IMPORTANT NOTICE TO STUDENTS

These slides are **NOT** to be used as a replacement for student notes.
These **slides** are sometimes **vague and incomplete on purpose** to spark a class discussion

An Introduction to Software Architecture By David Garlan & Mary Shaw – 94

CS 446/646 ECE452
May 18th, 2011

WATERLOO
CHERITON SCHOOL OF COMPUTER SCIENCE

Layered Systems

Organized hierarchy

- each layer has a unique role
  - provides a service to the layer above
  - acts as a client to the layer below
- separation of concerns?

Layered Systems

Components

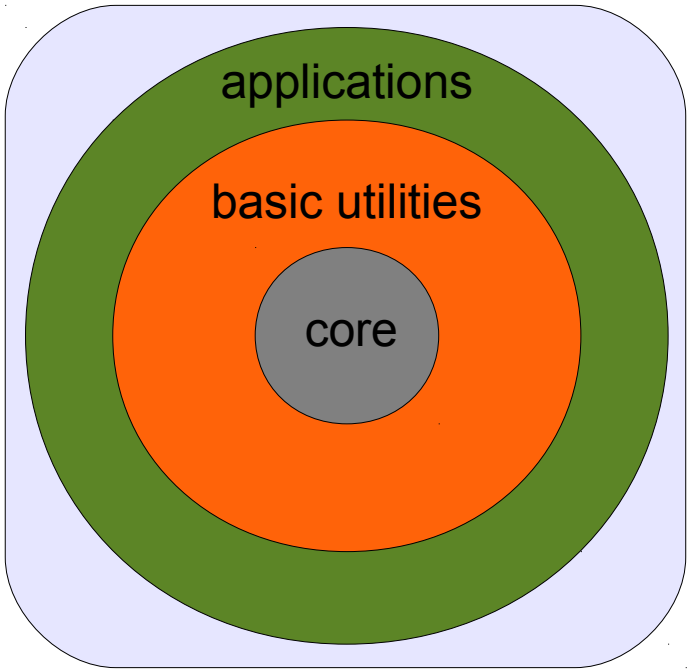
- layers: composed of groups of sub tasks/systems
- API: set of classes exposing an API layer

Connectors

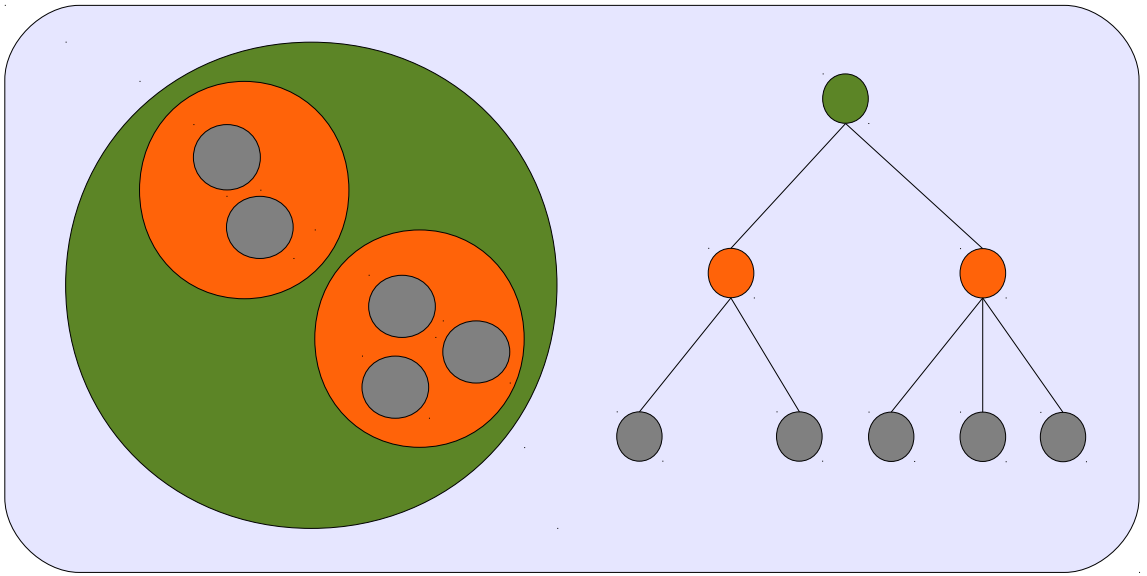
- communication protocols/interfaces
  - define the inter-layer interaction
  - should facilitate loose coupling
  - aim for standardized communication mechanism

Layered Systems

Different Layering Styles

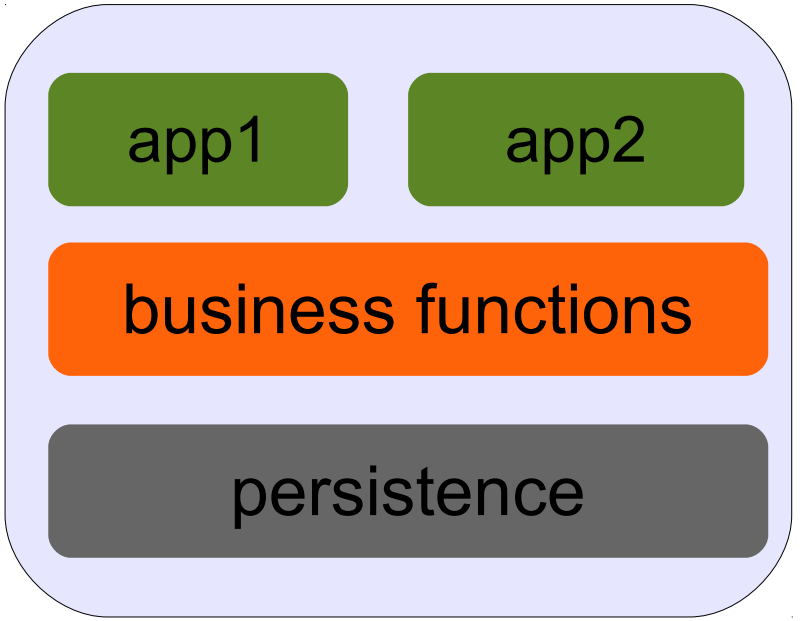


Onion skin model



Tree model

Layered Systems



Tiered Model

- specialization for enterprise applications
- tiers are generally physically separated (***so what?***)

Layered Systems

Invariants

- limit layer interactions to adjacent layers only
  - can be violated (how?)
- much richer interaction compared to pipeline
  - two way communication
- layers **must support** the **protocols** of its upper and lower **boundaries**

Layered Systems

Advantages

- increasing levels of ***abstraction***
- sub-component ***encapsulation***
- ***low coupling***
  - easy to maintain
  - a layer only interacts with a layer above and a layer below
- high (intra-layer) ***cohesion***
- ***modular reuse***
  - a layer can be replaced by another as long as the interface is not violated

Layered Systems

Disadvantages

- not all systems can be layered
  - why not?

Other Considerations

- performance
  - may force the high level functions to be tightly coupled with low level implementation
- layer abstraction
  - defining 'layer abstraction' is not always trivial

Repositories

Main idea

- **centralized source** of information with many components

Components

- central **data-store** component
  - represents system state/data
- collection of **data-use** components
  - collection of independent components operate on the central data-store

Connectors?

Repositories

Database

- **active**: incoming streams of transactions trigger processes to act on data-store

Blackboard

- **passive**: current state of the data-store triggers processes

Repositories

Advantages

- efficient when dealing with large amounts of data
  - known data schema
  - leads to ease of data sharing
  - centralized management

- clients are loosely coupled
  - why?

Repositories

Disadvantages

- data model
  - is static, bounded by defined schema
  - resistant to change as many depend on it
  - evolution is expensive

Interpreter Style

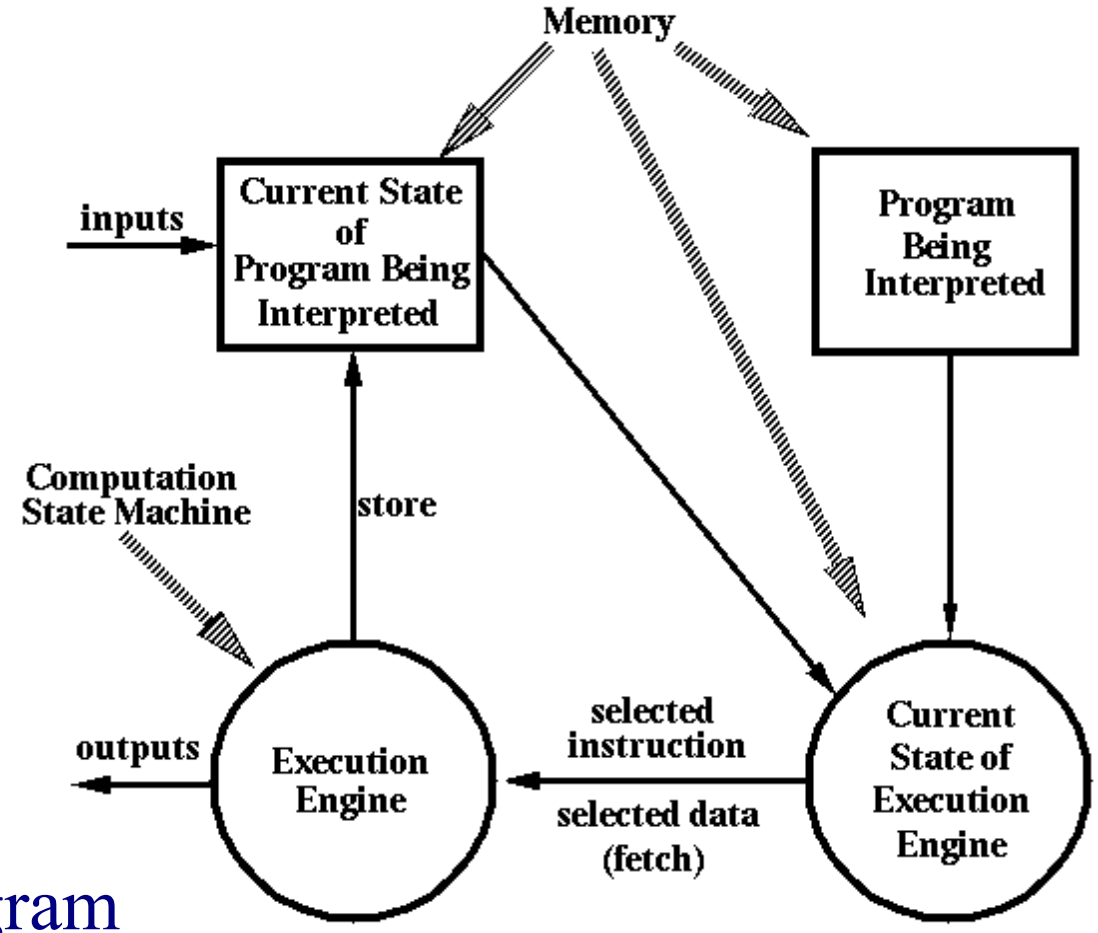
Main idea

- bridge functionality via software virtual machine
  - *"**suitable** for applications in which the most appropriate **language or machine** for executing the solution **is not directly available**"*

Interpreter Style

Components

- interpretation engine
  - to do the work
- memory
  - contains the psuedo-code & state
- control state of the engine
- current state of the program



Interpreter Style

Connectors

- procedure calls
- direct memory access
- Examples
  - programming language compilers (Java, small talk)
  - Scripting languages (awk, Perl)

Interpreter Style

Advantages

- simulation of non-implemented parts
- portability
  - across a variety of platforms

Disadvantages

- performance
  - computational complexity – slow execution

Further Reading

Microsoft Architectural Patterns and Styles

- http://msdn.microsoft.com/en-us/library/ee658117.aspx#ComponentBasedStyle