System Design: System Decomposition

CSE 308: Software Engineering
System Design

System design transforms an analysis model into a system design model.

Developers:
- define design goals of the system
- decompose the system into smaller subsystems
- select build strategies
System Design Activities

- System design is NOT algorithmic!
- Developers must make trade-offs
- Developers cannot anticipate all design issues

System design activities
- Identify design goals
- Design initial subsystem decomposition
- Refine subsystem decomposition to address design goals
System Design

System design, object design, and implementation comprise the construction of a system.

System design focuses on the processes, data structures, and SW/HW components necessary to implement the system.

This requires us to address conflicting criteria/constraints.
<table>
<thead>
<tr>
<th></th>
<th>Architecture</th>
<th>Software</th>
</tr>
</thead>
<tbody>
<tr>
<td>Components</td>
<td>Rooms</td>
<td>Subsystems</td>
</tr>
<tr>
<td>Interfaces</td>
<td>Doors</td>
<td>Services</td>
</tr>
<tr>
<td>Nonfunctional requirements</td>
<td>Living area</td>
<td>Response time</td>
</tr>
<tr>
<td>Functional requirements</td>
<td>Residential house</td>
<td>Use cases</td>
</tr>
<tr>
<td>Costly rework</td>
<td>Moving walls</td>
<td>Change interfaces</td>
</tr>
</tbody>
</table>
Analysis Products

- Analysis results in a requirements model that contains:
  - Nonfunctional requirements and constraints
  - Use case model
  - Object model
  - Sequence diagram for each use case
System Design Products

Analysis says nothing about how a system is realized

System design results in:

- Design goals: describe system qualities
- Software architecture
- Boundary use cases
Design Goals

Derived from nonfunctional requirements

Guide developer decisions in the event that trade-offs need to be made
Subsystem Decomposition

This is the bulk of system design

Divides the system into manageable pieces to deal with complexity

- each subsystem can be implemented independently
- this can be refined to meet specific design goals
Boundary Use Cases

Describe:

- system configuration
- system startup
- system shutdown
- exception-handling
System Design Concepts

- Subsystems provide services to other subsystems
  - service: set of related operations
  - define subsystems in terms of the services they provide

- Coupling: dependencies between two subsystems

- Cohesion: dependencies among classes within a subsystem

- Layering: organizes a system as a hierarchy of subsystems

- Partitioning: organizes subsystems as peers
Decompose systems into subsystems to manage complexity.

- A single developer/team should be able to handle a subsystem.
- Teams can work on individual subsystems with relatively little intercommunication.

- Systems are composed of classes and subsystems.
Services and Interfaces

- Service: set of related operations sharing a common purpose
- Set of operations defines a subsystem interface
- Object design focuses on the API, which extends the subsystem interface
- Defining subsystems in terms of their services allows us to focus on interface rather than implementation
- This helps to minimize the impact of change
Coupling and Cohesion

- **Coupling:** number of dependencies between subsystems
  - goal: strive for as loose coupling as possible

- **Cohesion:** number of dependencies within a subsystem
  - high cohesion: many objects are related to one another
  - goal: strive for as high coupling as possible

Generally, there is a tradeoff between coupling and cohesion.
Layering

Layer: a grouping of subsystems providing related services

Layers are ordered: a layer can only depend on lower layers

Closed architecture: layers can only access immediate neighbors

ex. OSI networking model

Closed, layered architectures lead to low coupling, but introduce speed/storage overhead
Partitioning

- Divide a system into peer subsystems
- Each subsystem depends on others, but can function independently
- In general, system decomposition makes use of both layering and partitioning
- Excessive layering/partitioning increases complexity
A software architecture includes:

- system decomposition
- global control flow
- handling of boundary conditions
- intersubsystem communication protocols
Repository

- Subsystems access a single data structure (central repository)
- Subsystems only interact through the repository
- Typically used for database management systems
- Useful for systems with constantly-changing data
- Disadvantage: repository can become a bottleneck
Model/View/Controller

Subsystems are classified as one of three types:

- **Model**: maintains domain knowledge
- **View**: display domain knowledge to user
- **Control**: manage sequence of user interactions

Changes in model state are propagated to view subsystem

Special case of the repository
Client/Server

- One subsystem (the server) provides services to instances of other subsystems (the clients), which interact with the user
  - ex. the Web (many servers, many clients)
- May act as a special case of the repository style
- Well-suited for distributed systems w/ lots of data
Peer-to-Peer

Generalization of client/server style
- subsystems can act as both clients and servers

More complex to design, due to deadlocks and callbacks
- callback: temporary, customized operation used to notify another subsystem
Three-Tier

Organizes subsystems into three layers:

- interface layer: includes all boundary objects
- application logic layer: includes all control/entity objects
- storage layer: storage/retrieval/querying of data
  similar to the repository subsystem
Four-Tier

- Decomposes interface layer into:
  - presentation client layer
  - presentation server layer
- Enables wide range of presentation clients
- Presentation server layer contains common forms
Pipe And Filter

Subsystems ("filters") process received data and send results to other subsystems

"pipes" - associations between subsystems

Each filter only knows the content/form of its input

Example: the Unix shell

Most useful for transforming streams of data without user interaction
System Design Summary

System design transforms an analysis model into a system design model

- includes definition of design goals, system decomposition, and selection of design strategies
Last Time

We discussed:

- system design concepts (subsystems and services)
- coupling and cohesion
- layers and partitions
- architectural styles
Start with a use case and the related analysis model (entity, boundary, and control objects, etc.)

We also need the list of nonfunctional requirements

this will guide us for system optimization
Identifying Design Goals

This step identifies what we should focus on. We can infer design goals from nonfunctional requirements, or from the application domain. We may need to elicit them from the client. Design goals MUST be stated explicitly. Typical design goals: reliability, security, modifiability.
Possible Design Criteria

- Performance
- Dependability
- End user criteria
- Cost
- Maintenance
Performance Criteria

Speed and space requirements for the system

Include:

- response time
- throughput
- memory usage
Dependability Criteria

- Robustness
- Reliability
- Availability
- Fault tolerance
- Security
- Safety
End User Criteria

Qualities that are important from a user’s POV

Includes:

- Utility
- Usability
- Learnability
Cost Criteria

Involves design and managerial considerations

Includes:

- Development cost
- Deployment cost
- Upgrade cost
- Maintenance cost
- Administration cost
Maintenance Criteria

- Extensibility
- Modifiability
- Adaptability
- Portability
- Readability
- Traceability of requirements
Design Goal Tradeoffs

You can’t get everything you want

Typical tradeoffs include:

- space vs. speed
- delivery time vs. functionality
- delivery time vs. quality
- delivery time vs. staffing
Identifying Subsystems

Initial subsystem decomposition should be derived from the functional requirements

- keep functionally-related objects together

Idea: objects identified in the same use case should be assigned to the same subsystem
Object Grouping Heuristics

- Assign objects identified in one use case into the same subsystem
- Create a dedicated subsystem for objects used for moving data among subsystems
- Minimize the number of associations crossing subsystem boundaries
- All objects in the same subsystem should be functionally related
The Facade Design Pattern

- Allows us to reduce class dependencies by creating a simple, unified interface
- Facade provides access only to a subsystem’s public services
- Other details are hidden
System Design

Incorporates the following activities:

- identification of design goals
- system decomposition
- refinement of system decomposition to address design goals
Addressing Design Goals

We will consider the following activities:

- selection of COTS/legacy components
- mapping subsystems to hardware
- designing a persistent data management infrastructure
- specification of an access-control policy
- designing the global control flow
- handling boundary conditions
Plan of Attack

- Overview of system design activities
- Discussion of activities to address design goals
- Managing system design
System Design Overview

Design goals
- guide developer decisions
- useful when trade-offs must be made

System decomposition
- divides a system into manageable pieces
- useful for handling complexity
Issues to Consider

- Hardware/software mapping
  - may require additional subsystems to be defined

- Data management
  - persistent data can be a system bottleneck

- Access control

- Control flow

- Boundary conditions
UML Deployment Diagrams

- Depict relationship among run-time components and hardware nodes
- Component: self-contained entity that provides service(s) to other components or actors
- Dashed arrows represent dependencies
Addressing Design Goals

- Mapping subsystems to processors and components
- Identifying and storing persistent data
- Providing access control
- Designing the global control flow
- Identifying boundary conditions
- Reviewing the system design model
Mapping Subsystems

- Many systems run on more than one computer
- Need to examine allocation of subsystems to computers and communication infrastructure
- Includes selection of a virtual machine (OS, etc.)
- Objects and subsystems are assigned to hardware nodes
  - we can use the Proxy design pattern for this
Dealing With Persistent Data

- Persistent data last beyond a single execution of the system
- Identifying persistent objects
  - ex. entity objects, classes that must survive shutdown
- Selecting a storage management strategy
  - ex. flat files, relational DBs, object-oriented DBs
Choosing a Database

If you have:

- voluminous data (ex. images)
- temporary data
- low information density

Choose flat files
Choosing a Database

If you have/need:

- concurrent accesses
- access at finer levels of detail
- multiple platforms for the same data

Choose a relational or object-oriented DB
Choosing a Database

If you have/need:

- complex queries over attributes
- large data sets

Choose a relational database
Choosing a Database

If you have/need:

- extensive use of associations to retrieve data
- medium-sized data set
- irregular associations among objects

Choose an object-oriented database
Providing Access Control

- Different actors may have access to different data/functions
- For each actor, we need to define access/operations
- Access matrix represents actors and classes
- Approaches include:
  - global access table
  - access control list
  - capability
Access Control

- An access matrix represents static access control
- Access rights are modeled as object attributes
- We can also implement dynamic access control via the Proxy design pattern
- Static and dynamic access control both rely on authentication
- Encryption can also be used for access control
Global Control Flow

Control flow: sequencing of actions in a system

Three possible mechanisms:

- procedure-driven control
- event-driven control
- threads
Identifying Boundary Conditions

- Boundary conditions include startup, initialization, and shutdown
- For each subsystem and persistent object, we examine:
  - configuration
  - startup and shutdown
  - exception handling
Exception Handling

Exception: event or error that occurs during system execution

Exception sources:
- hardware failure
- changes in the operating environment
- software fault

Exception handling: mechanism to deal with exceptions
Reviewing System Design

- System design is evolutionary and iterative
- Unfortunately, there is no external reviewer
- One option is to recruit other developers for review
- Besides meeting design goals, the system design model must be correct, complete, consistent, realistic, and readable
Managing System Design

- Documenting system design
- Assigning roles
- Addressing communication issues
- Managing iteration
System design document (SDD) describes design goals, subsystem decomposition, HW/SW mapping, data management, access control, control flow, and boundaries.

SDD is used to define interfaces and serve as an architectural reference.

Sections include: introduction, current SW architecture, proposed SW architecture, and subsystem services.

Written after the initial system decomposition is done.
Assigning Responsibilities

- **Architect**
  - ensures consistency in design decisions, interface styles

- **Architecture liaisons**
  - representatives from subsystem teams

- **Document editor**

- **Configuration manager**

- **Reviewer**
Communication

Sources of problems include:

- size: number of items, pieces of functionality, etc.
- change: subsystems and interfaces are in flux
- level of abstraction
- reluctance to confront problems
- conflicting goals and criteria
System design has three types of iterations:

- Brainstorming sessions for initial system design
- Interface revisions (includes vendor choice)
  - may include vertical/horizontal prototypes
- Remedies to lately-discovered errors and oversights

Design window: critical issues are only left open for a specified time
Next Time

- Managing reuse
- Reuse Activities
- Reuse concepts
- Object Design