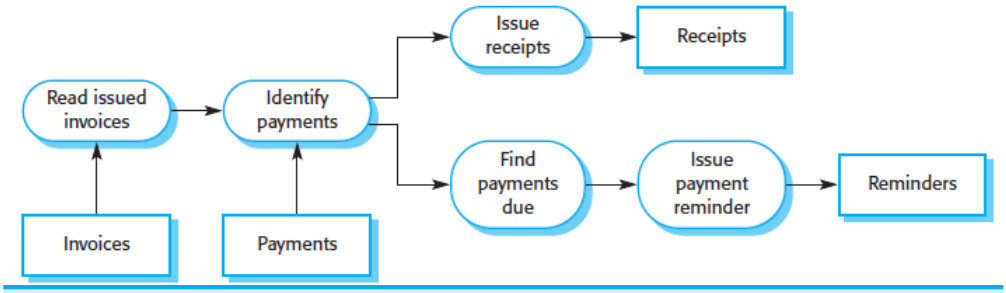
Distributed system model which shows how data and processing is distributed across a range of components. It is a set of stand-alone servers, a set of clients, and a network to connect the two.



Description	In a client–server architecture, the system is presented as a set of services, with each service delivered by a separate server. Clients are users of these services and access servers to make use of them.
Example	Figure 6.13 is an example of a film and video/DVD library organized as a client–server system.
When used	Used when data in a shared database has to be accessed from a range of locations. Because servers can be replicated, may also be used when the load on a system is variable.
Advantages	The principal advantage of this model is that servers can be distributed across a network. General functionality (e.g., a printing service) can be available to all clients and does not need to be implemented by all services.
Disadvantages	Each service is a single point of failure and so is susceptible to denial-of-service attacks or server failure. Performance may be unpredictable because it depends on the network as well as the system. Management problems may arise if servers are owned by different organizations.

Pipe and Filter Architecture

Data comes into a process, gets transformed and the resulting output is used as the input for the next process. Not a good choice for interactive systems.



Description	The processing of the data in a system is organized so that each processing component (filter) is discrete and carries out one type of data transformation. The data flows (as in a pipe) from one component to another for processing.
Example	Figure 6.15 is an example of a pipe and filter system used for processing invoices.
When used	Commonly used in data-processing applications (both batch and transaction-based) where inputs are processed in separate stages to generate related outputs.
Advantages	Easy to understand and supports transformation reuse. Workflow style matches the structure of many business processes. Evolution by adding transformations is straightforward. Can be implemented as either a sequential or concurrent system.
Disadvantages	The format for data transfer has to be agreed between communicating transformations. Each transformation must parse its input and unparse its output to the agreed form. This increases system overhead and may mean that it is impossible to reuse architectural components that use incompatible data structures.

Design Patterns

Architecture vs Design

Software architecture gives the high-levvel organization of the software and it identifies the main structural components in a system and the relationships between them.

Sofware design gives the code-level design and it identifies what each class doing, their relationships and scope.

Design patterns

A patter is a description of the problem and the essence of its solution. A design pattern is a way of reusing abstract knowledge about a problem and its solution.

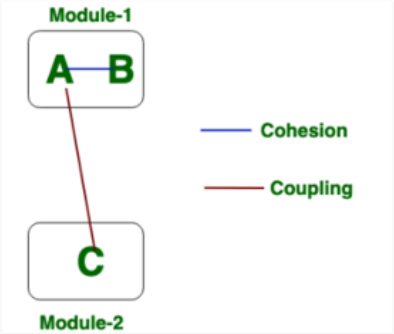
It should have High cohesion and low coupling.

Cohesion

Cohesion is the degree of interaction within a module.

There are 7 levels:

- Functional (best): all essential elements for a single task is in one module
- Sequential
- Communicational
- Procedural
- Temporal
- Logical
- Coincidental (worst): Elements have no conceptual relationship other than the location in the source code.

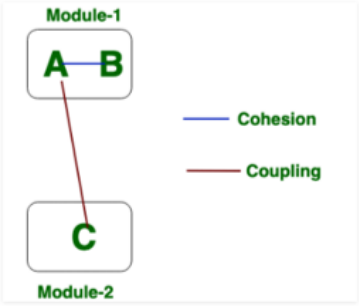


Coupling

Coupling is the degree of interaction between modules.

There are 5 levels:

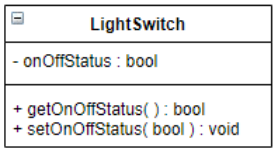
- Data (best): modules are independent from each other and communication by only passing data
- Stamp
- Control
- Common
- Content (worst): a module can modify the data of another module.



Encapsulation

Def: Hides the details from everything; is the process to contain information.

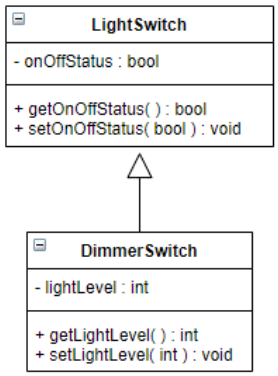
Implementation: variables of a class are private and functions are public so other classes can get and set the data.



Abstraction

Def: Shows only what is needed and hides the unwanted info. It is the process to gain info.

Implementation: Create an abstract class and extend to show more info or add complexity.



Design Problems and Patterns

To use patterns in your design, you need to recognize that any design problem may have an associated pattern that can be applied.

There is three catagory: Creational patterns which is focused on creating objects; Structural patterns which setups the relationship between objects; Behavioral patterns which defins how objects interact with each other.

Examples:

- Iterator
  - Provide a standard way of accessing the elements in a collection, irrespective of how that collection is implemented.
- Facade
  - Tidy up the interfaces to a number of related objects that have often been developed incrementally

- Observer
  - Tell several objects that the state of some other objects has changed
- Decorator
  - Allow for possibility of extending the functionality of an existing class at run-time