Software Architecture

Software architecture

- The design process for identifying the sub-systems making up a system and the framework for sub-system control and communication is architectural design
- The output of this design process is a description of the software architecture

Architectural design

- An early stage of the system design process
- Represents the link between specification and design processes
- Often carried out in parallel with some specification activities
- It involves identifying major system components and their communications

Advantages

- Stakeholder communication
  - Architecture may be used as a focus of discussion by system stakeholders
- System analysis
  - Means that analysis of whether the system can meet its non-functional requirements is possible
- Large-scale reuse
  - The architecture may be reusable across a range of systems

Architectural design decisions

- Is there a generic application architecture that can be used?
- (How) will the system be distributed?
- How will the system be decomposed into modules?
- How will the architectural design be evaluated?
- How should the architecture be documented?

Architecture reuse

- Systems in the same domain often have similar architectures that reflect domain concepts
- Application product lines are built around a core architecture with variants that satisfy particular customer requirements
- Examples?

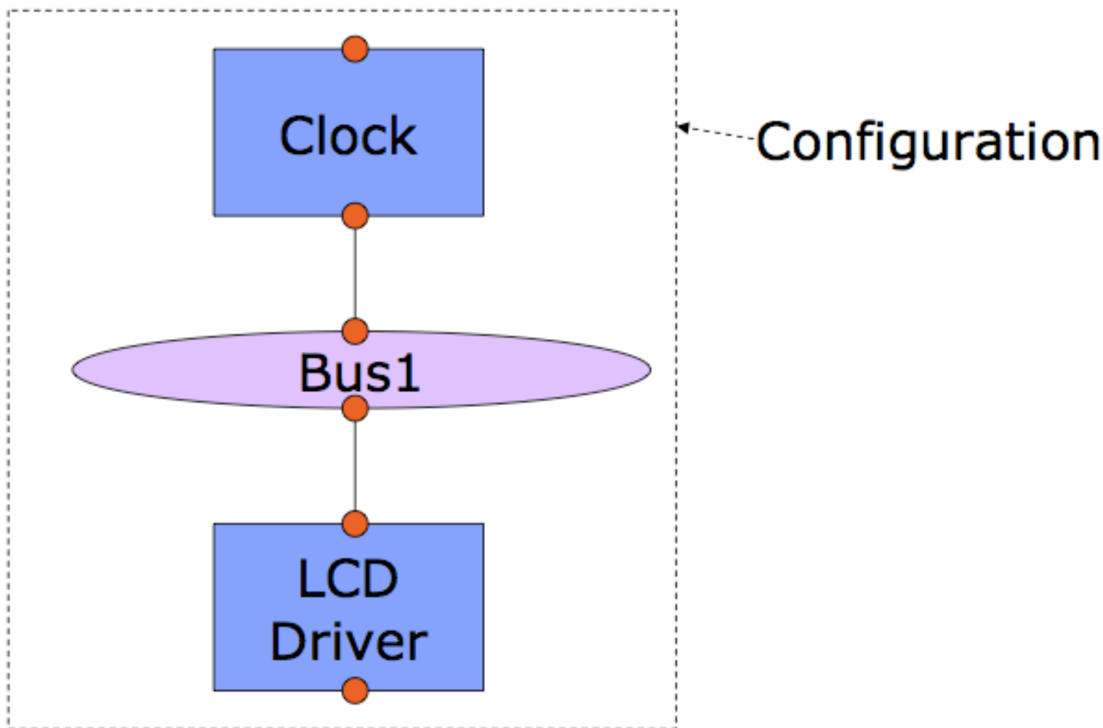
Architectural styles

- In order to create more complex software it is necessary to compose programming patterns
- For this purpose, it has been useful to induct a set of patterns known as "architectural styles"
- Examples:
  - Pipe and filter
  - Event based/event driven
  - Layered
  - Repository
  - Object-oriented
  - Model-view-controller

Software Architectural Styles

- Constituent parts:
  - Components
  - Connectors
  - Interfaces
  - Configurations

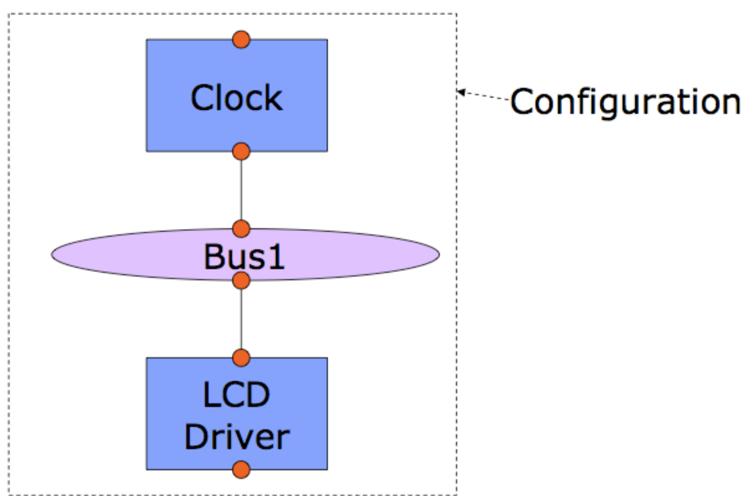
Example



What are components?

- Components are the loci of computation
  - Components "do the work" in the architecture
    - May be coarse-grained (an editor)
    - ...or fine grained (a clock emitting ticks)

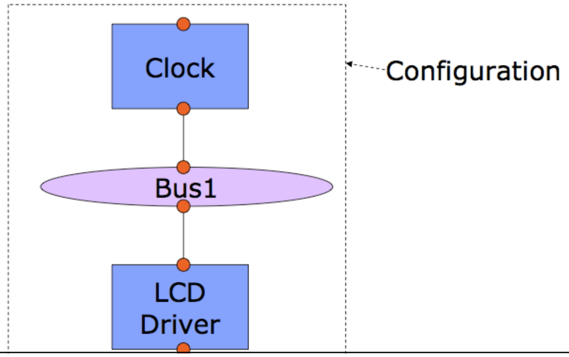
Eric M. Dashofy



What are connectors?

- Connectors are the loci of communication
  - Connectors facilitate communication among components
    - May be fairly simple (Broadcast Bus)
    - …or complicated (Middleware-enabled)
    - May be implicit (events)
    - …or explicit (procedure calls, ORBs, explicit communications bus)

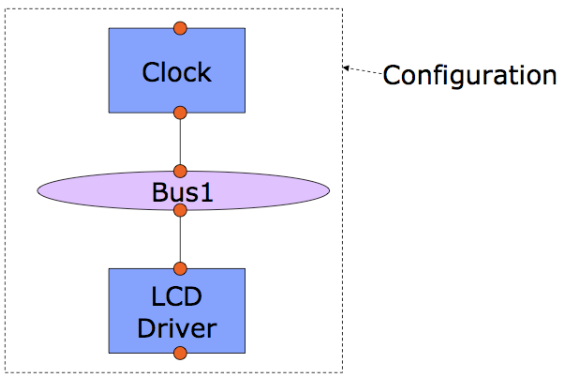
Eric M. Dashofy



What are interfaces?

- Interfaces are the connection points on components and connectors
  - They define where data may flow in and out of the components/connectors
    - May be loosely specified (events go in, events go out)
    - …or highly specified

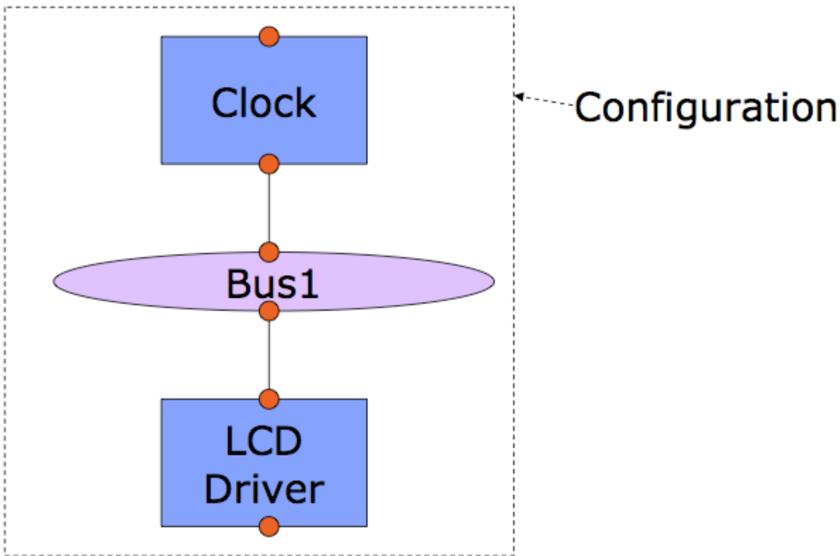
Eric M. Dashofy



What are configurations?

- Configurations are arrangements of components and connectors to form an architecture

Eric M. Dashofy

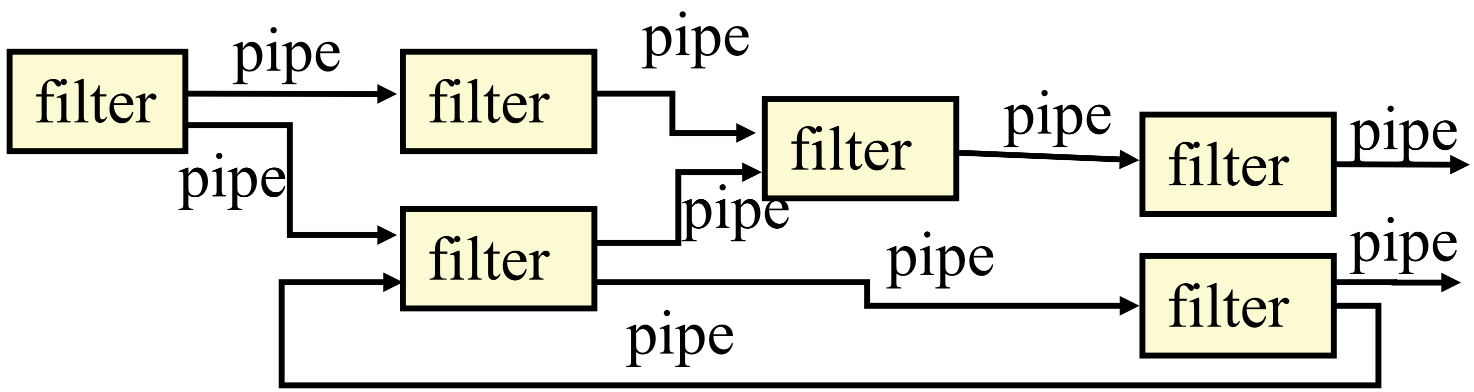


Common architectural styles

- Pipe-and-filter
- Repositories
- Event based systems (implicit invocation)
- Model-View-Controller
- Layered systems
- Object-oriented architectures

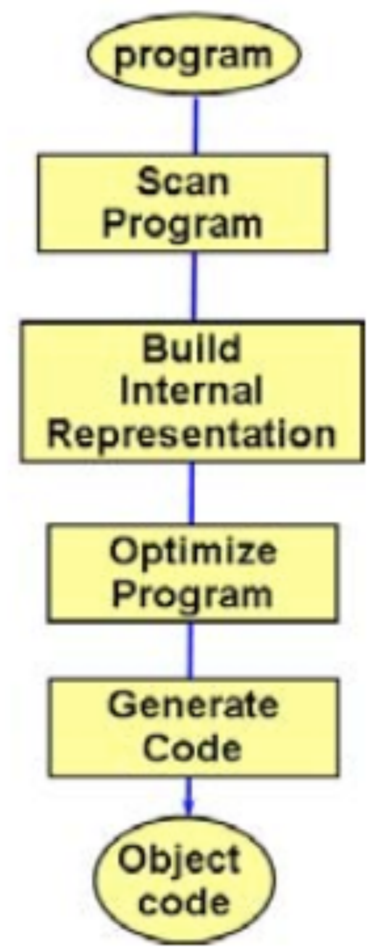
Pipe-and-filter

Source: *Adapted from Shaw & Garlan 1996, p21-2.*



- Data flow styles: Filters transform input into output
  - components: filters -computational i.e. retain minimal state
  - connectors: data streams
  - control: data flow between components
  - Topologies: arbitrary, acyclic, pipeline (linear)
- A series of independent, sequential transformations on ordered data.
- A component has a set of inputs and outputs.
- A component reads a stream of data and produces a stream of data.

Example: Traditional Compilers



Pipe-and-filter

- Examples:
  - UNIX shell commands
  - Compilers:
    - Lexical Analysis -> parsing -> semantic analysis -> code generation
  - Signal Processing
- Interesting properties:
  - filters are independent entities
  - filters don't need to know anything about what they are connected to

Example: Click

- Network routers are complicated pieces of code
- Eddie Kohler’s idea: Write routers as modular components connected by pipe-and-filter

Pipe-and-filter

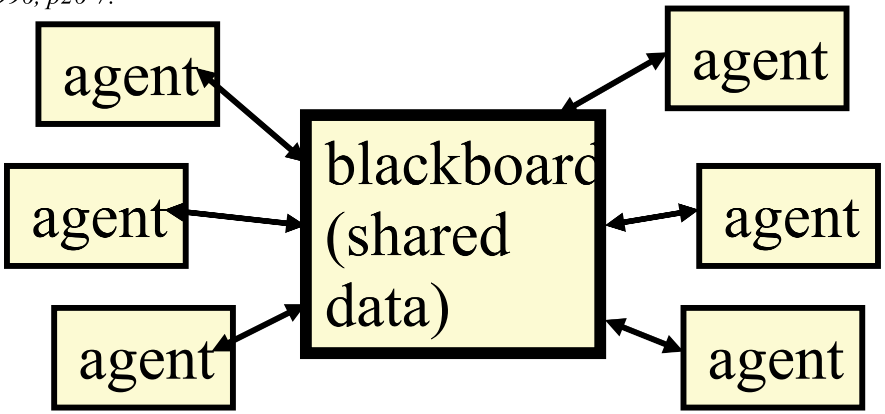
- Advantages
  - Overall behavior can be understood as a simple composition of the behaviors of individual filters
  - Support reuse
    - Existing filters can be hooked up to form a system
  - Easy maintenance and enhancement
    - New filters can replace old ones
  - They support parallel execution
  - Support specialized analysis, such as throughput and deadlock analysis

Pipe-and-filter

- Disadvantages
  - Not good for handling interactive applications
    - Complete transformation of input data to output data
  - Difficult to maintain correspondences between two separate but related streams
  - Extra overhead to parse and unparse data

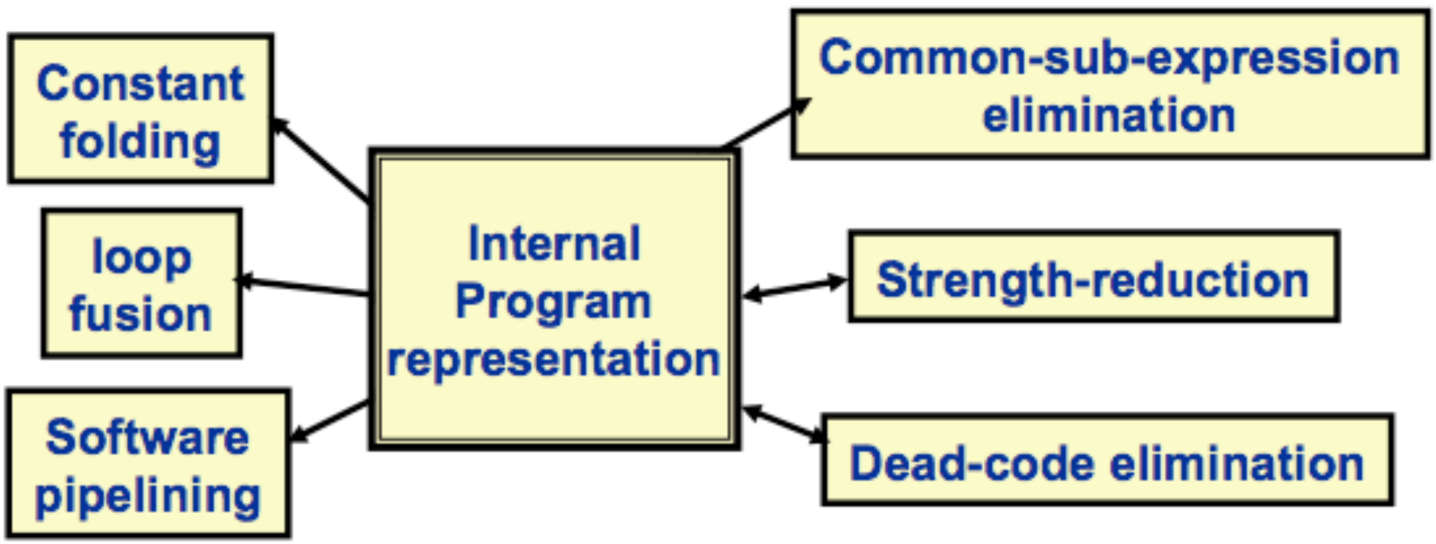
Repositories

Source: Adapted from Shaw & Garlan 1996, p26-7.



- Repositories: blackboard or database: Centralized data, usually structured
  - components: central data store, many computational objects
  - connectors: computational objects interact with central store directly or via method invocation
  - control: may be external, predetermined or internal
  - topology: star
- Maintaining and managing a richly structured body of information
- Data is long-lived and its integrity is important
- Data can be manipulated in different ways

Example: Compiler Optimization

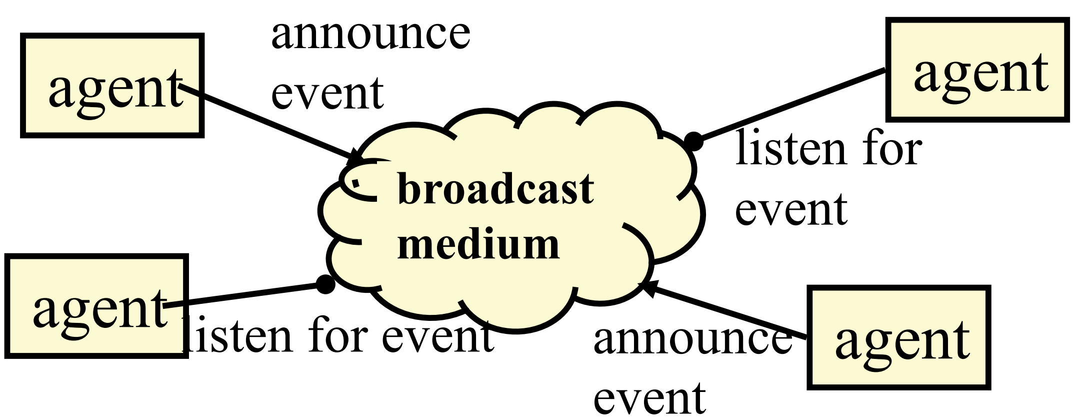


Repositories

- Examples
  - databases
  - modern compilers
  - programming environments
- Interesting properties
  - can choose where the locus of control is (agents, blackboard, both)
  - reduce the need to duplicate complex data
- Disadvantages
  - blackboard becomes a bottleneck

Event based (implicit invocation)

Source: *Adapted from Shaw & Garlan 1996, p23-4.*



- event based: implicit invocation: Independent reactive objects (or processes)
  - components: objects that register interest in "events" and objects that "signal events"
  - connectors: automatic method invocation
  - control: decentralized, de-coupling of sender and receiver
  - topologies: arbitrary

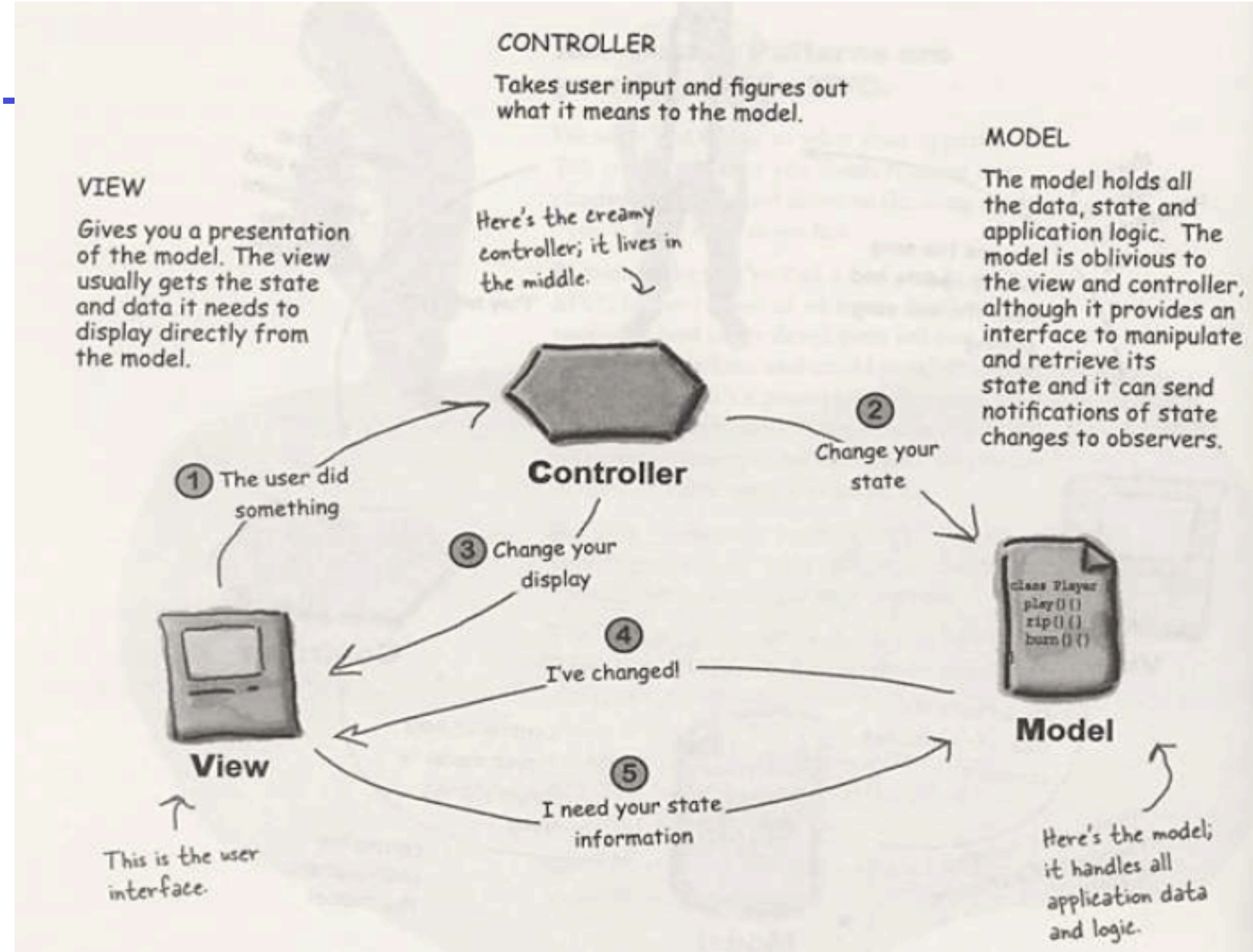
Event based (implicit invocation)

- Examples
  - debugging systems (listen for particular breakpoints)
  - database management systems (for data integrity checking)
  - graphical user interfaces
- Interesting properties
  - announcers of events don't need to know who will handle the event
  - Supports re-use
  - Supports evolution of systems (add new agents easily)
- Disadvantages
  - Components have no control over ordering of computations
  - Nor do they know when they are finished
  - Correctness reasoning is difficult

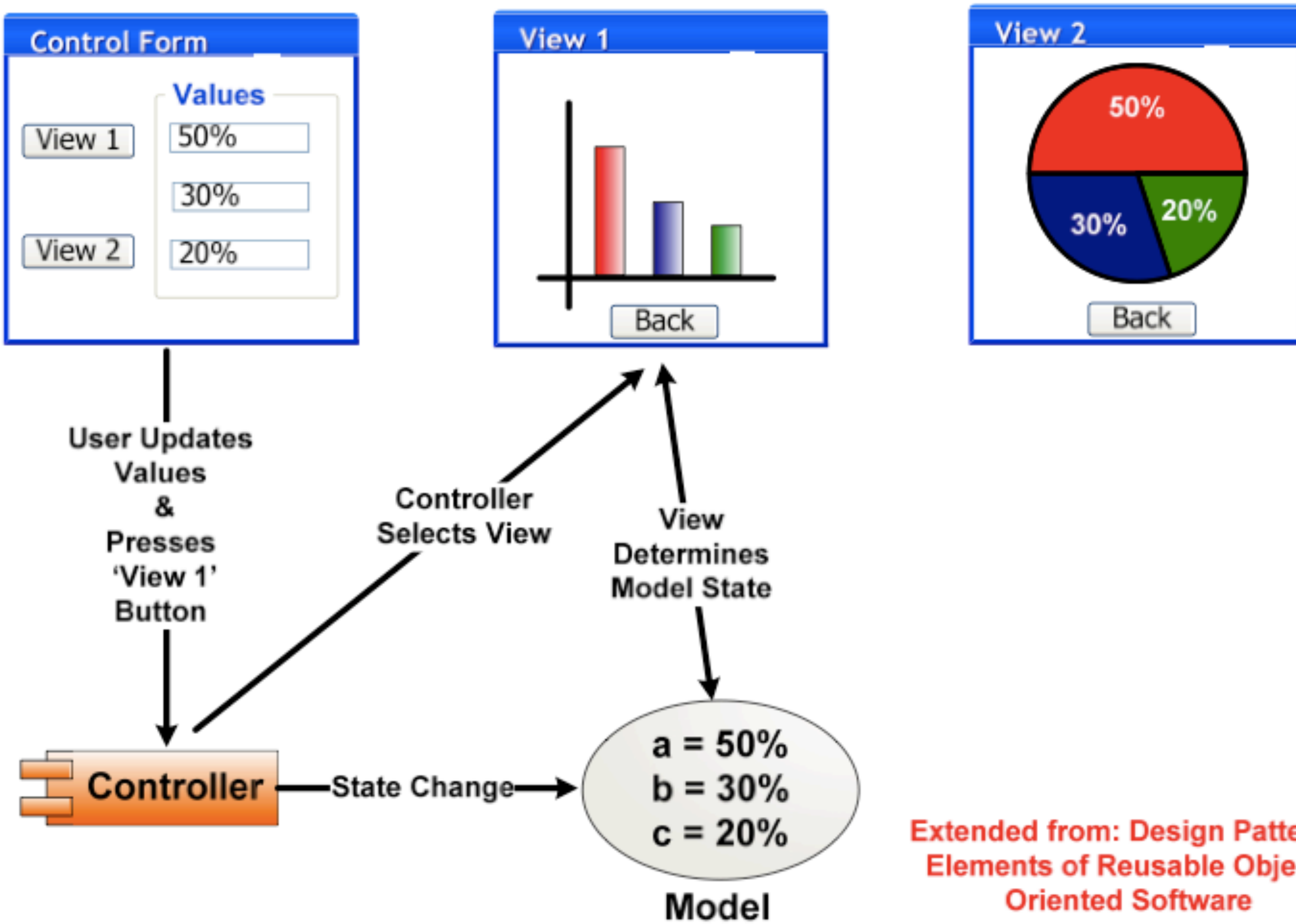
More Examples

- Web servers, file servers
- Embedded Systems
  - Sensor networks
- Business processes (Long running transactions)

Model-View-Controller



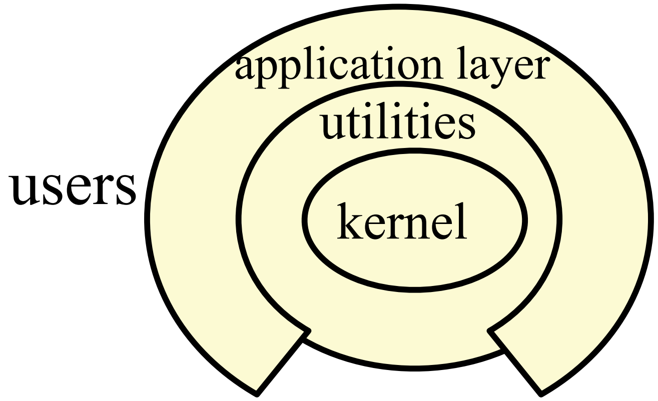
Example of Model-View-Controller



Extended from: Design Patterns Elements of Reusable Object-Oriented Software

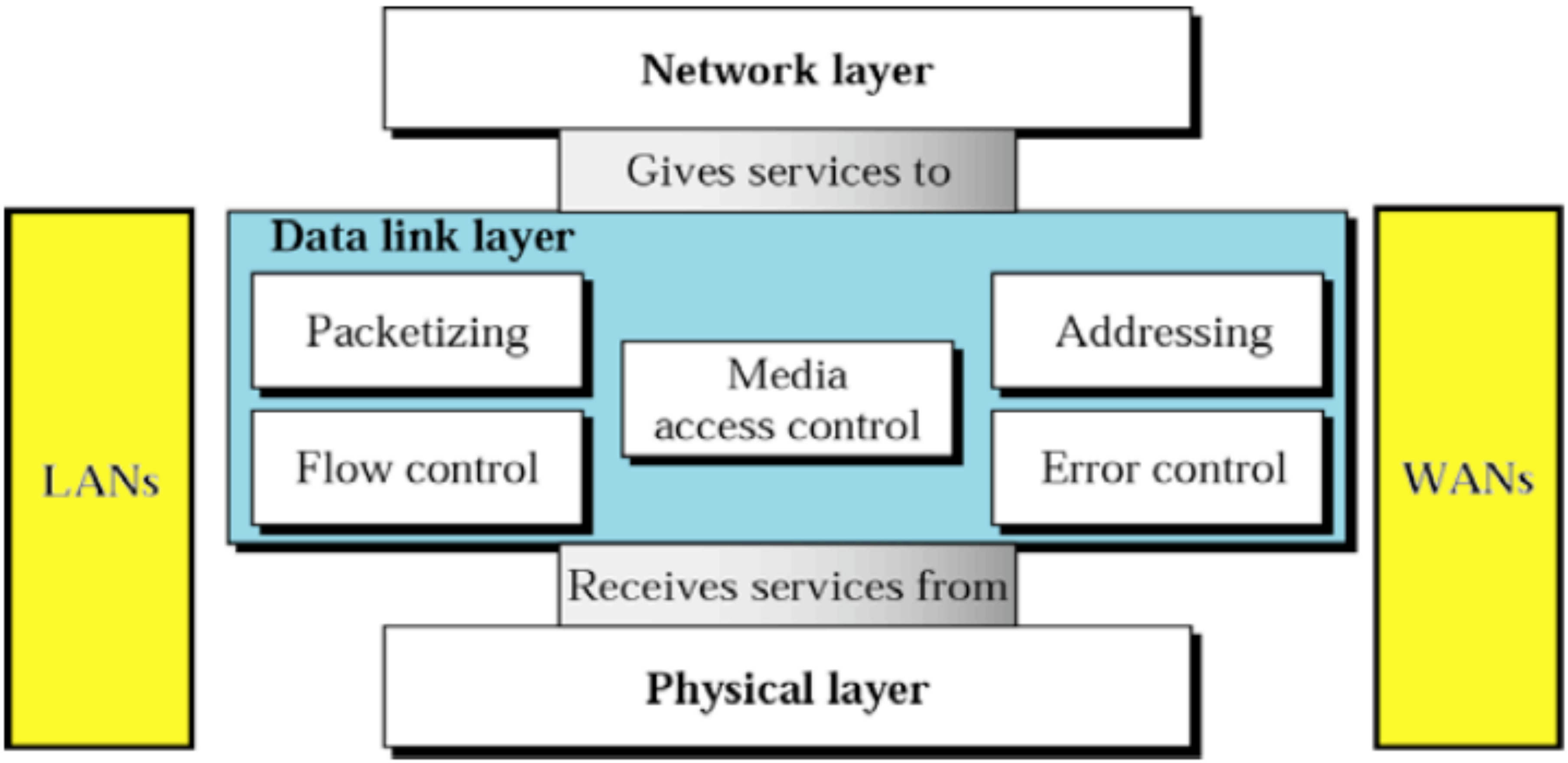
Layered Systems

Source: Adapted from Shaw & Garlan 1996, p25.



- Layered system: Independent processes
  - Components: processes
  - Connectors: protocols that determine how layers interact
  - Topologies: layered

Example: ISO Network Protocol



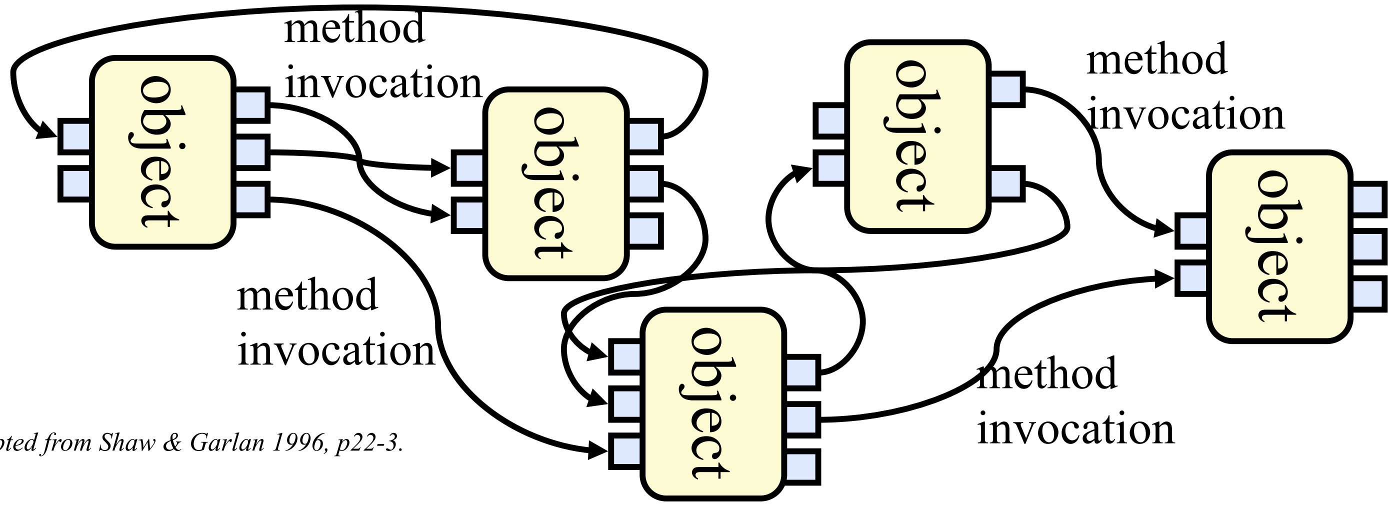
Layered Systems

- Examples
  - Operating Systems
  - Communication protocols
- Interesting properties
  - Support increasing levels of abstraction during design
  - Support enhancement (add functionality)
    - Change in one layer affects at most two layers
  - Reuse
    - can define standard layer interfaces
    - interchange implementations of same interface

Layered Systems

- Disadvantages
  - May not be able to identify (clean) layers
  - For performance reasons one layer may want to communicate with a non-adjacent layer
    - Especially true in real-time performance critical systems (cross layer design may be more efficient)

Object Oriented Architectures



Source: *Adapted from Shaw & Garlan 1996, p22-3.*

- Objected-oriented styles: data abstraction: Localized state maintenance (encapsulation)
  - components: managers
  - connectors: method invocation
  - control: decentralized
  - topologies: arbitrary

Object Oriented Architectures

- Examples:
  - abstract data types
  - object broker systems (e.g. CORBA)
- Interesting properties
  - data hiding (internal data representations are not visible to clients)
    - Can change implementation without affecting clients
  - can decompose problems into sets of interacting agents
- Disadvantage
  - objects must know the identity of objects they wish to interact with

Summary

- Architecture is what enables us to “scale up” our software to handle larger applications
- You must gain the ability to match the right architecture to the right application

Think About…

- How can architectures help reason about non-functional requirements?
  - Safety?
  - Security?
  - Dependability?
  - Performance?
  - Availability?
  - Maintainability?
  - …

Examples

- Performance
  - Localize critical operations and minimize communications. Use large rather than fine-grain components.
- Security
  - Use a layered architecture with critical assets in the inner layers
- Safety
  - Localise safety-critical features in a small number of sub-systems
- Availability
  - Include redundant components and mechanisms for fault tolerance
- Maintainability
  - Use fine-grain, replaceable components.

Conflicts

- Using large-grain components improves performance but reduces maintainability
- Introducing redundant data improves availability but makes security/consistency more difficult
- Localizing safety-related features usually means more communication so degraded performance

Distributed Architectures

Distributed systems architectures

- Client-server architectures
  - Distributed services which are called on by clients. Servers that provide services are treated differently from clients that use services.

- Distributed object architectures
  - No distinction between clients and servers. Any object on the system may provide and use services from other objects.

Middleware

- Software that manages and supports the different components of a distributed system. In essence, it sits in the middle of the system.
- Middleware is usually off-the-shelf rather than specially written software.
- Examples
  - Transaction processing monitors;
  - Data converters;
  - Communication controllers.

Multiprocessor architectures

- Simplest distributed system model.
- System composed of multiple processes which may (but need not) execute on different processors.
- Architectural model of many large real-time systems.
- Distribution of process to processor may be pre-ordered or may be under the control of a dispatcher.

Client-server architectures

- Application modeled as a set of
  - services provided by servers and
  - set of clients that use these services
  - Network for communication
- Clients know of servers but servers need not know of clients
- Clients and servers are logical processes

Client-server characteristics

- Advantages
  - Distribution of data is straightforward
  - Makes effective use of networked systems
  - Easy to add new servers or upgrade existing servers
- Disadvantages
  - No shared data model so sub-systems use different data organisation. Data interchange may be inefficient
  - Redundant management in each server;
  - No central register of names and services - it may be hard to find out what servers and services are available

Thin and fat clients

- Thin-client model
  - All of the application processing and data management is carried out on the server
  - Client is responsible for running the presentation software
- Fat-client model
  - Server is only responsible for data management
  - Software on the client implements the application logic and the interactions with the system user

Thin client model

- Used when legacy systems are migrated to client server architectures.
  - Legacy system acts as a server in its own right with a graphical interface implemented on a client.
- Disadvantage:
  - places a heavy processing load on both the server and the network
  - latency

Fat client model

- More processing is delegated to the client as the application processing is locally executed
- Most suitable for new systems where the capabilities of the client system are known in advance
- More complex than a thin client model especially for management. New versions of the application have to be installed on all clients.

Layered application architecture

- Presentation layer
  - Concerned with presenting the results of a computation to system users and with collecting user inputs
- Application processing layer
  - Concerned with providing application specific functionality e.g., in a banking system, banking functions such as open account, close account, etc.
- Data management layer
  - Concerned with managing the system databases

Three-tier architectures

- Allows for better performance than a thin-client approach
- Simpler to manage than a fat-client approach
- Recall: MVC
- A more scalable architecture - as demands increase, extra servers can be added.

Distributed object architectures

- There is no distinction in a distributed object architectures between clients and servers.
- Each distributable entity is an object that provides services to other objects and receives services from other objects.
- Object communication is through a middleware system called an object request broker.
- However, distributed object architectures are more complex to design than C/S systems.

Advantages

- Can delay decisions on where and how services should be provided
- New resources to be added to it as required
- Scaleable
- Possible to reconfigure the system dynamically with objects migrating across the network as required

CORBA

- CORBA is an international standard for an Object Request Broker - middleware to manage communications between distributed objects
- Middleware for distributed computing is required at 2 levels:
  - At the logical communication level, the middleware allows objects on different computers to exchange data and control information;
  - At the component level, the middleware provides a basis for developing compatible components. CORBA component standards have been defined.

Service-oriented architectures

- Based around the notion of externally provided services (web services)
- A web service is a standard approach to making a reusable component available and accessible across the web
  - A tax filing service could provide support for users to fill in their tax forms and submit these to the tax authorities

Services and distributed objects

- Provider independence
- Public advertising of service availability

- Opportunistic construction of new services through composition (e.g., mashups)

- Smaller, more compact, loosely coupled applications

Services standards

- Services are based on agreed, XML-based standards so can be provided on any platform and written in any programming language.
- Key standards
  - SOAP - Simple Object Access Protocol;
  - WSDL - Web Services Description Language;
  - UDDI - Universal Description, Discovery and Integration.

Automotive system