

- Design
 - Chapters – mainly 8 and 11 (for now)
 - sds.doc
 - look at grading criteria as well

Why Design

- good design -> good code
- easier to code, test, maintain, change
- easier to understand impact of requirements changes
- large projects – divide across teams, but have unifying design

Software design

- Software design is the process of building a program while satisfying a problem's functional requirements and without violating its non-functional constraints
- Software design is normally broken into two phases:
 - high level or *architectural* design and
 - low level or *detail* design
- (SDS also asks you for UI design)

Overview of Design Phase

- Input
 - specification – description of *what* the product is to do
- Output
 - design document – *how* the product is to achieve this
- Three main activities
 - architectural design
 - decompose the product into modules & connections b/w them
 - detailed design
 - data structures chosen & algorithms selected/designed
 - design testing
 - make sure design is correct – performed throughout phase
- Two key aspects of product to base design
 - actions & data -> action-oriented, *data-oriented*, or hybrid (OO)

- Design is the most creative phase
 - no set rules that you must follow
 - a lot of its success boils down to experience
 - instead there are principles and patterns which improve quality of design

Architectural design

- Architectural design is the process of identifying and assigning the responsibility for aspects of functional behavior to various modules or components of a software system
 - Management of non-functional constraints must be determined
 - The communication interfaces among the components must also be specified
 - sometimes can be derived naturally from the problem
 - some guidelines for select application domains in Chapter 13

Detail design

- Detail design is the process of specifying the logical behavior of each of the components
 - Algorithm selection
 - Data structure representation
 - combination of natural language, pseudo code, graphical representation

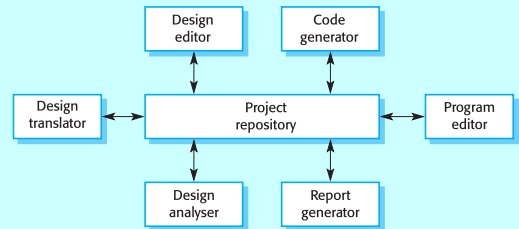
Architectural models

- 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.

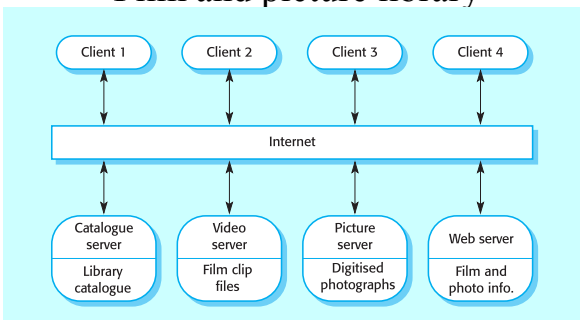
System organization (system architecture)

- Reflects the basic strategy that is used to structure a system.
- Three organizational styles are widely used:
 - A shared data repository style;
 - A shared services and servers style;
 - An abstract machine or layered style.

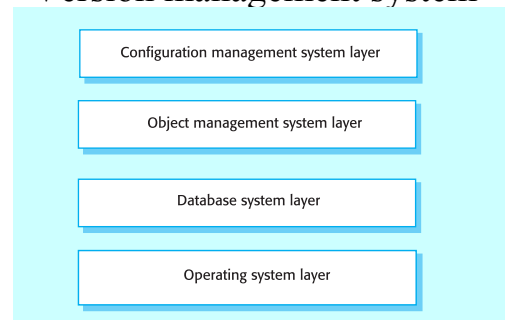
CASE toolset architecture



Film and picture library



Version management system



Architecture and non-functional properties

- 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
 - Localize 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.

Design Approach

- There are many approaches to design.
 - Some espouse a particular point of view as to how best to structure a system (for example, object-oriented design)
 - Some of them are intended for a particular class of application (e.g. real-time systems, transactional systems)
 - And some are intended for a specific part of the system structure (such as user-interface design).
- All approaches to design, however, contain three aspects that may be compared: method, representation and validation

Design Method

- A *method* is a systematic sequence of steps that a design team uses to solve a problem
- A method usually encourages a particular viewpoint on its users
 - For example, the object-oriented design method encourages designers to view problems in terms of their constituent objects before worrying about functional behavior
 - data flow analysis better for apps with pipelined functional components, e.g., compilers
- A method acts as a discipline on the designers, forcing them to order their thoughts and activities in certain ways
- All design methods must deal with the management of complexity, typically by means of abstraction and refinement

Design Representation

- Non-trivial problems require design solutions to be expressed using some form of representation
- The design representation may be either graphical or textual, but it is sufficiently formal that it can be checked for certain properties
- A design representation serves two audiences
 - The designers themselves. That is, the discipline imposed by the method and the representation encourages early detection of missing or inconsistent aspects of a proposed solution
 - Design representations are also read by other stakeholders. These might be coders or testers. Notably, they might subsequently be maintainers trying to understand the designers' original intent

Design Validation

- Designs must be checked to assure that they accomplish their intended goals.
- A good design method will have an associated validation technique
 - measure the extent of *coupling* and *cohesion* to check the quality of its designs
- Design inspections, walk-throughs and reviews are general techniques for assessing design quality. They can successfully complement method-specific validation techniques

Design Concepts

- Conceptual integrity / coherence
- Coupling / cohesion
- Information hiding
- Abstraction / refinement
- Rationale / tradeoffs

- *Coupling* - the extent to which two components depend on each other for successful execution
 - Low coupling is good
- *Cohesion* - the extent to which a component has a single purpose or function
 - High cohesion is good

- what does this mean:
 - modules single-minded/self-contained functions
 - address subset of requirements related to that function
 - simple interface, limited interaction -> bugs are often at interfaces
 - in terms of data, control, access to common content/data
 - easy to efficiently divide among team members

- Information Hiding

- Use of encapsulation to hide implementation details
- Reduces intercomponent coupling thereby supporting subsequent maintenance

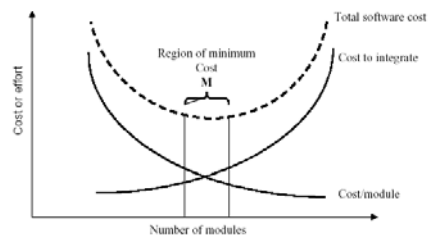
- Abstraction and Refinement

- All design methods support the idea of abstraction and refinement
- That is, designs are expressed at various levels of detail with correspondences between the levels
- Various conceptual devices (abstraction mechanisms) are used to refine a design at one level to a lower level
- Abstractions include procedural, data and control abstraction

- Rationale/Tradeoffs

- Design decisions are explicit choices of how to trade off two non-functional aspects of a design, such as speed versus size
- Design decisions should be explicitly documented
- Documentation of design decisions is called *design rationale*

Modularity and Software Cost



- consider low coupling/high cohesion
 - module should be ‘stand alone’, errors contained as much as possible
- consider requirements
 - change in requirements should minimize number of modules affected

Design dimensions

- make architecture decision
 - repository, service, layered
- make decomposition decision
 - structural components, or functional components
- determine control model
 - centralized, event-driven
- describe architecture & modules/subsystems
 - context model
 - behavioral/process model – data-flow, state machine
 - data model – semantic relationship, ER diagrams
 - object model

some guidelines

- are decomposition boundaries clear? rationale?
- is the correspondence between lower- and higher-level design representation clear
 - use numbering 1, 1.1, 1.2...
 - use colors
- can still show external components (shaded, gray...)
- data paths/interaction paths easy to follow?
 - may duplicate in design representation external component, just make it obvious that you've done so
- both dynamic and static aspects of your system shall be explained
 - with appropriate model
 - try to be consistent in the way you describe each (sub-) component
- reuse of components
 - describe in sufficient detail interface and/or changes to be made, not more

User-interface Design

- more on this later
- for now:
 - know the user, know the way the user interacts with system, the context in which system is used
 - describe with use-case, sequence diagrams
 - show prototype of major screens/interface elements

more UI

- be consistent in style, meaning of symbols...
- enable shortcuts for frequent uses
- dialogues
- provide meaningful/informative feedback
- simple error handling, reversal of actions