Achieving System Qualities Through Software Architecture II

The meaning of "design"

Architectural views

Modules and the module structure

AS LEAD SOFTWARE ENGINEER, I GIVE YOU THE FIRST UNIT OF OUR TEN THOUSAND COPY PRODUCTION RUN.
WOW! I WISH WE'D DESIGNED IT WITH THE FEATURES LISTED ON THE BOX. THAT WOULD HAVE BEEN AWESOME.
WHAT?
I'LL PUT THIS WITH THE OTHER REMINDERS OF HOW MY LIFE COULD HAVE BEEN EXCELLENT.
Dilbert.com DilbertCartoonist@gmail.com
3-15-10 © 2010 Scott Adams, Inc./Dist. by UFS, Inc.

Qualities Established in Architecture

Behavioral (observable)	Developmental Qualities
• Performance • Security • Availability • Reliability • Usability	• Modifiability(ease of change) • Portability • Reusability • Ease of integration • Understandability • Provide independent work assignments
Properties resulting from the properties of components, connectors and interfaces that exist at run time.	Properties resulting from the properties components, connectors and interfaces that exist at design time *whether or not they have any distinct run-time manifestation*.

Importance

- Customer experience: behavioral quality attributes drive the customer experience
- Development challenges: developmental quality attributes drive developmental difficulty
- Success depends on managing quality as well as functional requirements

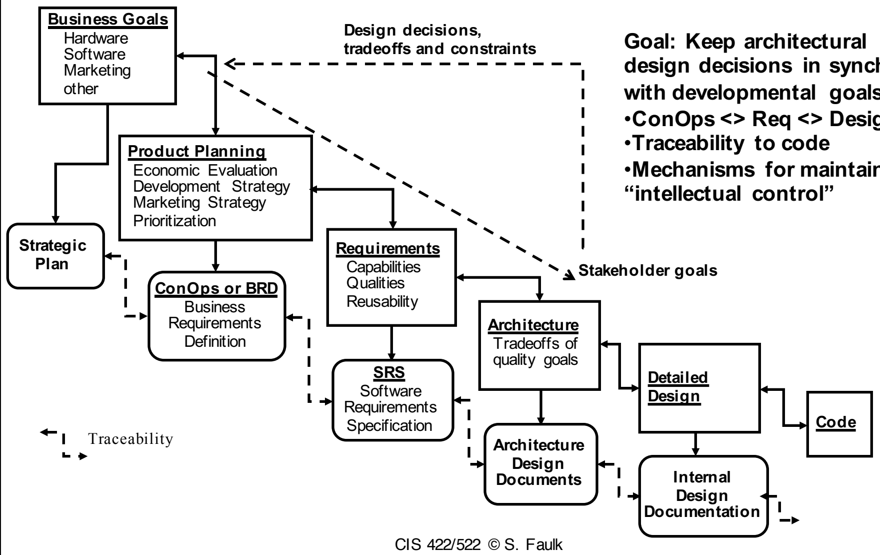
Functionality, Architecture, and Quality Attributes

- Functionality and quality attributes are orthogonal
- Quality attributes are typically *whole system properties*
  - Must be considered throughout design, implementation, and deployment
- Satisfactory results depends on:
  - Getting the big picture (architecture) right
  - Then getting the details (implementation) right

Example: Performance

- Ex: Performance depends on
  - How much inter-component communication is necessary (Arch)
  - What functionality has been allocated to each component (Arch)
  - How shared resources are allocated (Arch)
  - The choice of algorithms to implement functionality (Non-arch)
  - How algorithms are coded (Non-arch)

Product Development Cycle and Architecture



Goal: Keep architectural design decisions in synch with developmental goals
- **ConOps <> Req <> Design**
- **Traceability to code**
- **Mechanisms for maintaining "intellectual control"**

Software Engineering Architecture

- Goal is to keep developmental goals and architectural capabilities in synch
- Proceed from an understanding of desired qualities to an *acceptable* system design
  - Balance of stakeholder priorities and constraints
  - Requires making design tradeoffs
  - Documentation must communicate *how* this is accomplished

Implications for the Development Process

Implies need to address architectural concerns throughout the development process:

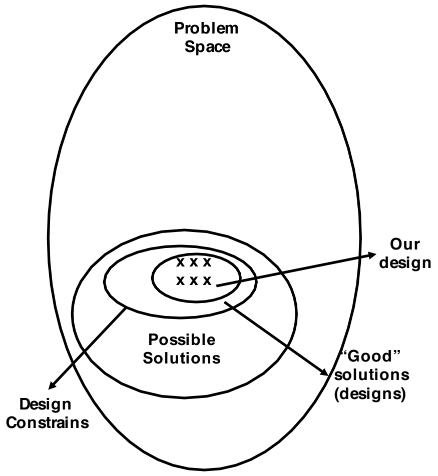
- Understanding the "business case" for the system
- Understanding the quality requirements
- Designing the architecture to meet quality goals**
- Representing and communicating the architecture
- Analyzing or evaluating the architecture
- Implementing the system based on the architecture
- Ensuring the implementation conforms to the architecture

What is “design?”

Meaning of “Design”

- What does it mean to say that we are going to “design the software?”
- What is the basis for making a design decision?
- How do we know when we are done?
- If we did a good job? What makes a good design?

The Design Space



- A Design: is (a representation of) a solution to a problem
  - Represents a set of choices
    - Typically very large set of possible choices
    - Must navigate through possibilities
    - Invariably requires tradeoffs
  - Possible choices are limited by *assumptions* and *constraints*
    - Must be ISO 2000 compliant, legacy compatible, etc.
    - May not use v.1 library routines
  - Some designs are better than others (notion of *good design*)

Design Means…

- Design Goals: the purpose of design is to solve some problem in a context of *assumptions* and *constraints*
  - Solution: acceptable balance of system qualities
  - Assumptions: what must be true of the design
  - Constraints: what should not be true
- Process: design proceeds through a sequence of decisions
  - A *good* decision brings us closer to the design goals
  - An idealized design process systematically makes good decisions
  - Any real design process is chaotic
- Good Design: *by definition* a good design is one that satisfies the design goals

Elements of Architectural Design

- Design goals
  - What are we trying to accomplish in the decomposition?
- Relevant Structure
  - How to we capture and communicate design decisions?
  - Which structures should we use?
- Decomposition principles
  - How do we distinguish good design decisions?
  - What decomposition (design) principles support the objectives?
- Evaluation criteria
  - How do I tell a good design from a bad one?

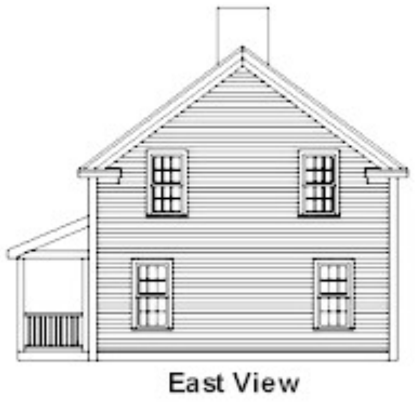
Architectural Views

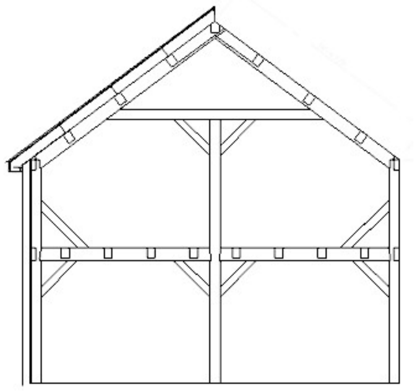
Which structures should we use?

Structure	Components	Interfaces	Relationships
Calls Structure	Programs (methods, services)	Program interface and parameter declarations	Invokes with parameters (A calls B)
Data Flow	Functional tasks	Data types or structures	Sends-data-to
Process	Sequential program (process, thread, task)	Scheduling and synchronization constraints	Runs-concurrently-with, excludes, precedes

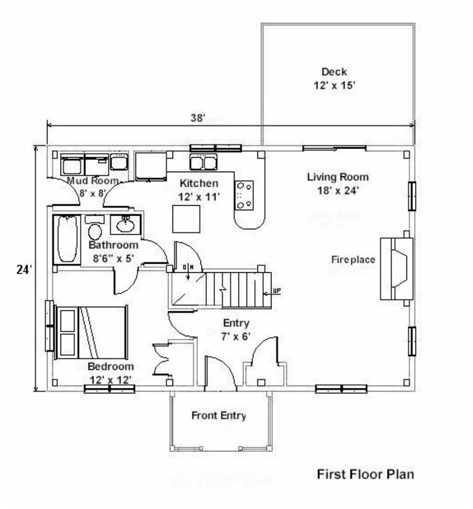
- Choice of structure depends the *specific design goals*
- Compare to architectural blueprints
  - Different blueprint for load-bearing structures, electrical, mechanical, plumbing

Elevation/Structural

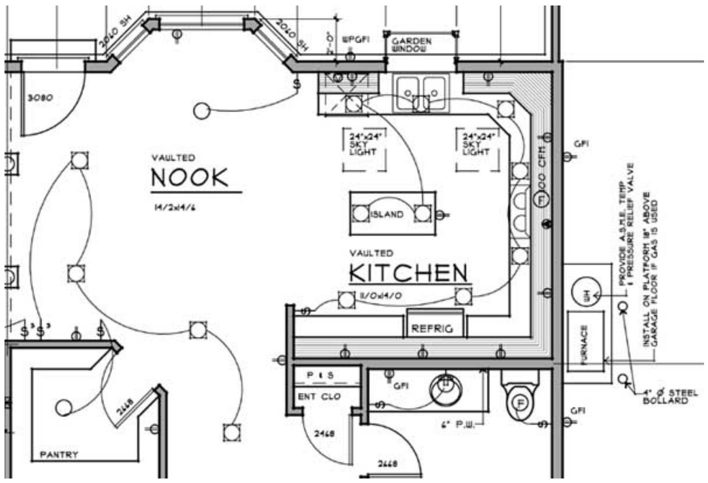




Floor Plan



Electrical Plan



Models/Views

- Each is a view of the same house
- Different views answer different kinds of questions
  - How many electrical outlets are available in the kitchen?
  - What happens if we put a window here?
- Designing for particular software qualities also requires the right architectural model or "view"
  - Any model can present only a subset of system structures and properties
  - Different models allows us to answer different kinds of questions about system properties
  - Need a model that makes the properties of interest and the consequences of design choices visible to the designer, e.g.
    - Process structure for run-time property like performance
    - Module structure for development property like maintainability

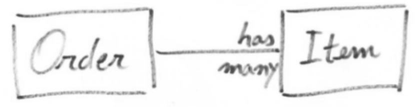


Figure 2: Conceptual Data Model – First Draft

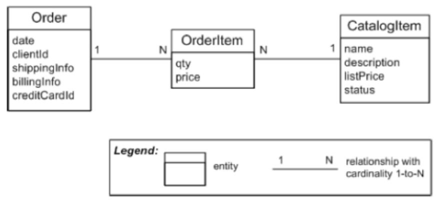


Figure 3: Logical Data Model

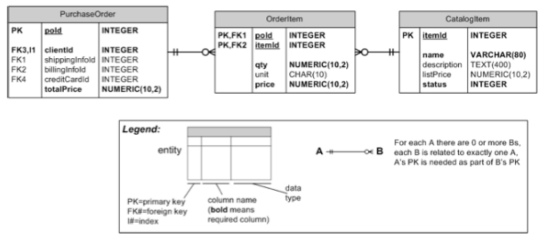


Figure 4: Physical Data Model

Example: Data Model View

- Data Model Architecture
  - Entities: data structures
  - Relations: cardinality, aggregation, generalization/specialization
  - Interface: attributes
- Model/communicate structure of complex data
  - What data is kept?
  - How is it related?
  - How is it structured and accessed in the system?

Which structures should we use?

Structure	Components	Interfaces	Relationships
Calls Structure	Programs (methods, services)	Program interface and parameter declarations	Invokes with parameters (A calls B)
Data Flow	Functional tasks	Data types or structures	Sends-data-to
Process	Sequential program (process, thread, task)	Scheduling and synchronization constraints	Runs-concurrently-with, excludes, precedes

- Choice of structure depends the *specific design goals*
  - Compare to architectural blueprints
- Choose minimal set of structures that
  - Make key design issues visible
  - Communicate key design decisions
- Which views would be useful for Address Book?

Some Key Architectural Structures

- Module Structure
  - Decomposition of the system into work assignments or information hiding modules
  - Most influential design time structure
    - Modifiability, work assignments, maintainability, reusability, understandability, etc.
- Uses Structure
  - Determine which modules may use one another's services
  - Determines subsetability, ease of integration (e.g. for increments)
- Process Structure
  - Decomposition of the runtime code into threads of control
  - Determines potential concurrency, real-time behavior

The Module Structure

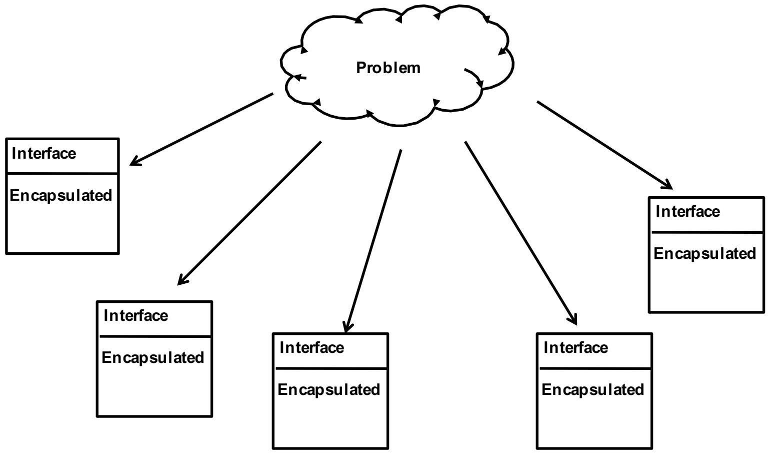
Modularization

- For any large, complex system, must divide the coding into work assignments (WBS)
- Each work assignment is called a “module”
- Properties of a “good” module structure
  - Parts can be designed independently
  - Parts can be tested independently
  - Parts can be changed independently
  - Integration goes smoothly

Modularization Goals

- Reduces complexity, improves manageability
- Coding
  - Can write modules with little knowledge of other modules
  - Replace modules without reassembling the whole system
- Managerial
  - Allows concurrent development
  - Avoids “Mythical Man Month” effect (“adding people to a late software project makes it later”)
- Flexibility/Maintainability
  - Anticipated changes affect only a small number of modules
  - Can calculate the impact and cost of change
- Review/communicate
  - Can understand or review the system one module at a time

Notional Modules

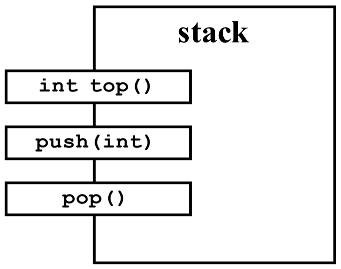


What is a module?

- Concept due to David Parnas (conceptual basis for objects)
- A module is characterized by two things:
  - Its interface: services that the module provides to other parts of the systems
  - Its secrets: what the module hides (encapsulates). Design/implementation decisions that other parts of the system *should not depend on*
- Modules are abstract, design-time entities
  - Modules are "black boxes" – specifies the visible properties but not the implementation
  - May, or may not, directly correspond to programming components like classes/objects
    - E.g., one module may be implemented by several objects

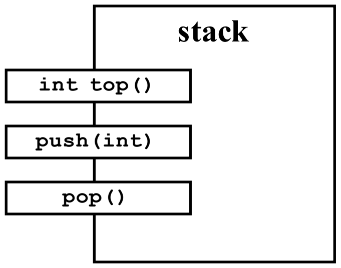
A Simple Module

- A simple integer stack
  - *push*: push integer on stack top
  - *pop*: remove top element
  - *top*: get value of top element
- What information is on the interface?
- What are the secrets?
- What information is missing?
- Why is this an abstraction?



A Simple Module

- A simple integer stack
- The *interface* specifies what a programmer needs to know to use the stack correctly, e.g.
  - *push*: push integer on stack top
  - *pop*: remove top element
  - *top*: get value of top element
- The *secrets* (encapsulated) any details that might change from one implementation to another
  - Data structures, algorithms
  - Details of class/object structure
- A module spec is *abstract*: describes the services provided but allows many possible implementations
- Note: a real spec needs much more than this (discuss later)



Why these properties?

Module Implementer

- The specification tells me exactly what capabilities my module must provide to users
- I am free to implement it any way I want to
- I am free to change the implementation if needed as long as I don't change the interface

Module User

- The specification tells me how to use the module's services correctly
- I do not need to know anything about the implementation details to write my code
- If the implementation changes, my code stays the same

***Key idea*: the abstract interface specification defines a contract between a module's developer and its users that allows each to proceed independently**

Is a module a class/object?

- The programming language concepts of classes and objects are based on Parnas' concept of modules
- To separate design-time concerns from coding issues, however, *they are not the same thing*
  - A module must be a work assignment at design time, does not dictate run-time structures
  - Coder free to implement with a different class structure as long as the interface capabilities are provided
  - Coder free to make changes as long as the interface does not change
- In simple cases, we will often implement each module as a class/object

Questions?