Why Architecture?

The architecture is not the operational software. Rather, it is a representation that enables a software engineer to:

(1) **analyze the effectiveness** of the **design** in meeting its stated requirements,

(2) **consider architectural alternatives** at a stage when making design changes is still relatively easy, and

(3) **reduce the risks** associated with the construction of the software.
Why is Architecture Important?

- Representations of software architecture are an enabler for communication between all parties (stakeholders) interested in the development of a computer-based system.
- The architecture highlights early design decisions that will have a profound impact on all software engineering work that follows and, as important, on the ultimate success of the system as an operational entity.
- Architecture constitutes a relatively small, intellectually graspable model of how the system is structured and how its components work together.
Data Design

- At the architectural level …
  - Design of one or more databases to support the application architecture
  - Design of methods for ‘mining’ the content of multiple databases
    - navigate through existing databases in an attempt to extract appropriate business-level information
    - Design of a data warehouse - a large, independent database that has access to the data that are stored in databases that serve the set of applications required by a business
Data Design

- At the component level …
  - refine data objects and develop a set of data abstractions
  - implement data object attributes as one or more data structures
  - review data structures to ensure that appropriate relationships have been established
  - simplify data structures as required
Data Design—Component Level

1. The **systematic analysis** principles applied to function and behavior should also be applied to data.
2. All **data structures** and the **operations** to be performed on each should be **identified**.
3. A **data dictionary** should be established and used to define both data and program design.
4. **Low level** data design **decisions** should be deferred until **late** in the design process.
5. The **representation** of data structure should be **known only** to those **modules** that must make **direct use** of the data contained within the structure.
6. A **library** of useful data structures and the operations that may be applied to them should be developed.
7. A software design and programming language should support the specification and realization of **abstract data types**.
Architectural Styles

Each style describes a system category that encompasses:

1. a set of components (e.g., a database, computational modules) that perform a function required by a system,
2. a set of connectors that enable “communication, coordination and cooperation” among components,
3. constraints that define how components can be integrated to form the system, and
4. semantic models that enable a designer to understand the overall properties of a system by analyzing the known properties of its constituent parts.

- Data-centered architectures
- Data flow architectures
- Call and return architectures
- Layered architectures
- Object-oriented architectures
Data-Centered Architecture
Call and Return Architecture
Layered Architecture
Object-Oriented Architecture

Figure 10.7  UML relationships for SafeHome security function archetypes (adapted from [BOS00])
**Architectural Patterns**

- **Concurrency**—applications must handle multiple tasks in a manner that simulates parallelism
  - operating system process management pattern
  - task scheduler pattern
- **Persistence**—Data persists if it survives past the execution of the process that created it. Two patterns are common:
  - a database management system pattern that applies the storage and retrieval capability of a DBMS to the application architecture
  - an application level persistence pattern that builds persistence features into the application architecture
- **Distribution**—the manner in which systems or components within systems communicate with one another in a distributed environment
  - A broker acts as a ‘middle-man’ between the client component and a server component.
Architectural Design

- The software must be placed into context
  - the design should define the external entities (other systems, devices, people) that the software interacts with and the nature of the interaction

- A set of architectural archetypes should be identified
  - An archetype is an abstraction (similar to a class) that represents one element of system behavior

- The designer specifies the structure of the system by defining and refining software components that implement each archetype
Architectural Context

target system: Security Function

- Safehome Product
- Internet-based system

- control panel
- homeowner

- sensors

- surveillance function peers

uses
Archetypes

Figure 10.7 UML relationships for SafeHome security function archetypes (adapted from [BOS00])
Component Structure

- SafeHome Executive
- External Communication Management
- GUI
- Internet Interface
- Security
- Surveillance
- Home management
- Function selection
- Control panel processing
- detector management
- alarm processing
Refined Component Structure

- SafeHome Executive
  - External Communication Management
    - GUI
    - Internet Interface
    - Control panel processing
    - Keypad processing
    - CP display functions
  - Security
    - detector management
    - alarm processing
    - scheduler
    - phone communication
    - alarm
- ...
Analyzing Architectural Design

1. Collect scenarios.
2. Elicit requirements, constraints, and environment description.
3. Describe the architectural styles/patterns that have been chosen to address the scenarios and requirements:
   - module view
   - process view
   - data flow view
4. Evaluate quality attributes by considered each attribute in isolation.
5. Identify the sensitivity of quality attributes to various architectural attributes for a specific architectural style.
6. Critique candidate architectures (developed in step 3) using the sensitivity analysis conducted in step 5.
An Architectural Design Method

customer requirements

"four bedrooms, three baths, lots of glass ..."

architectural design
Deriving Program Architecture
Partitioning the Architecture

- “horizontal” and “vertical” partitioning are required
Horizontal Partitioning

- define separate branches of the module hierarchy for each major function
- use control modules to coordinate communication between functions
Vertical Partitioning: Factoring

- design so that decision making and work are stratified
- decision making modules should reside at the top of the architecture
Why Partitioned Architecture?

- results in software that is easier to test
- leads to software that is easier to maintain
- results in propagation of fewer side effects
- results in software that is easier to extend
Structured Design

- **objective:** to derive a program architecture that is partitioned

- **approach:**
  - the Data Flow Diagram (DFD) is mapped into a program architecture
  - the Process Specification (PSPEC) and State Transition Diagram (STD) are used to indicate the content of each module

- **notation:** structure chart
Flow Characteristics

Transform flow

Transaction flow
General Mapping Approach

- **isolate** incoming and outgoing flow boundaries; for transaction flows, isolate the transaction center

- working from the **boundary outward**, map DFD transforms into corresponding **modules**

- **add control** modules as required

- **refine** the resultant program structure using effective modularity concepts
Transform Mapping

data flow model

"Transform" mapping
Factoring

direction of increasing decision making

typical "decision making" modules

typical "worker" modules
First Level Factoring

main program controller

input controller
processing controller
output controller
Second Level Mapping

mapping from the flow boundary outward
Transaction Flow

incoming flow

action path
Transaction Example

In reality, other commands would also be shown.
Level 2 Data Flow Diagram

1. Read command
2. Validate command
3. Determine type
4. Produce error msg
5. Error msg
6. Status
7. Invalid command
8. Valid command
9. Read record
10. Calculate output values
11. Send control value
12. Robot control
13. Start/stop
14. Assembly record
15. Report
16. Fixture setting
17. Format setting
18. Raw setting
19. Combined status
20. Record
21. Values
Transaction Mapping Principles

- **isolate the incoming flow path**
- **define each of the action paths** by looking for the "spokes of the wheel"
- **assess the flow** on each action path
- **define the dispatch and control** structure
- **map each action path flow** individually
Transaction Mapping
Isolate Flow Paths

- Read command
- Validate command
  - Command
  - Invalid command
  - Valid command
- Determine type
- Produce error msg
  - Error msg
  - Status
- Read fixture status
  - Combined status
  - Raw setting
- Determine setting
- Record
  - Record
  - Values
  - Output values
  - Format report
- Start/stop
  - Robot control
  - Send control value
  - Control value
  - Assembly record
  - Report

- Command
- Fixure setting
Map the Flow Model

- process operator commands
  - command input controller
    - read command
    - validate command
    - produce error message
  - determine type
    - fixture status controller
    - report generation controller
    - send control value

Each of the action paths must be expanded further.
Refining the Structure Chart

process operator commands

command input controller

read command
validate command
produce error message

fixture status controller
report generation controller
send control value

determine type

read fixture status
determine setting
format setting
read record
calculate output values
format report