The Specification Phase of Design

Software Engineering
Computer Science 620
Spring 2012
Prof. Leon Osterweil

- Requirements Spec.
- Test Plan
- Code
- Design

- 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 exercises this code
- (high level) Architecture consistent views
- (low level) specification
- Code must implement design
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

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


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
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: 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>

---

**Data Flow Diagram Design for First KWIC Decomposition**

```
title_list

input_titles

permuter

all_perms

sort_titles

sorted_perms

output_titles

sorted_perms
```

CS 620 Spring 2012 Univ. of Massachusetts

Copyright L. Osterweil, all rights reserved
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

permuter

circular shifts info

sort_titles

output titles

sorted_perms

all_perms

title_list

title_list

Title_Lists_Store

sorted_perms

all_perms

alpha list info

info
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....

<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>

**all_perms:**

- Don't duplicate storage of each line as a result of generating circular shifts

**sorted_perms:**

- Same idea as all_perms

---

After Input Titles

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

---

CS 620 Spring 2012 Univ. of Massachusetts

Copyright L. Osterweil, all rights reserved
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
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

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 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 has important design decisions visible in interfaces
- 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

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

Or as an invocation structure

- title-list
- input_titles
- permuter
- sort_titles
- output_titles
- Line_Storage
- Print_KWIC
- sorted_perms

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

Invocation interactions

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

represents procedure invocation
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
Connecting Design and Requirements

- Important to verify internal consistency of design
- Important to verify that design is consistent with requirements
- Important to use design to complete requirements
- All of the above are done better when design (and requirements) are defined more rigorously

Object Oriented Design

- Focus is on later stages of design and on coding
- Philosophically close to RDM
- Stresses importance of modelling the real world
- Primary focus is on data, not on activities
- Currently the most popular design methodology approach
- Wide variety of adaptations of this idea
- Often used subsequent to high-level/architectural design
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
• 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 descendants 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
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 (e.g., Java)

BOOD Diagrams

- Graphical representation given prominence
- Partly through proprietary software
  - Rational’s Rose
- “Cloud” charts
- Many subtypes of clouds
  - Depict many variations, attributes, etc.
- Need to depict/define more than just classes
Other OOD’s

• OMT (Rumbaugh)
  – Very similar to Booch
  – But more diagram(s)
• Jacobsen Use Cases
  – Strong similarity to JSD (soon)
  – Emphasis more on early phases
• Shlaer-Mellor
• Etc., etc.

UML (Unified Modeling Language): Combining BOOD and JSD(?)

• Merger of Booch, Rumbaugh, Jacobsen work
  – “The three amigos”
  – All worked for Rational (now IBM)
• Comprehensive suite of diagrams
• 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

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

Contrasting JSD and OOD

- JSD focus is on actions and processes
  OOD focus is on objects
- JSD focus is on modeling the "real world"
  OOD focus is on abstracting from the real world
- JSD seems most adept at conceptual, high level design
  OOD seems most applicable to addressing more implementation oriented issues
- OOD places more emphasis on Parnas-style modular encapsulation

Note: UML tries to cover both
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

Process for Doing This

Module-Based Design

1. Perform RDM
2. Identify Modules
3. Identify a Module
4. Specify Module Implementations
5. Develop Rqmt Element
6. Requirements

* Module Design is OK

~ Rqmts OK

~ Modules OK

~ A Rqmt OK
Example Requirement Specification Process

Requirements + Develop Rqmt Element

Declare and Define Rqmt

Declare Rqmt Element

Define Rqmt Element

Inter-requirement Consistency Check

Rqmt OK

Definition of Define Rqmt Element

Define Rqmt Element

Define Functional

Define Timing

Define Accuracy

Define Input/Output

Define Robustness

Rqmt OK
Better Definition of Define Rqmt Element

Design: Summary

• Broad phase: intertwines with requirements at the high level; with code at the low level

• Various approaches, but all entail modelling

• All advocate creating multiple coordinated views

• Specialization is evident

• Reasonable to use different design modelling approaches at different stages of one design activity

• Reasonable to use different design approaches for problems in different application areas

• Design artifacts must be connected to most other types of software artifacts by various consistency relations
The Coding Phase

- Goal: Produce executable code in a coding language
- Gets down to very specific details:
  -- Procedures/algorithms
  -- Data structures
  -- The interactions between them
  -- THE DEVIL IS IN THE DETAILS
- Coding is usually 10-15% of the effort on a software development project: We will spend little time on it in this course
- Coding should follow closely the specifications resulting from the final phases of design
  -- Modular structure of the code
  -- Object specifications (data modules) (Data abstractions)

Coding

- Goal: Create code that can be executed on a computer
- Developer writes source code
- Object code emitted from a compiler
  So, is code really just another model?
- Executable results from loading object code with libraries, utilities, etc.
- Important to keep all of these straight
- Some designed to support specific design methodologies
- Some are special-purpose, well adapted to application domains
What makes a programming language “good”?

If it meets the needs of its stakeholders
A “good” language is one that meets the needs of its stakeholders

- Different kinds of projects
  - Quality is super-important
  - Rapid deployment is key
  - Evolvability is paramount
  - Emphasis on user interface
  - Etc.
- Suggest languages with strengths like
  - Readability
  - Expressive power
  - Low level (close to the machine)
  - Dynamism and late binding
  - Etc.

On Languages

- Bad code can be written in any language
  - But some languages encourage bad practices
- Good code can be written in any language
  - But some languages encourage it/make it easier
  - And discourage bad practices
- Most modern languages try to encourage good practices
  - Like those we have been advocating (in discussing design)
    » Modularity
    » Information hiding
    » Data abstraction
    » Incorporation of design and requirements specification into code
    » Support for testing and analysis
Information Hiding in Implementation

- Implementation units should hide internal details as specified by a Modular design
  - Superior procedure semantics support this better
- Implementation units should communicate through well-defined interfaces (not global variables).
  - Some languages make global data easier than others
- Some languages make it hard to inspect internals of Modules.
  - Others make it easier
- Different decisions are harder or easier to hide
  - Algorithm
  - Data representation
  - Lower-level modules
  - Policy

Data Abstractions

- User's (client's)-eye view of the data types to be used
- Essentially the same as Parnas notion of a "data module" --and the notion of an "object"
- Cluster of "accessing primitives" / "methods" whose purpose is to provide the only mechanisms for manipulating data of a given type
- Problem: How to specify the semantics of these types --without specifying their implementation
- Being rigorous help separate (even slightly) different notions of an ADT from each other
Assertion Languages

- Assert statements to define assertions
  - Assertions defined by programmer
  - Locations identified by programmer
  - Reactions to violations defined by programmer
- Different assertion language semantics
  - Usually Boolean logic
- Sometimes private data space

Tool Suites

- Better tools make languages more useful
- Better editors
- Better diagnostics
- Better testing aids
- More powerful libraries
- Etc.
Ada

- Early language that supported information hiding
  - Use of External and Internal part dichotomy
  - Strict encapsulation
- Support for data abstraction
  - Packages
- Very wordy
- Support for disciplined concurrency
- No type hierarchy
- Very static language

C

- Gets you down “close to the machine”
- Little restriction on use of pointers
- Little restriction (help with) dynamic storage allocation
- Little support for encapsulation
C++

• Adds support for objects to C
• Thus, support for objects (encapsulation)
• Type hierarchies
• Still little discipline over pointers, storage allocation

Java

• Early language with special attention paid to dynamism and the web
• Designed to facilitate distributed applications
  – Host readily on various machines (across the web)
• Support by lots of tools
• Highly dynamic language
  – Various sorts of late binding
• But more discipline than C (eg. over use of pointers)
Lisp

• Very dynamic language
• Very little compilation done
  – Mostly interpretation
  – Create code “on the fly” and interpret it
• Excellent vehicle for rapid prototyping
• Virtually no concept of types
  – Types with Lisp extensions
• Primitive flow of control structures
• Very hard to encapsulate

Ruby/Python/Perl

• Highly dynamic
• Interpreted, not compiled
• Sometimes used as a scripting language, sometimes as
general-purpose programming language
• Object-orientedness varies
• Extensive libraries
• Supporting frameworks
  – RoR for web applications
Prolog

- Rule-based language
  - No real procedural flow of control
  - Emphasis on reaction
- Favorite language for trying to capture human knowledge
- Data is subordinated
- Structuring, modularization are difficult

Patterns

- Higher level implementation constructs
- Idioms (Rich and Waters, ~1985)
- The “Gang of Four” book
  - Inspiration from “real” architects (C. Alexander)
- Idioms in common use
- Suggest ways that humans think/human esthetics
- Transcend specific languages
- Some finding more direct support in newer languages
Coding closely tied up with “Testing”

- The essence is dealing with Faults
- They manifest themselves as Failures

Basic Definitions

- Failure: inconsistency between actual behavior of software and specification of intent
- Fault: software flaw whose execution caused the failure
- Error: human action that results in software containing a fault
- Testing: The systematic (?) search of a program's execution space for the occurrence of a failure
- Debugging: Searching for the fault that caused an observed failure
More Definitions

- **Testing**: The systematic (?) search of a program's execution space for the occurrence of a failure
- **Debugging**: Searching for the fault that caused an observed failure
- **Analysis**: The static examination of a program's textual representation for the purpose of inferring characteristics
- **Verification**: Using analytic inferences to formally prove that all executions of a program must be consistent with intent

Validation and Verification (V&V)

```
Informal Requirements
  ↓
Formal Requirements
  ↓
Software Implementation
  ↑
Validation
  ↓
Verification
```

Validation:
- Informal Requirements
- Formal Requirements
- Software Implementation

Verification:
- Validation