Coupling vs Cohesion:

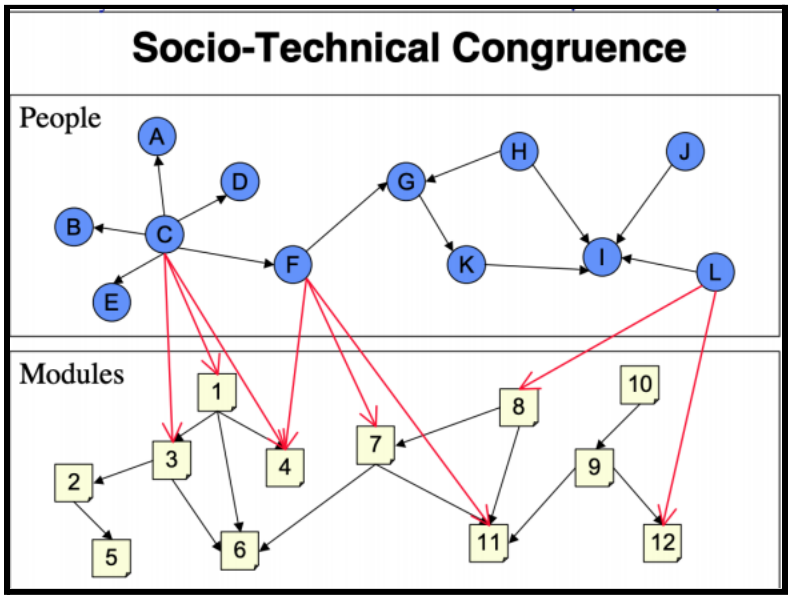
- **Coupling and Cohesion:**
- **Modules** are the building blocks of architectural applications and they often communicate with each other.
- A good architecture minimizes **coupling** between modules and maximizes the **cohesion** between modules.
- **Cohesion** refers to the degree to which the elements inside a module belong together.
- High-cohesion means that each class takes care of one thing, and one thing only. This references the **Single Responsibility Principle**.
- Low cohesion implies that a given module performs tasks which are not very related to each other and hence can create problems as the module becomes large.
- Modules with high cohesion tend to be preferable, because high cohesion is associated with several desirable traits of software including robustness, reliability, reusability, and understandability. In contrast, low cohesion is associated with undesirable traits such as being difficult to maintain, test, reuse, or even understand.
- Think of building a physical robot. Many small parts (highly cohesive), versus a few mega parts (low cohesion, monolithic).
- **Coupling** refers to the interdependencies between modules.
  I.e. Components that are mutually dependent are also called coupled.
- A **loosely coupled** system is one in which each of its components has, or makes use of, little or no knowledge of the definitions of other separate components.
- **Tight coupling/tightly coupled** is a type of coupling that describes a system in which hardware and software are not only linked together, but are also dependent upon each other.
- Loose coupling is important because it enables isolation and makes for future changes easier.
- We want to keep the coupling low and the cohesion high.
- **Measuring Coupling:**
- **Efferent coupling (ec):** Measures the number of classes on which a given class depends.
- **Afferent coupling (ac):** Measures how many classes depend on a given class.
- The **instability index** measures efferent coupling in relation to total coupling:
  $$\frac{ec}{ec + ac}$$
- The closer to zero the instability index is, the better.
- **Measuring Cohesion:**
- The cohesion of a class is the inverse of the number of method pairs whose similarity is zero.
- E.g. Assume a class has methods M1(p,q,r,s,t), M2(r,s,t), and M3(a,b,c). The lower case letters are instance variables participating in the methods. We see that for two methods (M1 and M2), their sets of participating variables intersect while the third (M3) doesn't.
  I.e. M1 and M2 share some of the same instance variables while M3 doesn't share any instance variables with M1 or M2.
  Hence the cohesion is 1/(2−0) = 1/2, which is small.

Conway's Law:

- **Conway's law** is an adage stating that organizations design systems which mirror their own communication structure.
- The law states: "The structure of a software system reflects the structure of the organization that built it."

Socio-Technical Congruence:

- Product development endeavors involve two fundamental elements:
    1. Technical: This involves processes, tasks, technology, etc.
       I.e. How people interact with the software.
    2. Social: This involves organizations and the individuals involved in the development process, their attitudes and behaviors.
       I.e. How people interact with each other.
- I.e.



- We can measure socio-technical congruence using a **coordination requirements matrix**.

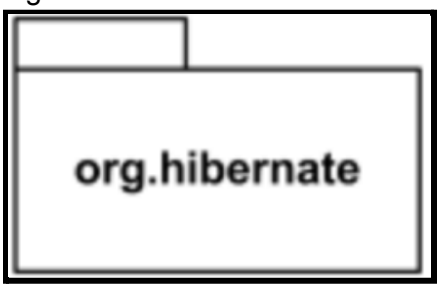
Software Architecture:

- The architecture of a system describes its major components, their relationships and how they interact with each other.
  A software architecture defines:
    1. The components of the software system.
    2. How the components use each other's functionality and data.
    3. How control is managed between the components.
- An example of a software architecture is the **client-server architecture**. The client-server architecture is a computing model in which the server hosts, delivers and manages most of the resources and services to be consumed by the client. This type of architecture has one or more client computers connected to a central server over a network or internet connection.
  Another example of a software architecture is the **3-layer architecture**. The 3-layer architecture has 3 main components:
    1. The presentation layer (UI)
    2. The application logic layer
    3. The database layer
- **Note:** MVC is not a 3-layer architecture. A fundamental rule in a 3-layer architecture is the presentation layer never communicates directly with the database layer. In a 3-layer architecture all communication must pass through the application logic layer. Conceptually the three-tier architecture is linear. However, the MVC architecture is triangular. The view sends updates to the controller, the controller updates the model, and the view gets updated directly from the model.

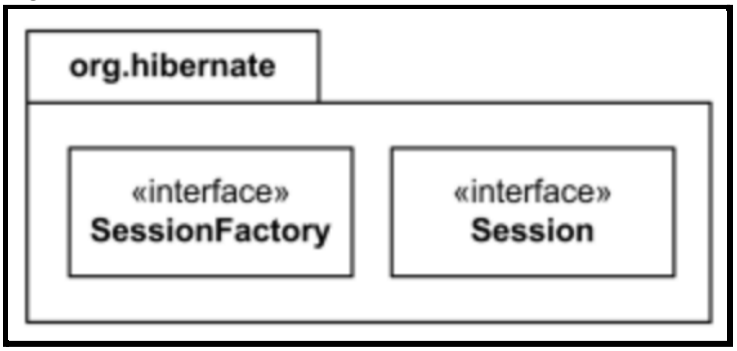
UML Packages:

- **Introduction:**
- A **package** is a namespace used to group together elements that are semantically related and might change together. It is a general purpose mechanism to organize elements into groups to provide a better structure for a system model.
- **UML package diagrams** are structural diagrams used to show the organization and arrangement of various model elements in the form of packages. A package is a grouping of related UML elements, such as diagrams, documents, classes, or even other packages. Each element is nested within the package, which is depicted as a file folder, and then is arranged hierarchically within the diagram. Package diagrams are most commonly used to provide a visual organization of the layered architecture within any UML classifier, such as a software system.
- Package diagrams are UML structure diagrams which show packages and dependencies between the packages.
  Note: Structure diagrams do not utilize time related concepts and do not show the details of dynamic behavior.
- If a package is removed from a model, so are all the elements owned by the package.
- A package could also be a member of other packages.
- A package in the Unified Modeling Language helps:
- To group elements.
- To provide a namespace for the grouped elements.
- Provide a hierarchical organization of packages.
- Benefits of UML package diagrams:
- They provide a clear view of the hierarchical structure of the various UML elements within a given system.
- These diagrams can simplify complex class diagrams into well-ordered visuals.
- They offer valuable high-level visibility into large-scale projects and systems.
- Package diagrams can be used to visually clarify a wide variety of projects and systems.
- These visuals can be easily updated as systems and projects evolve.
- **Terminology:**
- **Package:** A namespace used to group together logically related elements within a system. Each element contained within the package should be a packageable element and have a unique name.
- **Packageable element:** A named element, possibly owned directly by a package. These can include events, components, use cases, and packages themselves. Packageable elements can also be rendered as a rectangle within a package, labeled with the appropriate name.
- **Dependencies:** A visual representation of how one element or set of elements depends on or influences another. Dependencies are divided into two groups: access and import dependencies.
- **Access dependency:** Indicates that one package requires assistance from the functions of another package.
  I.e. One package requires help from functions of another package. (Making an API call for example)
- **Import dependency:** Indicates that functionality has been imported from one package to another.
  I.e. One package imports the functionality of another package. (Importing a package)
- **Notation:**
- A package is rendered as a rectangle with a small tab attached to the left side of the top of the rectangle. If the members of the package are not shown inside the package rectangle, then the name of the package should be placed inside.

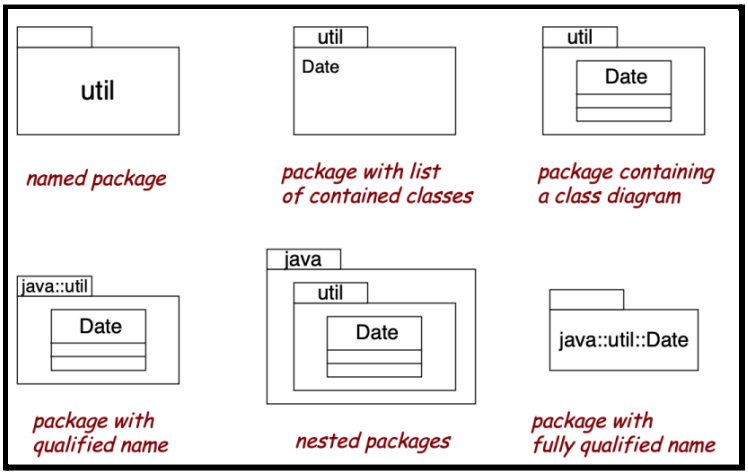
E.g.



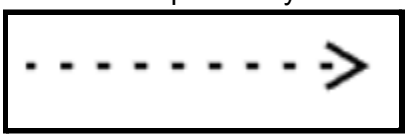
- The members/elements of the package may be shown within the boundaries of the package. If the names of the members of the package are shown, then the name of the package should be placed on the tab.
  E.g.



  Here, Package org.hibernate contains SessionFactory and Session.
- More examples:



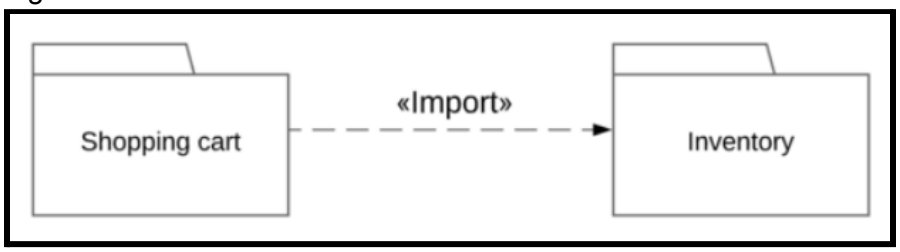
- To show a dependency between 2 packages, you draw a dotted arrow,



  , between the 2 packages such that the arrow is pointing to the independent package.

- To show an access dependency, write <<Access>> on the dotted arrow.
  E.g.

- To show an import dependency, write <<Import>> on the dotted arrow.
  E.g.



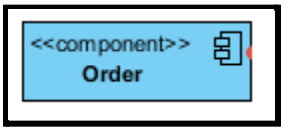
- **Criteria for Decomposing a System into Packages:**
- Different owners - who is responsible for working on which diagrams?
- Different applications - each problem has its own obvious partitions.
- Clusters of classes with strong cohesion - E.g. course, course description, instructor, student, etc.
- Or: Use an architectural pattern to help find a suitable decomposition such as the MVC Framework.
- **Other Guidelines for Packages:**
- Gather model elements with strong cohesion in one package.
- Keep model elements with low coupling in different packages.
- Minimize relationships, especially associations, between model elements in different packages.
- Namespace implication: An element imported into a package does not know how it is used in the imported package.
- We want to avoid dependency cycles.

Component Diagrams:

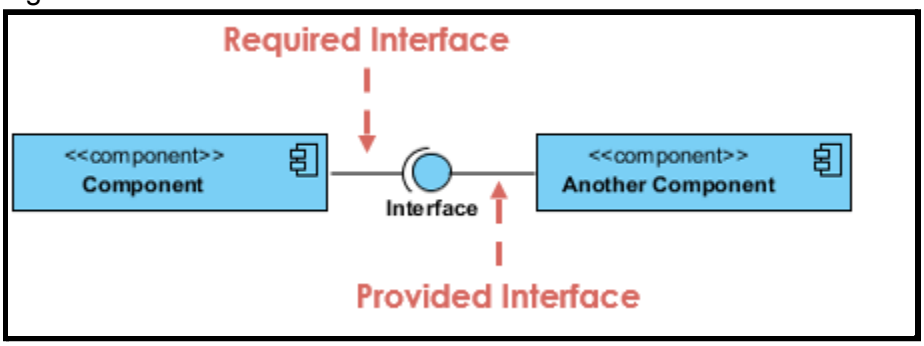
- **Introduction:**
- **Component diagrams** are used in modeling the physical aspects of object-oriented systems that are used for visualizing, specifying, and documenting component-based systems and also for constructing executable systems through forward and reverse engineering.
- A component diagram breaks down the actual system under development into various high levels of functionality.
- Each component is responsible for one clear aim within the entire system and only interacts with other essential elements on a need-to-know basis.
- Component diagrams can help your team:
  - Imagine the system's physical structure.
  - Pay attention to the system's components and how they relate.
  - Emphasize the service behavior as it relates to the interface.
- A component diagram gives a bird's-eye view of your software system. Understanding the exact service behavior that each piece of your software provides will make you a

better developer. Component diagrams can describe software systems that are implemented in any programming language or style.

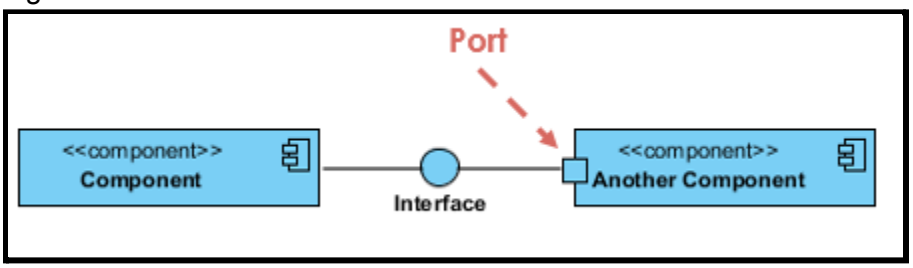
- **Notation:**
- **Component:** A rectangle with the component's name, stereotype text, and icon.
  E.g.



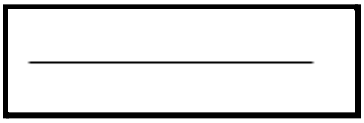
- **Interface:** There are 2 types of interfaces, **provided interface** and **required interface**.
  Provided interface: A complete circle with a line connecting to a component. Provided interfaces provide items to components.
  Required Interface: A half circle with a line connecting to a component. Required interfaces are used to provide required information to a provided interface.
  E.g.



- **Port:** A square along the edge of the system or a component. A port is often used to help expose required and provided interfaces of a component. Ports are used to hook up other elements in a component diagram.
  E.g.

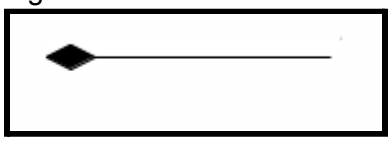


- **Association:** An association specifies a relationship that can occur between two instances. You represent an association using a straight line connecting 2 components.
  E.g.

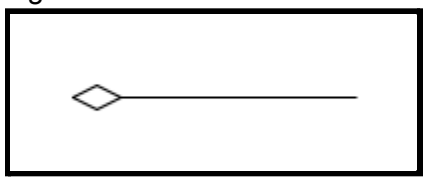


- **Composition:** Composition is a strong form of aggregation that requires a part instance be included in at most one composite at a time. If a composite is deleted, all of its parts are normally deleted with it. Composition is a type of association. You can represent a composition using an arrow where the arrowhead is filled in and points to the parent class.

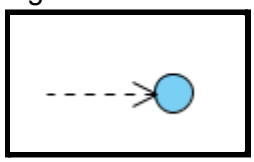
E.g.



- **Aggregation:** Aggregation implies a relationship where the child class can exist independently of the parent class. This means that if you remove/delete the parent class, the child class still exists. It is a special type of association and a weak form of association. You can represent an aggregation using an arrow where the arrowhead is not filled in and points to the parent class.
  E.g.

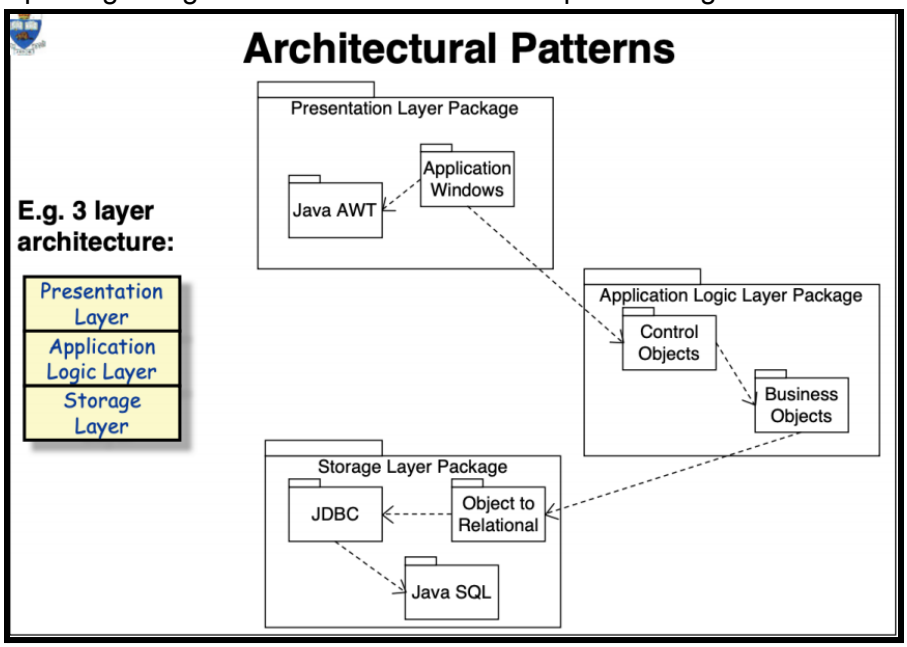


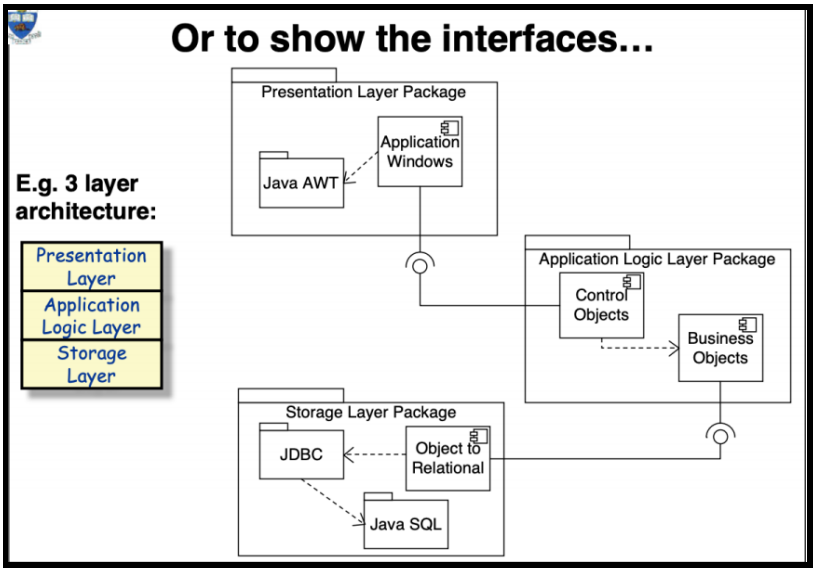
- **Dependency:** A dependency is a relationship that signifies that a single or a set of model elements requires other model elements for their specification or implementation. It is denoted as a dotted arrow with a circle at the tip of the arrow.
  E.g.



Examples:

- The following 2 UML diagrams show the same thing, a 3-layer application, but the first is a package diagram while the latter is a component diagram.



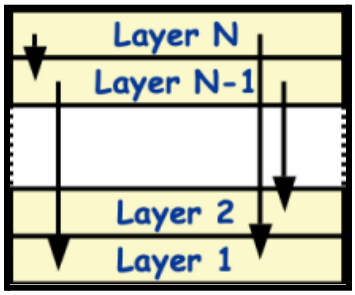


Layered Systems:

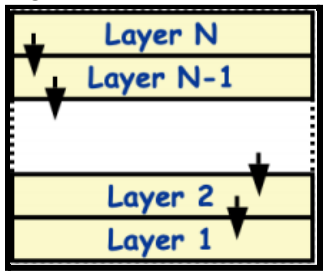
- Examples include operating systems and communication protocols.
- It supports increasing levels of abstraction during design.
- It supports enhancement (adding features and functionalities) and code reuse.
- It can define standard layer interfaces.
- A disadvantage is that it may not be able to define clean layers.

Open vs Closed Layered Architecture:

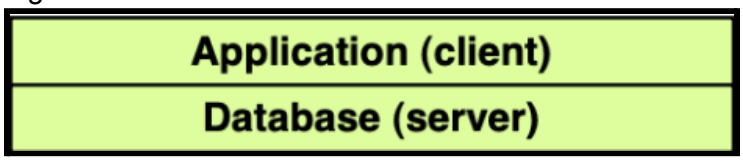
- **Open Layered Architecture:**
- A layer can use services from any lower layer.
- There is more compact code as the services of lower layers can be accessed directly.
- Breaks the encapsulation of layers, so there is an increase in dependency between layers.
- E.g.



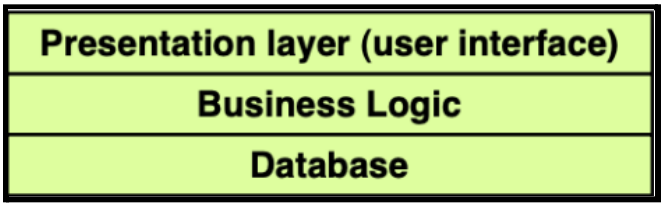
- **Closed Layered Architecture:**
- Each layer only uses services of the layer immediately below it.
- Minimizes dependencies between layers and reduces the impact of changes.
- E.g.



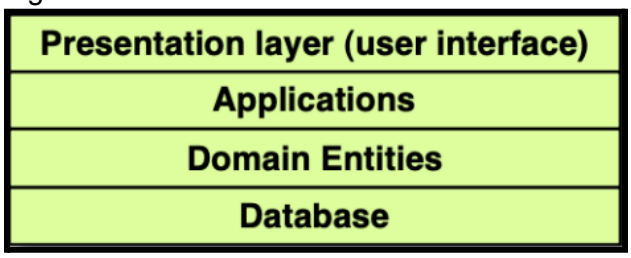
- With 2 layers, you have an application layer and a database layer. An example of a 2-layer architecture is a simple client-server model.
  E.g.



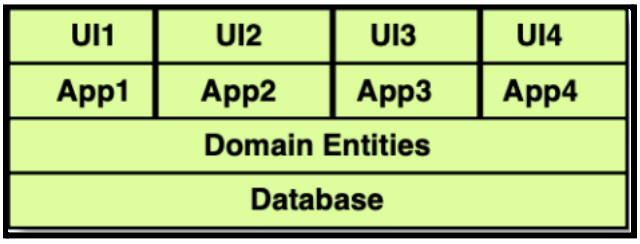
- With 3 layers, you have a presentation layer (UI), a business logic layer and a database layer. Here, we separated the business logic to make the UI and database layers more modifiable.
  E.g.



- With 4 layers, you have a presentation layer (UI), an application layer, a domain entity layer and a database layer. Here, we separated the application from the domain entity.
  E.g.

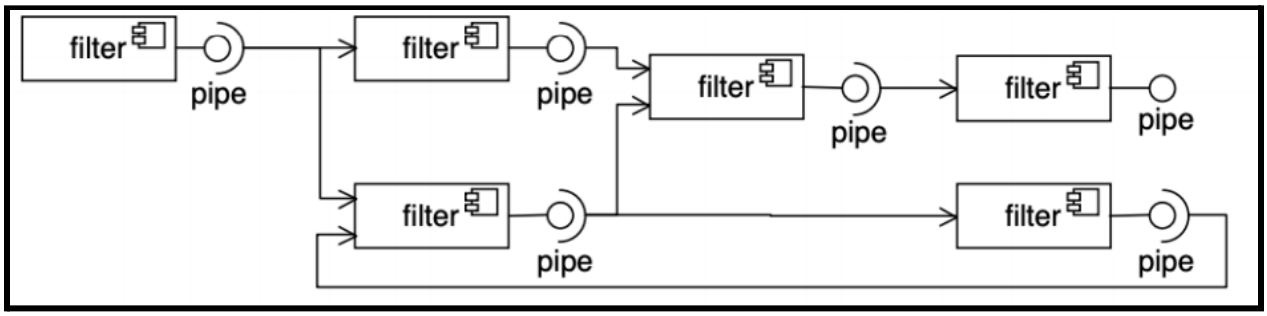


- With partitioned 4 layers, we have separate UIs for each application.
  E.g.



Pipe and Filters:

- Examples include Unix commands, compilers and signal processing.
- Filters don't need to know anything about what they're connected to.
- Filters can be implemented in parallel.
- The behaviour of the system is the composition of the behaviour of the filters.
- UML:



Object Oriented Architecture:

- Examples include abstract data types.
- Has encapsulation and abstraction.
- Can decompose problems into sets of interacting agents.
- Can be single or multi-threaded.
- A disadvantage is that objects must know the identity of the objects they wish to interact with.

Object Brokers:

- A variant of object oriented architecture.
- It adds a broker between the clients and servers.
- Clients no longer need to know which servers they are using.
- It can have many brokers and many servers.
- A disadvantage is that brokers can become bottlenecks which leads to degraded performance.

Event Based:

- Examples include debugging systems (listening for particular breakpoints), database management systems (data integrity checking) and graphical user interfaces.
- The announcers of events don't need to know who will handle the event.
- It supports re-use and evolution of systems and can add new agents easily.
- A disadvantage is that components have no control over ordering of computations.

Repositories:

- Examples include databases, blackboard expert systems and programming environments.
- Can choose where the locus of control is.
- Reduces the need to duplicate complex data.
- A disadvantage is that the blackboard can be a bottleneck.

Model-View-Controller (MVC):

- There is one central model with many viewers/views.
- Each view has an associated controller.
- The controller handles updates from the user of the view.
- Changes to the model are propagated to all the views.

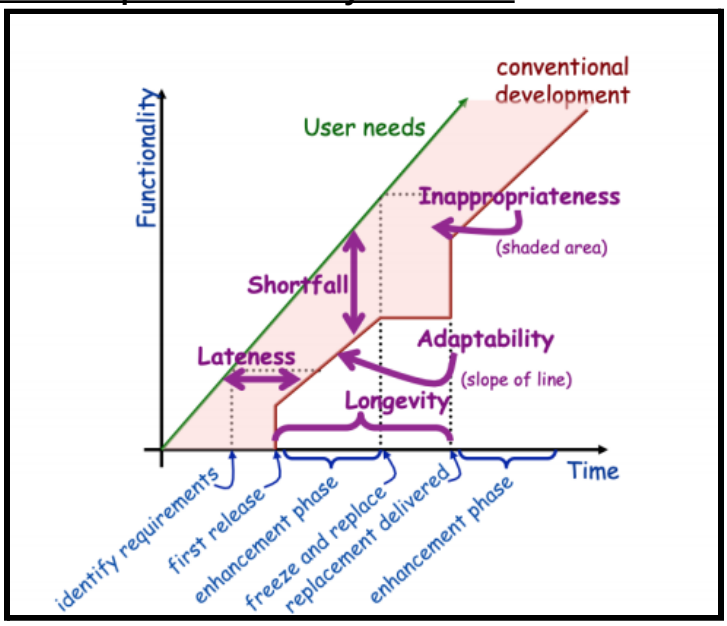
Program Types:

- **S-Type Programs (Specifiable):**
- The problem can be stated formally and completely.
- Acceptance: Is the program correct according to its specification?
- Evolution is not relevant as a new specification defines a new problem.
- S-Type software is one where specification is clear and detailed before the development even begins. Thanks to this detailed specification, it is clear what the solution should be and implementing it is trivial.
- S-programs are programs whose function is formally defined by and derivable from a specification.
- An S-program may be changed to improve its clarity or its elegance, to decrease resource usage when the program is executed, but any such changes must not affect the mapping between the input and output it achieves in execution.
- In S-programs, judgments about the correctness and the value of the programs relate only to its specification.
- E.g. Create a function that adds 2 numbers.
- **P-Type Programs (Problem-solving):**
- Here, we have an imprecise statement of a real-world problem.
- Acceptance: Is the program an acceptable solution to the problem?
- The software may continuously evolve as the solution is never perfect and can always be improved and the real world changes, so the problem changes.
- P-programs are programs for which the problem may be precisely formulated, but for which the solution must inevitably reflect an approximation of the real world.
- E.g. A program to play chess.
- **E-Type Programs (Embedded):**
- Here, the software becomes part of the world that it models.
- Acceptance: Depends entirely on opinion and judgment.
- This software is inherently evolutionary as changes in the software and world affect each other.
- E-programs are those programs that mechanize a human or societal activity.
- The program has become a part of the world it models.
- E.g. Operating Systems, business administration software, inventory management, etc.

Laws of Program Evolution:

- **Continuing Change:**
- Any software that reflects some external reality undergoes continual change or becomes progressively less useful.
- The change continues until it is judged that it is more cost effective to replace the system.
- **Increasing Complexity:**
- As software evolves, its complexity increases unless steps are taken to control it.
- **Fundamental Law of Program Evolution:**
- Software evolution is self-regulating with statistically determinable trends and invariants.
- **Conservation of Organizational Stability:**
- During the active life of a software system, the work output of a development project is roughly consistent, regardless of resources.
- **Conservation of Familiarity:**
- The amount of change in successive releases is roughly constant.

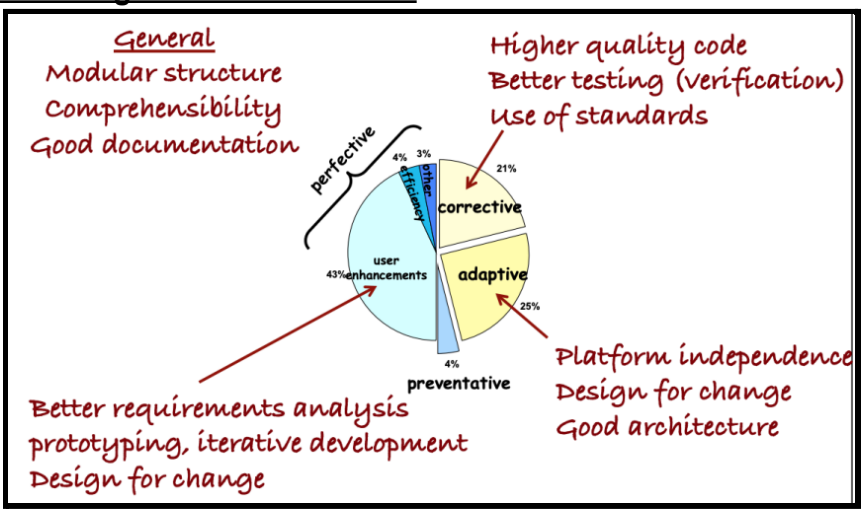
User Requirements Always Increase:



Software Geriatrics:

- **Causes of software aging:**
- Failure to update the software to meet changing needs. Customers will switch to a new product if the benefits outweigh the switching costs.
- Changes to software tend to reduce coherence and increase complexity.
- **Costs of software aging:**
- Owners of aging software may find it hard to keep up with the marketplace.
- Deterioration in space/time performance due to deteriorating structure.
- Aging software gets more buggy and each bug fix adds more errors than it fixes.
- **Ways of increasing longevity:**
- Design for change.
- Document the software carefully.
- Requirements and designs should be reviewed by those responsible for its maintenance.
- Software rejuvenation.

Reducing Maintenance Costs:



Factors That Drive The Cost of Maintaining Software:

- **Adaptive Maintenance:**
  - Accounts for 25% of the total maintenance cost.
  - Arises from modifying the software after its delivery to ensure that the product remains usable in a changing environment.
- **Corrective Maintenance:**
  - Accounts for 20% of the total maintenance cost.
  - Arises from resolving issues you identify during the initial deployment or release.
- **Perfective Maintenance:**
  - Accounts for 5% of the total maintenance cost.
  - Arises from improving software to make it perform efficiently.

Why Maintenance is Hard:

- Poor code quality
- Lack of knowledge of the application domain
- Lack of documentation
- Lack of glamour

Rejuvenation:

- **Reverse Engineering:**
- Includes re-documentation and design recovery.
- **Restructuring:**
- Includes refactoring (no changes to functionality) and revamping (only the ui is changed).
- **Re-Engineering:**
- Real changes are done to the code.
- It is usually done as a round trip:
  Design recovery → Design improvement → Re-implementation

Program Comprehension:

- During maintenance, programmers study the code about 1.5 times as long as the documentation and spend as much time reading code as editing it.
- Experts have many knowledge chucks.
- Experts follow dependency links while novices read sequentially.
- Much of the knowledge comes from outside the code.