What is the Nature of Design?

- Addresses the question: HOW?
- Goal: Indicate how to develop a solution system that will satisfy requirements
- Complements:
  - Requirements: WHAT
  - System Test Plan: HOW WOULD I KNOW IT IF I SAW IT
- Design is a very broad and encompassing area
  - Hard to separate it from requirements
  - Hard to separate it from code
- Too hard to be done in one large step
  - Especially because of execution platform variation

Design is essentially a modeling activity

Numerous Well-Known Design Notations and Methods

- Jackson System Development (both noun and verb)
- Rational Design Method (verb)
- Shlaer-Mellor (verb)
- Booch Object Oriented Design (both noun and verb, but mostly the noun)
- Using various notations (noun)
  - DFDs
  - FSAs
  - Petri Nets
  - UML
- ...

Design

- It is both a noun and a verb
- Design the noun
  - Specification of a plan for something
- Design the verb
  - A process for creating a “design, the noun”
- Both are fundamental and hard
  - We will address both
    - But most work has been done on the verb
  - We will start with “design, the noun”
What Do Designs Model (and Why)?

- Conceptual, architectural, high-level designs model how requirements might be met
  - Vehicles for "what-if" discussions
  - Help clarify requirements—by being related to them
  - Often merge and intersperse with requirements
  - Help suggest implementation issues/concerns
- Coding specifications model the form, content, structure of the eventual code
  - Increasing emphasis on evolvability, rapid modification, and flexible deployment

How are Designs Represented?

- Familiar approaches
  - Use of hierarchy to conquer size/complexity
  - Use of multiple views to capture different aspects
  - Use of pictures and diagrams to appeal to non-technical stakeholders
- Connected to requirements elements they respond to
- Connected to code elements that implement them

Architecture vs. Specification

- Architecture
  - High level system design
  - Concerned with components and the interactions among components
  - Not with the algorithms or data structures
- Specification (Low Level Design)
  - Emphasis on data structures and algorithms
  - Focus on implementation issues
    - Stepwise refinement
    - Evolvability
    - Use of abstraction

Traditional Software “Design”

- Early 70’s recognized the need for design
  - Emphasis on data structures and algorithms
    - Stepwise refinement
    - Data abstraction
  - More feasible to think of design as one phase
    - Focused on narrow set of issues
    - Still "Preliminary Design" vs "Detailed Design"
  - Corresponds roughly to what we will be calling “specification”

Architecture addresses new issues, requires a new term

- Change in how we do software development
  - Component based, distributed systems
- Previous HL design notations assumed procedure call model
- Need to focus on
  - Nature of components and their loci
  - Component interactions (Connectors)

Typical Architecture Issues

- Component interaction models
  - What are the components’ interfaces?
  - Who can use them? And how?
- How much flexibility is achievable? How modifiable?
  - Is plug and play possible?
- Where is network access used? How?
  - Message passing, broadcasting, etc?
- Late-binding issues
  - Non-determinism
  - Use of proxies
New issues in characterizing system objects

- Interaction protocols
  - Tightly coupled objects
    - Direct or Remote procedure calls
  - Loosely coupled
    - Event based notification, observers
- Degree of separation
  - Locality
  - Internet scale
  - "in the cloud"
- Modes of communicating with each other
  - message passing
  - broadcast
  - multi-cast

Architecture description (specification or design)

- A high level design that defines the components, connectors, constraints and the inter-relationships among these entities
  - Usually compositional
- The form (i.e. the type of graph) is not the point here
  - What it is trying to communicate (i.e. its position in the development process) is what is important
- Suggests the value of elaborate semantics and annotations of the nodes and edges

Example: System Interpreter Architecture

Example: A Simulator Architecture

Problems with This

- Picture creates an impression
  - But no specifics
- Raises questions
  - Answers few
- A good beginning
  - But how to proceed?
- Need a language
  - Capable of specifying details
  - Clear semantics
  - Etc.

Components, Connectors, Constraints: Central Software Architecture Entities

- Components--computational units
  - Subsystems
  - Classes
  - Objects
- Connectors--interaction model
  - Which components are connected to which?
  - How are they connected?
  - Are connectors just components with restricted semantics?
- Constraints
  - Guides and limits to the ways components and connectors can be configured
Architectural Styles

- Sets of constraints that are widely used because they offer understood capabilities and features
- Examples:
  - Pipe and Filter
  - Client/Server
    - REST
  - Publish/Subscribe
  - Model/View/Controller

Pipeline Architecture:
Each component has one input connector, one output connector

Pipe and Filter:
Pipeline architecture where some connectors have a “filter”

Filters:
Components that have particular properties (they “filter” the data moving thru the connectors)

Client/Server

Multiple Clients
Need To Specify Details

• What will a request look like?
• What will a reply look like?
• How will multiple simultaneous requests be served?
• Any constraints on requests, replies?
  – E.g. speed

Different Substyles

• How to specify different ways for client/server to perform
• REST Architecture
  – Server is “stateless”
  – No memory of details of client
  – A key property that www infrastructure is built upon

Service-Oriented Architecture

• Applications composed from components
• Components are accessed via the Web
  – Specified generically (as a "service")
  – Located by web searches (using proxies)
  – Accessed via the web
• How to compose such services?
  – What composition constructs
• How to be sure they provide correct services?
• How to maintain privacy and security?

SOA Variants

• SaaS (Software as a service)
• HaaS (Hardware as a service)
• SyaaS (Systems as services)
• Staas (Storage as a service)
• DaaS (Databases as a service)
• Taas (Tools as a service)
• Each emphasizes just what kinds of components are to be searched for and integrated

Cloud Architecture

• SOA approach, but
  – Don’t know/don’t care where or how services are provided via the Web
• Service may be different each time the system runs
• Similar problems, but now more worrisome
  – Correctness
  – Security
  – Privacy

Architecture description language (ADL)

• A language for defining an architecture
  – Components, connectors, constraints, configurations
• Supports specifying styles and details
• Often has associated capabilities
  – For editing
  – For visualization
  – For analysis
  – For system generation
  – For testing
Some Notable ADLs

- Different ADLs emphasize specification of different architectural issues and features
- Some examples
  - Darwin
  - Rapide
  - MetaH
  - ACME
  - Menage

Future Issue to be Addressed:
Consistency Verification

- Internal consistency of specification
- Internal consistency of architecture
- Consistency between specification and architecture
- Consistency between architecture and requirements
- Consistency between specification and implementation (in code)
- All of the above are done better when all these artifacts are defined more rigorously

Jackson System Development:
An Ancient Example

- JSD produces models of the real world and the way in which the system to be built must interact with it
- Primary focus of this is actions (or events)
- Actions can have descriptive attributes
- The set of actions are organized into sets of processes
- Processes describe which actions must be grouped together and what the "legal" sequences of actions are
- Processes can overlap in various ways
- Data are described in the context of actions
  (NOTE: In JSD data considerations are subordinate to actions)
- Processes are aggregated into an overall system model
  --Done with connectors

An Early Precursor to ADLs:
Focus on Some Basic Essentials

- JSD: Jackson Structured Design
  - Named after Michael Jackson
  - British software design pioneer
- Focus on high-level design: architecture
  - Lower levels addressed by JSP: Jackson Structured Programming
  - Together they span from requirements to coding
  - Now largely overtaken by use of various UML notations

References

- Overview:

Starts with “Process Models”—The Components

- Primary building blocks of a JSD design
  - Contain all actions characterizing a key real-world process
  - Actions are structured into a tree
  - Only the leaf nodes of the tree are real-world actions
  - Interior nodes are conceptual
  - Interior nodes can be annotated to show choice or iteration
  - Traversals of this tree constitute the only "legal" sequences of actions for this process
  - Model a process as a tree: defines a regular expression
  - Set of traversals is a regular set
  - Process models are usually multiply instantiated
**An Example:**
A Library Information System

- **Functional requirements:**
  - The way books are dealt with
    » Buy, lend, track, dispose
  - The way members are dealt with
    » Enrol, service requests (borrow, reserve, return)
  - Support bookkeeping requirements
    » Check status, answer queries

- **Non-functional requirements**:
  - Inquiries should be processed as soon as they are received
  - Reports have to be generated at the end of each day
  - Maintain privacy of transactions
  - Used primarily by librarians

**Main Architectural Features**

- **Some Key Components**
  - Book
  - Book repository
  - Member
  - Librarian

- **Some Key Connectors (i.e., links between them)**
  - Librarian interactions with Book Repository
  - Member interactions with Book
  - Etc.

**Some Key Architecture Decisions**

- **What are the characteristics of each component?**
  - What capabilities does it have?
  - What properties does it have?
  - Where is it located?
  - Etc.

- **How will the components interact with each other?**
  - By what media and protocols?
  - Subject to what restrictions?
  - Etc.

**Starts with Process Models**

- Primary building blocks of a JSD design
- Contain all actions characterizing a key real-world process
- Actions are structured into a tree
- Only the leaf nodes of the tree are real-world actions
- Interior nodes are conceptual
- Interior nodes can be annotated to show choice or iteration
- Traversals of this tree constitute the only "legal" sequences of actions for this process
- Model a process as a tree: defines a regular expression
- Set of traversals is a regular set
- **Process models are usually multiply instantiated**

**A JSD Component: The Book Process**

![Diagram of The Book Process]

**A JSD Component: The Book Process**

![Another diagram of The Book Process]
Component Defined as Actions and Entities

- Actions have the following characteristics:
  - An action takes place at a point in time
  - An action must take place in the real world outside of the system.
  - An action is atomic, cannot be divided into subactions.

- Entities have the following characteristics:
  - An entity performs or suffers actions in time.
  - An entity must exist in the real world, and not be a construct of a system that models the real world.
  - An entity must be capable of being regarded as an individual; and, if there are many entities of the same type, of being uniquely named.

Processes and Data

- Details of actions on data hang off of process model leaf nodes
- Input and Output actions too
  -- These are important during the network phase
- Global data is necessary too
  -- For functions that must combine data from >1 model process
  -- To assure consistency between processes
  -- To coordinate between different instances of the same process

Actions/Attributes

- Acquire
  - attributes: in the library, on loan
- Classify
  - attributes: in the library, requested, time out on loan
- Lend
  - attributes: in the library, on loan, time out on loan, loan date
- Renew
  - attributes: time out on loan, loan date
- Return
  - attributes: in the library, time out on loan, loan date
- Sell
  - attributes: in the library, out of loan

JSD Hierarchy
**Message Passing**

- Data stream carries a message from one process activity to an activity in another process (a DFG edge)
  - Must correlate with output leaf of sending model process
  - Must correlate with input leaf of receiving model process
- Data transfer assumed to be asynchronous
  - less restrictive assumption
- No timing constraints are assumed
- Messages are queued in infinitely long queues
- Messages interleaved nondeterministically when multiple streams arrive at same activity

**State Vector Inspection**

- Modeling mechanism used when one process needs considerable information about another
- State vector includes
  - values of all internal variables
  - execution text pointer
- Process often needs to control when its state vector can be viewed
- Process may need exclusive access to its vector
- Could be modeled as message passing, but
  - important to underscore characteristic differences
About those Graphs

- There are formal semantics of these graphs
  - Defined by Jackson and colleagues
- Components are defined precisely using
  - Hierarchy
  - Abstract interfaces
- Connectors are defined precisely too
- And their relations to each other

Comments/Evaluation

- This is an architectural design.
  - Should not assume the system will be built this way
- Based upon model of real world
- Careful (and experienced) analysis of the model generally suggests implementation tactics, though
- Treatment of data is very much subordinated/secondary
- Does a good job of suggesting possible parallelism
- Contrasts strongly with Objected Oriented notions
  (eg. Booch, UML), that we will see next
How Does One Go About Doing (Architectural) Design – the Verb?

- Process by which design is built is understandably complex
- Various authors have differing ideas about this
- A key essence of human creativity
  - Little understood
- Most efforts aim at supporting humans
- Much work on analyzing and verifying designs

JSD is done in Phases

- The modeling phase
  - Entity/action step
  - Entity structure step
- The network phase
  - Connect model processes and functions in a single system specification diagram (SSD)
- The implementation phase
  - Examine the timing constraints of the system
  - Consider possible hardware and software for implementing the system
  - Design a system implementation diagram (SID)

The Modeling Phase: JSD Models

- JSD produces models of the real world and the way in which the system to be built interacts with it
- Primary focus of this is actions (or events)
- Actions can have descriptive attributes
- The set of actions must be organized into set of processes
- Processes describe which actions must be grouped together and what the "legal" sequences of actions are
- Processes can overlap in various ways
- Data are described in the context of actions
  (NOTE: In JSD data considerations are subordinate to actions)
- Processes are aggregated into an overall system model
- Done with the aid of two canonical models of interprocess communication

The Network Phase: Communication Between Processes

- Weave Processes together incrementally to form the total system specification
- Also add new processes during this phase: eg. input, output, user interface, data collection
- Goal is to indicate how processes communicate with each other, use each other, are connected to user and outside world
- Linkage through two types of communication:
  --Message passing
  --State vector inspection
- Indicates which data moves between which processes
  --and more about synchronization

Implementation Phase

- JSD outputs suggest how to proceed with JSP
- Network Phase suggests ideal traversal paths through model processes and their local data
  - suggests internal implementation of model processes
  - studying use of model processes suggests internal structure of their data
- Communication by data streams and state vector inspection often suggest real implementations
  - But often not

“New” Development Approaches Exploiting Architecture Perspectives

- Model-based Development
- Component Based Development
- Service-Oriented Architecture
- Cloud Computing
Model-Based Development
- Start development by creating a model of the system to be built
  - Often specified using an architecture definition language
- Use the model to guide development of the system
- Hmmm. Sounds like "design before coding"
  - Think first (?)

Component-Based Development
- Think of development as the composition of pieces (components)
- So, start with a plan for how the pieces will fit together
  - A model?
  - A design?
- Start with some kind of catalog of what pieces are available
  - And what their interfaces are
- Fit the components into their places

Various Approaches to Doing This
- Build systems out of "components"
  - Parts that have already been built
- Components are large bodies of code
- But also accompanying
  - Test results
    - Explicit or Implicit
  - Design elements
  - Requirements
  - Etc.

Component-Based Development Example: A Japanese Software Factory
- Japanese hardware manufacturers built them in 70's, 80's
- Actual building designed to support software development
  - Ergonomic offices and amenities
- Strong computing support
- Library of existing, reusable components
- Clear and rigorously defined development processes
  - Emphasizing reuse of the existing components
- Heavy use of metrics measure degree of reuse

Reuse in the Software Factory
- Developers assigned tasks (generally coding)
- Task specification includes target reuse level
- Task specification includes suggestions about which components in reuse library are expected to be reusable
  - In some cases reuse probabilities are assigned
- Reusable components are more than code
  - Include related artifacts (eg. design, testcases)
- Developer is measured in part by how closely the reuse target has been met
- Adapted versions of reused components are contributed to the reuse library
  - Accompanied by related artifacts
  - With explanations of why adaptation was necessary
- Explanations of why components that were expected to be reused were not reused

Why Did This Work?
- Strong incentives to reuse (based on social pressure and productivity measurement)
- Investment in reuse
  - Costs extra to build components for reuse
  - Cost to maintain reuse database
  - Cost to resubmit adaptations to reuse database
- A software reuse process that works
- Focus on narrow software product line
  - One software factory builds only device drivers
  - Another may build operating system components
  - Another may build compilers for similar languages

Some software factory projects have achieved factor-of-ten improvements in "quality" and "productivity"