Design: The Specification Subphase

Software Engineering
Computer Science 620
Spring 2013
Prof. Leon Osterweil

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

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

How Does One Go About Designing

• Process by which design is built is understandably complex
• Various authors have differing ideas about this
• For this course, we separate WHAT from HOW
Numerous High-level Design Notations and Methods

- Jackson System Development
- RDM
- DFDs
- FSAs
- Shlaer-Mellor
- BOOD (Booch Object Oriented Design)
- UML
- ...

The Focus of the Specification Phase of Software Design is on *Modules*

Rational Design Methodology (RDM)

- Suggested by David L. Parnas and Paul Clements
- Based on paper by Parnas

David L. Parnas, "On the Criteria to be Used in Decomposing Systems into Modules", Comm. Of the ACM, 15 #12, Dec. 1972

Rational Design Methodology (RDM)

- Focus is on end-product of design, not process
  -- Act of design is hard/unpredictable
  -- Outcome is what is most important
- Focus on need for good requirements as a starting point
  -- Requirements and design hard to separate
  -- Combination is a Specification

An RDM design can not be expected to be constructed as a sequential succession of these steps--BUT IT SHOULD APPEAR AS THOUGH THAT WERE THE CASE

RDM Components

- Requirements Specification
- Module Guide
  -- Enumeration of all modules needed to implement system
  -- Hierarchically structured (tree)
- Module Interface Guide
  -- How modules can be accessed and exploited
- Uses Hierarchy
  -- Which modules depend upon which others
- Internal Structure of Modules
  -- May need to be hierarchical as well
  -- Lowest level of hierarchy is coding specifications
  These components span from requirements to code

What is a Module?
What is a Module?
• Notion of module is defined carefully by Parnas
• Module is the locus of responsibility for a function or task
  – Hides decision(s) about implementation
  – May be nested
  – Provides services only through strict, impenetrable interfaces
  – Intended to be replaceable by alternate(s) having the same interface(s)
• A modular system is typically built as hierarchical family of modules
  – Basis for conceptualization of system
  – Basis for implementation of system

Information Hiding
• Each design unit hides internal details of processing activities
• Design units communicate only through well-defined interfaces (as opposed, e.g. to global variables)
• Each design unit is specified by as little information as possible
• If internal details change, client units should need no change
• Example decisions to hide
  – Algorithms
  – Data representations
  – Lower-level modules
  – Policies

The Typical Alternative:
Design by Stepwise Refinement
• Top-down technique for decomposing an architecture into lower levels
• Proceed by:
  – Isolating design aspects that are not interdependent
  – Postponing representation choices as long as possible
  – Showing that each successive refinement step is a faithful expansion of the previous steps
Pretty Much Equivalent to “Divide and Conquer”

- Start with system function
- Break into major function
- Break each into sub-functions
- Concurrently refine program and data
- Continue until implementation is “immediate”

Problems with Stepwise Refinement

- What’s the basis for determining whether design aspects are interdependent?
- Later design decisions depend on earlier ones.
  - But what is the basis for choosing the initial decision to make?
- Once a representation decision is made, further decomposition decisions depend on it.
- Promotes development of a sequential design solution (as opposed to concurrent)
- If the initial function is “huge” how do you start to decompose it?

KWIC Index Example

Input: a file of titles
- “Computers in Crime” <reference 1>
- “The Fastest Computers” <reference 2>
- “Computer Fun” <reference 3>

- Output: an alphabetized, permuted index
  - Computer Fun <reference 3>
  - Computers in Crime <reference 1>
  - Computers, The Fastest <reference 2>
  - Crime, Computers in <reference 1>
  - Fastest Computers, The <reference 2>
  - Fun, Computer <reference 3>
  - In Crime, Computers <reference 1>
  - The Fastest Computers <reference 2>

Stepwise Refinement

Step 1: Print_Kwic (title_list);
Step 2: Print_Kwic:
  - input all titles;
  - generate and save all interesting circular shifts;
  - alphabetize saved lines;
  - print alphabetized lines;
Step 3b: generate and save all interesting circular shifts:
  - for each line in input do
  - begin
  - generate and save all interesting circular shifts of this line;
  - end;

Data Flow Diagram Design for First KWIC Decomposition

More Detailed DFD
Refinement of Title_Lists_Store

- title_list entries:
  - Packed 4 characters per word
- all_perms entries:
  - A vector of indices, showing starting address of each title
- sorted_perms entries: same idea...

all_perms:

<table>
<thead>
<tr>
<th>Address of this title</th>
<th>Address of 1st character of this permutation</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

sorted_perms: Same idea as all_perms

After Input_Titles

<table>
<thead>
<tr>
<th>Title_List</th>
</tr>
</thead>
<tbody>
<tr>
<td>C o m p</td>
</tr>
<tr>
<td>u t e r</td>
</tr>
<tr>
<td>e n</td>
</tr>
<tr>
<td>C o m p</td>
</tr>
<tr>
<td>p u t e</td>
</tr>
<tr>
<td>r s</td>
</tr>
<tr>
<td>n s</td>
</tr>
<tr>
<td>C r</td>
</tr>
</tbody>
</table>

After Permuter

<table>
<thead>
<tr>
<th>All_perms</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>All_perms</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>

Leads to more Detailed DFD

Title_Lists_Store

- title_list
- input_titles
- permuter
- titles_info
- permuted_titles_info
- is this last title?
- retrieve next title
- number of titles
- store

alpha_list_info
circular_shifts_info
sorted_perms
permuted_titles_info
all_perms
sorted_perms
input_titles
output_titles
all_perms
sorted_perms
sorted_perms
permuted_titles
is this last title?

Leads to more Detailed DFD

Title_Lists_Store

- title_list
- input_titles
- permuter
- sorted_perms
- permuted_titles_info
- sort_titles
- retrieved_titles
- number of titles
- current_title
- count
- yes
- no

leads to more detailed DFD

leads to more detailed DFD
More Detailed DFD

Design Decisions Implied

• All shifts will be stored (in the indices)
• All circular shifts will be generated before alphabetization begins
• Alphabetical orderings will be completed before printing begins
• All shifts of one line developed before any shifts of another line
• "Uninteresting" shifts eliminated at the time the shifts are generated
Recall: Problems with Stepwise Refinement

- What's the basis for determining whether design aspects are interdependent?
- Later design decisions depend on earlier ones. [Same for information hiding.]
  - But what is the basis for choosing the initial decision to make?
- Once a representation decision is made, all successive design decisions in that subtree of refinements may be dependent on it.
- Promotes development of a sequential design solution (as opposed to concurrent)
- If the initial function is "huge" how do you start to decompose it?

The Information Hiding Alternative

- Each design unit hides internal details of processing activities
- Design units communicate only through well-defined interfaces (as opposed, e.g. to global variables)
- Each design unit is specified by as little information as possible
- If internal details change, client units should need no change

Examples of Information to Hide

- Algorithms
- Data Representations
- Lower Level Modules
- Policies

Information Hiding in our Example

- Internal representation of data to be processed
- Representation of circular shifts
- Time at which circular shifts are computed
- Method of alphabetization (sorting)
- Time at which alphabetization is carried out
- Input formats
- Output formats

Modularized Design

- Line Storage is a module
- Defined in terms of its interfaces
- Other modules use this by method calls
- Internal implementation details invisible
  - Change of line storage implementation details
  - Parallel development of modules
  - Module interchanging

Before

```
title_list
input_titles
permuter
circular_shifts_info
sort_titles
alpha_list_info
output_titles
```

Title_Lists_Store

```
title_list
all_perms
sorted_perms
```

```
title_list
all_perms
sorted_perms
```
Before

Title_Lists_Store

Input_titles

Title_list

Permuter

Cicular_shifts

Output_titles

Sort_titles

Sorted_perms

Sorted_perms

Title_list

Sort_titles

Output_titles

Sorted_perms

Changes must be agreed upon by others.

After

Line_Storage

Input_titles

Permuter

Sort_titles

Output_titles

Sorted_perms

Print_KWIC

represents procedure invocation

Line_Storage Interface

- Line_Storage
  - char(r,w,c) — returns the c-th character in the w-th word in the r-th input line
  - setchar(r,w,c,d) — performs char(r,w,c) := d
  - words(r) — number of words in line r
  - numchars(r,w) — number of characters in w-th word of line r
  - others are also possible (e.g., numlines, setword), depending upon needs of other modules

Sort_titles Interface

- sort_titles
  - alph — performs module initialization
  - ith(i) — index of the circular shift that comes i-th in alphabetical order

Permuter Interface

- permuter
  - Assumptions:
    - if i<j then shifts of input line i precede shifts of input line j in the ordering of all shifts maintained by this module
    - initial shift of a given title is the original line, next is one-word rotation, etc.
    - cs_char(l,w,c) — returns the c-th character of the w-th word in the l-th circular shift
    - cs_words() — number of words in l-th circular shift
    - (num_shifts() — number of shifts generatable from input line r — is a redundant, but related notion)
    - ...
    - cs_setup — performs module initialization

Design Decisions, Revisited

- All shifts will be stored
  - As opposed to computed on demand
  - Assumes you have enough memory to store everything
Design Decisions, Revisited

- All shifts will be stored
  - As opposed to computed on demand
  - Assumes you have enough memory to store everything
- All circular shifts generated before alphabetization begins
  - Precluding use of an insertion sort running concurrently or as a coroutine with the shift generator

Design Decisions, Revisited

- All shifts will be stored
  - As opposed to computed on demand
  - Assumes you have enough memory to store everything
- All circular shifts generated before alphabetization begins
  - Precluding use of an insertion sort running concurrently or as a coroutine with the shift generator
- Alphabetical orderings completed before printing begins
  - Precluding concurrency and demanding more storage
  - (e.g. after first half printed, storage could be reused)

Design Decisions, Revisited

- All shifts will be stored
  - As opposed to computed on demand
  - Assumes you have enough memory to store everything
- All circular shifts generated before alphabetization begins
  - Precluding use of an insertion sort running concurrently or as a coroutine with the shift generator
- Alphabetical orderings completed before printing begins
  - Precluding concurrency and demanding more storage
  - (e.g. after first half printed, storage could be reused)
- Do all shifts of one line before any shifts of another
  - Perhaps faster to do all first shifts first, then
  - alphabetization of them, then second shifts...

Design Decisions, Revisited

- All shifts will be stored
  - As opposed to computed on demand
  - Assumes you have enough memory to store everything
- All circular shifts generated before alphabetization begins
  - Precluding use of an insertion sort running concurrently or as a coroutine with the shift generator
- Alphabetical orderings completed before printing begins
  - Precluding concurrency and demanding more storage
  - (e.g. after first half printed, storage could be reused)
- Do all shifts of one line before any shifts of another
  - Perhaps faster to do all first shifts first, then
  - alphabetization of them, then second shifts...

- "Uninteresting" shifts eliminated when shifts generated
  - Burying this policy decision within the shift generator

Differences

- Are in the way the modules are divided into work assignments and in the the interfaces between modules
- Changeability
  - E.g., Changing property 1 (internal data representation) could cause change in all modules of first scheme (and in only one of second scheme)
- Independent Development
  - Scheme 1: formats and table organizations are complex and (too) essential to efficiency
  - Scheme 2: interfaces more abstract, containing function names and their parameters
- Comprehensibility
  - In order to understand the output module in Scheme 1 you need to understand previous modules, the "whole system", as opposed to just one module in Scheme 2.

Some Observations

- Scheme 1: makes each major step in processing a module
- Scheme 2: uses information hiding, where modules need not correspond to processing steps
  - E.g. alphabetization may or may not correspond to a processing phase
  - Every module in Scheme 2 is characterized by its knowledge of a design decision which it hides from the others
    - (Start decomposition with a list of design decisions!)
    - Interfaces reveal as little as necessary about internal module workings
      - Scheme 1 may leave important design decisions visible in interfaces
      - Scheme 2 enables more concurrent development
    - Clean decomposition and hierarchical structure are independent properties of system structure
Structure of an RDM Design Specification

- Module List
  - Enumeration of all modules
  - How modules can be accessed and exploited
  - Interface methods, for example
- Module Interface Specifications
  - Which modules depend upon which others
  - And in which ways
  - Description of the information being hidden by the module
- Interaction Hierarchy
  - Probably should be hierarchical
  - Lowest level of hierarchy should be close to coding specifications

Module List

- Could be a list of modules
- Could have the list structured
- Redundant with later specifications

E.g. DFD for KWIC Decomposition

- title_list
- input_titles
- permuter
- sort_titles
- sorted_perms
- output_titles

Or as an invocation structure

- title-list
- input_titles
- permuter
- sort_titles
- output_titles
- Line Storage
- sorted_perms
- Print_KWIC

Module Interface Specifications

- Here are some examples that we just saw

What secret(s) does each hide?

Line_Storage Interface

- Line_Storage
  - char (r,w,c) returns the c-th character in the w-th word in the r-th input line
  - setchar (r,w,c,d) performs char (r,w,c) := d
  - words(r) number of words in line r
  - numchars(r,w) number of characters in w-th word of line r
  - others are also possible (e.g., numlines, setword), depending upon needs of other modules
**sort_titles Interface**

- **sort_titles**
  - `alph` --- performs module initialization
  - `ith (i)` --- index of the circular shift that comes i-th in alphabetical order

**permuter Interface**

- **permuter**
  - **Assumptions:**
    - if `i < j` then shifts of input line `i` precede shifts of input line `j` in the ordering of all shifts maintained by this module
    - initial shift of a given title is the original line, next is one-word rotation, etc.
    - `cs_char(l,w,c)` --- returns the c-th character of the w-th word in the l-th circular shift
    - `cs_words(l)` --- number of words in l-th circular shift
    - `num_shifts(r)` --- number of shifts generatable from input line r --- is a redundant, but related notion
    - `cs_setup` --- performs module initialization

**Invocation interactions**

- **title_list**
- **input_titles**
- **permuter**
- **sort_titles**
- **output_titles**
- **sorted_perms**
- **Print_KWIC**
- **Line_Storage**

**Interaction Hierarchy**

- A structure showing which modules interact with each other in which ways

**Internal Structure of Modules**

- Saw some of that in some of the module interfaces
Information being hidden

- Data structures
- Algorithms
- Implementation tricks
- Other modules used
- Other external capabilities used

Structure of an RDM Design Specification

- Module List
  - Enumeration of all modules
- Module Interface Specifications
  - How modules can be accessed and exploited
  - Interface methods, for example
- Interaction Hierarchy
  - Which modules depend upon which others
  - And in which ways
- Internal Structure of Modules
  - Probably should be hierarchical
  - Lowest level of hierarchy should be close to coding specifications
- Description of the information being hidden by the module