Overview - Design

• Introduction to Design
• Architectural Design
• Modules
• Structured Design
• Objects
• Object-Oriented Design
• Detailed Design
• Integration Testing
Goals and Objectives

• Develop a coherent representation of a software system that will satisfy the requirements
• Identify inadequacies in the requirements
• Can develop review plan that demonstrates coverage of the requirements
  – yields confidence in design
• Can develop test plan that covers design
  – yields confidence in both design and implementation
Introduction to Design

**How?**

- Requirements Analysis
  - Specification
  - Validation
  - Design
    - Verification
    - Implementation and Integration
      - Testing
      - Operation and Maintenance
        - Revalidation

Validation

Change

CSC 402 Software Engineering I
Relationship to other lifecycle phases

• Requirements
  – Specifies the “app domain” not the “machine”
  – Provides conceptual boundaries
    € keeps design focused

• Implementation
  – When design specifications are sufficient for coding assignments (now it can be “executable” on a machine)
    € each assignment, theoretically, can be given to a programmer unaware of the overall system architecture
    • and the code will satisfy the associated requirements
Basic Design Process

• The design process develops several models of the software system at different levels of abstraction
  – Trial and Error in many ways
  – Starting point is an informal “boxes and arrows” design
  – Add information to make it more consistent and complete
  – Provide feedback to earlier designs for improvement
Top-Down Design

• Recursively partition a problem into sub-problems until tractable (solvable) problems are identified

Diagram:
- System level
  - Subsystem level
    - Module level
Design Activities

- Architectural design
  - Subsystem identification
    € services and constraints are specified
  - Module design
    € modular decomposition is performed; relationships specified

- Detailed design
  - Interface design
    € module interfaces are negotiated, designed and documented
  - Data structure and algorithm design
    € module details (data structures and algorithms) to provide system services are specified
Design Products

• Refined requirements specification
• Description of systems to be constructed
  – software architecture (diagrams and rationale)
  – modular decomposition (hierarchy)
  – abstract module interface specifications
  – detailed module designs
• Documentation of decisions and rationale
• Data dictionary of all defined objects
• Validation review plan
• Integration test plan
Desirable Characteristics/ Common Problems

• Uniform
• Complete
• Rigorous
• Confirmable, verifiable, testable
• Supportable by tools
• Desensitized to change
• Accommodates independent coding

• Depth-first design: only partial satisfaction of requirements
• Failure to consider potential changes
• Too detailed: overly constrains implementation
• Ambiguous: misinterpreted during implementation
• Undocumented: designers become essential
• Inconsistent: system cannot be integrated
Architectural Design

• Architectural Design
  – decomposition of large systems that provide some related set of services + establishing a framework for control and communication

• Architectural styles establish guidelines
  – a relatively new area of research

• No generally accepted architectural design process (Well, maybe UML, Rational Unified Process…) 
  – some important sub-processes
    € System structuring: structuring of the system into a number of subsystems, where a subsystem is an independent software unit
    € Control modeling: establishing a general model of control relationships between the parts of the system
    € Modular decomposition: decomposing each identified subsystem into modules
Architecture, Subsystems and Modules

• Architecture consists of interacting subsystems
  € describes the subsystem decomposition in terms of subsystem responsibilities, dependencies among subsystems, subsystem mapping to hardware, and major policy decisions such as control flow, access control and data storage. (Bruegge)

• Subsystems
  – a component whose operation does not depend on the services provided by other subsystems
  – communicates with other subsystems via defined interfaces
  – is further decomposed further into modules during design
Basic concerns

• Define design goals
  – identify and prioritize qualities of the system to be optimized
    € find these in nonfunctional requirements

• Decompose system into smaller subsystems
  – to support the design goals
Software Architecture

- **Components (subsystems)**
  - The elements out of which the system is built
  - Examples: filters, databases, objects, ADTs

- **Connectors**
  - The interaction or communication mechanisms
  - The glue that combines the components
  - Examples: procedure calls, pipes, event broadcast, messages, secure protocols

- **Constraints**
  - Limitations on the composition of components and connectors
Architectural Style

• Example architectural styles
  – Batch sequential
  – Pipe and filter
  – Main program and subroutines
  – Blackboard
  – Interpreter
  – Client-server
  – Communicating processes
  – Event systems
  – Object-oriented
  – Layered Systems

Families of systems defined by patterns of composition
Architectural Design: System Structuring

• Model of the system structure and decomposition
  € how subsystems share data
  € how they are distributed
  € how they interface with each other

• Three standard models
  – *Repository model*: how subsystems exchange and share information
    • E.g., all shared data is held in a central database or each sub-system maintains its own database
  – *Distribution model*: how data and processing is distributed across a range of processors
    • E.g., Client-server or peer-to-peer processes
  – *Abstract machine model*: the interfacing of subsystems as abstract machines each of which provides a set of services to others
    • E.g., each subsystem defines an abstract machine
Architectural Design: Control Modeling

- Control of subsystems so that services are delivered to the right place at the right time

- Two general approaches
  - *Centralized control*
    - One subsystem has overall responsibility for control and starts/stops other subsystems
      - call-return model (sequential)
      - manager model (concurrent)
  - *Event-based control*
    - each subsystem responds to externally generated events (from other subsystems or the environment)
      - broadcast model
      - interrupt-driven model
Simple Overview (very basic Architecture)

• Design major subsystems
  – use UML, class diagrams, event trace or sequence diagrams
    € don’t forget the error handling subsystem
  – keep #subsystems small (5 to 9 rule of thumb)
  – keep communication between subsystems to a minimum
  – explain what each subsystem does

• Traceability matrices
  – show allocation of requirements to subsystems
  – show subsystems and the requirements they handle
    € assign a requirement to to as few subsystems as possible
      • why? (coupling and cohesion, covered later in modules…)
Simple Overview

• Subsystem Interfaces
  – traceability matrices can help
    € subsystems that appear together may need to communicate
  – one required function - one interface
    € minimize #ways to invoke or call a subsystem
  – know who will be calling the subsystem and why
    € record this rationale
  – keep complex data structures out of the interfaces

• Environment Interface
  – don’t worry about O.S., that really is background
  – UI is important
  – other environmental interfaces?
Architecture should address:

- external interfaces, user interface
- db if needed, other data storage issues
- key algorithms
- memory management, key string storage (such as error messages)
- concurrency
- security
- localization
- networking
- portability
- programming language
- error handling
Architecture Documentation

• System Overview
• Architectural Goals and Constraints
• Subsystems and Organization
  – diagrams and rationale
• Can find templates on line if you wish
Design: Modular decomposition

- After decomposition of the system into subsystems, subsystems must be decomposed into modules
  - *No rigid distinction between system and modular decomposition*

- Two important approaches for decomposing subsystems into modules:
  - *Data-flow (structured design)*
    - system is decomposed into functional modules which accept input data and transform it to output data
    - process-based decomposition
    - achieves mostly procedural abstractions
  - *Object-oriented (object-oriented analysis and design)*
    - system is decomposed into a set of communicating objects
    - object-based decomposition
    - achieves both procedural + data abstractions
Design: Hierarchy

- Hierarchies support modular decomposition
  - *Uses relation*: \( a \) uses \( b \) only if the correct functioning of \( a \) depends on the existence of a correct implementation of \( b \)
    
    \( \varepsilon \) modular decomposition can be specified by *uses*, where
    - Level 0 is the set of all programs that use no other program
    - Level \( i \) (\( i > 0 \)) is the set of all programs that use at least one program on level \( i - 1 \) and no program at level \( \geq i \).
    - Note: the *uses* relation does not always provide a hierarchy
  - *Is-composed-of relation*: \( a \) is-composed-of \( b \) if \( b \) is a component of \( a \) and encapsulated within \( a \)
    
    \( \varepsilon \) modular decomposition can be specified by *is-composed-of*, where
    - non-terminals are virtual code
    - terminals are the only units represented by code
    - Then, the *uses* relation is specified over the set of terminals only
    - Note: the *is-composed-of* relation is acyclic
Modules

• Definition: a software entity encapsulating the representation of an abstraction and providing an abstract interface to it

• Module interaction
  – Module hides implementation details so that the rest of the system is insulated and protected from the details AND vice versa
  – Modules communicate only through well-defined interfaces

• Negotiating module interfaces
  – design interface to component to be insensitive to change
  – determine likely usage patterns and purposes
  – disseminate minimal information as useful generalities

  € abstract Interfaces: one specification, many possible implementations
  – suppress unnecessary detail of a design decision
Modular Decomposition: Abstraction

- Abstraction is a tool that supports focus on important, inherent properties and suppression of unnecessary detail
  - permits separation of conceptual aspects of a system from the implementation details
  - allows postponement of design decisions
- Three basic abstraction mechanism
  - procedural abstraction
    € specification describes input/output
    € implementation describes algorithm
  - data abstraction
    € specification describes attributes, values, properties, operations
    € implementation describes representation and implementation
  - control abstraction
    € specification describes desired effect
    € implementation describes mechanism
Modular Decomposition: Information Hiding

- Information hiding is a decomposition principle that requires that each module hides its internal details and is specified by as little information as possible
  - forces design units to communicate only through well-defined interfaces
  - enables clients to be protected if internal details change

- Sample entities to encapsulate
  - abstract data types
  - algorithms
  - input and output formats
  - processing sequence
  - machine dependencies
  - policies (e.g. security issues, garbage collection, etc.)
Modular Decomposition: Cohesion and Coupling

- Cohesion
  - the degree to which the internals of a module are related
- Coupling
  - the degree to which the modules of a design are related
- The ideal system has highly cohesive modules that are loosely coupled
  - high cohesion -> well-designed reusable module
  - low coupling -> coherent design, resistant to change
Types of Cohesion

- coincidental
  - multiple, completely unrelated actions
- logical
  - series of related actions, often selected by parameters
- temporal
  - series of actions related in time
- procedural
  - series of actions sharing sequence of steps
- communicational
  - procedural cohesion but on the same data
- informational
  - series of independent actions on the same data
- functional: exactly one action

(Bad)

(Good)
Types of Coupling

- **content**
  - one module directly references content of another

- **common**
  - both modules have access to same global data

- **control**
  - one module passes an element of control to another

- **stamp**
  - one module passes a data structure to another; which only uses part of the passed information

- **data**
  - one module passes only homogeneous data items

(Good) (Bad)
Some examples of cohesion

• Logical cohesion
  – Input/Output libraries
  – Math libraries

• Temporal cohesion
  – Program initialization

• Communicational cohesion
  – “calculate data and write it to disk”
  – Closely related: sequential cohesion
    € the output of one element is the input to another
Some examples of coupling

- **Control coupling**
  - One module passes control flags (parameters or global variables) that control the sequence of processing steps in another module

- **Stamp coupling (alternative definition)**
  - Similar to common coupling (modules that share global data) except that globals are shared selectively among routines that require the data
  - Ada packages support stamp coupling since variables defined in a package specification are shared between all modules which use the package.
Structured Design

• System is completely specified by the functions that is to perform
  – Top-down, iterative refinement of functionality
    € break the [system] function into subfunctions
    € determine hierarchy and data interaction
  – Function refinement guides data refinement
  – Hierarchical organization is a tree with one module per subfunction

• Pros and Cons
  – modules are highly functional
  – best suited when state information is not pervasive
  – data decisions must be made earlier
  – changes in data ripple through entire structure
  – little chance for reusability
Structured Design Process

• Identify flow of data and incorporate detail and structure iteratively
  – given specification loop
    € identify data flow and transformations
      • nouns as data, verbs as transformations
    € derive data flow diagrams
    € identify "natural aggregates"
      • identify highest level input and output units
      • remaining units are central transforms
      • form level of structure chart
      • control module (coordinate)
        – input module (afferent)
        – central module(s) (transform)
        – output module (efferent)
      • form structure chart
  – until implementation is immediate
Data Flow Diagrams

• Software system as flow of data from logical processing unit A (transformation) to B
  – do not include control information
  – data flow diagram elements
    € round-cornered rectangle = transformation
    € vector = data flow
    € vector operation = data flow link
      • * (and)
      • + (or)
      • + (exclusive or)
    € arc with data flow link = bracketing to override precedence
      • and over or over exclusive or
    € rectangle = data store
    € circle = user interaction (input/output)
Data Flow Templates

- Simple flow:
  - A to C

- Conjunct flow:
  - A
  - B (marked with an asterisk)
  - C

- Disjunct flow:
  - A
  - B (marked with a plus sign)
  - C

- Exclusive flow:
  - A
  - B (marked with a plus sign)
  - C
Structure Charts

• Depict software structure as a hierarchy of modules and data communication
  – may have control info defining selection and loops
  – structure chart elements
    € rectangle = module
    € four types of module based on data flow
      • control (coordinate)
      • input (afferent)
      • central (transform)
      • output (efferent)
Structure Charts (cont’d)

€ vector = control relationship
€ arrow with circular tail = directed data relationship
  • data couple (open), control couple (closed)
€ round-cornered rectangle = data store
€ circle = user interaction (input/output)
Structure Chart Templates

Afferent

\[ X \]

\[ \downarrow \]

\[ X \]

\[ \uparrow \]

\[ \circ \]

\[ \downarrow \]

\[ \circ \]

\[ \circ \]

\[ A \]

\[ \downarrow \]

\[ \bullet \]

\[ B \]

\[ \downarrow \]

\[ \circ \]

\[ \downarrow \]

\[ \circ \]

\[ \downarrow \]

\[ \circ \]

\[ \downarrow \]

\[ \circ \]

\[ \downarrow \]

\[ \circ \]

\[ \downarrow \]

\[ \circ \]

\[ \downarrow \]

\[ \circ \]

\[ \downarrow \]

\[ \circ \]

\[ \downarrow \]

\[ \circ \]

\[ \downarrow \]

\[ \circ \]

\[ \downarrow \]

\[ \circ \]

\[ \downarrow \]

\[ \circ \]

\[ \downarrow \]

\[ \circ \]

\[ \downarrow \]

\[ \circ \]

\[ \downarrow \]

\[ \circ \]

\[ \downarrow \]

\[ \circ \]

\[ \downarrow \]

\[ \circ \]

\[ \downarrow \]

\[ \circ \]