Design

Objectives

- To introduce design and to discuss its importance
- To describe the general design process
- To explain architectural design decisions that have to be made
- To introduce three complementary architectural styles covering organization, decomposition and control
- To discuss reference architectures are used to communicate and compare architectures

Conceptual design

- Tells the customer what the system will do
- Answers:
  - Where will the data come from?
  - What will happen to the data in the system?
  - What will the system look like to users?
  - What choices will be offered to users?
  - What is the timing of events?
  - What will the reports and screens look like?
- Characteristics of good conceptual design
  - in customer language with no technical jargon
  - describes system functions
  - independent of implementation
  - linked to requirements

Technical design

- Tells the programmers what the system will do
- Includes:
  - major hardware components and their function
  - hierarchy and function of software components
  - data structures
  - data flow
- Methods:
  - Modular decomposition
  - Data-oriented decomposition
  - Event-oriented decomposition
  - Outside-in design
  - Object-oriented design

Three design levels

- Architecture: associates system components with capabilities
- Code design: specifies algorithms and data structures for each component
- Executable design: lowest level of design, including memory allocation, data formats, bit patterns

Design styles

- Pipes and filters
- Object-oriented design
- Implicit invocation
- Layering
- Repositories
- Interpreters
- Process control
- Client-server

Discuss later

Object-oriented specifications

- Each entity in the system is an object.
- A method or operation is an action that can be performed directly by the object or can happen to the object.
- Encapsulation: the methods form a protective boundary around an object.
- Class hierarchies of objects encourage inheritance.
- Polymorphism: same method for different objects, each with different behavior
- Will not discuss in great detail. You should already be experts in OO techniques.

OO design

- Usually uses an OO requirements representation
- System design identifies and represents objects and classes, plus details of each objects attributes and behaviors.
- System design also identifies interactions and relationships.
- Program design inserts computational features in the models.
- Program design inserts class library details.
- Program design considers nonfunctional requirements to enhance design.

Important design issues

- Modularity and levels of abstraction
- Collaborative design
- Designing the user interface
  - metaphors, mental model, navigation rules, look and feel
  - cultural issues
  - user preferences
- Concurrency
- Design patterns and reuse

Issues to consider in trade-off analysis

(Lane, in Shaw and Garlan 1996)

Functional dimensions	*Structural dimensions*
External event handling:	Application interface abstraction level
• No external events	• Monolithic program
• Process events while waiting for input	• Abstract device
• External events preempt user commands	• Toolkit
	• Interaction manager with fixed data types
	• Interaction manager with extensible data types
	• Extensible interaction manager
User customizability	Abstract device variability
• High	• Ideal device
• Medium	• Parameterized device
• Low	• Device with variable operations
	• Ad hoc device
User interface adaptability across devices	Notation for user interface definition
• None	• Implicit in shared user interface code
• Local behavior changes	• Implicit in application code
• Global behavior change	• External declarative notation
• Application semantics change	• External procedural notation
	• Internal declarative notation
	• Internal procedural notation
Computer system organization	Basis of communication
• Uniprocessing	• Events
• Multiprocessing	• Pure state
• Distributed processing	• State with hints
	• State plus events
Basic interface class	Control thread mechanisms
• Menu selection	• None
• Form filling	• Standard processes
• Command language	• Lightweight processes
• Natural language	• Non-preemptive processes
• Direct manipulation	• Event handlers
	• Interrupt service routines
Application portability across user interface styles	
• High	
• Medium	
• Low	

Characteristics of good design

- Component independence
  - coupling
  - cohesion
- Exception identification and handling
- Fault prevention and tolerance
  - active
  - passive

Techniques for improving design

- Reducing complexity
- Design by contract
- Prototyping design
- Fault-tree analysis

Design evaluation and validation

- Mathematical validation
- Measuring design quality
- Comparing designs
  - one specification, many designs
  - comparison table
- Design reviews

Weighted comparison of Shaw and Garlan designs

Attribute	***Priority***	*Shared data*	*Abstract data type*	*Implicit invocation*	*Pipe and filter*
Easy to change algorithm	1	1	2	4	5
Easy to change data representation	4	1	5	2	1
Easy to change function	3	4	1	4	5
Good performance	3	5	4	2	2
Easy to reuse	5	1	4	2	5

Design reviews

- Preliminary design review
  - examines conceptual design with customer and users
- Critical design review
  - presents technical design to developers
- Program design review
  - programmers get feedback on their designs before implementation
- We'll discuss inspections, a more formalized review, shortly

Questions for any design review

Is it a solution to the problem?
Is it modular, well-structured, and easy to understand?
Can we improve the structure and understandability?
Is it portable to other platforms?
Is it reusable?
Is it easy to modify or expand?
Does it support ease of testing?
Does it maximize performance, where appropriate?
Does it reuse components from other projects, where appropriate?
Are the algorithms appropriate, or can they be improved?
If this system is to have a phased development, are the phases interfaced sufficiently so that there is an easy transition from one phase to the next?
Is it well-documented, including design choices and rationale?
Does it cross-reference the components and data with the requirements?
Does it use appropriate techniques for handling faults and preventing failures?

Documenting the design

- design rationale
- menus and other display-screen formats
- human interfaces: function keys, touch screen descriptions, keyboard layouts, use of a mouse or joystick
- report formats
- input: where data come from, how they are formatted, on what media they are stored
- output: where data are sent, how they are formatted, on what media they are stored
- general functional characteristics
- performance constraints
- archival procedures
- fault-handling approach

Software architecture

- This process used to be called high level design or preliminary design.
- 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.
- In your project you will be given an architecture and need to understand it before making changes.

Advantages of explicit architecture

- 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.

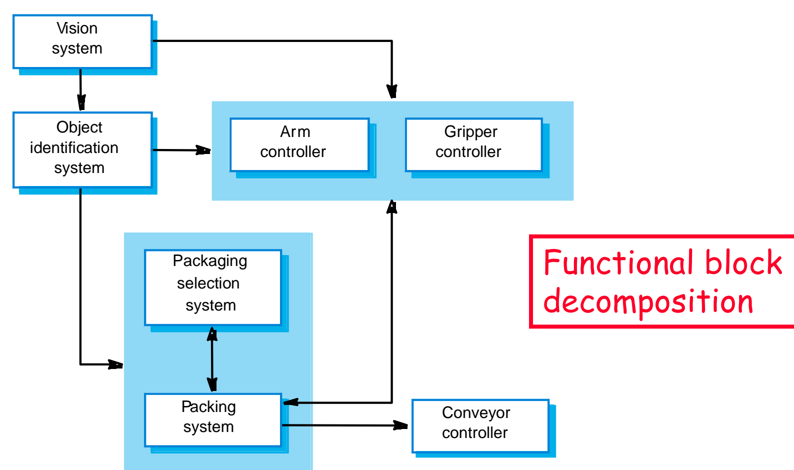
Architectural conflicts

- Basic conflicts are between performance and (safety, reliability, or maintainability)
- Using large-grain components improves performance but reduces maintainability.
- Introducing redundant data improves availability but makes security more difficult.
- Localizing safety-related features usually means more communication so degraded performance.

System structuring

- Concerned with decomposing the system into interacting sub-systems.
- The architectural design is normally expressed as a block diagram presenting an overview of the system structure.
- More specific models showing how sub-systems share data, are distributed and interface with each other may also be developed.

Packing robot control system



Functional block decomposition

Box and line diagrams

- Very abstract - they do not show the nature of component relationships nor the externally visible properties of the sub-systems.
- However, useful for communication with stakeholders and for project planning.

Architectural design decisions

- Is there a generic application architecture that can be used?
- How will the system be distributed?
- What architectural styles are appropriate?
- What approach will be used to structure the system?
- How will the system be decomposed into modules?
- What control strategy should be used?
- 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.
- The architectural model of a system may conform to a generic architectural model or style.
- Application architectures are covered in Chapter 13 and product lines in Chapter 18.

Sample architectural models

- Used to document an architectural design.
- Static structural model that shows the major system components.
- Dynamic process model that shows the process structure of the system.
- Interface model that defines sub-system interfaces.
- Relationships model such as a data-flow model that shows sub-system relationships.
- Distribution model that shows how sub-systems are distributed across computers.

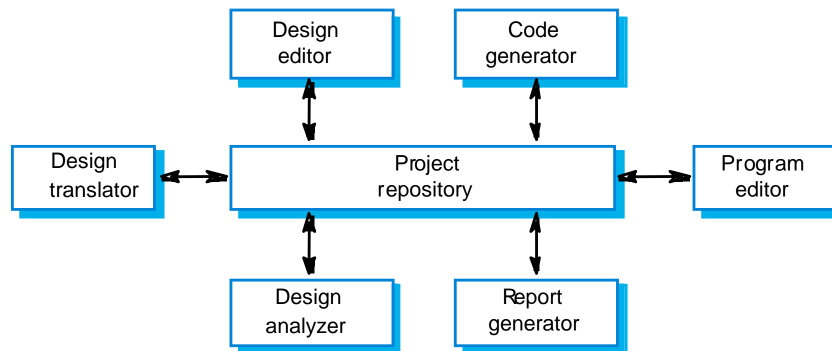
System organization

- Reflects the basic strategy that is used to structure a system.
- Three organizational styles are widely used:
  1. A shared data repository style;
  2. A client - server style;
  3. An abstract machine or layered style.

1. The repository model

- Sub-systems must exchange data. This may be done in two ways:
  - ❑ Shared data is held in a central database or repository and may be accessed by all sub-systems;
  - ❑ Each sub-system maintains its own database and passes data explicitly to other sub-systems.
- When large amounts of data are to be shared, the repository model of sharing is most commonly used.

Example: CASE toolset architecture



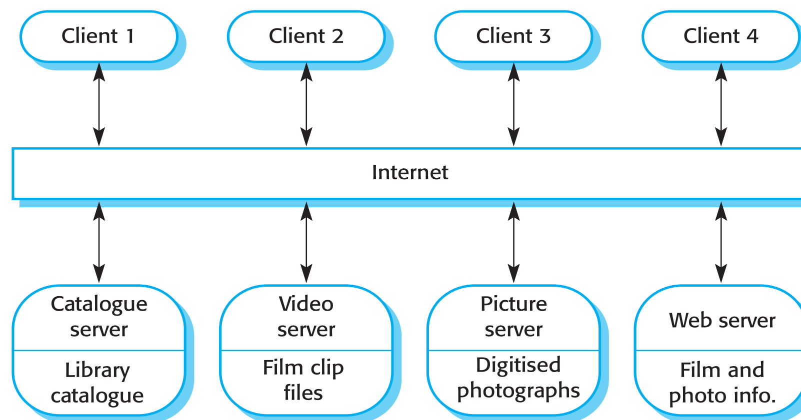
Repository model characteristics

- Advantages
  - Efficient way to share large amounts of data;
  - Sub-systems need not be concerned with how data is produced. Centralized management e.g. backup, security, etc.
  - Sharing model is published as the repository schema.
- Disadvantages
  - Sub-systems must agree on a repository data model. Inevitably a compromise;
  - Data evolution is difficult and expensive;
  - No scope for specific management policies;
  - Higher performance, but harder to maintain that other approaches;
  - Difficult to distribute efficiently.

2. Client-server model

- Distributed system model which shows how data and processing is distributed across a range of components.
- Set of stand-alone servers which provide specific services such as printing, data management, etc.
- Set of clients which call on these services.
- Network which allows clients to access servers.

Film and picture library



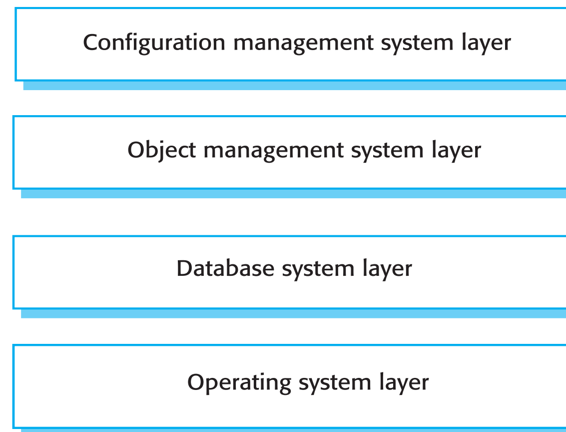
Client-server characteristics

- Advantages
  - Distribution of data is straightforward;
  - Makes effective use of networked systems. May allow for cheaper hardware;
  - Can more easily process data in parallel;
  - Strict protocol between client and server. Can more easily add new clients or new servers or upgrade existing servers.
- Disadvantages
  - No shared data model so sub-systems use different data organization. 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.

3. Abstract machine (layered) model

- Also called the virtual machine model (e.g., it is how the Java Virtual Machine [JVM] is built into web browsers)
- Used to model the interfacing of sub-systems.
- Organizes the system into a set of layers (or abstract machines) each of which provide a set of services.
- Supports the incremental development of sub-systems in different layers. When a layer interface changes, only the adjacent layer is affected.
- However, often artificial to structure systems in this way.

Version management system



Modular decomposition styles

- Styles of decomposing sub-systems into modules.
- No rigid distinction between system organization and modular decomposition.
- A sub-system is a system in its own right whose operation is independent of the services provided by other sub-systems.
- A module is a system component that provides services to other components but would not normally be considered as a separate system.

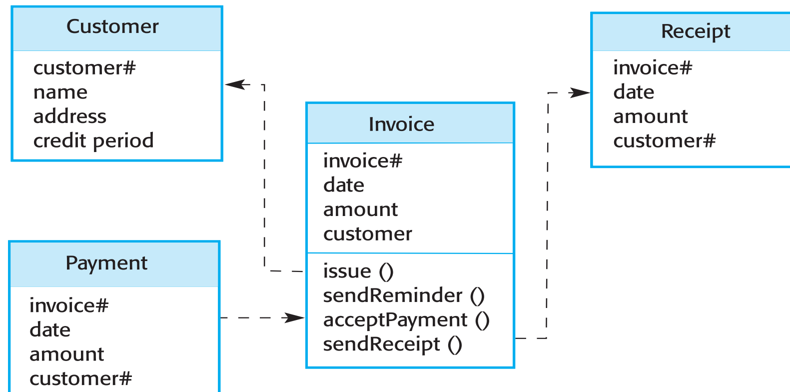
Modular decomposition

- Two modular decomposition models covered
  - An object model where the system is decomposed into interacting object;
  - A pipeline or data-flow model where the system is decomposed into functional modules which transform inputs to outputs.
- If possible, decisions about concurrency should be delayed until modules are implemented.

Object models

- Structure the system into a set of loosely coupled objects with well-defined interfaces.
- Object-oriented decomposition is concerned with identifying object classes, their attributes and operations.
- When implemented, objects are created from these classes and some control model used to coordinate object operations.
- You should be fairly good at building object models by now.

Invoice processing system



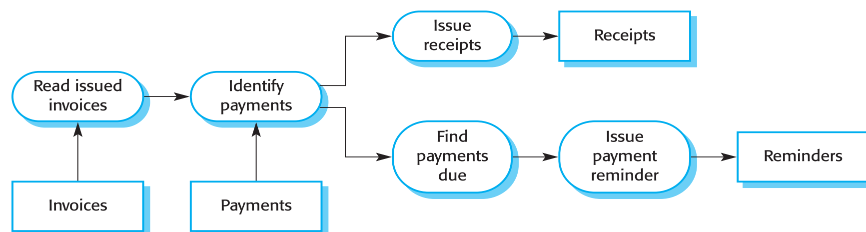
Object model advantages

- Objects are loosely coupled so their implementation can be modified without affecting other objects.
- The objects may reflect real-world entities.
- OO implementation languages are widely used.
- However, object interface changes may cause problems and complex entities may be hard to represent as objects.

Function-oriented pipelining

- Functional transformations process their inputs to produce outputs.
- May be referred to as a pipe and filter model (as in UNIX shell).
- Variants of this approach are very common. When transformations are sequential, this is a batch sequential model which is extensively used in data processing systems.
- Not really suitable for interactive systems.

Invoice processing system



Pipeline model advantages

- Supports transformation reuse.
- Intuitive organization for stakeholder communication.
- Easy to add new transformations.
- Relatively simple to implement as either a concurrent or sequential system.
- However, requires a common format for data transfer along the pipeline and difficult to support event-based interaction.

Control styles

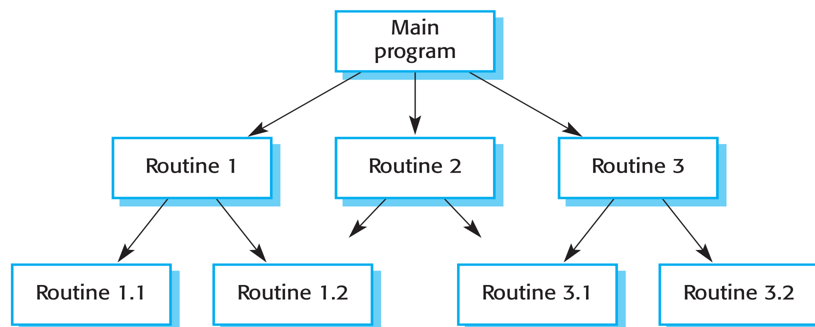
- Are concerned with the control flow between sub-systems. Distinct from the system decomposition model.
- Centralized control
  - One sub-system has overall responsibility for control and starts and stops other sub-systems.
- Event-based control
  - Each sub-system can respond to externally generated events from other sub-systems or the system's environment.

Centralized control

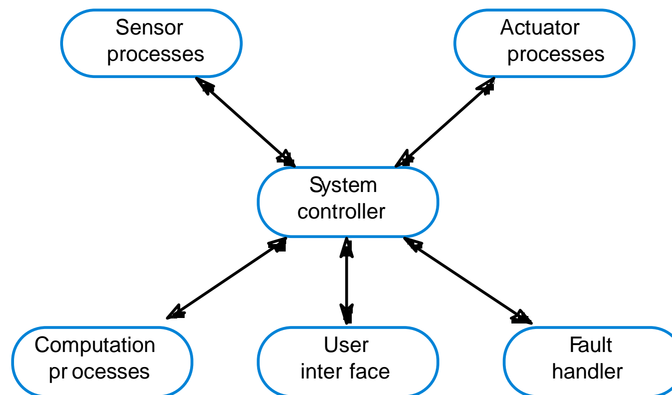
- A control sub-system takes responsibility for managing the execution of other sub-systems.
- Call-return model
  - Top-down subroutine model where control starts at the top of a subroutine hierarchy and moves downwards. Applicable to sequential systems.
- Manager model
  - Applicable to concurrent systems. One system component controls the stopping, starting and coordination of other system processes. Can be implemented in sequential systems as a case statement.

Call-return model

Traditional program control



Real-time system control



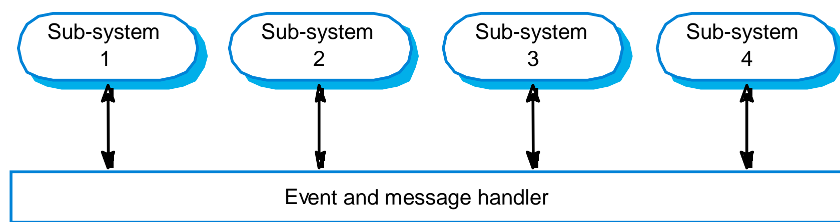
Event-driven systems

- Driven by externally generated events where the timing of the event is out with the control of the sub-systems which process the event.
- Two principal event-driven models
  - Broadcast models. An event is broadcast to all sub-systems. Any sub-system which can handle the event may do so (e.g., ethernet);
  - Interrupt-driven models. Used in real-time systems where interrupts are detected by an interrupt handler and passed to some other component for processing.
- Other event driven models include spreadsheets and production systems.

Broadcast model

- Effective in integrating sub-systems on different computers in a network.
- Sub-systems register an interest in specific events. When these occur, control is transferred to the sub-system which can handle the event.
- Control policy is not embedded in the event and message handler. Sub-systems decide on events of interest to them.
- However, sub-systems don't know if or when an event will be handled.

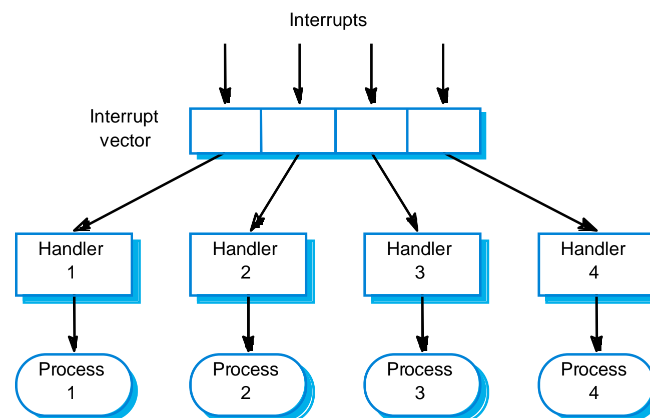
Selective broadcasting



Interrupt-driven systems

- Used in real-time systems where fast response to an event is essential.
- There are known interrupt types with a handler defined for each type.
- Each type is associated with a memory location and a hardware switch causes transfer to its handler.
- Allows fast response but complex to program and difficult to validate.

Interrupt-driven control



Module design issues

From previous discussion on architectural styles:

- Want to use data abstractions to divide modules into separately understood and executing components.
- Object oriented design is a mechanism to build independent operations in each module according to these principles.

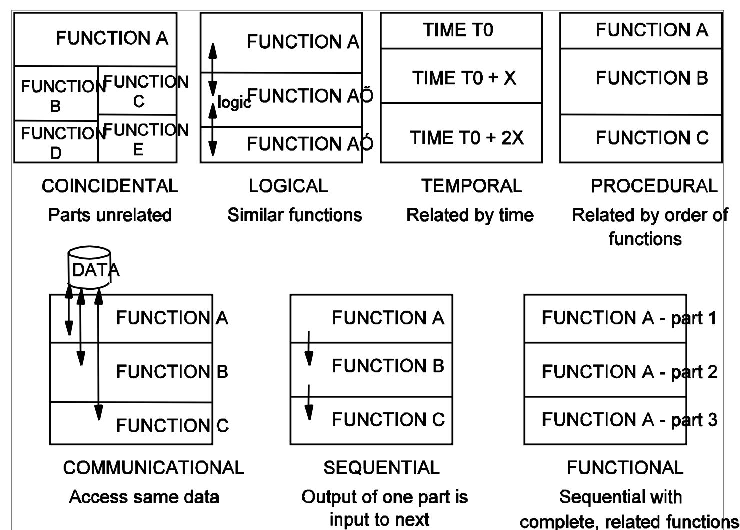
Myers and Constantine developed (in 1970s) a way to characterize degree of "separation of concerns" in module designs: Cohesion and coupling.

- Cohesion: The degree of interaction within a module
- Coupling: The degree of interaction between two modules

Goal: A design should have high cohesion and low coupling.

Note: These are interesting concepts, but one shouldn't take them too seriously - somewhat subjective.

Cohesion



Cohesion

Remember: Higher is considered better in a good design.

1. Coincidental cohesion - A module has coincidental cohesion if it performs multiple, unrelated actions.

Usual derivation: Clumping together unrelated functions for "convenience."

Example: A module of all "leftover" functions that are under 10 statements long:

Read next character

Compute square root of argument

Command to clear window on screen

...

Cohesion - II

2. Logical cohesion - A module has logical cohesion when it performs a series of related actions.

Examples:

Modules to do input and output

Modules to perform editing operations

Modules to apply similar number conversions

3. Temporal cohesion - A module has temporal cohesion when it performs a series of actions related in time.

All functions in the module occur in the same time period, but are not necessarily functionally related.

Example:

In a compiler, read character, create token, produce symbol table entry, as the scanner phase

Functions to read input data and process it

(Question: aren't these more weakly related than logical cohesion?)

Cohesion-III

4. Procedural cohesion - A module has procedural cohesion if it follows a series of actions related by the sequence of steps to be followed by the product.

Example:

All operations involving the symbol table of a compiler

All X-windows command operations

5. Communicational cohesion - A module has communicational cohesion if it performs a series of actions related by the sequence of steps to be followed by the product and if all actions are performed on the same data.

Example:

Functions for symbol table records involving name fields

Functions for symbol table records involving scalar data fields

Cohesion-IV

6. Informational cohesion - A module has informational cohesion if it performs a number of actions, each with its own entry point, with independent code for each action, all performed on the same data structure.

Example:

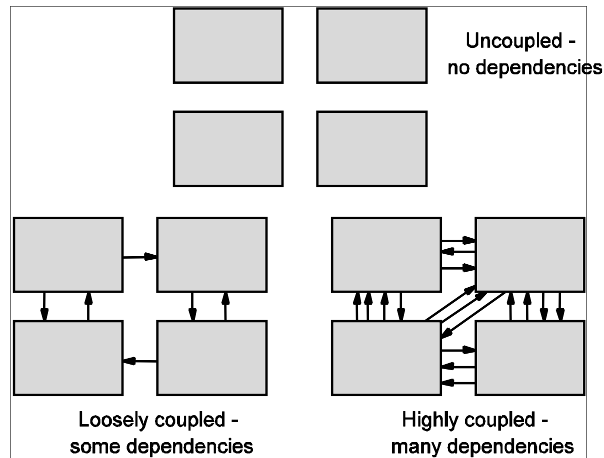
Abstract data types are examples of this, but these are not necessarily abstract data types.

7. Functional cohesion - A module that performs exactly one action or achieves a single goal has functional cohesion.

Example:

X-Windows, Parse program

Coupling



Coupling

Remember - Low coupling is considered better in a good design. Following are high to low coupling order:

5. Content coupling - Two modules are content-coupled if one directly references the content of the other.

Example: One module uses the internal data structures of another. (Definitely NOT an abstract data type.)

4. Common coupling - Two modules are common-coupled if they have access to the same global data.

Example: Most modules designed in industry.

Coupling-II

3. Control coupling - Two modules are control-coupled if one passes an element of control to the other.

Example: One module passes a switch variable that tells other module what action to take. (e.g., called modules does: If switch

then do operation 1

else do operation 2)

2. Stamp coupling - Two modules are stamp-coupled if data structures are passed as arguments, but the called module operates on only some of the data.

Example: A "user" record is passed from module A to module B, but B does not use all of the fields in the record.

Coupling-III

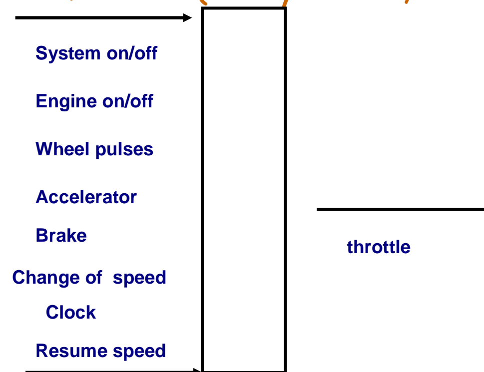
1. Data coupling - Two modules are data-coupled if all arguments are homogeneous data items. That is, all arguments are either simple arguments or data structures where all the fields are referenced.

In this example, a data abstraction would be passed as a simple argument to modules. Only the defining module accesses the components of the record.

Again: These are interesting goals, desirable in a design, but their characterization if somewhat subjective, so it shouldn't be an overwhelming goal.

Example: Cruise Control

IEEE Software (Mary Shaw, 11/1995)

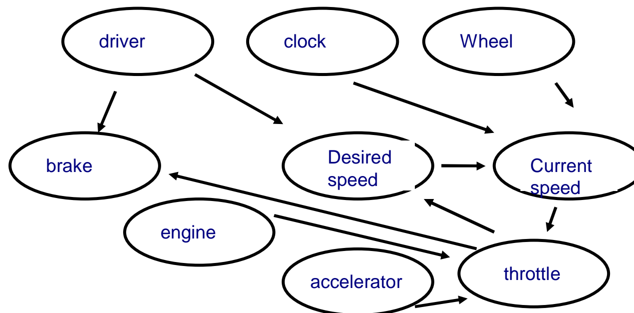


Cruise control - II

System on/off – control cruise control
Engine on/off – car active
Wheel pulses – one per revolution
Brake – if pressed, cruise control suspended
Accelerator – How far accelerator depressed
Change of speed – Increase or decrease speed if cruise control is on
Resume speed – If cruise control is on, last maintained speed resumes
Clock – Pulse every millisecond

Example Architectural Styles

- Object—oriented architectures: Systems are discrete objects that encapsulate state and definitions of operations.
- Similar to the encapsulated data types previously discussed. The various models of OO design differ in how they define objects, define sequencing among objects, and how a design satisfies requirements. – Booch's object oriented diagram



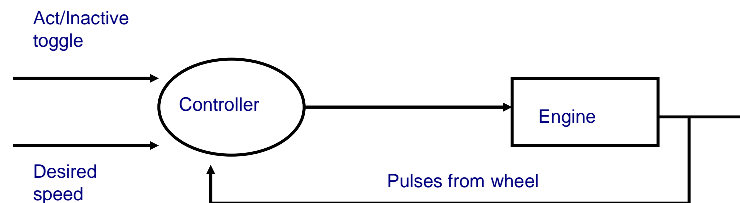
State-Based architecture

- Focus on the major modes, or states, of the system, and on the events that cause transition between 2 states.
- Example Gannon – Atlee

Current mode	Ignited	Running	Too fast	Brake	Act.	Deact	Resume	New mode
Off	@T	---	---	---	---	---	---	Inactive
Inactive	@F	---	---	---	---	---	---	Off
	t	t	---	f	@T	---	---	Cruise
Cruise	@F	---	---	---	---	---	---	Off
	---	@F	---	---	---	---	---	Inactive
	---	---	@T	---	---	---	---	
	---	---	---	@T	---	---	---	Override
	---	---	---	---	---	@T	---	
Override	@F	---	---	---	---	---	---	Off
	---	@F	---	---	---	---	---	Inactive
	t	t	---	F	@T	---	---	Cruise
	t	t	---	f	---	---	@T	

Feedback control architectures

A special form of dataflow architecture adapted for embedded systems. They control a process by continuously available signals. (Shaw design)



Real time architectures

Add stringent response time characteristics. Each column is set of processes to execute for that state. (Ward and Keskar design)

Process/Action	Capture current speed	Increase speed	Maintain speed
Stop accelerating	0	0	0
Accelerate	0	1	0
Maintain speed	0	0	1
Accel. To desired speed	0	1	0
Get and maintain speed	1	0	1
Stop maintaining speed	0	0	0
Turn CC off	0	0	0
Cruise control ready	0	0	0

Design evaluation criteria

- Separation of concerns:
  - Does design separate independent parts of the system?
  - Is redundant information avoided?
  - Are closely related parts grouped together? Locality allows for an "interchangeable parts" strategy.
- Generally:
  - OO focuses on real-world entities
  - State designs focus on a system's operational modes
  - Process-control designs focus on feedback relationships
  - Real-time designs focus on events and their order of execution

Design evaluation criteria

- Perspicuity of design:
  - Does the expression of the design correspond to the problem being solved?
  - Is there traceability to the important requirements?
- Ability to check and analyze design:
  - Is the design easy to analyze and check for correctness, performance, and other properties of interest?
  - If more than one model is used, is the interaction between them easy to follow?
- Abstraction power:
  - Does the design highlight and hide appropriate details?
  - Does it aid the designer to avoid premature implementation details?

Design evaluation criteria

- Safety:
  - Can you ensure that the system will not enter an unsafe state?
  - Can you ensure that you can always return to a safe termination state? (e.g., in the cruise control example, coasting down a steep hill is outside of the domain of the cruise control since only the throttle and not the brakes are controlled.)
  - Is the model of the real—world within the system the same as outside the system? (e.g., is the definition of speed in the cruise control the same as a vehicle's actual speed?)

Design evaluation criteria

- Integration with hardware:
  - How well do the modeled controls match the physical controls?
  - How do the manual and automatic modes interact?
  - What are the characteristics of real—time response? (How rapidly does the system react to control inputs? How accurately does the system maintain speed?)

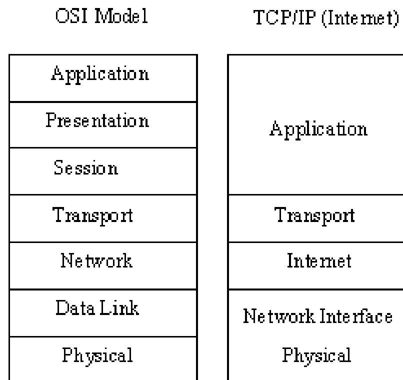
Reference architectures

- Architectural models may be specific to some application domain.
- Two types of domain-specific model
  - Generic models which are abstractions from a number of real systems and which encapsulate the principal characteristics of these systems. Covered in Chapter 13.
  - Reference models which are more abstract, idealized model. Provide a means of information about that class of system and of comparing different architectures.
- Generic models are usually bottom-up models; Reference models are top-down models.

Reference architectures

- Reference models are derived from a study of the application domain rather than from existing systems.
- May be used as a basis for system implementation or to compare different systems. It acts as a standard against which systems can be evaluated.
- OSI model is a layered model for communication systems.

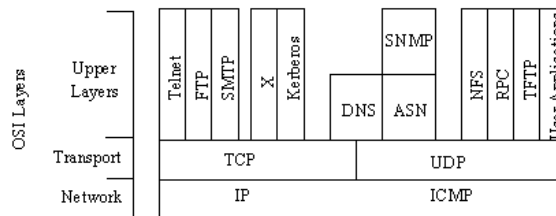
OSI reference model



TCP-IP Protocols

Telnet - RemoteLogin
FTP - File Transfer Protocol
SMTP - Simple Mail Transfer Protocol
X - X Windows System
Kerberos - Security
DNS - Domain Name System
ASN - Abstract Syntax Notation
SNMP - Simple Network Management Protocol

NFS - Network File Server
RPC - Remote Procedure Calls
TFTP - Trivial File Transfer Protocol
TCP - Transmission Control Protocol
User Datagram Protocol
IP - Internet Protocol
ICMP - Internet Control Message Protocol



ECMA Framework Reference Model

A framework provides a common set of services needed to support application programs written for the domain of the environment. For a software engineering environment that means framework services support tools which aid in the design, development, deployment and management of the software development process.

In 1988 ECMA began work on describing such a framework reference model. The initial version was published in 1990. It included heavy emphasis on the data repository aspects of the framework.

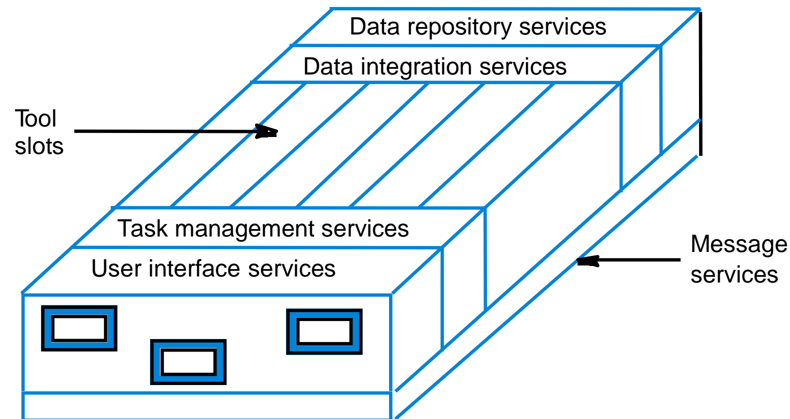
Beginning in 1989, NIST's Integrated Software Engineering Environment (ISEE) Working Group, working jointly with ECMA, extended the model to include a greater variety of services. The original model was revised in 1991 and again in 1993.

The model is just a catalog of services that may be applicable to an environment framework. There was no implied architecture in describing any of the services. HOWEVER, it is often used as a preliminary architecture for such systems.

Reference Model Services

1. The 21 object management services provide for the storage, retrieval and management of a persistent object store.
2. The 6 process management services provide for developing process models of the development life cycle and the management of these processes within the environment.
3. The 5 communication services provide for communication among tools using communication paths like messages, RPCs and shared data storage.
4. The 8 policy enforcement services provide the confidentiality, security and integrity requirements on an environment.
5. The 10 user interface services provide the interface between the executing program and the user who is interacting with the environment.
6. The 6 framework administration services provide for the installation and tailoring of tools, users and other objects into the framework.
7. The 10 operating system services provide basic system functionality.

The ECMA reference model



"Toaster" Model – Usual representation

Standards

Who defines a language?:

- Example is: I = 1 & 2 + 3 | 4 legal in C, and what is assigned to I?

Three ways typically to answer this:

- ❑ Read language manual (Problem: Can you find one?)
- ❑ Read language standard (Problem: Have you ever seen it?)
- ❑ Write a program to see what happens. (Easy to do!)

Most do option 3, but current compilers may not give correct answer.

- Standards are a mechanism so that users and vendors of various products can build systems easily using interchangeable parts. One would like all C compilers to accept the same program. (A desirable goal, but often not achievable, as we will soon see.)

Language standards defined by national and international standards bodies: ISO, IEEE, ANSI.

Role of government in standards

- Standards in US are voluntary. -- No government enforcement of standards.
- National Institute for Standards and Technology (NIST) only provides standards through its Federal Information Processing Standards (FIPS) program that are applicable to federal agencies. NIST FIPS generally endorse commercial standards.

Standardization - in theory

Process for creating standards are relatively similar among all of these groups:

1. A vested interest in some technology meet to discuss a needed standard.
2. A working group is set up to define standard in some particular technology.
3. The working group advertises for a balloting group from among members of parent organization.
4. Members of the working group agree on features for this new standard.
5. Members of the balloting group vote on proposed standard.
6. If passed, standard sent to parent organization for approval.
7. If parent group passes standard, it is published.
8. Sometimes published as a "trial use" standard.
9. Standard effective for 5 years. All are supposed to be reviewed and revised or withdrawn at that time.

Standardization - in practice

But process is flawed:

- Contentious features often omitted to gain consensus.
- Vendors do not want major changes from current products to avoid redesign and angering current customers
- When to standardize? If too early, may not be sure of correct features; if too late, too many diverse products.
- Only vendors have a vested interest in the results, so join working group. A big "yawn" for users until standard is published, and then too late.
- Balloting group self selected by working group.
- No votes "counted" by working group and "resolved."
- Users don't care until standard approved, and then it is too late!
- Standards organizations (e.g., IEEE) make a huge profit in publishing standards documents. There is a conflict of interest in doing what is right versus raising funds. (e.g., Software engineering standards in a book that is well over $100 per copy)
- Process is slow for 1990s. Can easily take 3-5 years (or more) to create and publish a standard.

Evolution of standards

In general, standards describe effects of conforming behavior (except for Ada). Behavior of non-conforming program not specified, so any extensions to a standards conforming compiler may still be standards conforming.

Standards supposed to be reviewed every 5 years:

- FORTRAN 1966, 1977, 1990
- Ada 1983, 1995
- Not quite 5 years, but at least periodically

When to standardize?

Problem: When to standardize a language?

- If too late -- many incompatible versions -- FORTRAN in 1960s was already a de facto standard, but no two were the same.
- If too early -- no experience with language -- Ada in 1983 had no running compilers
- Just right -- Probably Pascal in 1983, although now a "dead'' language

Other languages:

- C in 1988
- LISP in 1990 -- Way too late
- De facto standards: ML - One major implementation - SML
- Smalltalk - none
- Prolog - none
- SQL - Database access (Date?)

Architectural standards

- There are very few software architectural standards - POSIX ("UNIX" kernel), OSI (data communication)
- Most standards are handled as corporate and consortium standards where the "name" is owned by some entity:
  - UNIX by (who owns UNIX today? [Lawsuit by SCO against IBM for infringing copyright by using Linux code])
  - X-Windows by X-Windows consortium
  - Motif by OSF (Are they still around?)
  - Many Internet standards (by Internet Working Group?) -- TCP/IP, FTP, SMTP, TELNET

Architectural standards

- More ...
  - Windows API (application program interface) by Microsoft. This was the basis for the Justice Department antitrust suit against Microsoft.
  - PCTE -- Portable Common Tool Environment by ECMA (European Computer Manufacturers Association (Note: They did original "toaster" model)
  - CORBA -- Common Object Request Broker Architecture by the Object Management Group (OMG)
  - Java -- by Sun
  - Active-X by Microsoft (as an alternative to Java)

Standardization summary

- Industry standards allow for more rapid evolution of the standard, but loss of control of who can change the standard.
- Problem -- How to get various international groups to agree on the same standard?
- Major unsolved problem for today -- How to evolve the formal standardization process to be more receptive to rapid changes without allowing specific corporations to control the process?
- Issues to consider:
  - Are standards being used effectively today?
  - Do we even need them?
  - Who should sponsor them? -- which organizations
  - Who should write them? -- users or vendors
  - Which comes first -- the underlying technology or the standard?

Key points

- The software architecture is the fundamental framework for structuring the system.
- Architectural design decisions include decisions on the application architecture, the distribution and the architectural styles to be used.
- Different architectural models such as a structural model, a control model and a decomposition model may be developed.
- System organisational models include repository models, client-server models and abstract machine models.

Key points

- Modular decomposition models include object models and pipelining models.
- Control models include centralized control and event-driven models.
- Reference architectures may be used to communicate domain-specific architectures and to assess and compare architectural designs.
- Standards are useful, but non-technical issues often dominate their development.