Specification and Coding

Software Engineering
Computer Science 520-620
Spring 2014
Prof. Leon Osterweil

**Requirements Spec.**

Characteristics of System to be built must match required characteristics

Hi level design must show HOW requirements can be met

Test Results must match required behavior

**Test Plan**

Test plan exercises this code

**Design**

(high level) Architecture consistent views

(low level) specification

Code must implement design

**Code**
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
Design

• It is both a noun and a verbal
• 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 focus on the former

Almost the same thing can be said about “specification”—it is both a noun and a gerund

One Issue:
How are Specifications 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 architecture elements they respond to
• Connected to code elements that implement them
A Different Issue: How Does One Go About Creating a Specification (low level design)

- It is a complex process
- Various authors have differing ideas about this
- Most are poorly defined and informal
  - Hard to understand
  - Hard to compare and assess
- We start by focusing on the first issue: representation
- Then we focus on the second issue: Representing processes

Rational Design Methodology (RDM)

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

The Focus of the Specification Phase of Software Design is on the product, a structure of *Modules*

Representation is primarily by familiar graph structures

**BUT**

Important semantics are attached to the nodes and edges

---

**Rational Design Methodology (RDM)**

- Focus is on end-product of design, not process
  --Act of design is hard/unpredictable
  --Outcome is what is most is 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
Rational Design Methodology (RDM) Focuses on the Products

- Act of design is hard/unpredictable
  - RDM has little to say about how to do it
  - Outcome is what is most important
- An RDM design can not be expected to be constructed as a sequential succession of steps
  - BUT IT SHOULD LOOK THAT WAY
- Focus on helping others understand the design

Starts with good Requirements

- Requirements and design are hard to separate (Parnas)
- Combination is what Parnas calls a *Specification*
- Each is useful in helping to fill out the other
- Key feature of the design is its focus on solution units
  - MODULES
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)
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
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: Specification by Stepwise Refinement

- Focus on the specification process
- 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?
The Classical Example: KWIC Index

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;

More Detailed DFD

Title_list

input_titles

title_list

permuter

circular shifts info

alpha list info

sort_titles

sorted_perms

output titles

Title_Lists_Store
CFG for permuter

 Leads to 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>
<th>Title List</th>
</tr>
</thead>
<tbody>
<tr>
<td>C o m p</td>
<td>C o m p</td>
</tr>
<tr>
<td>u t e r</td>
<td>u t e</td>
</tr>
<tr>
<td>(space) F u n</td>
<td>(space) F u n</td>
</tr>
<tr>
<td>(end of title) C o m</td>
<td>(end of title) C o m</td>
</tr>
<tr>
<td>p u t e</td>
<td>p u t e</td>
</tr>
<tr>
<td>r s (space) l</td>
<td>r s (space) l</td>
</tr>
<tr>
<td>n (space) C r</td>
<td>n (space) C r</td>
</tr>
</tbody>
</table>
After Permuter

Title List

<table>
<thead>
<tr>
<th></th>
<th>C</th>
<th>o</th>
<th>m</th>
<th>p</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>u</td>
<td>t</td>
<td>e</td>
<td>r</td>
</tr>
<tr>
<td>(space)</td>
<td>F</td>
<td>u</td>
<td>n</td>
<td></td>
</tr>
<tr>
<td>(end of</td>
<td>C</td>
<td>o</td>
<td>m</td>
<td></td>
</tr>
<tr>
<td>title)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>p</td>
<td>u</td>
<td>t</td>
<td>e</td>
</tr>
<tr>
<td></td>
<td>r</td>
<td>s</td>
<td></td>
<td>I</td>
</tr>
<tr>
<td>(space)</td>
<td>n</td>
<td></td>
<td>C</td>
<td>r</td>
</tr>
</tbody>
</table>
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
• This facilitates
  – Change of line storage implementation details
  – Parallel development of modules
  – Module interchanging
Decisions about storage of titles, permutations, sorted permutations are not hidden.
Changes must be agreed upon by others.

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 \(\text{char} (r,w,c) := d\)
  - 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

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

• 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...

• `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 interfaces between modules
- Evolvability
  - 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
- 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

Module List

- Could be a list of modules
- Could have the list structured
- Redundant with later specifications
E.g. DFD for KWIC Decomposition

```
<table>
<thead>
<tr>
<th>title_list</th>
</tr>
</thead>
<tbody>
<tr>
<td>input_titles</td>
</tr>
<tr>
<td>permuter</td>
</tr>
<tr>
<td>all_perms</td>
</tr>
<tr>
<td>sort_titles</td>
</tr>
<tr>
<td>sorted_perms</td>
</tr>
<tr>
<td>output_titles</td>
</tr>
</tbody>
</table>
```

Or as an invocation structure

```
<table>
<thead>
<tr>
<th>title-list</th>
</tr>
</thead>
<tbody>
<tr>
<td>input_titles</td>
</tr>
<tr>
<td>permuter</td>
</tr>
<tr>
<td>sort_titles</td>
</tr>
<tr>
<td>Print_KWIC</td>
</tr>
<tr>
<td>output_titles</td>
</tr>
</tbody>
</table>
```

represents procedure invocation
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
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

**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

- Might be some utilities used to facilitate doing this
- If so, then indicate that here
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
Object Oriented Design

- Focus is on specification and coding
- Philosophically close to (derived from) RDM
- Stresses importance of creating abstractions of the entities in the real world
- Primary focus is on data, not on activities
- Currently popular design methodology approach
- Wide variety of adaptations of this idea
- Often used subsequent to high-level/architectural design

Here is an object structure from Tokeneer

Certificate relationships

3.2 Biometrics

All of the complexity of the biometrics is hidden within the biometric library, which we will be simulating in a very simple way, enabling the test drivers to decide whether a fingerprint will or will not match a template.

3.3 Door/Latch

The door has four possible states: the cross-product of open/closed and locked/unlocked.

Open means the door does not present a hazard from entering or leaving the enclosure.

Closed means the door prevents a human from entering or leaving the enclosure. To enter or leave, the door must first be opened.
History

- Related to/descended from Parnas notion of Module
- In keeping with popular emphasis on superior use of abstraction
- Consistent with ideas about abstract data types
- Strongly motivated by examples of superior code written in languages such as Modula, Smalltalk
  -- OOD is intended as the starting point for development of code in such superior languages
- More impetus provided by interest in reuse and evolution
- Interest strengthened by disillusion with older design ideas (eg. iterative refinement)

OOD Characteristics

- Primary organization of design is as a collection of objects
- Activities are organized according to objects they affect
- Claim: This is more "natural"
- Stresses importance of insulation from effects of change (like RDM)
- More focus on potentially reusable components
- Claim: Design is clearer, more modifiable
- Strongly suggests implementation modules
- Meshes nicely with modern languages that emphasize strong support for Modularity (eg. Ada, C++, Java)
What is an Object?

According to Booch: It is an entity that

- Has state
  --Distinguishes this clearly from a function/activity
- Is characterized by actions it suffers/requires of other objects
- Is an instance of a class (type)
  --But the type may have only one instance
- Is denoted by a name
  --But may have many names (aliases)
- Has restricted visibility to (and by) other objects
- Is divided into two parts: specification and implementation
  --Implementation may be in terms of other objects

Characteristics of Classes/Objects

- A Class
  --Usually has instances
  --Has operations upon it
- Class is defined in terms of its operations (methods)
  --Not in terms of its structure/representation/etc.
- Class may have attributes
  --Often defined as values returned by methods
- Operations are defined as part of the type, but operate on instances, not on the type itself
  --Similarly for attributes
- Two types of operations:
  --Constructors change object state (eg. by creating it, destroying it, changing it in some way)
  --Selectors gain access to all or part of the state
  Booch adds iterators, enables visiting all objects in a class
Inheritance

- Not all authors (e.g., Booch) consider inheritance to be essential to object-orientedness.
- Inheritance is a way of organizing/classifying the classes in a system, organization, etc.
  --It facilitates reuse.
- Classes organized into hierarchies.
- Child classes are elaborations on their ancestor classes.
  --Add new methods and/or attributes.
- Helps develop new classes (build upon the old ones).
- Helps developers find classes (follow the hierarchy down).
- But, the world is generally not strictly hierarchical.
  --Often classes may need to inherit from more than one line of descent (multiple inheritance).
  --Often classes have methods that descendents don't need.
  --Often classes need to override methods from ancestors.
- These needs lead to various multiple inheritance schemes.
- None of this seems integral to OOD.

Components of an OOD Product

- List of classes
- Attributes for each class
- Operations (methods) for each class
- Interobject visibility
- Object interfaces
- Implementations of objects

Note how close this is to an RDM specification.
Objects (Classes) and Attributes

- Objects identified by browsing requirements text or DFD
- Abbott: The nouns in a natural language reqts. spec are prime candidates for objects
  --adjectives are attributes/verbs are methods
- From a DFD: look for the major operands to the major data transformation steps
- Objects are often nested
- Set of objects is all-too-often not sufficiently well nested
  --Causes problems for large systems with many (hundreds, thousands?) of objects
- Large, experienced software organizations are starting to develop and maintain libraries of (reusable) objects

Operations (Methods)

- Semantics of an object (class) are completely provided by the set of methods on it
- Similar to Parnas notion of defining a Module in terms of its accessing primitives
- Identify methods by:
  --Looking at verbs in natural language spec.
  --Identifying activities in DFD's that manipulate the object
- Assure that set of methods provides complete state maintenance facilities and all services needed by other objects from other classes
Interobject Visibility

• System to be built is a network of collaborating and communicating objects

• Methods on classes are there to support needs of other classes: which ones need which others?

• Class should expose/make available all that is needed, but no more

• Document what is needed (and by whom) to support development of class

• Conversely, documentation of what is available from a class helps developers of other objects develop what they need more easily

Object (Class) Interface

• Formal, rigorous specification of what the class offers, and how to use it

• Often done in an actual coding language (eg. Ada, Java)

• One of two parts of a class definition: this is the public, visible one

• Components of the interface:
  --Name of the class
  --Its lineage in a hierarchy
  --Its attributes
  --Its methods (with complete calling sequences)
Object (Class) Implementation

- Create appropriate internal representation
- How to maintain object state
- How to implement the various methods
- Use of other classes where indicated
- Generally done directly in a coding language (eg. Java)

BOOD Diagrams

- Graphical representation given prominence
- Partly through proprietary software
  - e.g. IBM/Rational’s Rose
- “Cloud” charts
  - Nothing to do with “cloud computing”, though
- Many subtypes of clouds
  - Depict many variations, attributes, etc.
- Need to depict/define more than just classes
Design Process Not Addressed by BOOD

• Process is described in a book
  – With many examples
  – Vague hints and suggestions
• Focus is more on end product
• But DesignING is hard and complex
• Can be represented by:
  – DFGs
  – FSMs
  – CFGs
  – Etc.
• That can also represent system dynamics issues

Multirepresentation Systems

• Have seen that different representations are of different uses
• One diagram may be useful in different ways to different stakeholders
• But most stakeholders require a variety of diagrams
• Several different diagrams can be expected to be needed to satisfy different stakeholders
• Problems with different views/diagrams
  – Are they all representing the same software product?
  – How to assure that they are all consistent with each other?
  – If the product changes, then ALL views must change correspondingly
STATEMATE supports Requirements Specification

- Key feature is maintenance of consistency among views
  - Done by projecting views of abstract model
  - Change abstract model through changes to views

- Use of Hierarchical Decomposition for understandability

- Different diagrammatic views
  - Statecharts (Enhanced FSMs)
  - Enhanced Data Flow Diagrams

- Statecharts, FSMs both support specification of different requirements types simultaneously
  - Functional
  - Robustness
  - Safety

UML (Unified Modeling Language)

- Merger of the “The three amigos”
  - Booch (OO Design)
  - Rumbaugh (OOD, DFDs, FSAs)
  - Jacobsen (Use cases)

- Brought together by Rational (now IBM)
- Comprehensive suite of diagrams
- Comprehensive tool suite too
The Unified Modeling Language

• Some semantics in place
  – But not all
  – International task forces (!) working on this
• Process for using them was developed too
  – Rational Unified Process (RUP)
• UML blew away the opposition
  – Not clear this was good

(Some) UML representations

• Class Diagrams
• Use Cases
• Sequence Diagrams
• Package Diagrams
• State Diagrams
• Activity Diagrams
• Collaboration Diagrams
• Deployment Diagrams

Commonly used to represent design and requirements too
UML Augmented by RUP (Rational Unified Process)

• RUP is a detailed specification of the process of developing software
• Aimed at development using UML
• But more generic than that
• Tries to address the complexities of development
  – E.g. the need for iteration

Major UML Problems/Objections

• What are semantics of all of these features of all of these diagrams?
  – Task forces working on them
  – Maybe there is just too much there (?)
• Diagram semantics overlap
  – Which diagram to use when
  – How to tell when they are inconsistent
• Extensibility
  – Use of “stereotype” feature
  – How to reconcile semantics of new features with existing ones
UML Tries to cover everything

- A diagram type for everything
- But they are not well connected to each other
- Few rules on what to use when
- Long reach with uncertain grasp

Evaluating Which Design Method to Use

- Different design methods incorporate different semantics
- What do you want to define/communicate to various stakeholders?
- Need to verify consistency with requirements is often a key driver in this decision
- What types of requirements are of interest to stakeholders?
  - Choose design representation accordingly
Design Processes

• They are hard, requiring considerable iteration
• Key issue is rework
• Rework: Revisiting previously-done development steps, but now in a new context
  – Hard to depict this with pictures
  – The context is artifacts and their values, histories
• Fred Brooks’s new book:
  – The Design of Design

Much more about this in about a week or so