Introduction to Data Structures

Credits

- Some of these slides were written by me
- Some information was found on the internet!
- Some information was from Michael Main, Data Structures and Other Objects Using Java
- Some information was from Wikipedia

Today's Goal

- Talk about standard data structures and where they are used
- Talk about sorting and searching
- Talk about running time analysis

Data Structures Overview

- Primitive Types
  - Bit, byte, etc.
- Collections
  - Bags
  - Array Lists (Sequences)
  - Linked Lists (Sequences)
- Abstract Data Types
  - (Interfaces, Abstract Classes)
- Stacks – Last in, first out
- Queues – First in, first out
- Recursion
- Trees
  - Binary Search
  - Heaps
- Searching
  - Serial/Linear Search
  - Binary Search
  - Hashing
- Sorting
  - Quadratic Sorts
  - Quick Sort
  - Merge Sort
  - Heap Sort
- Running Time Analysis
  - Big-Oh

Bit

- Tiniest data structure
- On or Off
- 1 or 0
- Fundamental building block of all data

“Byte”

- 8 bits
- Bytes store numbers
- We use numbers to represent everything else
- Numbers are stored, ultimately in binary
- Example:
  00000001 is 1
  00000100 is 4
  00001100 is 12

Nybble / nyble/nibble

- Half a byte, 4 bits

Byte

- A byte can contain a number
- A byte can contain a character
- Byte = 25;
- Byte = ‘A’
- What if we want a word?

Array of bytes

char bytes[5] ;

bytes[0] = ‘M’

Bytes[1] = ‘I’

Bytes[2] = ‘K’

Bytes[3] = ‘E’

Bytes[4] = 0

(Remember , computers like to start counting from zero)

Bigger Numbers

- Integers
- What if we want a number bigger than the largest value a byte can hold
- Integer types:
  - Short - 2bytes (packaged as 1)
  - Int - (4 bytes packaged as 1)
  - Long (8 bytes packaged as 1)

Review

- We understand multi byte arrays for storing words (array)
- We understand multi byte structures for storing numbers (types of int)
- We can have an array of ints too

One Dimensional Array of integers

Int ints[5] ;

ints[0] = 3

ints[1] = 4

ints[2] = 1

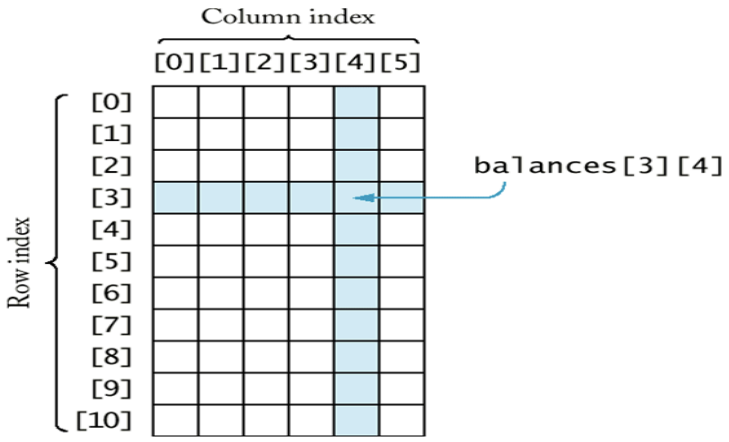
ints[3] = 2

ints[4] = 0

So what is int #2?

What is int #4?

Two Dimensional Array



Array of words

```
return $stopwords = array("a", "about", "above", "above", "across", "after", "afterwards", "again", "against",
"all", "almost", "alone", "along", "already", "also","although","always","am","among", "amongst", "amoungst", "amount",
"an", "and", "another", "any","anyhow","anyone","anything","anyway", "anywhere", "are", "around", "as", "at", "back","be",
"became", "because","become","becomes", "becoming", "been", "before", "beforehand", "behind", "being", "below", "beside",
"besides", "between", "beyond", "bill", "both", "bottom","but", "by", "call", "can", "cannot", "cant", "co", "con",
"could", "couldnt", "cry", "de", "describe", "detail", "do", "done", "down", "due", "during", "each", "eg", "eight",
"either", "eleven","else", "elsewhere", "empty", "enough", "etc", "even", "ever", "every", "everyone", "everything",
"everywhere", "except", "few", "fifteen", "fify", "fill", "find", "fire", "first", "five", "for", "former", "formerly",
"forty", "found", "four", "from", "front", "full", "further", "get", "give", "go", "had", "has", "hasnt", "have", "he",
"hence", "her", "here", "hereafter", "hereby", "herein", "hereupon", "hers", "herself", "him", "himself", "his", "how",
"however", "hundred", "ie", "if", "in", "inc", "indeed", "interest", "into", "is", "it", "its", "itself", "keep", "last",
"latter", "latterly", "least", "less", "ltd", "made", "many", "may", "me", "meanwhile", "might", "mill", "mine", "more",
"moreover", "most", "mostly", "move", "much", "must", "my", "myself", "name", "namely", "neither", "never", "nevertheless",
"next", "nine", "no", "nobody", "none", "noone", "nor", "not", "nothing", "now", "nowhere", "of", "off", "often", "on",
"once", "one", "only", "onto", "or", "other", "others", "otherwise", "our", "ours", "ourselves", "out", "over", "own",
"part", "per", "perhaps", "please", "put", "rather", "re", "same", "see", "seem", "seemed", "seeming", "seems", "serious",
"several", "she", "should", "show", "side", "since", "sincere", "six", "sixty", "so", "some", "somehow", "someone",
"something", "sometime", "sometimes", "somewhere", "still", "such", "system", "take", "ten", "than", "that", "the",
"their", "them", "themselves", "then", "thence", "there", "thereafter", "thereby", "therefore", "therein", "thereupon",
"these", "they", "thicky", "thin", "third", "this", "those", "though", "three", "through", "throughout", "thru", "thus",
"to", "together", "too", "top", "toward", "towards", "twelve", "twenty", "two", "un", "under", "until", "up", "upon", "us",
"very", "via", "was", "we", "well", "were", "what", "whatever", "when", "whence", "whenever", "where", "whereafter",
"whereas", "whereby", "wherein", "whereupon", "wherever", "whether", "which", "while", "whither", "who", "whoever",
"whole", "whom", "whose", "why", "will", "with", "within", "without", "would", "yet", "you", "your", "yours", "yourself",
"yourselves", "the");
```

One characteristic of arrays

- Array Lookup is fast!
- You know the index, you go right to the ‘box’.

Think of mail boxes

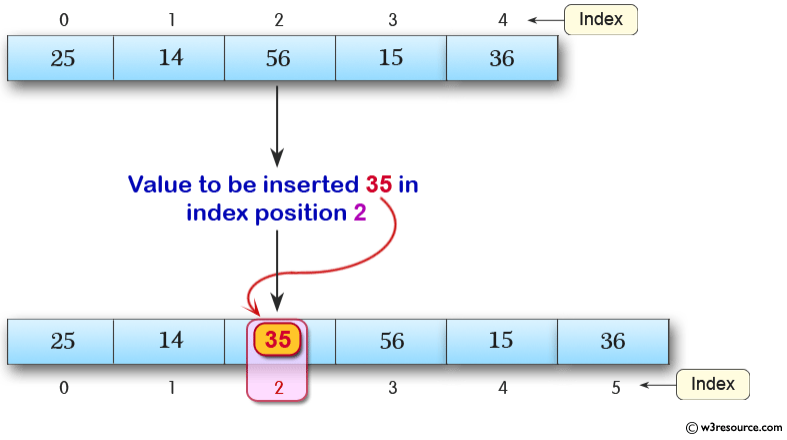
Collection

- A collection is a grouping of data items that have some sort of relation or shared significance.
- Array is one data structure that can be used for a collection
- An array has the benefit of fast lookups (If you know where to look!)

Array Weakness

- Suppose we want to insert into the middle of an array?
- What happens?

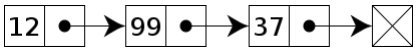
Insert New Value into Array

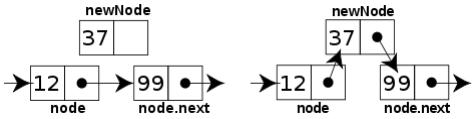


https://www.w3resource.com/java-exercises/array/java-array-exercise-9.php

Alternative

- Linked List
- Every node References the Next node





https://en.wikipedia.org/wiki/Linked_list

Linked Lists

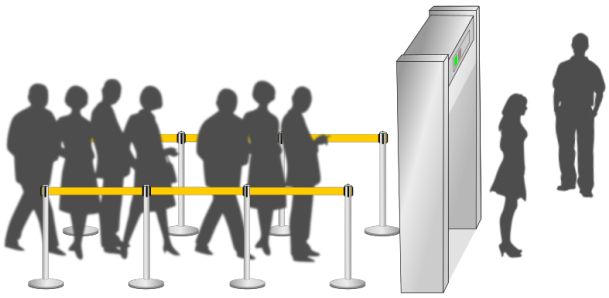
- Can be used for a collection
- Quicker than array to add data anywhere
- Slow to lookup as you need to search all items.
- Good for applications that get a lot of new data, but look data up infrequently
- Or applications that only look at the front or back of the list.

Array vs. Linked List

- Array: Good for fast random access lookups
- Array: Slow for adding new data to keep it sorted
- Linked List: Fast for adding new data anywhere.
- Linked List: Slow'ish for finding data

Queues

- First in, First out data structure



Queues

- First item added to a queue is the first item removed.
- Can be built from an array or a linked list

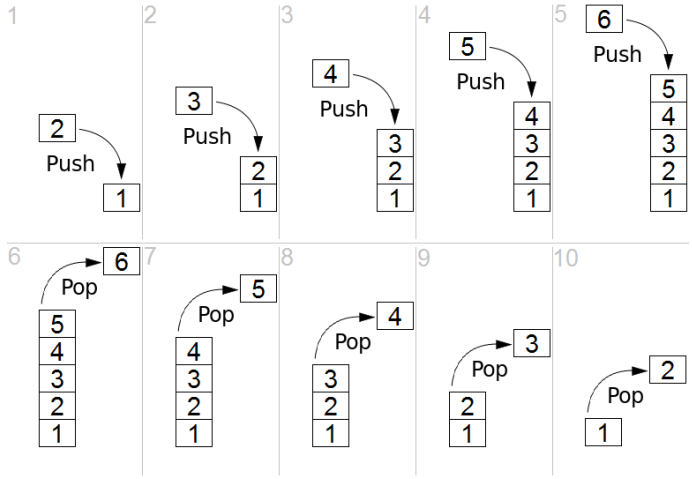
Queues

- Where are queues used?
- Imagine a web-site that services files to thousands of users. You can't service them all at once, so you serve first come first serve!
- Thanks stackoverflow.com!
- Used heavily by operating systems
  - Devices queues, process queues,

Stacks

- Last in, first out data structure
- Can also be built with array or linked list
- Where used?
  - Undo in word! (or anything)
- Extensive operating system use

Stack



https://en.wikipedia.org/wiki/Stack_(abstract_data_type)

Summary so far

- Bits
- Bytes
- Ints/Longs/Shorts
  - These are all primitive types
- Arrays
- Linked List
- Queues
- Stacks

Abstract Data Types

- We know what a data type can do
- How it is done is hidden for the user
- Put another way
  - users are not concerned with how the task is done but rather with what it can do.
- Stack and Queue are examples of abstract data types

Queue

- Enqueue
- Dequeue
- Remove

Stack

- Push
- Pop
- Empty

Information Hiding

- Information Hiding
  - Only expose what is necessary to use the class/object
  - Instance variables should all be private
  - You should use ‘getter’ and ‘setter’ methods to manipulate these values.
  - This is important because:
    - You can validate the values before storing them.
    - Your methods that use the instance data can be sure they are correct.
    - Remember, the author of a method is responsible for it not ‘crashing’.

Running Time Analysis

- Algorithm Running Analysis

Running time analysis

- Operational Ceiling
  - The ceiling or limit at which the application can operate before failure.
- Worst Case
  - The maximum number of operations
- Best Case
  - Is the best we can hope to do!
  - The minimum number of operations required
- Average Case
  - Average amount of operations required.

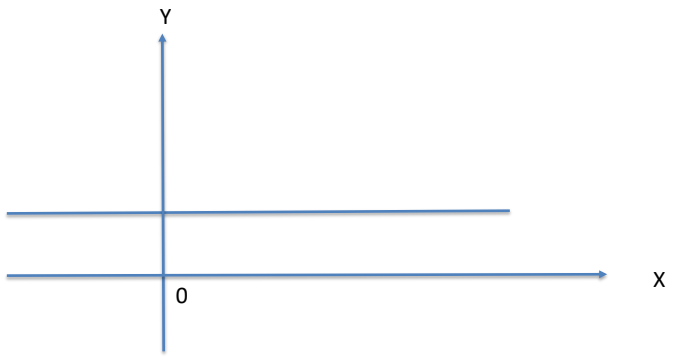
Running time analysis (why?)

- Does the algorithm work fast enough for my needs?
- How much might changing the size of input affect my performance?
- Is one algorithm better than another?

Big-Oh notation (constant)

- **Constant time**
  - **O(1)**
  - Operation takes the same amount of time no matter what n is.
- For example:
  - Reference a location in an array.
  - A = i[20]
  - A = i[35];
  - Takes as much time to reference the spot in the array, no matter how big the array i[] is.

Constant Time



Flat Line!
0 Slope!

Big-Oh Notation (logarithmic)

- Remember "Logarithms"
- *$y=b^x$ means $\log_b(y) = x$*

  Examples for $\log_{10}$:
  - $\log_{10} 10 = 1$
  - $\log_{10} 100 = 2$
  - $\log_{10} 1000 = 3$
  - $\log_{10} 10000 = 4$

Big-Oh Notation (logarithmic)

- Logarithmic Time
- O(log n)
- If the 'n' you are changing falls into bands, it is likely logarithmic time. For example:
  - Your n changes from 10 to 15 and you still need 1 operation but when your 'n' changes from 15 to 200 you need 2 operations.
- Example: Calculate Number of digits in a number
  - 913 3
  - 233 3
  - 32 2

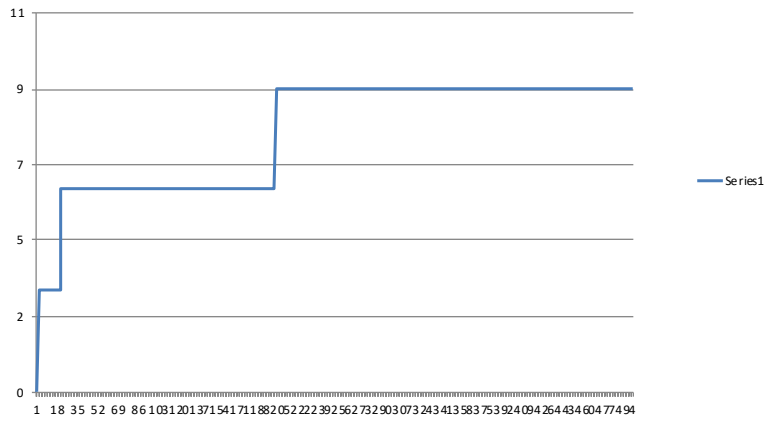
How many times can you divide and have the number still be greater than 1

```
public class Divide {
    public static void divideIt(int number)  {
        int temp = 0;
       timer time=startTime();

        while (number > 1)          // 1 - compare
          {
            number = number / 10; // 1 assignment, 1 divide
          }

         timer time2=EndTime();

    }

     System.out.print(number+","+EndTime-StartTime+",");

}
```

How many times can you divide and have the number still be greater than 1

```
public class Divide {
    public static void divideIt(int number)  {
        int temp = 0;
       timer time=startTime();

          something_that_I_cant_even_see(number);

         timer time2=EndTime();

    }

     System.out.print(number+","+EndTime-StartTime+",");

}
```

Plotted Output



Big-Oh Notation (Linear)

- Linear Time
- O(n)
- Requirements:
  - Each time 'n' changes the operations count changes.
  - When plotted the operations count follows a straight line.

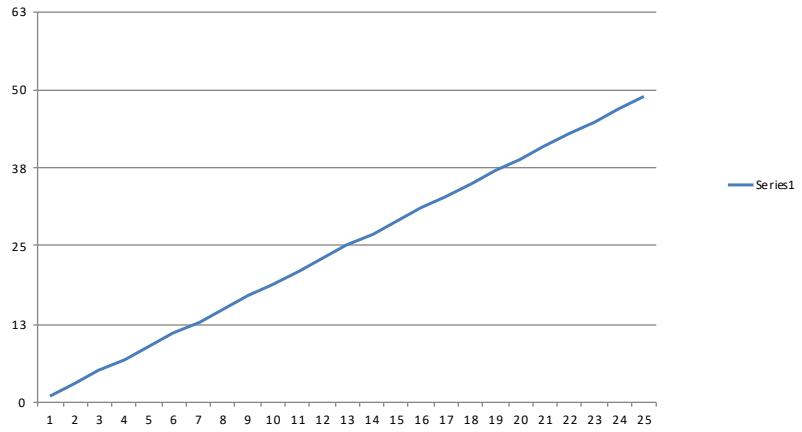
Linear example:

```
public static double doSomethingLinear(double myN)
 {
       timer time=startTime();

      // Generally one loop is linear
       for (i=0;i<myN;i++)
       {
        retVal = retVal * myN;
       }

       timer time2=EndTime();

       System.out.print(myN+","+EndTime-StartTime+",");
       return retVal;
 }
```

Sample Data

- 0,1
- 1,3
- 2,5
- 3,7
- 4,9
- 5,11
- 6,13
- 7,15
- 8,17
- 9,19
- 10,21
- 11,23
- 12,25
- 13,27
- 14,29
- 15,31
- 16,33
- 17,35
- 18,37
- 19,39
- 20,41

Picture



Line with some slope

Big-Oh notation (Quadratic)

- Quadratic Time
- $O(n^2)$
- Requirements:
  - Each time 'n' changes the operations count changes…BUT
  - When plotted the operations count DOES NOT follow a straight line, instead it shoots up rapidly.

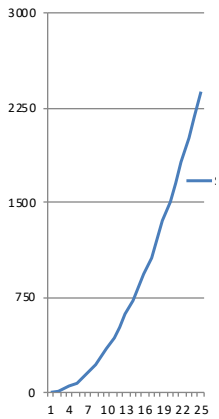
Quadratic Example

```
public static long profound_thing_do (long count)
{
 int i,j; long retVal = 0;  int temp=0;

    time1 = startTime();

     // generally 2 loops are quadratic
    for (int i=0;i<count;i++)
      for(j=0;j<count;j++)
           do_something_profound(count);

   time2 = EndTime()
   System.out.println(myN+","+time2-time1);
   return retVal;
}
```

Possible Data

- 0,1
- 1,8
- 2,23
- 3,46
- 4,77
- 5,116
- 6,163
- 7,218
- 8,281
- 9,352
- 10,431
- 11,518
- 12,613
- 13,716
- 14,827
- 15,946
- 16,1073
- 17,1208
- 18,1351
- 19,1502
- 20,1661

Graph, when plotted Quadratic



Remember — Running Time Analysis

- Don't try to count instructions
- Put 'counters' inside your loops
- Change 'n'
- Run
- Plot with excel

REMEMBER:Running time analysis

- Operational Ceiling
  - The ceiling or limit at which the application can operate before failure.
- Worst Case
  - The maximum number of operations
- Best Case
  - Is the best we can hope to do!
  - The minimum number of operations required
- Average Case
  - Average amount of operations required.

End of Part #1

- Data types
- Arrays
- Linked Lists
- Queues
- Stacks
- Algorithm Complexity
- Half a semester in 65 slides

Searching

- Searching is one of the most important operations performed
- Having elements in order can be a big help.
- Linear search
- Binary search
- Hash table

Linear Search/ Serial Search

- Serial Search or Linear Search
  - Linear time
- Searches for desired element starting at first element.
- Goes one element at a time until it is found.
- Benefits
  - List does not have to be ordered
  - Easy to write
- Drawback: O(n) time lower bound

Linear Searching – for example

- For (i=0;i<n;i++)
  - (If data[i] == value_I_am_looking_for)
    - break;
- Easy to see 'Lower Bounds' would be linear
  - **O(n)**
- Best case really fast!
- Average case better than worst case..

Binary Search

- Divide and conquer search
- **Lower bounds, if done properly, is big-oh**
  - **O($\log_n$) Logarithmic!!!**
- Worst case?
  - Element not found
    - Still faster than linear search
- Best case
  - Element found in center of list

Binary Search Pseudocode

Pseudocode:

If (range contains only one element):

    Look for desired value

Else

    Get midpoint of range

    Is this my element?

    Determine which half of range contains the value

    'binary search' that half of the range

Binary Search Example

- Find 37
- Start: 9,11,13, 14, 27, 37, 39,40, 41
  - *9 elements*

Possible Exam Question

- Find 9
- Start: 9,11,13, 14, 27, 37, 39,40, 41
- Assume middle = first + size/2
- What is the first element searched?
- What is the second element searched?
  - Start: *9,11,13, 14*, 27, 37, 39,40, 41
- Write the order in which each element is examined.

Hashing Slides (Some from)

- www.mis.nsysu.edu.tw/~syhwang/Courses/IR/**Hashing.ppt**
- San-Yih Hwang a former student of the U of M.

Hashing

- Hashing is an efficient information retrieval strategy providing access to information based on a key
- Time complexity, performance, the goal is "avoiding collisions"
  - (A collision occurs when two or more keys map to the same location)
- Space complexity, compactness.
  - Basically we can avoid collisions, but waste a lot of space.

Example: Library Card Catalogs

Sample Catalog Record
Author: Kesey, Ken.
Title: One flew over the cuckoo's nest, a novel.
Published: New York, Viking Press [1962]
LC Call No.: PZ4.K42On
Subjects: Psychiatric hospital patients--United States--fiction
Control No.: 62008602

https://en.wikipedia.org/wiki/Library_catalog

Hashing

- Information is stored in a hash table
- Information is accessed using a key
- hash table: an array of locations which can hold data record
  - Sometimes called ***buckets or slots***.
- A hash function maps between a key and a bucket.
  - I.e., it given a key, it tells which bucket to look in for the data

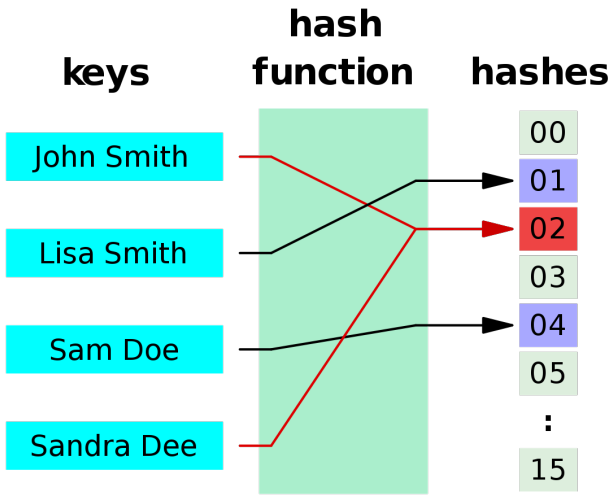
Hashing function

- A **perfect hash function** would distribute data across the buckets such that *no collisions ever occurred*
- Perfect hash functions are hard to find.
- Sometimes a hash function returns the same bucket for two different keys. This is a collision .

Hash Tables

- **Open addressing**
  - Each slot contains a key-value pair (both!)
  - Key-value, because we likely don't have a perfect hash function.
  - If a collision occurs the algorithm calculates another location.
    - perhaps next location

Open address hashing



Review Hash Tables

- Can be O(constant)
- We use a **hash function** to map a 'key' to an integer index. <-- important/remember
- We use generated index to store and lookup data in an array of objects.
- A **perfect hash function** maps elements to distinct integers, with no collisions!

Summary

- Hash tables associate a collection of records with keys.
- The record location is based on the hash value create from the record's key.
- Collisions cause the next available slot is used.
- Searching with the key is then quick.
- When deleting a record the location is marked as deleted so searches skips that spot.

Sorting

Sorting!

- Bubble sort
- Selection sort
- Insertion sort
- Heap sort - nlogn
- Quick sort – n logn
- ~~Merge sort – n logn~~

Bubble Sort

- Compare First Two Numbers
  - Swap if you need to
- Compare Second Two Numbers
  - Swap if you need
- Then start over
- Do that process 'n' times, where 'n' is the length of the list.

Sorting

- Bubble sort

```
for (i=n-1; i>=0; i--)
  for (j=0; j<i; j++)
     if (data[j] > data[j+1])
          swap(data[j], data[j+1])
```

- Time complexity $O(n^2)$

Insertion Sort

- Make a new empty, sorted list
- Take each number from the unsorted list
- Put it into the correct position on the sorted listTime complexity $O(n^2)$
- https://en.wikipedia.org/wiki/Insertion_sort

Example

- 64 25 12 22 11

- 64
- 25, 64
- 12,25,64,
- 12,22,25,64
- 11,12,22,25,64

Selection Sort

- Loop
- Find the minimum element
- Put it at the beginning (or the new beginning!)
- Time complexity $O(n^2)$

```
64 25 12 22 11
11 64 25 12 22
11 12 64 25 22
11 12 22 64 25
11 12 22 25 64
```

https://en.wikipedia.org/wiki/Insertion_sort

Selection Sort Example

Sorted sublist	Unsorted sublist	Least element in unsorted list
( )	(11, 25, 12, 22, 64)	11
(11)	(25, 12, 22, 64)	12
(11, 12)	(25, 22, 64)	22
(11, 12, 22)	(25, 64)	25
(11, 12, 22, 25)	(64)	64
(11, 12, 22, 25, 64)	( )	

https://en.wikipedia.org/wiki/Selection_sort

Selection Sort, Example #2 (Swap, not move)

```
64 25 12 22 11
11 25 12 22 64
11 12 25 22 64
11 12 22 25 64
```

Quicksort

- Select a 'pivot' element of the list
  - The ‘magnitudes’ of the elements relative to the pivot
  - Elements less than the pivot are moved left
  - Elements greater than or equal to the pivot are moved to the right
  - At this point all we have to do is sort the left sublist and the right sublist ...
  - Quicksort left, quicksort right!
  - Time complexity $O(nlog_n)$ – normally

Quicksort

- 4,7,9,1,3,2,8
- Pivot: 4 (index is 3)
- 1,3,2,4,7,9,8 <- reorder such that everything less than 4 is left
- Quicksort left <- everything greater is right
  - 1,3,2
  - Pivot 1 (index is 0)
- Quicksort right
  - 7,9,8
  - Pivot 7

So why would you even consider?

- Why consider bubble, selection, or insertion sorts?
  - Very quick to write
  - Stable, no worry about stack size
  - Can sort files on a disk

Bogo Sort – From Wiki

- Based on the generate and test paradigm. It is not useful for sorting, but may be used for educational purposes, to contrast it with other more realistic algorithms
- If bogosort were used to sort a deck of cards, it would consist of checking if the deck were in order, and if it were not, throwing the deck into the air, picking the cards up at random, and repeating the process until the deck is sorted.
- Its name comes from the word *bogus*.

Trees

Problem with lists

- Accessing a item from a linked list takes linear time
- Accessing a item from an array list takes constant time
- Binary trees can improve upon this and take logarithmic time for the average case
  - Worst case degenerates to linear

Tree Vocabulary

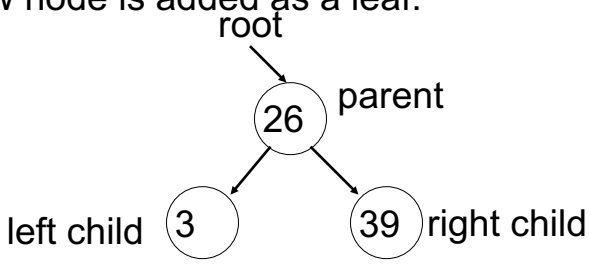
- Remember these…
  - Preorder is root, left, right
  - Postorder is left, right, root
  - In order is left, root, right

Trees

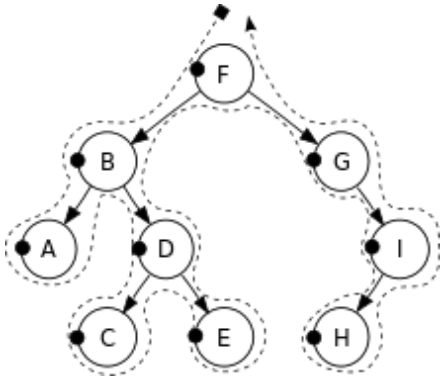
- Binary Tree – each node has at most 2 leaves
  - Not necessarily ordered
- Binary Search Tree – each not has at most 2 leaves and is ordered!

Binary Search Trees

- A binary search tree is a binary tree in which **every node's** left subtree holds values less than the node's value
- Every right subtree holds values greater than the node's value.
- A new node is added as a leaf.

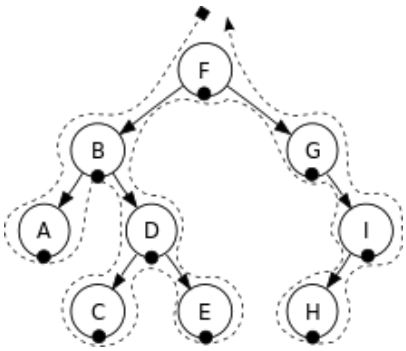


Tree-Traversal Pre-Order



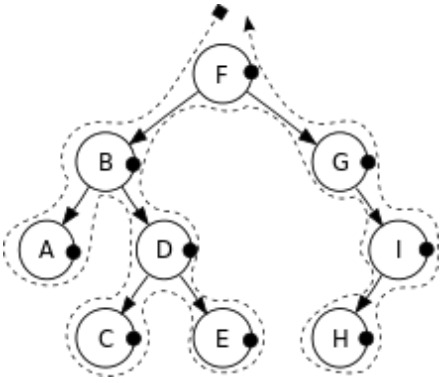
https://en.wikipedia.org/wiki/Tree_traversal

Tree-Traversal In-Order



https://en.wikipedia.org/wiki/Tree_traversal

Tree-Traversal Post Order



https://en.wikipedia.org/wiki/Tree_traversal

Remember. What is the difference?

- Binary Tree
- Binary Search Tree

What would this tree look like?

If we added these values to a binary search tree..

71 22 19 8 10 28 -2

What is the resulting tree?

Tree's

- What is good about pre-order?
  - We can print out a binary search tree and read it back in to build exactly the same binary tree.
- What is good about in-order?
  - The binary ***search*** tree is printed ***in-order***?
  - Can't we make a sort with this? Sure!

Heaps!

Heaps

- A heap is a binary tree in which the elements can be compared with each other using *total order semantics*.
- A **total order semantic means** all the elements of a class can be placed in a single line, proceeding from smaller to larger along the line.
- A binary search tree needs this too

Total Order Semantic

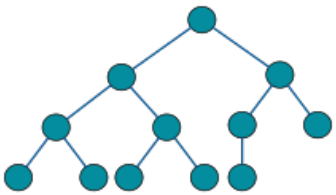
- Equality: (x==y) if they are identical
- Totality: Only one of (x<y) or (x==y) or (x>y)
- Consistency:
  - (x > y) means (y < x)
  - (x!=y) means !(x==y)
  - Blah blah blah
- Transitivity
  - If (x < y) and (y < z) then (x<z)

Heaps – 2 Rules

- The element contained by each node is greater than or equal to the elements of that node's children.
- The tree is a complete binary tree so that every level except the deepest contains as many nodes as possible; at the deepest level, all the nodes are as far left as possible.

Complete Binary Tree

- Every level except the deepest must have as many nodes as possible.
- Deepest level, all nodes are as far left as possible

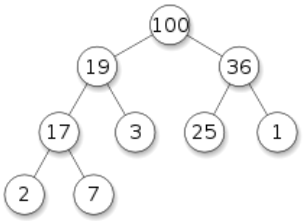


https://en.wikipedia.org/wiki/Binary_tree

98

Heaps

- Objects are organized such that each value is at least as large as its children.
- (Or at least as small depending on how you look at it)
- Heaps are complete trees

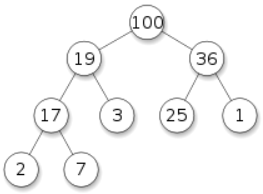


Max Heap Example

https://en.wikipedia.org/wiki/Heap_(data_structure)

Heap Demo

- Heap Demo
- https://youtu.be/WCm3TqScBM8



Heap Practice

Where are heaps used?

- Not for searching data
- Priority queues
  - Priority queue = a queue where all elements have a “priority” associated with them
  - Remove in a priority queue removes the element with the smallest ‘priority’

Heap Sorting

- Put items in a heap
- Take items out of a heap

Heaps

- Perhaps reduce the number of threads in thread heavy applications.
- You can implement a heap-based priority queue and a timer

The End

Designing Uncomplicated Software

Article · October 2018

DOI: 10.26439/interfases2018.n011.2954

CITATIONS
0

READS
31

2 authors:

Michael Dorin
University of St. Thomas
9 PUBLICATIONS **5** CITATIONS
SEE PROFILE

Sergio Montenegro
University Wuerzburg
120 PUBLICATIONS **437** CITATIONS
SEE PROFILE

Recibido: 15 de agosto del 2018
Aceptado: 16 de octubre del 2018
doi: 10.26439/interfases2018.n011.2954

Designing uncomplicated software

Michael Dorin
mike.dorin@stthomas.edu
University of St. Thomas. Minnesota, EE. UU.

Sergio Montenegro
sergio.montenegro@uni-wuerzburg.de
Universität Würzburg. Würzburg, Germany

Abstract

The Agile Manifesto prescribes less focus on tools and processes, and more focus on human interactions. This is a very important and powerful concept; however, many development organizations have interpreted it in terms of no procedures and no processes. This is understandable as many activities, such as the design workflow, are thankless and laborious. When a proper design is missing, the resulting source code may become overly complicated and difficult to maintain. The software design does not have to be arduous as this workflow can be done without pain through an adaptation called Responsibility-Driven Design. This adaptation assigns personalities to the internal components of the software to humanize the operation. The new design workflow is completely compatible with agile concepts such as customer interaction, and produces a credible candidate architecture ultimately resulting in the creation of a less complicated software.

Keywords: ***Agile Manifesto, software design, human interactions, Responsibility-Driven Design***

Resumen

Diseño de *software* no complicado

El Manifiesto Ágil prescribe disminuir el foco en las herramientas y procesos para centrarlo en las interacciones humanas. Este es un concepto muy importante y potente; sin embargo, muchos equipos de desarrollo lo han traducido en términos de no procedimientos y no procesos. Esto es comprensible ya que muchas actividades, entre ellas el flujo de trabajo del diseño, son ingratas y laboriosas. Cuando no se realiza un diseño apropiado, el resultado puede ser un código demasiado complejo y difícil de mantener. El diseño de un *software* no tiene que ser arduo y el flujo de trabajo puede aliviarse con una adaptación denominada "diseño conducido por la responsabilidad" (Responsibility-Driven Design). Esta adaptación asigna personalidades a los componentes internos del *software* para humanizar la tarea. El nuevo flujo de diseño es completamente compatible con los conceptos de agilidad, como la interacción con el cliente, y produce una arquitectura candidata con credibilidad que resultará en la creación de un *software* no complicado.

Palabras clave: ***Manifiesto Ágil, diseño de*** **software**, ***interacciones humanas, diseño conducido por la responsabilidad***

1. Introduction

For whatever reason, designing a software has not been as glamorous as simply writing it. In the days when design relied heavily on flowcharts and data flow diagrams, programmers would complain about management requiring those steps. Some organizations believe architecture design is too expensive and time consuming. Another contributing factor to this, at least at the beginning of a project, is that the software problem to be solved is not well understood (Foote & Yoder, 1997).

Immediately writing a code is seen as a way for engineers to begin understanding the domain with the thought of writing the "real code" later, which more often than not will never happen. In the eyes of the customer and management, the code is working, and the team is demonstrating progress.

At this point, piecemeal growth of the software begins and development starts to grow in an uncontrolled fashion (Foote & Yoder, 1997). Put another way, rather than a design and architecture structuring the code, the code defines the design and architecture. This results in an overly complicated code base which is hard to expand and maintain (Foote & Yoder, 1997).

Many alternatives to this approach have been invented such as the Responsibility-Driven Design. As stated by Rebecca Wirfs-Brock, "Responsibility-Driven Design is a way to design that emphasizes behavioral modeling using objects, responsibilities, and collaborations. In a responsibility-based model, objects play specific roles and occupy well-known positions in the application architecture." (Wirfs-Brock & Wilkerson, 1989). This concept of Responsibility-Driven Design is beneficial for analyzing and designing workflows of software engineering.

If Responsibility-Driven Design is handled at an even more basic level than as described by Wirfs-Brock, a much unencumbered design workflow is produced which nicely partitions the modules of a system.

The goal of the new workflow is to create a better, less complicated software from a human perspective. As part of this ongoing research, students performed analyses and designs using these techniques, the results of which are provided herein.

2. Background

2.1 Complicated code

When discussing a complicated code, it is important to agree on what the term "complicated code" means. Complicated code is a code which is hard to understand and explain by a human who is reading it. Problems with a complicated code are well known and well documented (Banker, Datar, Kemerer, & Zweig, 1993).

The first step in preventing a complicated code is to reacquaint with what makes software complicated. The paper "Coding for Inspections" described a survey carried out to identify the most basic problems seen by software engineers when reviewing a code. Concern for software complexity is not new as McCabe and Halstead designed complexity measurement metrics more than forty years ago (Dorin, 2018). Though a Google search identifies dozens of newer metrics, Halstead and MCabe are readily available without cost. Software is considered undesirable to review when it has higher cyclomatic complexity and higher Halstead difficulty (Dorin, 2018).

Another aspect identified is how stylistic issues also made source code more complicated to review. Table 1 lists the stylistic violations identified in "Coding for Inspections and Reviews" which most bothered reviewers (Dorin, 2018).

Table 1. *Most unpleasant to review styles*

Style Name
There should be space around operators
Do not write over 120 columns per line
Average length of functions
Indent blocks inside of a function
Put matching braces in same column
Use less than 5 parameters in function
Do not use the question keyword
Avoid deeply nested blocks
Use braces for even one statement

2.2 Irreducible complexity

With the thought of avoiding a complicated code in mind, one might think about irreducible complexity. Irreducible complexity is not a universally accepted concept in the biological sciences. Michael Behe defines irreducible complexity as a single system composed of several well-matched, interacting parts that contribute to the basic function, wherein the removal of any one of the parts causes the system to effectively cease functioning. Put simply, if we take a piece away, the system no longer performs as it was intended to (Behe, 2009). Supporters of intelligent design believe this shows that evolution cannot be completely responsible for life on this planet; there had to be an intelligent creator involved.

The merits of biological intelligent design will not be debated here, but one cannot avoid noticing the parallels between computer/software evolution and biological evolution. A code is said to evolve, but a code cannot evolve without the hand of the creator. Some have suggested a code "rots" if left alone long enough. In practice though we can recognize it is not the code that is rotting, but the environment that it was designed to run on is changing. In computer software, there is no way not to recognize the hand of an arguably intelligent creator. An accounts receivable program may one day evolve into a full accounting system program, but it will not do so by mutation.

As a software engineer, this concept should be kept in mind during analysis and design. If a design is overly complicated, the software engineer should work to eliminate extra complexity, with the final target being reduced until the program can be reduced no more without destroying product functionality. Extraneous parts that do not contribute to the program's functionality should be removed.

3. Components of the new workflow

3.1 Dress rehearsal

Military organizations around the world have used rehearsals for centuries. It is said that the Romans rehearsed battles using sand tables with icons to visualize the battlefield (Smith, 2010). The modern army believes the rehearsal is a tool for commanders to make sure parties involved understand the intent and concept of the operations. Rehearsals provide opportunity to identify inadequacies in a plan that were not previously recognizable. Rehearsals contribute to external and internal coordination. (Army, 2015) In other words, rehearsals of all shapes and sizes are used to ensure efficient battlefield operations. Dress rehearsals can be entire battlefield simulations with whole army units participating, or they can be small, where individuals take on the role of entire units. Events are simulated in real-time and participants act out their responsibility at different points of the exercise (Army, 2015). Obviously, the military is not alone in using rehearsals and this is a powerful tool that can be well used in software engineering.

3.2 Play writing

It may be considered odd that information on writing a play would be included in a discussion of software engineering life cycle models. However, when considering employing rehearsals as tool, using a play as a structure should not be overlooked. In his book, "Writing Your First Play", Roger Hall outlines elements of a play (Hall, 2012). Section 1 covers action and how dynamic action employs verbs. In software engineering, verbs can be used to represent methods or functions in your code.

Section 2 discusses obstacles and conflict, such as the conflict faced by stakeholders who do not have the required software.

Though this technique can work with any architecture design, the Model-View-Controller design pattern (MVC) works very well for this approach. MVC defines a plan for organizing components.

The model portion handles data storing and the algorithms for processing data. The view portion is responsible displaying information and results to the user. The controller is in charge and sends commands to the model and the view (Rosenbloom, 2018).

There are many resources describing how to write a successful play; however, applying artistic information to software design is not always obvious. In playwriting, it is important to come up with a main character, then decide on the conflict or problem (Victor, 2009). Afterwards, it should be decided on a beginning point and show the story in actions and "speech". Don't overdo it: one group of students wrote their play based on Star Wars characters and, upon rereading at a later date, they could not remember the roll of each character.

There is one more suggestion which can be a benefit to the success of a play, especially for new authors. Characters with special skills should be provided or generated before playwriting begins. In the sample play, characters with different skill sets participate in completing the required task.

For example, the "Artista" character is responsible for communication. A "Jefa" character is responsible for the overall operation. Other characters for security, data management, and direct communications are included. See Table 2 for a complete list. The "Profesor" and the "Estudiante" are the only two human characters in the play and they represent the users of the software.

The main character of the play is the "Profesor," though the "Jefa" has an active supporting role. As play writers should give characters a significant problem to solve immediately, in the sample play, the problem to solve is how the professor best communicates with students during class.

3.2.1 Example Play: The happy class

Table 2. *Play characters*

Name	Character Title	Responsibility
Claudia	Oficinista (File clerk)	Stores and retrieves data
Diego	Estudiante (Student)	A student using the system (Human)
Gonzalo	Guachimán (Security)	Provides user validation
Patricio	Profesor (Professor)	A professor using the system (Human)
Rebecca	Jefa (Boss)	Manages software operations
Sergio	Telefonista (Telephone operator)	Provides internal communications
Valeria	Artista (Artist)	Generates output to users

Setting

Three software people (Valeria, Rebecca, Patricio) are sitting around piles of paper showing user stories and use cases. A low-resolution prototype is taped to the wall. Cups are full of coffee. The three are very pleased with the quantity and quality of their requirements gathering analysis but harbor some doubts with respect to moving to the next phase.

Narration

Valeria: This is sure good coffee. Do you think we have enough?

Rebecca: I hope not, I want to wrap up and go home… but now what?

Patricio: Now we have to come up with a candidate architecture, but where do we begin? We have gathered so much information and talked to so many people. We even have fantastic low-resolution user interface prototypes.

Valeria: Perhaps we start small. Rebecca, please grab me a minor use case.

Valeria: Ok, I have the "*Profesor* Logs In" use case, but it is still not obvious how to continue.

Rebecca: I know. Let us pretend to be the software. Perhaps we can get an idea of how to construct this thing!

Patricio: Don't be silly.

Rebecca: Wait, wait, let us just give it a try and see where it takes us.

Patricio: Ok, I will pretend to be the *Profesor* in the use case. Rebecca, you'll be the Software.

Patricio: I'll start.

Patricio: Hola, Rebecca. I want to set up a class.

Rebecca: I am not sure what you want or how to help. Valeria, can you show him what he can do?

Valeria: Ok, I will pretend to be in charge of showing stuff.

Valeria: Ok, here are your options. (Valeria shows Patricio a sheet of paper. Patricio pretends his options are written on it.)

Patricio: I think this is getting closer, but rather than calling you by name, I am going to call you by a title to help this stay organized. I will be the Profesor. Rebecca, you seem to be the boss so I will call you "*Jefa*." Valeria, you seem to be a communicator, so I will call you "*Artista*".

Profesor: Hola, Jefa. I want to set up a class.

Jefa: Artista, please show this *Profesor* his options.

Artista: Profesor, welcome, here is our main screen. Professors need to sign in.

Jefa: STOP! I don't know how people authenticate. I just know how to boss people around. We need somebody like a '*guachimán*' to handle this. (Just then they notice Gonzalo is sitting in the corner.)

Patricio: Hey, Gonzalo, come here for a second. We need you to be a *guachimán* in our software world.

Gonzalo: Hey, a *guachimán*? Wow! That sounds like fun!

Patricio: Ok, let us continue again. Remember: Rebecca is the Jefa.

Profesor: Hey, *Jefa*, I want to set up a class.

Jefa: *Artista*, please show this *Profesor* his options.

Artista: *Profesor*, welcome, here is our main screen: Professors need to sign in.

Profesor: *Guachimán*, here is my info.

Guachimán: *Jefa*, the *Profesor* has signed in.

Jefa: *Artista*, please show the *Profesor* how to create a class.

Artista: *Profesor*, please provide a class name and let us know when you are ready to start.

Profesor: *Jefa*, I want to start a class named SEIS610. It is my favorite class and a fantastic professor teaches it.

Rebecca: STOP. I don't know how to create a class. We need somebody to keep track of all of this. Wait, Claudia, come here. Can you pretend to be an *Oficinista* for us? Please, just for a bit. Ok, Claudia, when I call out "*Oficinista*", you answer.

Claudia: Do I have to call you "*Jefa*"?

Rebecca: Yes, you do.

Rebecca: Ok, now let us continue

Jefa: *Oficinista*, please create a class named SEIS610.

Oficinista: Ok, *Jefa*, here is that class you wanted.

Rebecca: Stop again. Ok, Claudia has created a class for us but how do I talk to it. Claudia does not know anything about talking to classes. It's like we need a telephone operator. Sergio, come here for a second. We need you to pretend to be a telephone operator.

Sergio: Do I have to?

Rebecca: Yes, you do; and yes, you need to call me "*Jefa*".

Rebecca: Alright, let us continue again.

Jefa: *Telefonista*, can you please give me a new channel?

Telefonista: Yes, *Jefa*, here you go.

Jefa: Oficinista, hey, sorry to bug you again, but can you store this channel information?

Oficinista: *Jefa*, consider it done!

Jefa: *Artista*, can you show the *Profesor* a classroom based on the new class with this ID and channel information?

Artista: *Profesor*, here is your class. Make sure you tell your students the ID is 1234! (Diego walks by… he decides to be funny and pretends to be a student.)

Diego: Hey, look at me. I am an *Estudiante*!

Estudiante: Hola, *Jefa*, I want to participate in class with ID 1234.

Jefa: *Oficinista*, do we have a class with ID 1234? If so, can you give it to me?

Oficinista: Yes, *Jefa*, we do have that class; here is the info.

Jefa: *Artista*, can you display a class to this *Estudiante* with this info?

Artista: *Estudiante*, here is your class.

Estudiante: *Jefa*, can you tell the Profesor that I don't understand the problem just described?

Jefa: *Oficinista*, what is the channel that class 1234 uses?

Oficinista: Class 1234 uses this channel.

Jefa: *Telefonista*, can you relay this question to this channel to the *Profesor*?

Telefonista: Yes, I will, and I have.

Patricio: Well done, gang! I think this gives us a fascinating picture! The end!

3.3 Responsibility-Driven Design

Data flow diagrams remain a very popular form of analyzing a system. Data flow diagrams, as their name implies, are data centric. Responsibility-Driven Design proposes that, instead of thinking about data and algorithms which process data, one should think about objects with responsibilities.

Responsibilities are made up of two basic items: what it knows and what it can do. Basically, objects are bundles of data and operations on that date (Wirfs-Brock & Wilkerson, 1989).

Responsibilities are identified by highlighting the nouns and the verbs in the requirements, such as user stories. Verbs are candidates for actions a class can perform, and nouns are candidates for information that the class should maintain (Wirfs-Brock & Wilkerson, 1989).

In Responsibility-Driven Design, objects have a very specific part of the application. Each object is responsible for doing one portion of the work. Objects do only one job, and they must do that one job well. Objects then communicate with each other to fulfill the larger goals of the application.

Figure 1. Example sequence diagram

3.4 Unified Modeling Language (UML)

The Unified Modeling Language (UML) defines a standard set of diagrams used in designing software (Larman, Kruchten, & Bittner, 2001). UML sequence diagrams visually describe the actions of objects in a time sequence. An example of a sequence diagram is shown in Figure 1.

In the Unified Modeling Language, class diagrams show the relations and dependencies among classes. Class diagrams are used to show the overall architecture of a design. An example of a class diagram is shown in Figure 2.

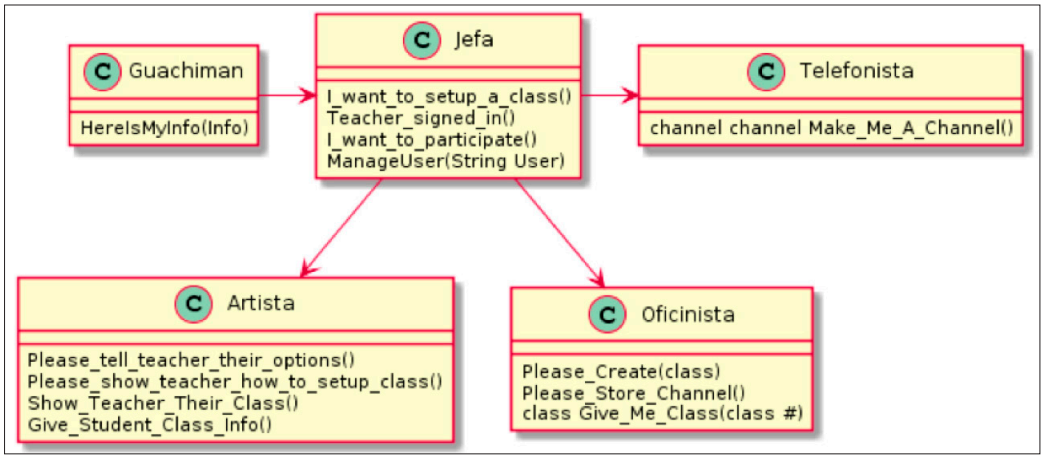


Figure 2. Example class diagram

4. Proposed analysis and design workflow

In this section, a simplified form of Responsibility-Driven Design is shown as a way for engineers to relate to software modules. For example, as a software engineer, you should try to describe how your software will work from the perspective of people doing the work, with the caveat that each person may do only one thing and must do it well. Put another way: write a play. A main character must be chosen, and conflict is needed (Dorf, 2018). In the world of software engineering, the main character is likely the human the software is being written for, and the conflict is what the main character is missing when the software does not exist. Developers must decide what roles are needed to resolve this conflict and the characters are given a significant problem to solve immediately (Dorf, 2018). Human personalities are given to the software modules. At this point, a beginning point is determined, and the engineering analysis comes from the story via actions and speech. Responsibility-Driven Design coupled with plays produces an analysis that is easily communicated to all stakeholders.

When the play is finished, it is then converted to a UML sequence diagram. Strict UML rules should not be enforced as the goal is to arrive at a candidate architecture for the software system. When the sequence diagram is complete, a UML class diagram can be made. Once again, strict UML rules should not be enforced. At this point, a candidate design which includes identified classes, associations, and method names is ready. An example of a play as well as related sequence and class diagrams are included in the appendices.

5. Results

To determine if this approach has merit, software design projects that were assigned to graduate students at the University of St. Thomas were analyzed. The Software Engineering beginning class has been consistently organized for the past three years. Students were required to form teams of two or three persons and design a major software project. Students were allowed to select the theme of their own projects, but in general students were guided towards projects where the user interface was a prominent part of the application. Some example projects include a WhatsApp-like application which translates texts to the native language of the receiver, classroom management applications, games, and medical-patient management systems.

In the first year studied (2015), the pre-play, students were asked to generate two-column use cases from user stories and then derive a design. Students generally had no trouble with the initial use case, which showed user and system interactions. These described "the user does this," "the system does that" type interaction. However, at this time many students were unable to identify the classes required to build a system. De-constructing the system into smaller objects was for many a frustrating task.

When assessing team progress, it was apparent that generally only one student in the group understood how to undertake this task adequately. This problem was reflected through summative assessment where nearly 50 percent of the students were unable to correctly create multiple two-column use cases, and then perform an analysis to derive required UML diagrams. Also, nearly 25 percent of the students who had created correctly two-column use cases and adequately identified classes were unable to properly suggest functions or methods within those classes. Informally, students also indicated frustration with this approach.

In mid-2016 performance, style plays where introduced as a method of analysis, and it became evident that the level of participation in the group activity rose dramatically. Performance style plays solved a significant problem facing the students the partitioning of the system object.

Students had less trouble identifying classes. Resolution of this difficulty was helped through the suggestion of characters with specialized skills for the play. Students now could envision a collection of specialists performing the tasks required for the system to operate. During the pre-play, it was difficult to envision how to divide up the work of the system. The post-play, which provided suggested characters by assigning tasks, became very practical and was no longer perceived as impossible.

Students also no longer had trouble identifying the methods required of each class, as methods were built upon the dialog between the characters in the play.

All members of the team took part in the creation of the play, and the post-play summative assessment rose to nearly 80 percent success. Thirty-nine final exams from two sections of pre-play classes and 119 final-exams from four sections of post-play classes were reviewed. Though the numbers of reviewed pre-play and post-play exams differed, the success percentages were consistent among classes. Additionally, post-play students who were not wholly successful were also not completely lost. In general, their issues were not severe. For example, "methods" might show up in the wrong class or "methods" might be missing. With a little bit more practice, these students could master this topic.

Table 3. *Successful projects*

Period	Value
Pre-play success	50%
Post-play success	80%

6. Conclusion

In this paper, a new approach to the analysis and design workflows is presented with the goal of avoiding a complicated software. Terminology and characteristics of a complicated software are provided. How to creatively perform the software engineering analysis and design workflows by writing plays inspired by "Responsibility-Driven Design" is shown. Information on creating UML sequence and UML class diagrams is given. Moreover, summative assessment is used as a measure of overall success and failure of the approach. Further research is warranted to analyze the most specific issues students had pre-play and how the performance style play could solve those issues. In addition, formal evaluation of large programming projects should also be done to verify good design quality, and good programming practice is necessary.

References

Army, U. (2015). Fm 6-0 commander and staff organization and operations. Washington.

Banker, R. D., Datar, S. M., Kemerer, C. F., & Zweig, D. (1993, November). Software complexity and maintenance costs. *Commun. ACM, 36*(11), 81–94. DOI: 10.1145/163359.163375

Behe, M. J. (2009). Irreducible complexity: Obstacle to darwinian evolution. *Philosophy of biology: An Anthology*, 32, 427.

Dorf, J. (2018). *Playwriting 101 how to write a play*. Retrieved from http://www.playwriting101.com

Dorin, M. (2018). Coding for inspections and reviews.

Foote, B., & Yoder, J. (1997). Big ball of mud. Pattern languages of program design, 4, 654-692. Hall, R. (2012). Writing your first play. Focal Press.

Larman, C., Kruchten, P., & Bittner, K. (2001). How to fail with the rational unified process: Seven steps to pain and suffering. Valtech Technologies & Rational Software.

Rosenbloom, A. (2018). A simple MVC framework for web development courses. In *Proceedings of the 23rd western canadian conference on computing education* (pp. 13:1-13:3). New York, NY, USA: ACM. DOI:10.1145/3209635.3209637

Smith, R. (2010). The long history of gaming in military training. *Simulation & Gaming, 41*(1),6-19.

Victor, W. (2009). *Creative writing now, how to write a play*. Retrieved from https://www. creative-writing-now.com/how-to-write-a-play.html (Accessed: 2018-08-11)

Wirfs-Brock, R., & Wilkerson, B. (1989). Object-oriented design: A responsibility-driven approach. In *ACM sigplan notices* (Vol. 24, pp. 71-75).

SEIS-610

Diagram Review

1

Diagrams This Semester

- Data Flow Diagrams (DFD)
- Flow Charts/Decision Trees
- Petri Nets
- State Machines
- CRC Cards (I know, not really a diagram)
- Class Diagrams
- Sequence Diagrams

2

Data Flow Diagram

FIGURE 12.3 The data flow diagram for Sally's Software Shop: second refinement.

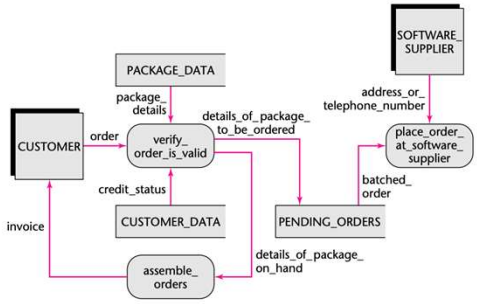
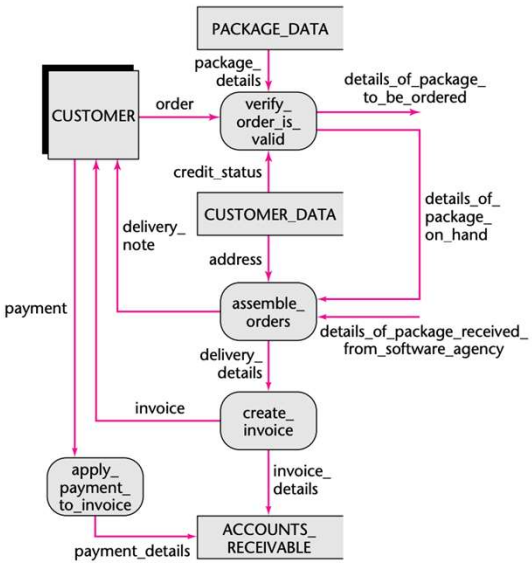
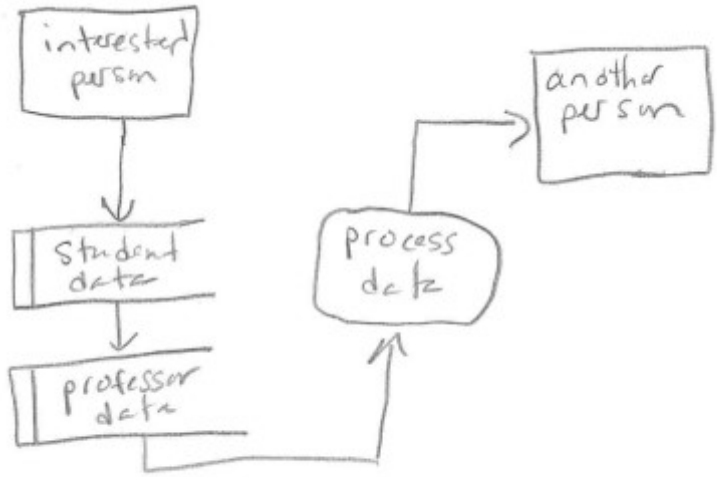


FIGURE 12.4 The data flow diagram for Sally's Software Shop: part of third refinement.



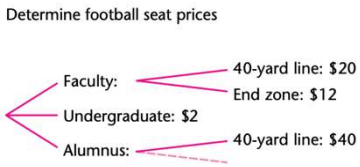
3

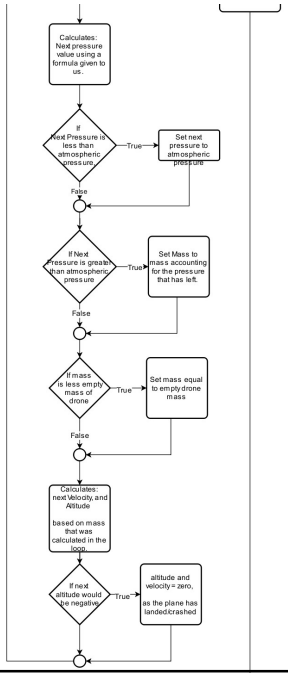


4

Decision Tree and Flow Charts

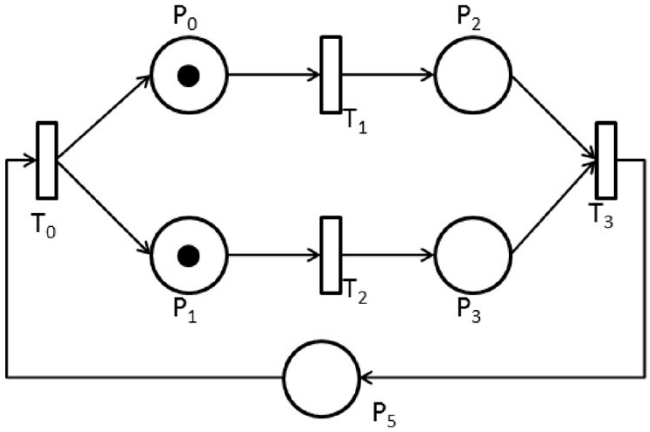
FIGURE 12.7 A decision tree describing seating prices for college football games.





5

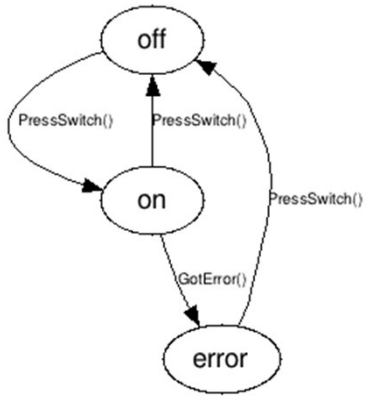
Petri Nets



https://www.intechopen.com/books/petri-nets-manufacturing-and-computer-science/a-petri-net-based-approach-to-the-quantification-of-data-center-dependability

6

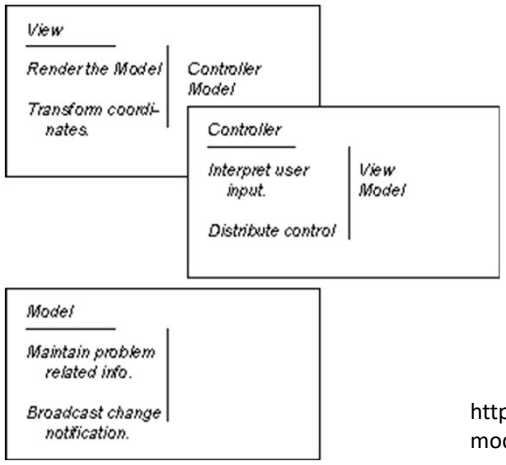
State Machine



https://stackoverflow.com/questions/5923767/simple-state-machine-example-in-c

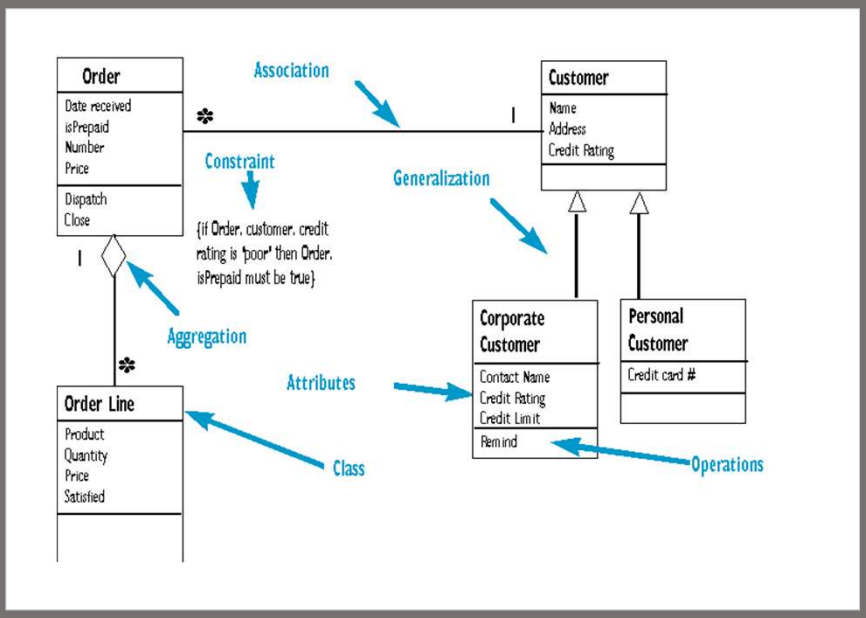
7

CRC Cards



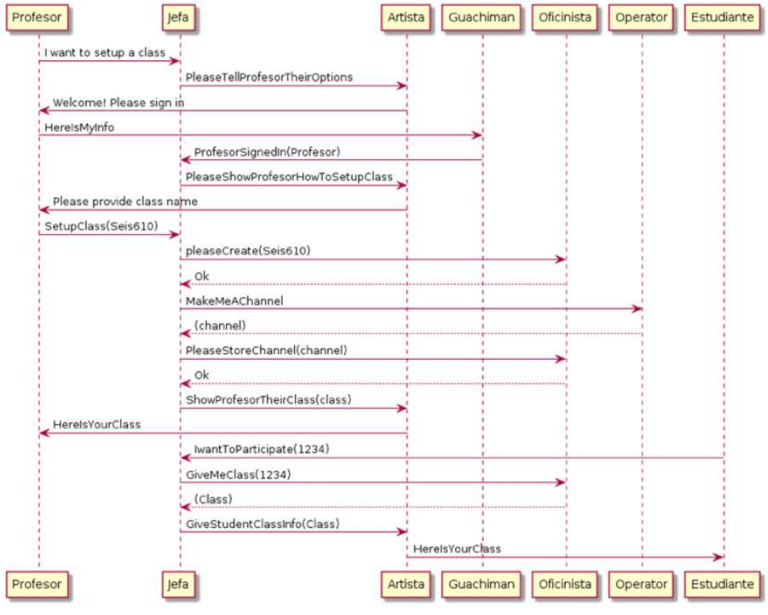
https://stackoverflow.com/questions/29286240/crc-card-for-model-view-controller

8



Class Diagram

9



Sequence Diagrams

10

The End

11

Risk Mitigation **Michael Dorin**

SEIS-601/610 Moon Landing

2/19/2019

a. No experience editing, compiling, or building java programs.

On 2/13/2019, the SEIS-601 class has practiced editing, compiling, and building java programs. Several programs were built that evening. We have a high degree of confidence that this is no longer a risk.

b. No experience outputting data from a java program.

As part of the effort on 2/13/2019, programs requiring input were created and tested.

c. No experience inputting data into a java program.

As part of the effort on 2/13/2019, programs requiring output were created and tested.

d. No experience saving data in persistent storage.

This remains a risk. However, a five minute YouTube video https://youtu.be/BxCbxfpwC7Q seems to show adequate information for persistence for our purposes. We have decided that the risk is mitigated enough to continue.

e. No experience processing data.

During the 2/13/2019 practice exercise, we wrote programs to calculate results of formulas related to free fall. Data was input, calculations were done, and results were output. We are confident in processing data.

f. No physics background.

A bit of research has found equations for a zero angle drop. (See below) We believe this sufficient for our purposes and will not stop the project.

1. velocity = velocity_original + acceleration * time;

2. upward_acceleration = resultant_force (newtons, N) divided by mass (kilograms, kg).

3. To find resultant force:
   resultant_force = thrust – weight
4. To find weight: mass * gravity
5. simplifying upward_acceleration = (thrust - mass * gravity)/mass

   Simplifying again: upward_acceleration = thrust/mass – gravity

6. height=height_original+velocity_original*time+ 1/2* acceration * time^2

The equations are from here:
https://www.sciencelearn.org.nz/resources/397-calculating-rocket-acceleration
here: http://hyperphysics.phy-astr.gsu.edu/hbase/mot.html#mot4

g. No experience with graphics.

This remains a risk, but one that we can live with. Project boundaries already indicates that this will be text only initially. We will do our best to mitigate this risk later, but is not a risk worthy of stopping the project.

SIES610

Rational Unified Process
Supplemental Information

1

Vocabulary

- **Requirements Gathering** discovering the requirements of a system (user facing)
- **Analysis** emphasizes investigation of the problem and solution. (engineering facing)
- **Design** emphasizes a conceptual solution that fulfills requirements
- **Object-Oriented Analysis** there is an emphasis on finding and describing objects
- **Object-Oriented Design** there is an emphasis on defining how objects collaborate to fulfill requirements.

2

More Vocabulary

- Functional Requirement
  - Something the product must do.
- Non-Functional Requirement
  - Something the product will be judged by.
  - Example: The product must be secure.
  - Example: The product must be easy to use.
  - Example: The product will be used in libraries so should not have sound.

3

Agile

- Agile applies to time-boxed, iterative (i.e. *iterations*) and evolutionary development.
- There are several approaches to agile
  - Scrum
  - Extreme programming
  - Etc.
- Unified process is only one of them
- Agile modeling likes UML as a sketch

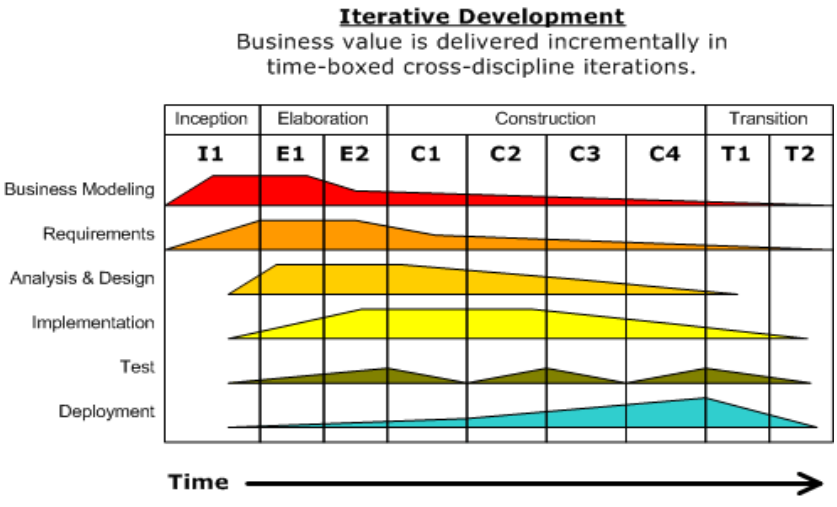
4

Unified Process Phases (UP)

- **Inception** – Vision, business case, requirements, limitations/boundaries, vague estimates
- **Elaboration** – refine vision, emphasis on analysis and design, other workflows can happen
- **Construction** – Iterative implementation, but other workflows still happen
- **Transition** – Passing to customer, emphasis is on the handoff.

5

Iterative Development



6

Inception

The smallest phase in the project. If the Inception Phase is long then it may be an indication of excessive up-front specification, which is contrary to the spirit of the Unified Process. **Plan for 1 pass through inception, unless the project is huge.**

Goals for the Inception phase

- Establish a vision for the project.
- Establish the boundary conditions.
  - What is not going to be done??
- Outline the user stories and key requirements
- Establish a justification for the project.
  - include business risks.
- Non-functional requirements
- Initial Candidate Architecture (Block diagram)
- Identify software engineering risks.
- Prepare a preliminary project schedule and cost estimate.
- Functional requirements described, 10% of use stories analyzed in detail.

7

Inception - FURPS

- **Functional** – Features, capabilities (User Stories)
- **Usability** – Human factors, help, documentation,
- **Reliability** – frequency of failure, recoverability, predictability
- **Performance** – response times, throughput, resource usage.
- **Supportability** – adaptability, maintainability, internationalization, configurability
- **+** - Implementation, Interface, Operations, Packaging, Legal, and so forth.

Red are the Non-functional requirements

8

Inception

- Use cases/User Stories
  - Lots of use case names/features identified
  - Maybe analyze 10% of them
- Non-functional requirements
- Maybe: Rough draft of user interface
  - Hand drawn pictures
- Business Case
  - Wild estimate of cost

9

Sprints

- 'Time boxed' iterations
  - 2 to 6 weeks, but keep closer to 2
  - Keep consistent. If your time box is 2 weeks, keep it 2 weeks.

10

Elaboration

- A healthy majority of the system requirements should be captured.
- Requirements are analyzed
- known risk factors are addressed and the system architecture is validated
- Common processes undertaken in this phase include the creation of architectural diagrams.
- Spend more time on analysis and design and less on requirements
- Perhaps some implementation
- End of the Elaboration phase
  - the system architecture is stabilized
  - If prototypes exist, they demonstrate that the architecture will support the key functionality **and mitigate risk factors.**

11

Elaboration

- Re-Read user stories
- Write a play (analysis)
- Draw sequence diagrams (design)
- Turn sequence diagrams into class diagrams (design)
- Repeat

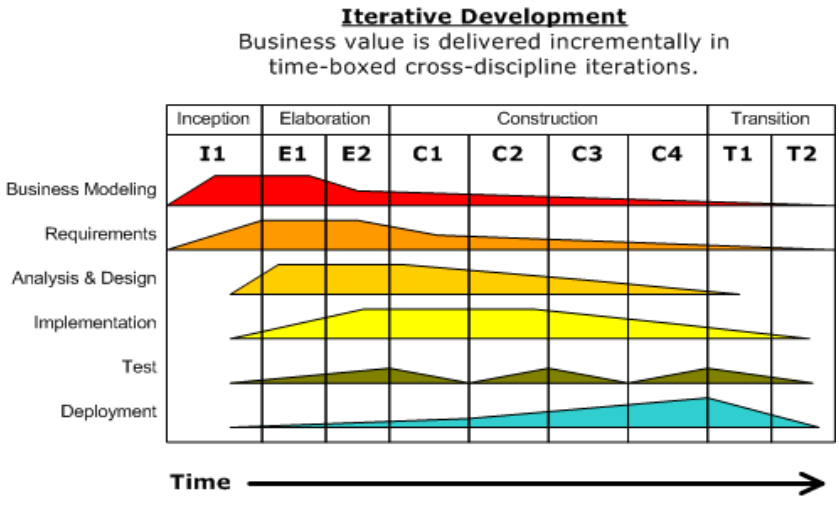
12

Construction Phase

- Construction should be the largest phase in the project. In this phase the remainder of the system is built on the foundation laid in Elaboration.
- System features are implemented in a series of short, timeboxed iterations. (Sprints)
- Each iteration results in an executable release of the software.
- Our UML (Unified Modelling Language) diagrams used during this will be put to good use here

13

Iterative Development



14

Transition Phase

- The final project phase.
- In this phase the system is deployed to the target users.
- Feedback received may result in further refinements
- Transition phase is also built on iterations.
- The Transition phase also includes system conversions and user training.

15

User Stories

- User Stories form the functional requirements
- Michael Cohn User Story Template:
  - From here:
  - https://www.mountaingoatsoftware.com/blog/advantages-of-the-as-a-user-i-want-user-story-template
- "As a <type of user>, I want <some goal> so that <some reason>."
  - [The 'so that' is **NOT** optional.]
- For example:
- As a **moderator** I want to create a new game by entering a game and an optional description so that I can start inviting estimators.

16

Use Cases

- Use cases are more formal than user stories.
  - primarily functional or behavioral requirements that indicate what the system will do.
- You can find and refine requirements by holding workshops, focus groups, demonstrations of code as it progresses.

17

- The End

18

SEIS 610

Chapter 13

1

in: *Z User Group*, *Organizations*, *Z notation*

Z User Group

NEW **Last ABZ conference:**

ABZ 2018: Southampton↑, United Kingdom↑, 5-8 June 2018

Previous ABZ conferences:

ABZ 2016: Linz↑, Austria↑, 23-27 May 2016

ABZ 2014: Toulouse↑, France↑, 2-6 June 2014

ABZ 2012: Pisa↑, Italy↑, 18-22 June 2012

ABZ 2010: Orford, Quebec↑, Canada↑, 23–25 February 2010

First ABZ conference:

Virtual Library

Computing

Software

2

Agenda

- Review Object Oriented-Analysis (Chapter 13)
  - Analysis Workflow (13.1)
  - Extracting Entity Classes (13.2)
    - Noun Extraction (13.5.1)
  - CRC Cards (13.5.2)
    - (Only because they remind me of the play's we are writing)
  - Test Workflow (13.7)
  - Extracting Boundary and Control Classes (13.8)
  - Specification Document (13.18)
  - Actors and Use Cases (13.19)
  - Metrics (13.21)
  - Challenges to Object-Oriented Analysis (13.22)

3

Object Oriented Analysis

- Entity Class – information that is long lived
- Boundary Class – models' interaction between product and its actors
- Control Class – Models complex computations and algorithms

4

Object Oriented Analysis

- Entity Class – information that is long lived
- Boundary Class – models interaction between product and its actors
- Control Class – Models complex computations and algorithms
- MVC
- MVC is an architectural model for the UI based world!
- Entity, Boundary, Control is for business/and other modeling

5

MVC

- Model –Represents the problem domain, maintain state, and provide methods for accessing and mutating the state of the application.
- The Controller's job is to translate incoming requests into outgoing responses.
- View : The View's job is to translate data into a visual rendering for response to the Client

- http://www.bennadel.com/blog/2379-a-better-understanding-of-mvc-model-view-controller-thanks-to-steven-neiland.htm

6

Quick Pause

- Design Patterns
- "In software engineering, a **design pattern** is a general repeatable solution to a commonly occurring problem in software **design**. A **design pattern** isn't a finished **design** that can be transformed directly into code. It is a description or template for how to solve a problem that can be used in many different situations."
- https://sourcemaking.com/design_patterns

7

Object Oriented Analysis

- Entity Class – information that is long lived
- Boundary Class – models interaction between product and its actors
- Control Class – "Controls" complex computations and algorithms
- https://www.youtube.com/watch?v=JWcoiXNoKxk&feature=youtu.be&t=15m14s

(start 15 minutes in!)

8

Extracting Entity Classes (13.2)

- Functional Modeling
- Entity Class Modeling
- Dynamic Modeling

9

Functional Modeling

- A functional modeling perspective concentrates on describing the dynamic process.
- The main concept in this modeling perspective is the process, this could be a function, transformation, activity, action, task etc.
- A well-known example of a modeling language employing this perspective is data flow diagrams.
- It is illustrative to think like this?
  - Think of data and processes identified in the DFD?
  - You can turn that data into classes
- https://en.wikipedia.org/wiki/Function_model
- Another way of looking at this is with use cases

10

1. User A presses the Up floor button at floor 3 to request an elevator. User A wishes to go to floor 7.
2. The Up floor button is turned on.
3. An elevator arrives at floor 3. It contains User B, who has entered the elevator at floor 1 and pressed the elevator button for floor 9.
4. The elevator doors open.
5. The timer starts.
   User A enters the elevator.
6. User A presses the elevator button for floor 7.
7. The elevator button for floor 7 is turned on.
8. The elevator doors close after a timeout.
9. The Up floor button is turned off.
10. The elevator travels to floor 7.
11. The elevator button for floor 7 is turned off.
12. The elevator doors open to allow User A to exit from the elevator.
13. The timer starts.
    User A exits from the elevator.
14. The elevator doors close after a timeout.
15. The elevator proceeds to floor 9 with User B.

11

1. User A presses the Up floor button at floor 3 to request an elevator. User A wishes to go to floor 1.
2. The Up floor button is turned on.
3. An elevator arrives at floor 3. It contains User B, who has entered the elevator at floor 1 and pressed the elevator button for floor 9.
4. The elevator doors open.
5. The timer starts.
   User A enters the elevator.
6. User A presses the elevator button for floor 1.
7. The elevator button for floor 1 is turned on.
8. The elevator doors close after a timeout.
9. The Up floor button is turned off.
10. The elevator travels to floor 9.
11. The elevator button for floor 9 is turned off.
12. The elevator doors open to allow User B to exit from the elevator.
13. The timer starts.
    User B exits from the elevator.
14. The elevator doors close after a timeout.
15. The elevator proceeds to floor 1 with User A.

12

Use Case

- **Use case:** User/System actions that are required to reach or abandon a goal. A set of success and failure scenarios that describe an actor reaching or abandoning their goal.
- **Use case scenario:** A single path through the use case.
- **Use case instance**: A sequence of actions a system performs that yields an observable result of value to a particular actor.
- **Use case model:** All the written use cases that describe a system's functional requirements.
- **Actor:** Role external party/parties that interact with the system. Does not have to be human! Anything with a behavior.

13

Use Case Types

- **brief**—Terse one-paragraph summary, usually of the main success scenario.
- **casual**—Informal paragraph format. Multiple paragraphs that cover various scenarios.
- **fully dressed**—All steps and variations are written in detail, and there are supporting sections, such as preconditions and success guarantees.
- Two column is my favorite variation

14

13.19 Use Case Vocab

- **Primary Actor:** The primary actor is trying to achieve a goal. Also the actor that initiated use case.
- **Supporting Actor:** actor—provides a service (for example, information) to the SuD.
  - An automated payment authorization service is an example.
  - could be an organization or person.
- **Offstage Actor:** Has an interest in the behavior of the use case, but is not primary or supporting; for example, a government tax agency.

15

Finding Use Cases

- Choose a system boundary
  - Software? Hardware? A person?
- Identify the primary actors
  - Those that will have goals fulfilled using the software
- Enumerate their goals
- Define use cases that satisfy their goals.

16

About Use Cases

1. The name should start with a strong verb.
2. A use case is a set of scenarios.
   - A scenario is a list of steps.
3. Each step should state what the user does and/or what the system responds.
4. **The steps must not mention how the system does something. Keep the steps essential or logical -- no colors, clicks, typing!**
5. Each step needs to be analyzed in detail before it becomes code.
6. Keep It Simple: use the simplest format you need.
7. Refine interesting use cases first.
8. Use cases are not object oriented
   - That would be too specific
   - They should not identify objects
9. Finally, an activity diagram might do the trick! (No use case needed)

17

Sample 2 Column

A	B	C
Number	UC1	Questions to anser
Name	Subscribe/Create Account	
Description	New users need to create an account: Create password with security questions and answers, enter student account information (ID & password).	
Actor	Student and System	
Student	**System**	
[This use case begins when the student arrives at class with their mobile device. The instructor has already made known the URL of the HappyClass applicaton.]		
Student Connects to Happy Class		
	System displays Intro Screen	
Student Selects Registeration.		
	System prompts for necessary data. (Create username(ID) and password and fill out the answers of three questions.)	what data is necessary?
Student Provides requested data		
	System stores information	
	System display login screen	

18

More about use cases

1. Make sure you store use cases so that they are easily found, edited, and used.
2. Put use cases on a project web site. (confluence or wiki)
3. Keep track of different versions.
4. Writing use cases is a team sport.
5. Focus on a particular user (give them a name) in each use case and each step.
6. **Don't get bogged down in all the special ways it can go wrong until you've finished the main success story.**
7. http://www.csci.csusb.edu/dick/samples/usecases.html
8. Finally Use Case Diagrams are pretty, but not sufficient.

19

Include/Extend

- Extend is used when a use case conditionally adds steps to another first class use case.
- Include is used to extract use case fragments that are duplicated in multiple use cases.
  - These fragments are always run.

20

Entity Class Modeling

- Determine the entity classes and attributes
- Model their relationships
- Create a class diagram

21

13.5.1 Noun Extraction (Entity class modeling)

- Review use case or user stories
- Identify the nouns
- Keep track of the verbs

22

CRC Cards (13.5.2) (Entity Class Modeling)

- Class-Responsibility-Collaboration Cards
- Interaction among team members can highlight missing or incorrect aspects of a class.
- Relationships between classes are highlighted.
- Does not do much for entity classes
  - But you should know the domain by now
  - You should be able to produce the entity classes
- Standard index cards divided into sections

23

Review CRC Cards

- http://agilemodeling.com/artifacts/crcModel.htm

24

CRC Card (re-iterating creation)

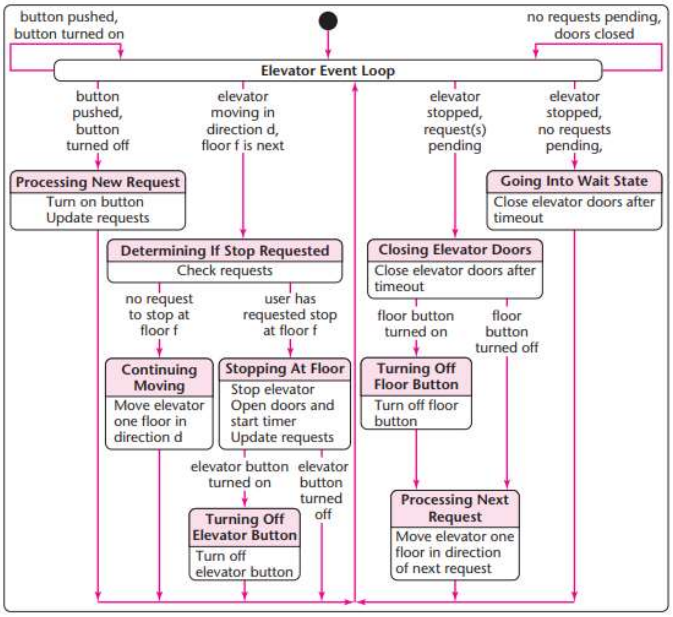
1. Find 3 to 5 classes
   - Noun extraction? Guess?
2. Define Responsibilities
   - What do they do and what do they know?
3. Define Collaborators
   - What do they need done or what information do they need?

25

Dynamic Modeling (13.6, page 414)

- The **dynamic model** represents the time–dependent aspects of a system.
- It is concerned with the temporal changes in the states of the objects in a system
- https://www.tutorialspoint.com/object_oriented_analysis_design/ooad_dynamic_modeling.htm
- Think State diagrams and Petri Nets

26



Page 415 in your text

27

How to write a play (twist on crc)

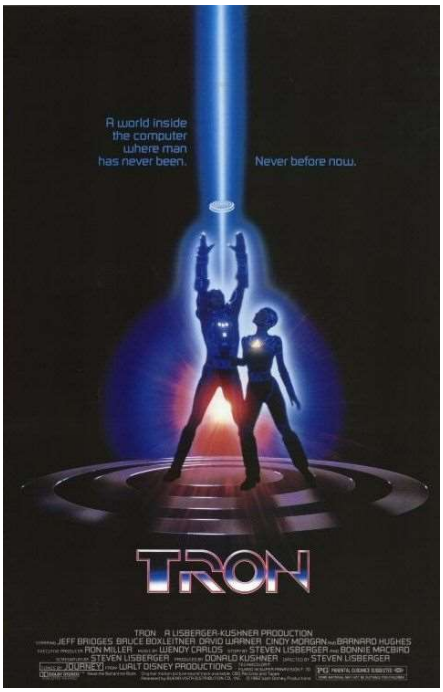
- Come up with a main character.
- Decide on conflict
  - Your play should have a conflict. Give your character a major problem that he or she has to solve immediately.
- Decide on a beginning point
- Show the story in actions and "speech"
  - Ok just actions for us
- Don't over do it!

- http://www.creative-writing-now.com/how-to-write-a-play.html
- http://www.dummies.com/how-to/content/playwriting-for-dummies-cheat-sheet.html

28

Play's!

- Idea for modeling using a 'play' is based on identifying responsibility
- Play's seem more fun than CRC cards.
- CRC cards might be more agile over the long run.



29

Test Workflow (13.7)

- Review Classes
- Review Models
- Start thinking of a test plan

30

Test workflow (13.7)

- Now is the time to start writing the test plan
- We should have a good collection of:
  - entity, boundary and control classes.
- Or if you prefer:
  - Model, View and Controller classes
- We should have a **good dynamic model**
- We should have **a good static model**
- We can write and refine test cases.
  - Why do I say refine??

31

Next slides

- Attribution:
- http://www.guru99.com/test-case.html

32

Test Cases Best Practices

1. Keep simple and transparent
2. Keep user in mind
3. Avid Repetition
4. Make no assumptions
5. Ensure 100% Coverage
6. Test Cases must be identifiable
7. Implement Testing Techniques
   a. Boundary Value Analysis
   b. Equivalence Partition
   c. Stat Transitions
   d. Error Guessing
8. Self Cleaning
9. Repeatable and self-standing
10. Peer review

http://www.guru99.com/test-case.html

33

Test Workflow

- https://www.youtube.com/watch?v=BBmA5Qp6Ghk
  - More detail: http://www.guru99.com/test-case.html
- https://youtu.be/9abf1eQephA

34

Guru99.com Test template

Test Case ID	Test Scenario	Test Steps	Test Data	Expected Results	Actual Results	Pass/Fail
TU01	Check Customer Login with valid Data	1.Go to sitehttp://demo.guru99.com 2.Enter UserId 3.Enter Password 4.Click Submit	Userid = guru99 Password = pass99	User should Login into application	As Expected	Pass

35

Test Case Management Tools

- **For documenting Test Cases**
  - With tools you can expedite Test Case creation with use of templates
- **Execute the Test Case and Record the results:**
  - Test Case can be executed through the tools and results obtained can be easily recorded.
- **Automate the Defect Tracking:**
  - Failed tests are automatically linked to the bug tracker , which in turn can be assigned to the developers and can be tracked by email notifications.
- **Traceability:**
  - Requirements, Test cases, Execution of Test cases are all interlinked through the tools, and each case can be traced to each other to check test coverage.
- **Protecting Test Cases:**
  - Test cases should be reusable and should be protected from being lost or corrupted due to poor version control. Test Case Management Tools offer features like
- **Naming and numbering conventions**
  - **Versioning**
  - Read only storage
  - Controlled access
  - Off-site backup
- Popular Test Management tools are : Quality Center and JIRA

36

Extracting Boundary and Control Classes (13.8)

- Each input screen, output screen, report, etc. Everything generated to show the user is a candidate for a boundary class
- Control classes are identified by algorithms that need implementing.
- They should be visible in a dfd.

37

Extracting Boundary and Control Classes (13.8) (Doing MVC)

- Each input screen, output screen, report, etc. Everything generated to show the user is a candidate for view. (View!)
- "model" classes are identified by algorithms that need implementing.
  - Non trivial operation/calculation, turn into class.
- They are also visible in your dfd.
- Controller
  - It is a class.
  - All the things that need to be bossed around!

38

Specification Document (13.18)

- A **Software Requirements Specification** (SRS) is a technical document that describes in detail the externally visible characteristics of a software product
- Parts of the SRS include: Environmental requirements: OS, platform, interoperability, standards, etc.
  - Non-functional requirements: security, usability, efficiency, etc.
  - Feature specifications: precisely describe each feature
  - Use cases: examples of how a user accomplishes a goal by using one or more features
- Test Plan
- SRS and Test Plan give enough detail
- http://www.jrobbins.org/ics121f03/lesson-spec-design.html

39

13.21 Metrics

- Book suggests number of pages in UML diagram
  - Count how many classes are defined
  - Count how many sequence diagrams are created
- Count how many errors you find during reviews

40

13.21 Metrics for the Object-Oriented Analysis Workflow

- As with the other core workflows
  - It is essential to measure the five fundamental metrics: size, cost, duration, effort, and quality
  - It is essential to keep accurate fault statistics
- A measure of size of the object-oriented analysis
  - Number of pages of UML diagrams

41

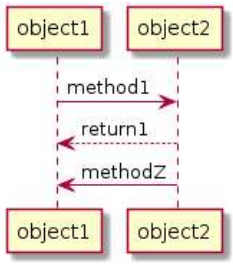
13.22 Challenges of the Object-Oriented Analysis Workflow

- Do not cross the boundary into object-oriented design
- Do not concern yourself with methods to classes *yet*
  - Reallocating methods to classes during stepwise refinement is wasted effort

42

Sequence Diagrams

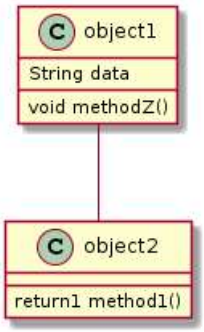
- A **Sequence diagram** is an interaction diagram that shows how objects operate with one another and in what order. It is a construct of a message sequence chart.
- Do not spend a lot of time on the books sequence diagrams or use case diagrams.
- Solid line means message to.
- Thank you wiki



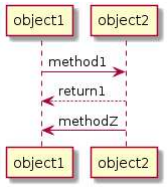
43

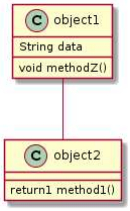
Class Diagrams

- Diagram that describes the structure of a system
  - the system's classes, their attributes
  - operations (or methods)
  - and the relationships among objects
- Class diagrams are static
- These are a key part architecture of a system
- Thank you wiki



44





45

Sample Sequence Diagram

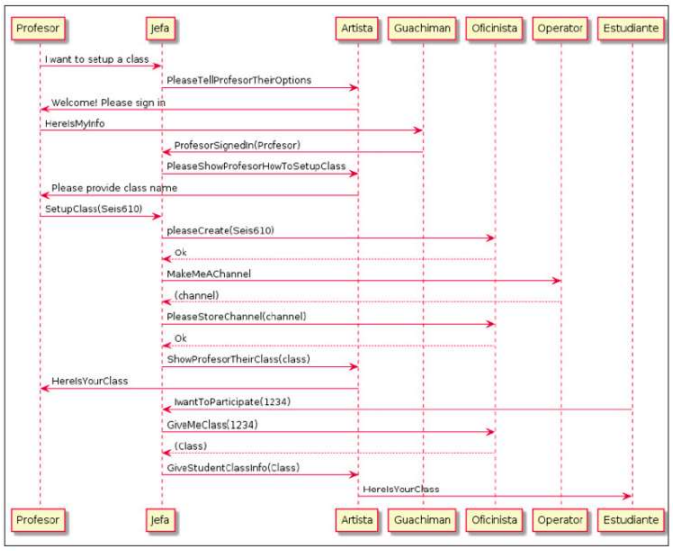
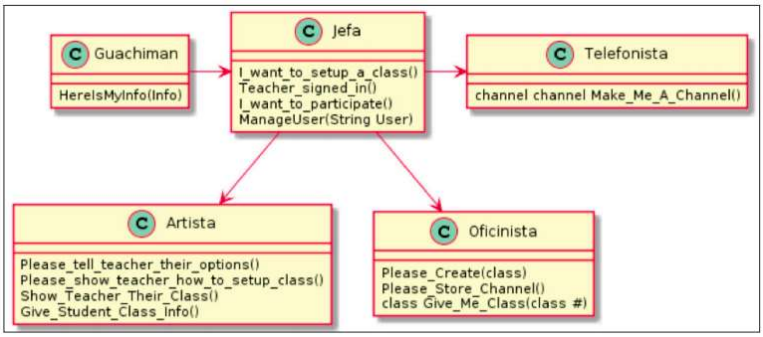


Figure 1. Example sequence diagram

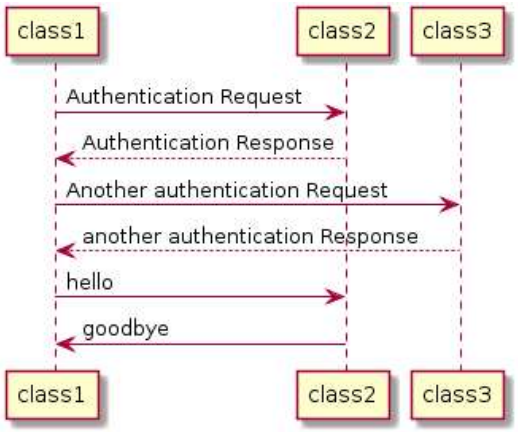
46

Sample Class Diagram



47

Quiz Part 1: Draw the class diagram for this sequence diagram



48

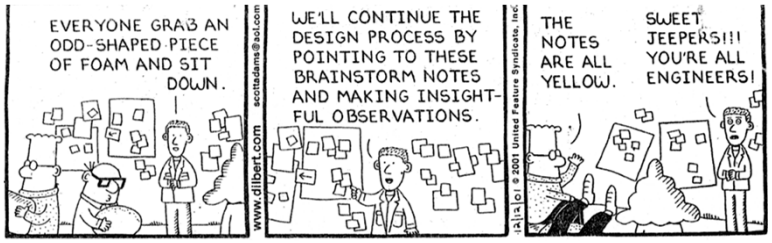
- The end!

49

SEIS 610

Chapter 14

Agenda



- Design (Chapter 14)
  - Design and Abstraction (14.1)
  - Operation-Oriented Design (14.2)
  - Data Flow Analysis (14.3)
  - Transaction Analysis (14.4)
  - Data Oriented Design (14.5)
  - Object Oriented Design (14.6)
  - Test Workflow (14.10)
  - Real-time Design (14.13)

Chapter 14 Design

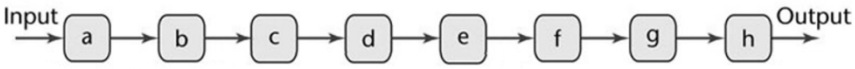
- **Architectural design** (14.1)
  - Modular decomposition of the product is our goal
  - We start with what we guess is the foundation
  - We build our (detailed) design on top!
- **Operation-Oriented** Design (14.2)
  - Remember high cohesion and low coupling
  - Books two approaches Data Flow Analysis and Transaction Analysis

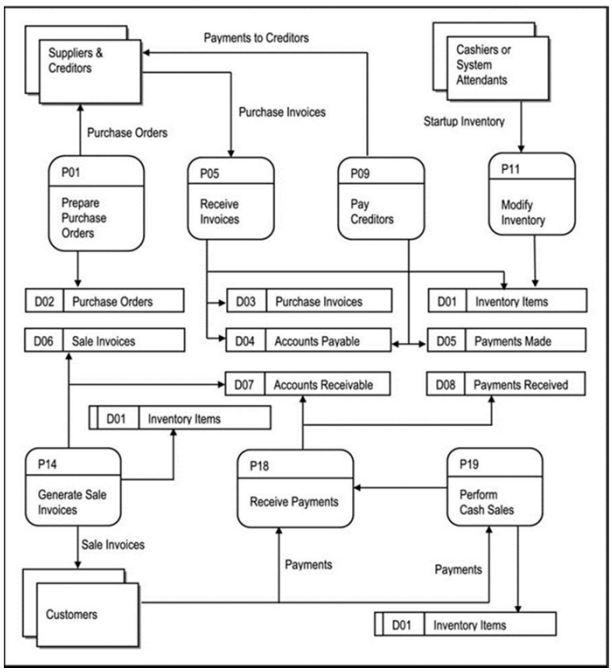
Data Flow Analysis (14.3)

- When DFD is done
- We have all the information we need
- Determine the point of highest abstraction
  - For input
  - For output
- Basically, where input looses the quality of being input and is only internal data.

Data Flow Analysis

- Software transforms input into output
- There happens to be no data stores in this picture
- Picture assumes all input goes into one process
- Picture assumes all output goes out one process
- This is really not how we have thought of it





Example

http://apprize.info/usability/engineering/engineering.files/image054.jpg

This is more indicative of how we have thought of things.

Data Flow Analysis (14.3)

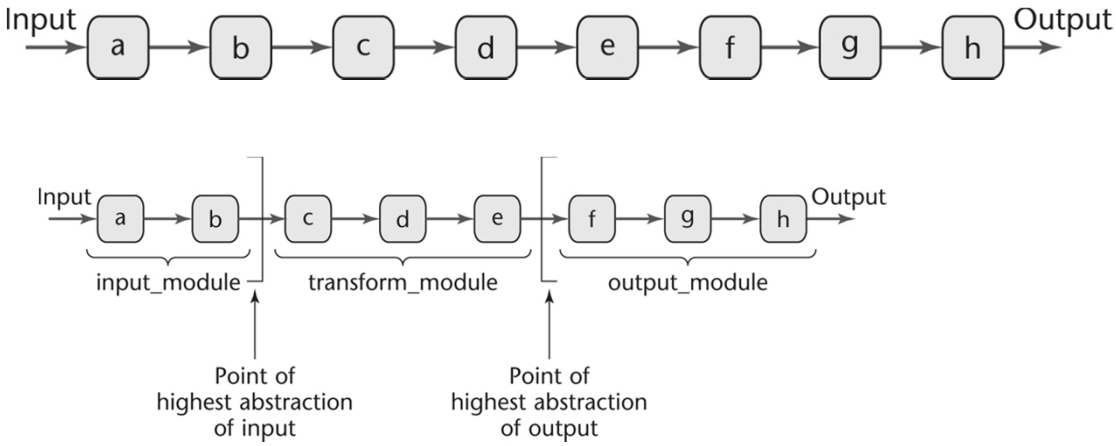
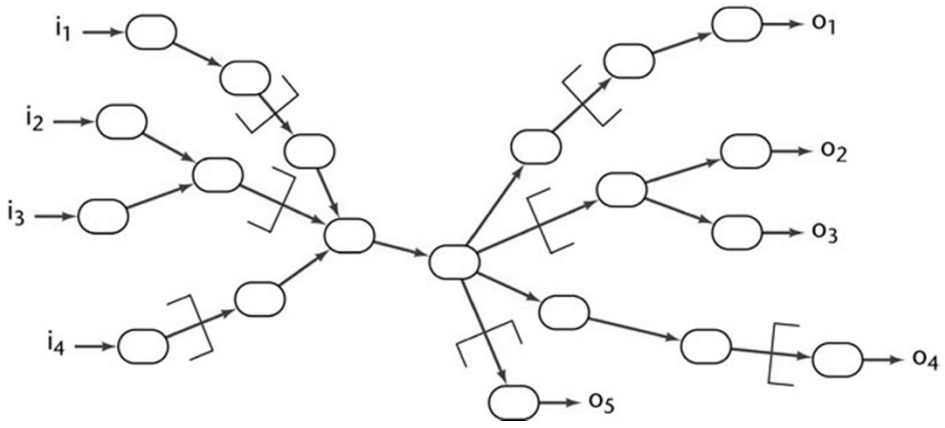
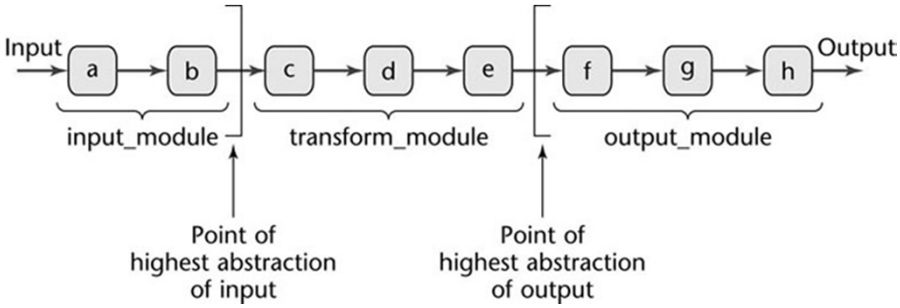


Figure 14.8



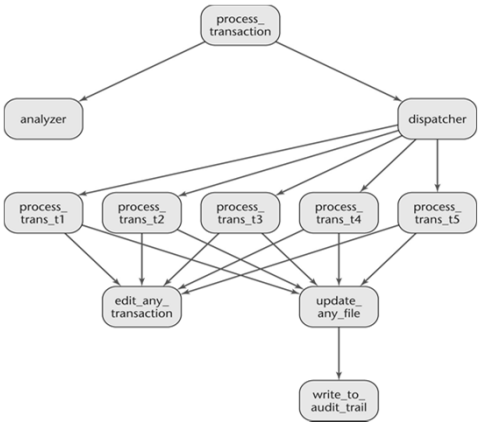
Data Flow Analysis

- Lather, rinse, repeat.
- Decompose product into modules
- Repeat until you believe you have high cohesion



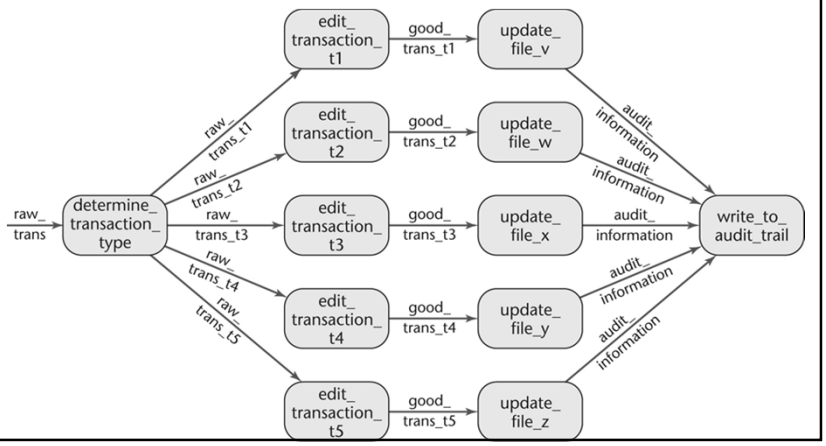
Transaction Analysis

- DFA might not work for transaction processing products
- This solution has 'control' coupling



Transaction Analysis

- Instead, perhaps have two handlers…One to edit and one to update.
- Instantiate as necessary.



Suppose we have a transaction

(jumping to Object Oriented)

- Transaction
  - Transaction amount
  - Transaction date

Suppose we have a transaction

- Transaction
- Data
  - Transaction amount (floating point number)
  - Transaction date (string)

Suppose we have a transaction class

- Transaction
- Data
  - Transaction amount (floating point number)
  - Transaction date (string)
- Methods
  - Set transaction amount
  - Edit transaction amount

Now think we have five different types of transactions!

- Loan transaction
- Savings transaction
- Sales transaction
- Return transaction
- Payment transaction

Vocab (you need to know this)

- Polymorphism
  - The condition of occurring in several different forms.
  - the occurrence of different forms among the members of a population or colony, or in the life cycle of an individual organism.
- Inheritance
  - Attributes from ancestor available to descendants
- Dynamic Binding
  - Correct object created when asked for!
- Information Hiding
  - Making only what is necessary available publicly
- Encapsulation
  - Bundling methods and data into objects. (or perhaps mechanisms and data)
- Abstraction
  - **F**ocus on the essential properties of a 'thing' instead of one specific example. (lynda.com)
  - We define essential aspects of a system.

Now suppose we have a file class

- Data
  - File name
  - File handle
- Methods
  - Update

Polymorphism

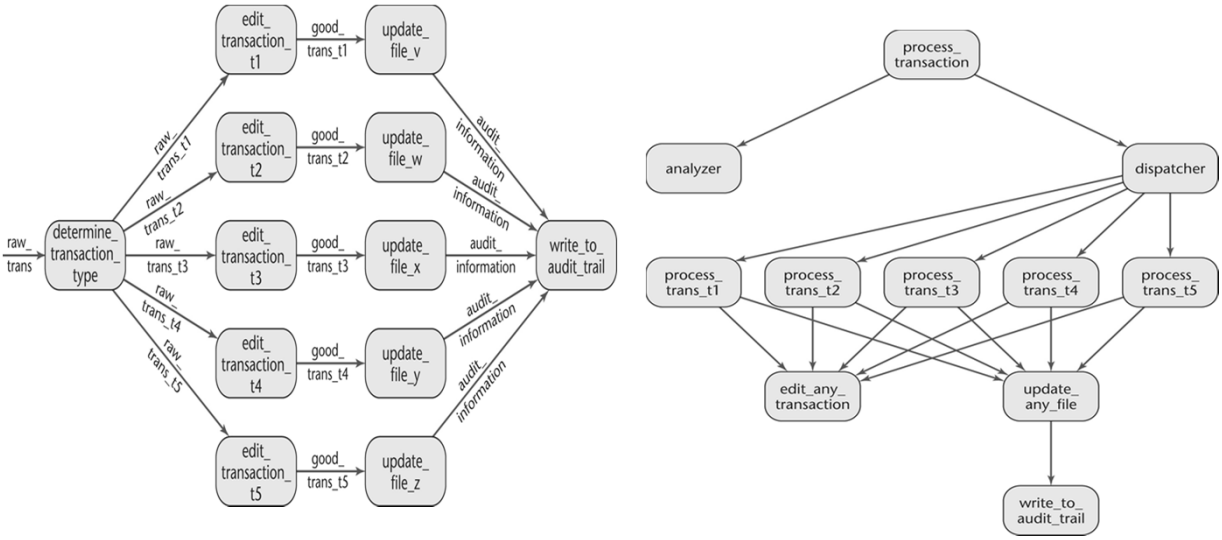
- Transaction
  - LoanTransaction
  - SavingsTransacction
  - SalesTransaction
  - ReturnTransaction
  - PaymentTransaction
- TransactionFile
  - LoanTransactionFile
  - SavingsTransactionFile
  - SalesTransactionFile
  - ReturnTransactionFile
  - PaymentTransactionFIle

So if we were to do this in Java

```
Transaction transaction = new Transaction();
File file = new TransactionFile();

transaction.edit(123.00);
file.update(transaction);
```

So if we were to do this in Java

```
Transaction transaction = new LoanTransaction();
File file = new LoanTransactionFile();

transaction.edit(123.00);
file.update(transaction);
```



Data-Oriented Design (14.5)

- Basic principle
  - The structure of a product must conform to the structure of its data
- Three very similar methods
  - Michael Jackson [1975], Warnier [1976], Orr [1981]
- Data-oriented design
  - Has never been as popular as action-oriented design
  - With the rise of OOD, data-oriented design has largely fallen out of fashion

14.6 Object Oriented Design

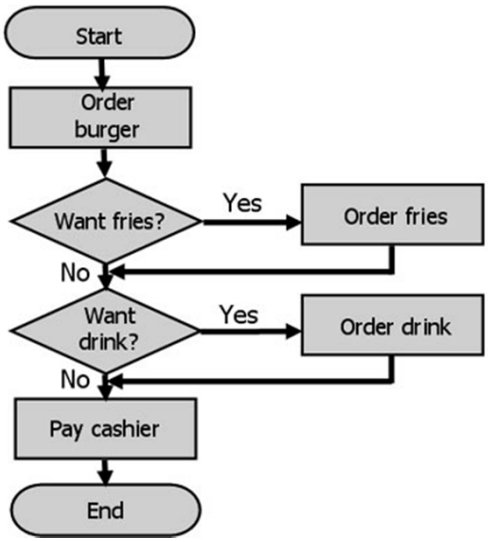
- Design product in terms of classes extracted during our analysis
  - Classes are identified during the object-oriented analysis workflow
  - Classes are designed during the design workflow
- SEIS 610
  - We write our play
  - We have our sequence diagrams
- Create a class diagram
  - With methods!
- Perform Detailed Design as required
  - State Diagrams
  - Activity Diagrams

Design Workflow (14.9)

- Complete the sequence diagram (dynamic)
- Complete the class diagram (static)
- Perform the detailed design
  - Dynamic diagrams
    - State charts
    - Flow charts
    - Petri Nets
    - DFD

Activity Diagrams

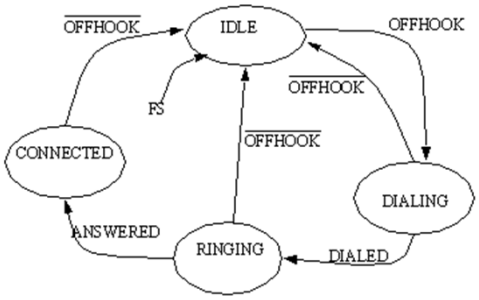
- "Old fashioned flowcharts"
- Use for methods with difficult logic.



- http://www.teach-ict.com/as_a2_ict_new/ocr/A2_G063/331_systems_cycle/analysis_tools/miniweb/images/flowchart.jpg

State Diagrams

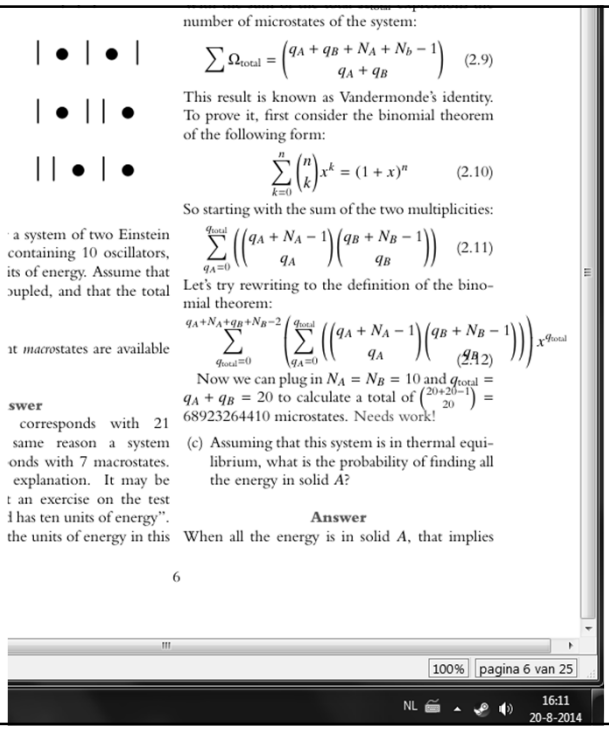
- Use for classes which have Interesting state logic.



http://www.thelearningpit.com/hj/plcs_files/plcs-353.gif

Text!

- Yes, you can use text!
- http://i.imgur.com/Bbn2CZj.png?1



| • | • |

| • | | • |

| | • | • |

a system of two Einstein containing 10 oscillators, its of energy. Assume that oupled, and that the total

ıt *macro*states are available

swer

corresponds with 21 same reason a system onds with 7 macrostates. explanation. It may be t an exercise on the test l has ten units of energy". the units of energy in this

number of microstates of the system:

$$\sum \Omega_{\text{total}} = \binom{q_A + q_B + N_A + N_b - 1}{q_A + q_B} \quad (2.9)$$

This result is known as Vandermonde's identity. To prove it, first consider the binomial theorem of the following form:

$$\sum_{k=0}^{n} \binom{n}{k} x^k = (1+x)^n \quad (2.10)$$

So starting with the sum of the two multiplicities:

$$\sum_{q_A=0}^{q_{\text{total}}} \left(\binom{q_A + N_A - 1}{q_A} \binom{q_B + N_B - 1}{q_B} \right) \quad (2.11)$$

Let's try rewriting to the definition of the binomial theorem:

$$\sum_{q_{\text{total}}=0}^{q_A+N_A+q_B+N_B-2} \left(\sum_{q_A=0}^{q_{\text{total}}} \left(\binom{q_A + N_A - 1}{q_A} \binom{q_B + N_B - 1}{q_B} \right) \right) x^{q_{\text{total}}} \quad (2.12)$$

Now we can plug in $N_A = N_B = 10$ and $q_{\text{total}} = q_A + q_B = 20$ to calculate a total of $\binom{20+20-1}{20} = 68923264410$ microstates. Needs work!

(c) Assuming that this system is in thermal equilibrium, what is the probability of finding all the energy in solid A?

Answer

When all the energy is in solid A, that implies

6

100% pagina 6 van 25

NL 16:11 20-8-2014

Do we have to do all diagram types?

- Nope!
- **You should always do sequence and class diagrams**
- You may need activity diagrams
- You may need state diagrams
- You may want Petri Nets
- You may like Data Flow Diagrams
- You may need a darn good description in text.
- But you don't need all of the above!

Keep in mind

- Information hiding
- Abstraction
- Responsibility-driven design
  - That is what we are doing!!

- The end!

- P.S. Forgot Test Workflow

Motto: Review Review Review

Object-Oriented and Classical Software Engineering

Eighth Edition, WCB/McGraw-Hill, 2011
Chapter 10

Stephen R. Schach

CHAPTER 10

"KEY MATERIAL FROM PART A"

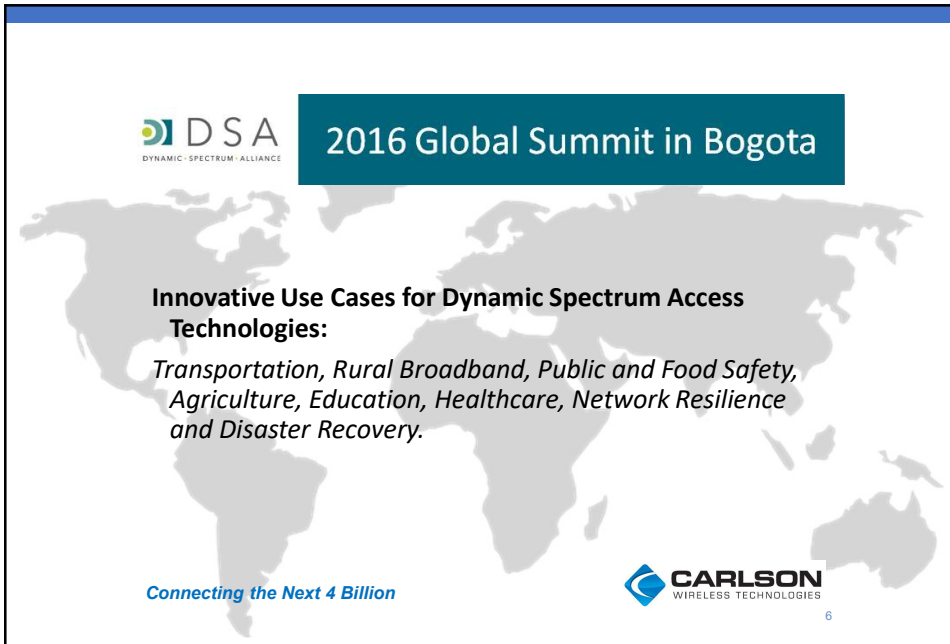
Why are we starting here?

Project Selection

- St. Thomas Mission

Why are we starting here?

- Project Based Class
- LAST SEMESTER: 2 Projects to choose from
  - Online Teaching System for Spanish Classes
  - Network Management System for Dynamic Spectrum TVWS systems.
- Just like real life!



Highlights

- Main theme was making Internet access affordable to connect the lowest income 4 billion earthlings
  - Highlighted a number of success stories in Africa and Latin America
    - Sub-$2000 installations enabling sub-$3.00/month service in small towns and villages
    - Mawingu Networks already covering >6000 km² in Kenya with local resident doing construction and setup (video)
  - Economist Richard Thanki presentation on the value of these efforts
- DSA concepts explained to regulators from developing countries
  - Message was received; minds were changed in favor of unlicensed spectrum
  - Speakers covered from technology and economics to financing
- Without exception, all trials and deployments report no interference with incumbents
- Two vendors discussed TVWS hardware availability
  - 6Harmonics 802.11af + 802.22 ASIC solution later this year
  - Carlson Wireless with MediaTek/Aviacomm chipset on mini-PCI being optimized; ready soon
    - Meeting difficult FCC OOBE limits
    - Initial production quantities at ~$37.00
- Began discussion of new Wi-Fi regulatory "alliance" with Microsoft, Google, Facebook

Access Affordability

Billions of People on Earth	Average Annual Income	Affordable Monthly Communications Spend
1st	$49,206	$205
2nd	$12,722	$53
3rd	$5,540	$23
4th	$2,987	$12
5th	$1,771	$7
6th	$1,065	$4.40
7th	$540	$2.25

Overview

- Software development: theory versus practice
- Iteration and incrementation
- The Unified Process
- Workflow overview
- Teams
- Cost–benefit analysis
- Metrics
- CASE
- Versions and configurations
- Testing terminology
- Execution-based and non-execution-based testing
- Modularity
- Reuse
- Software project management plan

10.1 Software Development: Theory vs. Practice

- Ideally, software is developed like everything else
  - Linear
  - Starting from scratch

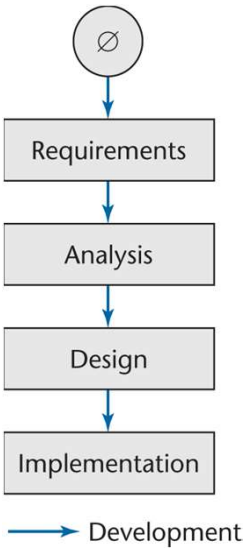


Figure 10.1

Software Development: Theory vs. Practice

- In the real world, software development is totally different
  - We are not building bridges
  - We are not building buildings
  - We make (and then fix) mistakes (usually)
  - The client's requirements change while the software product is being developed
    - *Moving target problem*

10.2 Iteration and Incrementation

- In real life, we cannot speak about "the design phase"
  - Instead, the operations of the design phase are spread out over the life cycle
  - We keep returning to earlier workflows
- The basic software development process is *iterative*
  - Each successive version is closer to its target than its predecessor

What's the difference?

- https://watirmelon.blog/2015/02/02/iterative-vs-incremental-software-development/

Incremental

Iterative

Miller's Law

- At any one time, we can concentrate on only approximately seven *chunks* (units of information)
- To handle larger amounts of information, use *stepwise refinement*
  - Concentrate on the seven aspects that are currently the most important
  - Postpone aspects that are currently less critical
  - Every aspect is eventually handled, but in order of current importance
- This is an *incremental* process

Incrementation

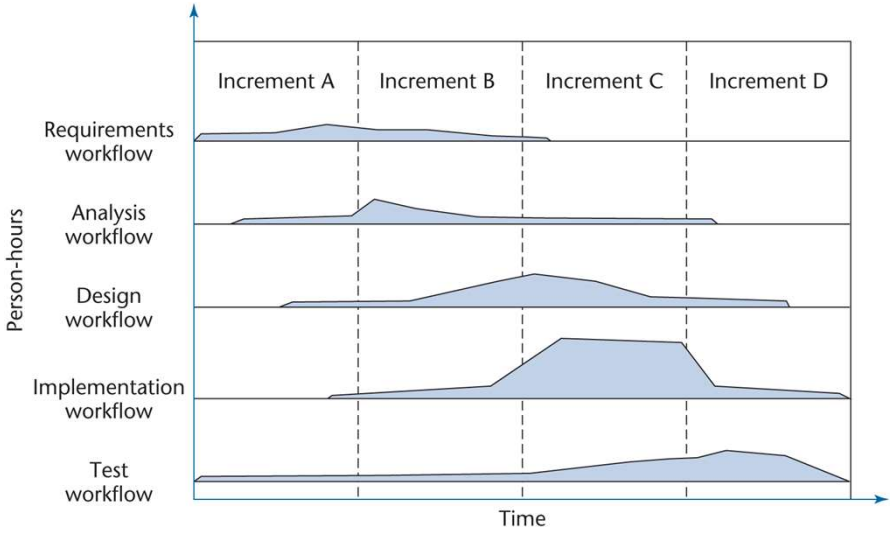


Figure 10.2

Incrementation

- The number of increments will vary — it does not have to be four
- Sequential phases do not exist in the real world
- Instead, the five *core workflows* (activities) are performed over the entire life cycle
  - Requirements workflow
  - Analysis workflow
  - Design workflow
  - Implementation workflow
  - Test workflow

Workflows

- All five core workflows are performed over the entire life cycle
- However, at most times one workflow predominates
- Examples:
  - At the beginning of the life cycle
    - The requirements workflow predominates
  - At the end of the life cycle
    - The implementation and test workflows predominate
- Planning and documentation activities are performed throughout the life cycle

Iteration

- Iteration is performed during each incrementation

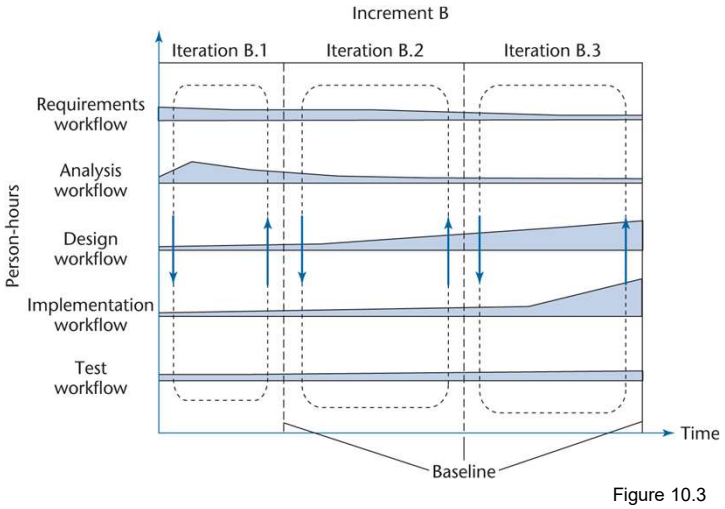


Figure 10.3

Iteration

- Again, the number of iterations will vary—it is not always three

10.3 The Unified Process

- The *software process* is the way we produce software
- It incorporates
  - The methodology
    - With its underlying software life-cycle model and techniques
  - The tools we use
  - The individuals building the software

The Unified Process

- Despite its name, the *Unified Process* is not a process!
  - It's a *methodology*
- The Unified Process is an adaptable methodology
  - It has to be modified for the specific software product to be developed

The Unified Process

- The Unified Process uses a graphical language, the *Unified Modeling Language* (UML) to represent the software being developed
- A *model* is a set of UML diagrams that represent various aspects of the software product we want to develop

The Unified Process

- The object-oriented paradigm is iterative and incremental in nature
  - There is no alternative to repeated iteration and incrementation until the UML diagrams are satisfactory

10.4 Workflow Overview

- *Requirements workflow*
  - Determine exactly what the client needs
- *Analysis workflow*
  - Analyze and refine the requirements
    - To achieve the detailed understanding of the requirements essential for developing a software product correctly and maintaining it easily

Workflow Overview

- *Design workflow*
  - Refine the artifacts of the analysis workflow until the material is in a form that can be implemented by the programmers
- *Implementation workflow*
  - Implement the target software product in the chosen implementation language(s)

Workflow Overview

- *Test workflow*
  - Testing is carried out in parallel with the other workflows, from the beginning
  - Every developer and maintainer is personally responsible for ensuring that his or her work is correct
  - Once the software professional is convinced that an artifact is correct, it is handed over to the software quality assurance group for independent testing

10.5 Teams

- Software products are usually too large (or too complex) to be built by one software engineering professional within the given time constraints
- The work has to be shared among a group of professionals organized as a *team*

Teams

- The team approach is used for each of the workflows
- In larger organizations there are specialized teams for each workflow

10.6 Cost–Benefit Analysis

- *Cost–benefit analysis* is a way of determining whether a possible course of action would be profitable
  - Compare estimated future benefits against projected future costs
- Cost–benefit analysis is a fundamental technique in deciding whether a client should computerize his or her business
  - And if so, in what way

10.7 Metrics

- We need measurements (or *metrics*) to detect problems early in the software process
  - Before they get out of hand

Metrics

- There are five fundamental metrics
  - Each must be measured and monitored for each workflow:
- 1. Size (in lines of code or, better, in a more meaningful metric)
- 2. Cost (in dollars)
- 3. Duration (in months)
- 4. Effort (in person-months)
- 5. Quality (number of faults detected)

Metrics

- Metrics serve as an early warning system for potential problems
- Management uses the fundamental metrics to identify problems
- More specialized metrics are then utilized to analyze these problems in greater depth

10.8 CASE

- *CASE* stands for *Computer-Aided Software Engineering*
  - Software that assists with software development and maintenance

CASE Taxonomy

- A CASE *tool* assists in just one aspect of the production of software
  - Examples:
    - A tool that draws UML diagrams
    - A report generator, which generates the code needed for producing a report

CASE Taxonomy

- A CASE *workbench* is a collection of tools that together support one or two activities
  - Examples:
    - A requirements, analysis, and design workbench that incorporates a UML diagram tool and a consistency checker
    - A project management workbench that is used in every workflow
- A CASE *environment* supports the complete software process

10.9 Versions and Configurations

- During development and maintenance, there are at least two versions of the product
  - The old version, and
  - The new version
- There will also be two or more versions of each of the component artifacts that have been changed

Versions and Configurations

- The new version of an artifact may be less correct than the previous version
- It is therefore essential to keep all versions of all artifacts
  - A CASE tool that does this is called a *version control tool*

Versions and Configurations

- A *configuration* is
  - A set of specific versions of each artifact from which a given version of the complete product is built
  - A *configuration-control tool* can handle problems caused by development and maintenance by teams
    - In particular, when more than one person attempts to change the same artifact
- A *baseline* is a configuration of all the artifacts in the product
  - After each group of changes has been made to the artifacts, a new baseline is attained

Versions and Configurations

- If a software organization does not wish to purchase a complete configuration-control tool, then, at the very least,
  - A version-control tool must be used in conjunction with
  - A *build tool*, that is, a tool that assists in selecting the correct version of each compiled-code artifact to be linked to form a specific version of the product
    - Build tools, such as *make*, have been incorporated into a wide variety of programming environments

10.10 Testing Terminology

- A *fault* is injected into a software product when a human makes a mistake
- A *failure* is the observed incorrect behavior of the software product as a consequence of a fault
- The *error* is the amount by which a result is incorrect
- The word *defect* is a generic term for a fault, failure, or error

Testing Terminology

- The *quality* of software is the extent to which the product satisfies its specifications
- Within a software organization, the primary task of the *software quality assurance* (SQA) group is to test that the developers' product is correct

10.11 Execution-Based and Non-Execution-Based Testing

- There are two basic forms of testing:
  - *Execution-based testing* (running test cases), and
  - *Non-execution-based testing* (carefully reading through an artifact)

Non-Execution-Based Testing

- In a *review*, a team of software professionals carefully checks through a document
  - Examples:
    - specification document
    - design document
    - code artifact
- There are two types of review
  - *Walkthrough* – less formal
  - *Inspection* – more formal

Non-Execution-Based Testing

- Non-execution-based testing has to be used when testing artifacts of the requirements, analysis, and design workflows
- Execution-based testing can be applied to only the code of the implementation workflow
- Non-execution-based testing of code (code review) has been shown to be as effective as execution-based testing (running test cases)

10.12 Modularity

- A *module* is
  - A lexically contiguous sequence of program statements,
  - Bounded by boundary elements (that is, { … } pairs),
  - Having an aggregate identifier
- Examples:
  - Procedures and functions of the classical paradigm
  - Objects
  - Methods within an object

Three Design Objectives

- Ensure that the *coupling* (degree of interaction between two modules) is as low as possible
  - An ideal product exhibits only *data coupling*
    - Every argument is either a simple argument or a data structure for which all elements are used by the called module
- Ensure that the *cohesion* (degree of interaction within a module) is as high as possible

Modularity

- Maximize *information hiding*
  - Ensure that implementation details are not visible outside the module in which they are declared
    - In the object-oriented paradigm, this can be achieved by careful use of the **`private`** and **`protected`** visibility modifiers

10.13 Reuse

- *Reuse* refers to using components of one product to facilitate the development of a different product with a different functionality
  - Examples of a reusable component:
    - Module
    - Class
    - Code fragment
    - Design
    - Part of a manual
    - Set of test data, a contract
    - Duration and cost estimate

Reuse

- Reuse is most important because
  - It takes time (= money) to
    - specify,
    - design,
    - implement,
    - test, and
    - document

    a software component
- If we reuse a component, we must retest the component in its new context

10.14 Software Project Management Plan

- The components of a *software project management plan*:
  - The work to be done
  - The resources with which to do it
  - The money to pay for it

Three Work Categories (The work)

- Project function
  - Work carried on throughout the project
  - Examples:
    - Project management
    - Quality control

The Resources

- Resources needed for software development:
  - People
  - Hardware
  - Support software
- Use of resources varies with time
  - The entire software development plan must be a function of time

The Money!

- Buy or build?
- Or do anything at all.
- *Money* is a vital component of the plan
  - A detailed budget must be worked out
  - The money must be allocated, as a function of time, to the project functions and activities
- Key components of the plan include
  - The c*ost estimate,* and
  - The *duration estimate*

Three Work Categories (Back to work!)

- Activity
  - Work that relates to a specific phase
  - A major unit of work
    - With precise beginning and ending dates,
    - That consumes *resources*, and
    - Results in *work products* like the budget, design, schedules, source code, or users' manual
- Task
  - An activity comprises a set of *tasks* (the smallest unit of work subject to management accountability)

Completion of Work Products

- A *milestone* is the date on which the work product is to be completed
- It must first pass *reviews* performed by
  - Fellow team members
  - Management
  - The client
- Once the work product has been reviewed and agreed upon, it becomes a *baseline*

Software Project Management Plan

- A *work package* is
  - A work product, *plus*
    - Staffing requirements
    - Duration
    - Resources
    - The name of the responsible individual
    - Acceptance criteria for the work product
    - The detailed budget as a function of time, allocated to
      - Project functions
      - Activities

The end!

SEIS 610

Chapter 11

Requirements

Agenda

- Review Requirements
  - Business Model, Etc. (11.1-11.5)
  - Rapid Prototyping (11.13, 11.15)
  - Human Factors (11.14)

Chapter 11 – put another way

Chapter 11: Requirements
- The Requirements ~~Phase~~ Workflow
- Determining What the Client Needs
- Understanding the Domain
- The Business Model
- Initial Requirements
- Rapid Prototyping
- Human Factors
- Rapid Prototyping as a Specification Technique
- Challenges of the Requirements

Requirements – Business Model (11.1-11.5)

- First: Understand the domain
  - Acquire familiarity with application domain (Important)
  - You can't hope to automate a process for somebody without understanding the problem they are trying to solve.
  - This is why we have a glossary!
- Business Model
  - Business model of the 'domain'
  - How do they make money?
  - Why is this product valuable

Requirements – Business Model

- Why should/should software help?
  - How much will the software cost to create?
  - Is the software going to be sold or used internally?
  - If it is going to be sold as a product:
    - For how much?
    - How much do competitive products cost?
    - What features do competitive products have?
    - Where will this product fit within the product offerings?
    - Can you think of two people besides family members who will buy it??

Business Model

- Ask yourself: “Why should the software help?”
  - How much is it costing not to have the software?
  - How do you figure that out? Understand the domain!
- Understanding the domain
  - Interviewing
  - Surveys and Questionnaires
  - Direct Observation

Business Model

- Initial Requirements
  - User Stories
    - **As a user, I would like to _____ so that I can _____**
  - Use cases
    - More formal, user/system interaction
- Functional Requirements
  - An action the target must be able to perform
  - Created during requirements and refined during analysis workflows
- Non-functional Requirements
  - Specifics related to the product itself
    - Platform constraints, response times, reliability
  - Best addressed during requirements and analysis, but may have to be handled during design.

Business Model

- Initial Requirements
  - User Stories
    - As a user, I would like to _____ so that I can _____
  - Use cases
- Functional Requirements
  - An action the target must be able to perform
  - Created during requirements and refined during analysis workflows
- Non-functional Requirements (FURPS)
  - Specifics related to the product itself
    - Platform constraints, response times, reliability
  - Best addressed during requirements and analysis, but may have to be handled during design.

11.5 Initial Requirements

- Heavily influenced by ‘business model’
- Requirements may be modified
- Functional requirement
  - Specifies an action the target must be able to perform
- Non-functional requirement, Quality requirement
  - Platform constraints
  - Response times
  - Reliability

11.5 Initial Requirements

- Functional
  - Handled during requirements and analysis workflows
- Non-functional
  - Sometimes handled during design workflow
- In agile, RUP
  - Your use cases
  - Your user stories
  - Become the requirements

Software Rapid Prototype (11.13, 11.15)

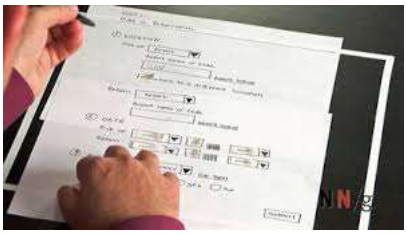
- Prototyping
  - The process of developing a trial version of the system
  - Gives engineers and users a chance to ”test drive” the software
  - Perhaps a way of correcting the weakness of “waterfall”
- Benefits
  - Improve usability factors
  - Understanding of requirements
  - Even effort reductions are seen
- Problems
  - Many times standards are not enforced for the prototype
  - Less coherent design and integration results
  - Then not thrown away!

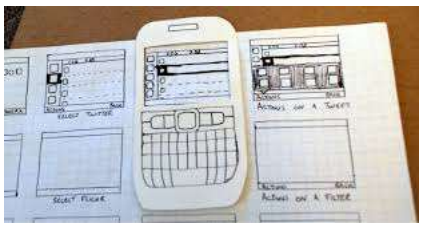
Rapid Prototyping (11.13, 11.15)

Software Rapid Prototype

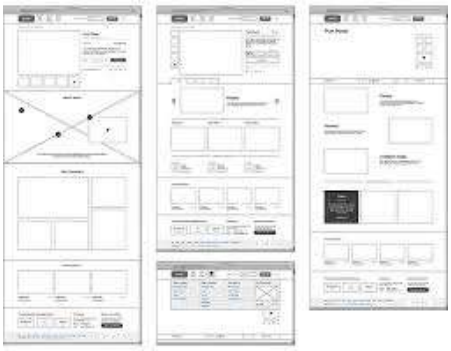
- Throw-away vs. keep-it
- Throw-away
  - Just like it sounds—discard the prototype when finished
- Keep-it (evolutionary programming)
  - ”Well we made it this far, may as well not throw it away”
  - Likely developed with loose standards
  - Less coherent design and integration results
  - Big ball-of-mud design going forward

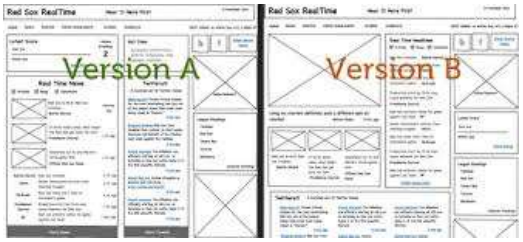
Paper Prototypes



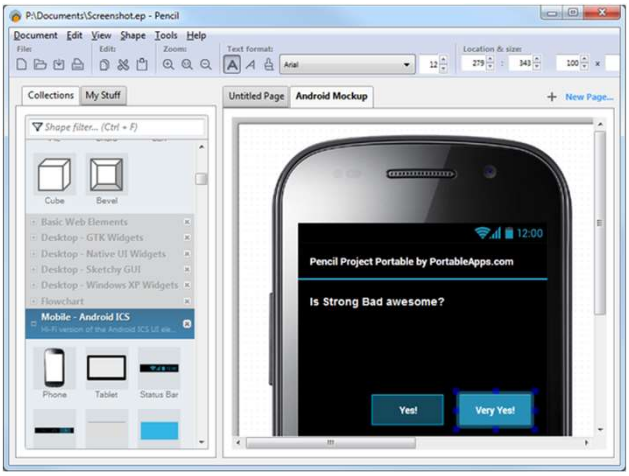


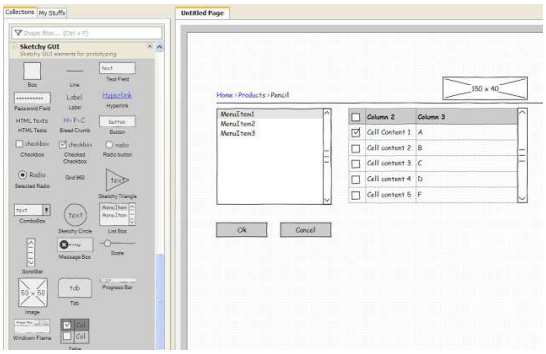
Paper Prototypes –Wireframe





Sample Rapid Prototypes – Evolus' Pencil





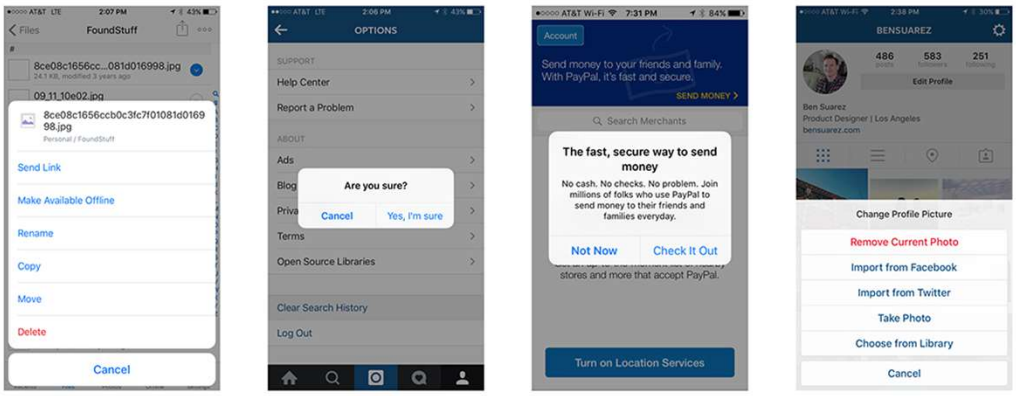
http://pencil.evolus.vn/

Human Factors (11.14)

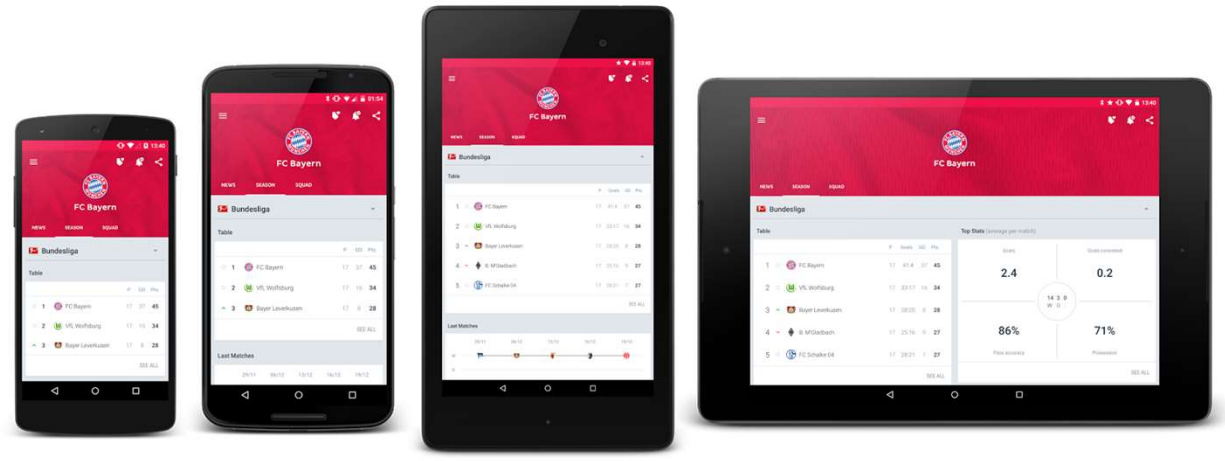
- User friendliness
- Size of letters
- ADA requirements
- Read up before you start for your platform
  - http://beijerinc.com/pdf/whitepaper/interface_design_best_practices.pdf
  - http://developer.android.com/guide/practices/index.html
  - http://www.applicoinc.com/blog/ios-mobile-app-development-guide/
- Other

Windows Metro

IOS

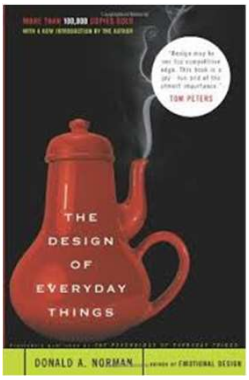

Files
FoundStuff
Send Link
Make Available Offline
Rename
Copy
Move
Delete
Cancel
OPTIONS
Help Center
Report a Problem
Are you sure?
Cancel
Yes, I'm sure
Terms
Open Source Libraries
Clear Search History
Log Out
Account
Send money to your friends and family.
With PayPal, it's fast and secure.
SEND MONEY
Search Merchants
The fast, secure way to send money
No cash. No checks. No problem. Join millions of folks who use PayPal to send money to their friends and families everyday.
Not Now
Check It Out
Turn on Location Services
BENSUAREZ
Edit Profile
Change Profile Picture
Remove Current Photo
Import from Facebook
Import from Twitter
Take Photo
Choose from Library
Cancel

Android


FC Bayern
Bundesliga
2.4
0.2
86%
71%

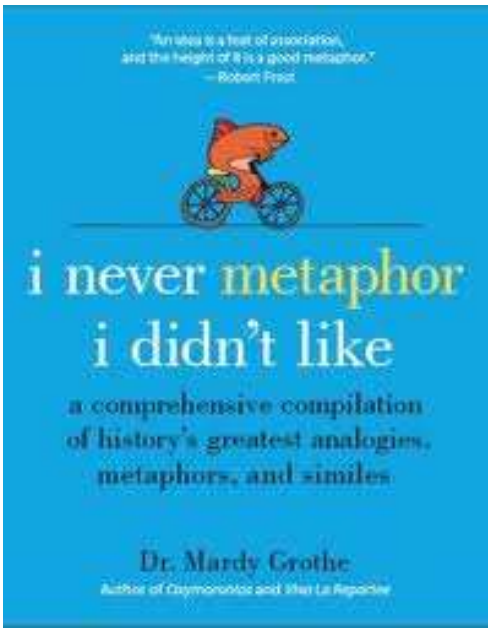
User Interface Thoughts

- Design of Everyday Things
- Don Norman



Affordances/Metaphors

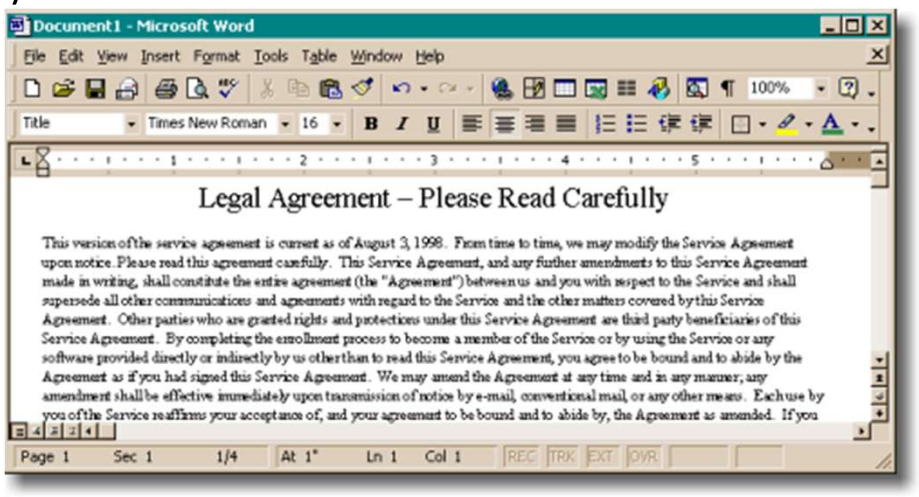
- Affordances: Well-designed objects make it clear how they work just by looking at them.
  - (Design of Everyday Things)
- Metaphors
  - An Interface metaphor is a set of user interface visuals, actions and procedures that exploit specific knowledge that users already have of other domains.
  - The purpose of the interface metaphor is to give the user instantaneous knowledge about how to interact with the user interface.
  - They are designed to be similar to physical entities but also have their own properties
  - From wikipedia



Any idea how to zoom?

http://www.joelonsoftware.com/uibook/chapters/fog0000000060.html

Any idea how to zoom?

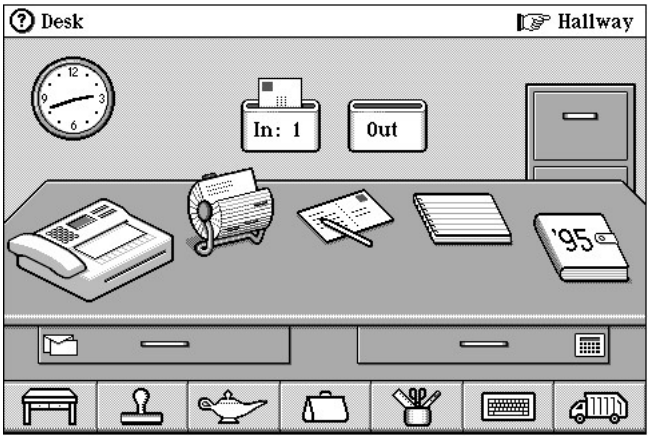


http://www.joelonsoftware.com/uibook/chapters/fog0000000060.html

Icons

www.webdesignhot.com

General Magic's defunct Magic Cap operating system



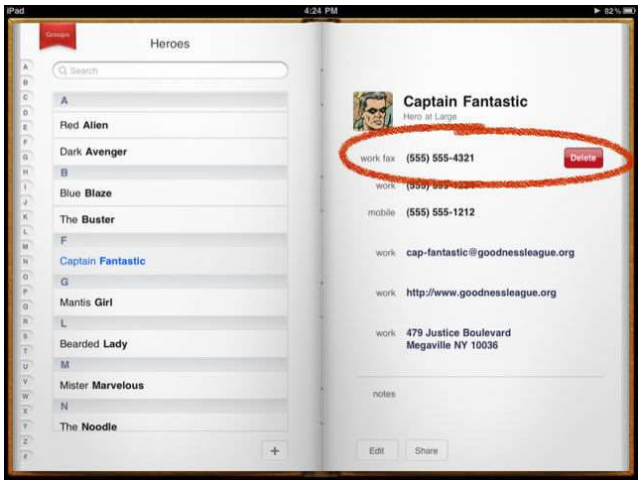
www.codinghorror.com

Microsoft Bob

http://boscoh.com/programming/beautiful-user-interfaces

E-book Metaphor

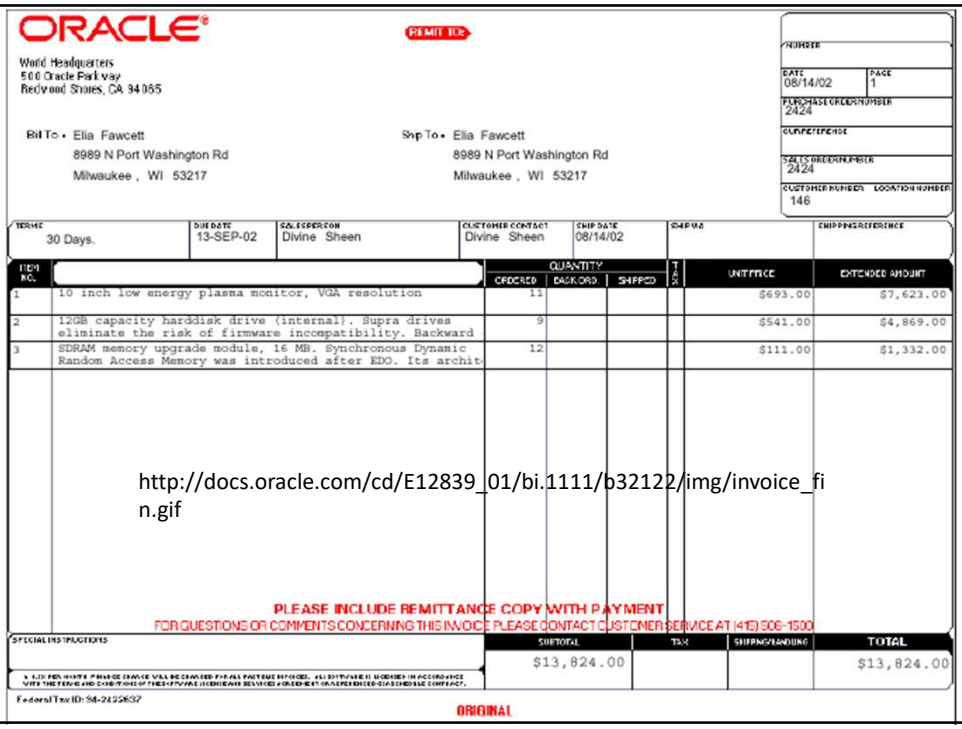
alfabravo.com



globalmoxie.com

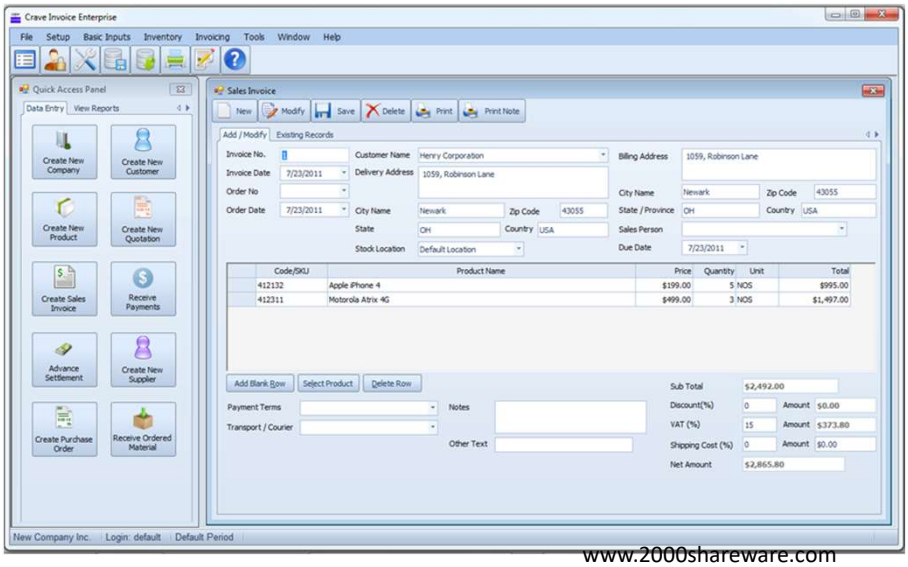
Drewtech.com

http://www.youtube.com/watch?v=Nj697OF6Cmo



http://docs.oracle.com/cd/E12839_01/bi.1111/b32122/img/invoice_fin.gif

Invoice Program?



www.2000shareware.com

Inventory (real)

inmotionsystems.wordpress.com

Inventory/Shelf Metaphor

- www.rockpapershotgun.com

Visual Interaction Design: Beyond the Interface Metaphor

- http://old.sigchi.org/bulletin/1997.2/vid.html
- "When a metaphor is applied to a system, it gives the system a particular set of ***affordances***. "
- "Metaphor is a container for a particular set of **affordances**."
- "The book metaphor includes a set of affordances, including those for page-turning, reading text, bookmarking, and so on."
- "The blank sheet of paper metaphor affords marking and erasing, and so on."

Affordances/Metaphors

- Affordances only have meaning when considered with respect to a particular group of users.
  - The front door to your home affords passage to you and your family, but not to a giraffe.
  - A grade school desk affords sitting to a child, but not to an adult.
  - This article affords reading only to people who read English and have enough motivation to continue reading.
- Affordances must be designed with the user in mind.

http://old.sigchi.org/bulletin/1997.2/vid.html

Affordances/Metaphors

- When we create an interface metaphor, we are, in essence, dumping the contents of the metaphor (its affordance set) onto the computer system.
- Some of those affordances fit nicely onto the system's feature set (else that metaphor would not have been chosen), others do not have a corresponding feature in the system, and some of the system's features are left affordance-less, invisible.

Affordances/Metaphors

- Metaphor is good as a stage of design, suggesting to us what features might be appropriate in the system or supplying us
- A stepping-off point for the look and behavior of the interface.
- But we need to get beyond the metaphor, even allowing the system to grow to where it no longer resembles that original metaphor at all.

Reusing the Rapid Prototype (11.15)

- Problems with rapid prototypes
  - Assumption you are farther than you are
  - Hurt feelings. (really)
    - High resolution prototypes have a lot of time put in
  - Desire not to "waste" the rapid prototype
  - Rapid prototypes poorly coded

11.17 Metrics for the Requirements Workflow

- Volatility and speed of convergence are measures of how rapidly the client's needs are determined

Metrics for the Requirements Workflow

- The number of changes made during subsequent phases
- Changes initiated by the developers
  - Too many changes can mean the process is flawed
- Changes initiated by the client
  - Moving target problem

11.18 Challenges of the Requirements Phase

- Employees of the client organization often feel threatened by computerization
- The requirements team members must be able to negotiate
  - The client's needs may have to be scaled down
- Key employees of the client organization may not have the time for essential in-depth discussions
- Flexibility and objectivity are essential

Challenges

- Team must inform client to decide what is important
  - Developers may have to withdraw if no solution
- Flexibility and objectivity are essential for requirements elicitations
  - Should approach each interview with no preconceived ideas
  - Should never make assumptions about requirements

- The end!

SEIS 610

Agenda

- Review
- Classical Analysis
  - Specification Document (12.1)
  - Information Specifications (12.2)
  - Structured Systems Analysis (12.3)
  - Semi Formal Techniques (12.5)
  - Finite State Machines (12.7)
  - Petri Nets, Other Formal Methods (12.8, 12.9)
  - Comparison (12.11)

Review Requirements – Business Model (11.1-11.5)

- First: Understand the domain
  - Acquire familiarity with application domain (Important)
  - You can't hope to automate a process for somebody without understanding the problem they are trying to solve.
  - This is why we have a glossary!
- Business Model
  - Business model of the 'domain'
  - How do they make money?
  - Why is this product valuable

Review: Requirements – Business Model

- Why should/should software help?
  - How much will the software cost to create?
  - Is the software going to be sold or used internally?
  - If it is going to be sold as a product:
    - For how much?
    - How much do competitive products cost?
    - What features do competitive products have?
    - Where will this product fit within the product offerings?
    - Can you think of two people besides family members who will buy it??

Review: Business Model

- Initial Requirements
  - User Stories
    - **As a user, I would like to _____ so that I can _____**
  - Use cases
    - More formal, user/system interaction
- Functional Requirements
  - An action the target must be able to perform
  - Created during requirements and refined during analysis workflows
- Non-functional Requirements
  - Specifics related to the product itself
    - Platform constraints, response times, reliability
  - Best addressed during requirements and analysis, but may have to be handled during design.

Chapter 12

- Classical Analysis

Classical Analysis (12)

- Specification Document (12.1)
  - Documents should give engineers/developers and client/stakeholders a good understanding of what the product **must do**.
  - No matter what methodology is used
- This must contain acceptance criteria
- Solution strategies and plans are proposed.
- Bottom line
  1. Document gives good feeling about the product
  2. Document gives good feeling about acceptance criteria
  3. Document gives good feeling about a select solution strategy. (in class world)

Constraints

- Read page 360 & 361
- Where do constraints go in the Rational Unified Process?

Solution Strategy

- Where does the solution strategy go in the Rational Unified Process?

Informal Specifications (12.2)

- Written in a natural language such as English
- Page after page of text describing what the product should do.
- Weakness is ambiguity
  - Lawyers battle this constantly!
  - So do teachers
  - Most of us are not lawyers
  - No matter how well we explain in English we are always missing something.
- Realistically, correctness proof's such as 12.2.1 are not going to happen.

Informal Specifications

- What are these in the rational unified process?
- How do we battle ambiguity?

Glimpse: Analysis in OO/RUP

- Specification Document
  - Requirements document
    - Use cases, stories, FURPS, etc.
  - Design document
    - Sequence diagrams
    - Class diagrams
    - state, activity diagrams
  - Test Document
    - Detailed with traceability
- Informal Specification issues are mitigated with the 3-leg stool!
  - What you want, How we are gong to it, What it really does!

Structured Analysis (12.3)

- Data Flow Diagram (DFD)
- A graphical representation depicting the flow of data in an 'information' system.
- A tool analysis's can use to collect information necessary for system requirements.

Data Flow Diagrams

- Good sources
- http://web.simmons.edu/~benoit/lis486/s13/readings/Notes-Analysis-2
- https://www.visual-paradigm.com/tutorials/data-flow-diagram-dfd.jsp

Data Flow Diagram

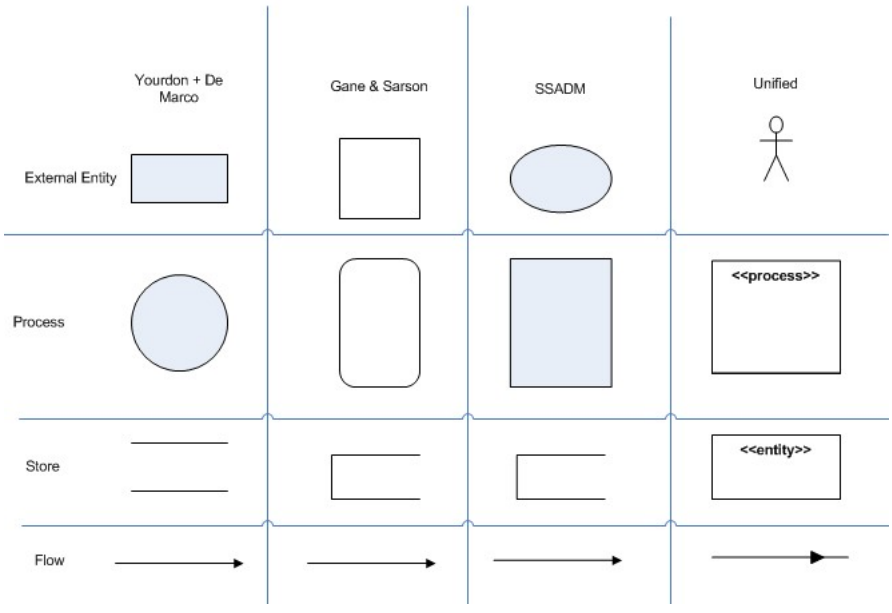
- Identify Data Flows
  - Requirements
    - Use cases/etc.
  - Rapid Prototype
- Data flows
  - Start with source or destination of data
  - OR Start with data store
- Data
  - Transformed by processes

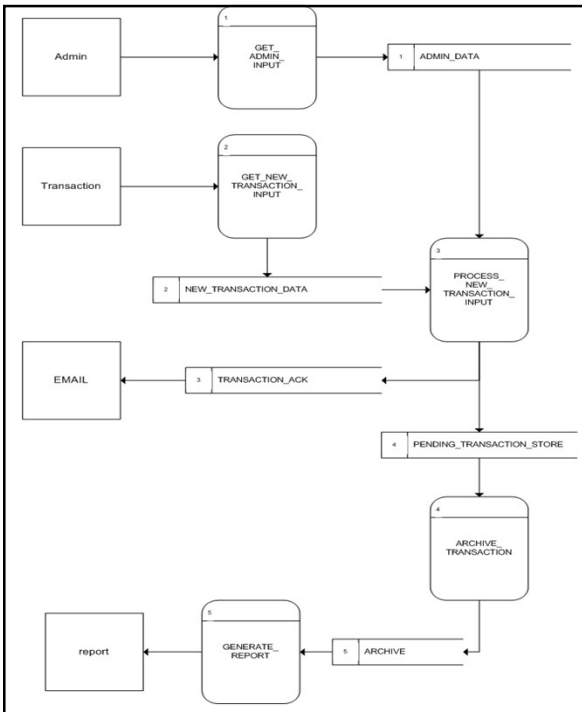
Structured Systems Analysis (Data Flow Diagrams) (12.3)

- Process oriented
- Help identify existing business processes
  - Hint: Think understanding the domain
- Can help re-engineer business processes
- Examines how data flows through the system

Data Flow Diagrams (comprised of)

- External Entity (source or destination-'sink' of data)
  - Human
  - System
  - Subsystem
  - External to the system we are studying
- Process
  - Business activity where manipulation and transformation of data occurs
- Data Store
  - Represents persistent storage





http://www.paulherber.co.uk/visio-dfd/examples.php

Data Flow Diagrams

- A process must have inputs
- A process must have outputs
- Source can have outputs
- Sink can have inputs
- Source/Sink can have inputs and outputs
- Data store can have inputs and outputs

Sources and Sinks (Source or Destination)

- Sources & sinks are referred to as external entities because they go outside the system.
- We don't need to consider the following:
  - Interactions that occur between sources and sinks
  - What a source or sink does with the data or how it operates (a black box)
  - How a control or redesign a source or sink since the info system deals with data as they are (what goes in and out of the box)
  - How to provide sources and sinks direct access to stored data because they cannot manipulate the data: the system must receive or distribute data between the system and its environment

Processes

- **No process can only have outputs**. That would be making data from nothing. If it only has outputs, it must be a source.
- **No process can only have inputs**. If it has only inputs, then it must be a sink.
- A process should have a verb in its name.

Data Store

- Data **can't** move from one store to another store, it must be moved by a process.
- Data cannot move directly from a source to a data store
- Data must be moved from a data store to a sink

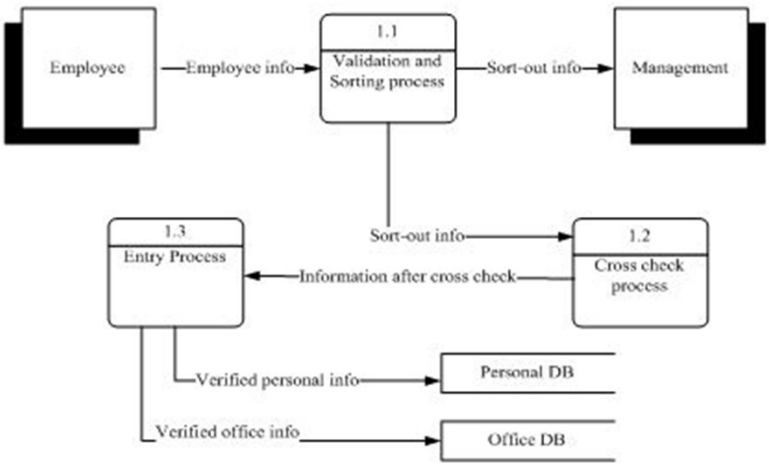
Practice Exercise

- Payroll DFD Diagram

How to perform structured systems analysis

1. Draw the dataflow diagram
2. Decide which sections to computerize
3. Determine the details of the data flows
4. Define the logic of the processes
5. Define the data stores
6. Define the physical resources
7. Determine IO specifications
8. Perform sizing
9. Determine the HW requirements

Sample Data Flow



How to perform structured systems analysis

1. Draw the dataflow diagram
2. Decide which 'processes' to computerize
3. Determine the details of the data flows
4. Define the logic of the processes
5. Define the data stores
6. Define the physical resources
7. Determine IO specifications
8. Perform sizing
9. Determine the HW requirements

How to perform structured systems analysis

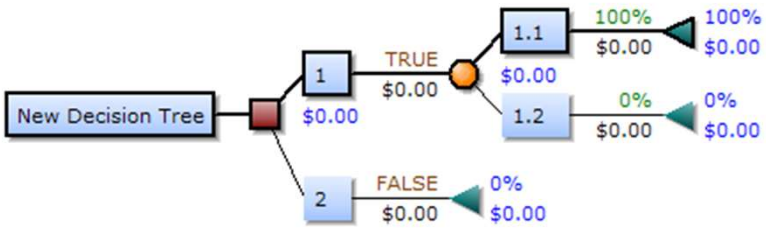
1. Draw the dataflow diagram
2. Decide which sections to computerize
3. Determine the details of the data flows
4. Define the logic of the processes
5. Define the data stores
6. Define the physical resources
7. Determine IO specifications
8. Perform sizing
9. Determine the HW requirements

How to perform structured systems analysis

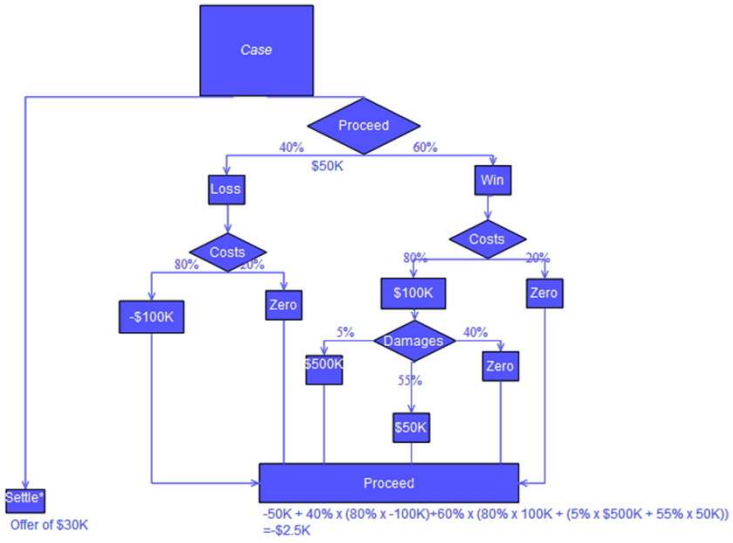
1. Draw the dataflow diagram
2. Decide which sections to computerize
3. Determine the details of the data flows
4. Define the logic of the processes (eek! How do I do that!?!!)
5. Define the data stores
6. Define the physical resources
7. Determine IO specifications
8. Perform sizing
9. Determine the HW requirements

Define the logic of the processes

- Decision Tree

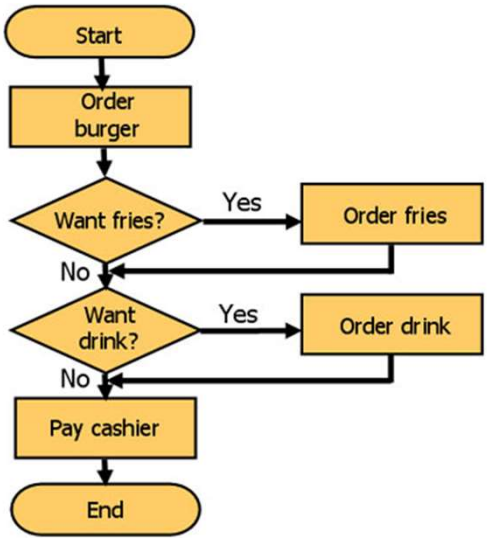


Decision Tree using flowchart symbols



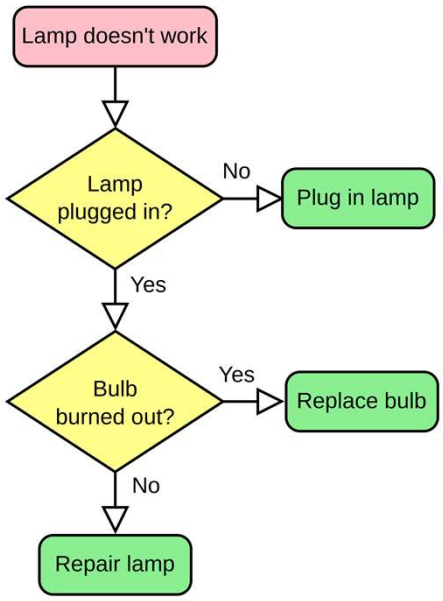
Activity Diagrams

- "Old fashioned flowcharts"
- Use for methods with difficult logic.



- http://www.teach-ict.com/as_a2_ict_new/ocr/A2_G063/331_systems_cycle/analysis_tools/miniweb/images/flowchart.jpg

Even cleaner flowchart



How to perform structured systems analysis

1. Draw the dataflow diagram
2. Decide which sections to computerize
3. Determine the details of the data flows
4. Define the logic of the processes
5. Define the data stores
6. Define the physical resources
7. Determine IO specifications
8. Perform sizing
9. Determine the HW requirements

How to perform structured systems analysis

1. Draw the dataflow diagram
2. Decide which sections to computerize
3. Determine the details of the data flows
4. Define the logic of the processes
5. Define the data stores
6. Define the physical resources
7. Determine IO specifications
8. Perform sizing
9. Determine the HW requirements

How to perform structured systems analysis

1. Draw the dataflow diagram
2. Decide which sections to computerize
3. Determine the details of the data flows
4. Define the logic of the processes
5. Define the data stores
6. Define the physical resources
7. Determine IO specifications
8. Perform sizing
9. Determine the HW requirements

How to perform structured systems analysis

1. Draw the dataflow diagram
2. Decide which sections to computerize
3. Determine the details of the data flows
4. Define the logic of the processes
5. Define the data stores
6. Define the physical resources
7. Determine IO specifications
8. Perform sizing
9. Determine the HW requirements

How to perform structured systems analysis

1. Draw the dataflow diagram
2. Decide which sections to computerize
3. Determine the details of the data flows
4. Define the logic of the processes
5. Define the data stores
6. Define the physical resources
7. Determine IO specifications
8. Perform sizing
9. Determine the HW requirements

Other semiformal techniques.

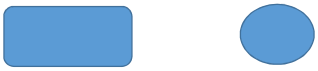
- PSL/PSA
- SADT
- SREM
- Structural Analysis
- Important that you know the existence of these
- Why???

Skip 12.6, Entity Relationship

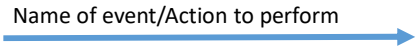
- Skip 12.6

12.7 Finite State Machines

- States represented by rectangle or circle

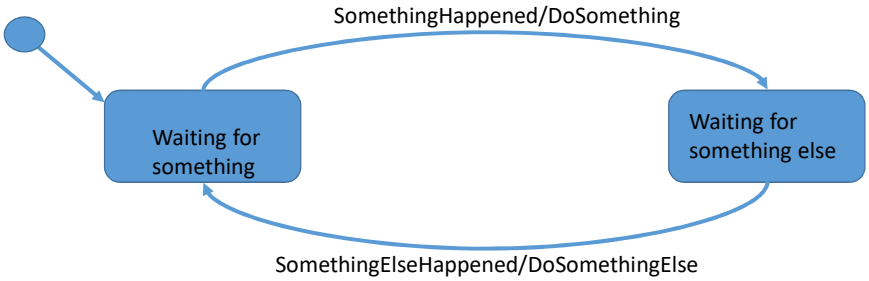


- Events Represented by arrows



- Event name and action performed are on top of arrow.
  - Sometimes action is not shown.

State Machine Basics



States:
1. Waiting for Something
2. Waiting for Something Else

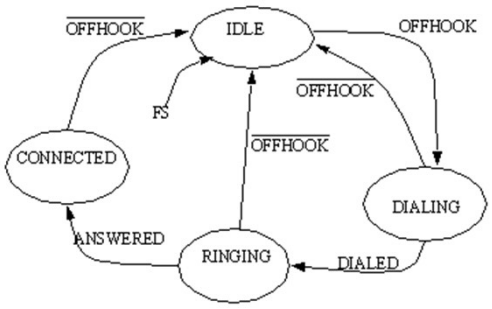
Events:
1. Something Happened
2. SomethingElseHappened

Actions
1. DoSomething
2. DoSomethingElse

1. We start in Waiting for Something.
2. S

Finite State Machines

- State Diagrams
- One way of representing interesting state logic.



http://www.thelearningpit.com/hj/plcs_files/plcs-353.gif

Finite State Machines

Toaster Example

Petri Nets

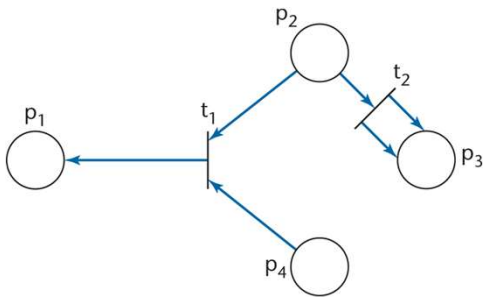
- Closely related to state machines
- Conceptually can deal with timing issues
  - Note, missing in state diagrams …BUT
  - Timeouts are events that can be scheduled to deal with timing issues

Petri Nets

- 4 Parts
  - Set of places
    - Circles
  - Set of transitions
    - Solid bar
  - Input function
    - Arrows from places
  - Output Function
    - Arrows to places

Petri Nets (contd)

- Set of places P is $\{p_1, p_2, p_3, p_4\}$
- Set of transitions T is $\{t_1, t_2\}$
- Input functions:
  $I(t_1) = \{p_2, p_4\}$
  $I(t_2) = \{p_2\}$
- Output functions:
  $O(t_1) = \{p_1\}$
  $O(t_2) = \{p_3, p_3\}$



Petri Nets (contd)

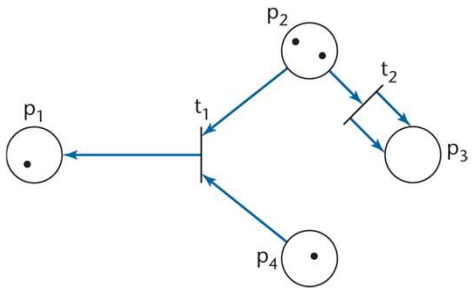


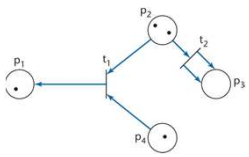
Figure 12.19

- Four tokens: one in p_1, two in p_2, none in p_3, and one in p_4
  - Represented by the vector (1, 2, 0, 1)
- A transition is enabled if each of its input places has as many tokens in it as there are arcs from the place to that transition

Petri Nets (contd)

- Transition t_1 is enabled (ready to fire)
  - If t_1 fires, one token is removed from p_2 and one from p_4, and one new token is placed in p_1
- Transition t_2 is also enabled
- Important:
  - The number of tokens **is not conserved**

Petri Nets (contd)



- Petri nets are indeterminate
  - Suppose t_1 fires

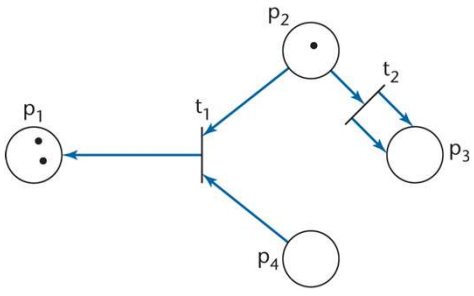


Figure 12.20

- The resulting marking is (2, 1, 0 ,0)

Petri Nets (contd)

- Now only t_2 is enabled
  - It fires

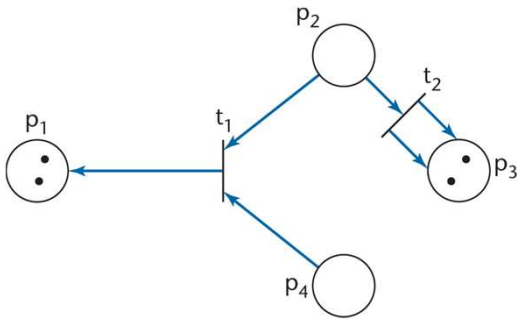


Figure 12.21

- The marking is now (2, 0, 2, 0)

Petri Nets

- Example

Other formal methods

- Z
  - Formal specification language
  - Next conference is July 2014, buy your tickets now!
- Anna
- Gist
- CSP
- VDM
- Once again, why should we know these exist?

Comparison of Specification Methods

- Very few of the exotic methods are in widespread use
- Analysis (and design) are problems software engineers continue to grapple with.
- If the method you are using is coming up short, it may be time to walk through engineering history
- Perhaps your problem has already been solved!

Comparison of Specification Methods

- Why study classical analysis, when object-oriented analysis is considered so much better?

Why study classical analysis, when object-oriented analysis is considered so much better?

- Understanding the problem domain.
- Understanding the business model.

- The end!

Implementation Chapter 15

SEIS 610

Agenda

- Choice of programming language (15.1)
- Good programming practice (15.3)
- Coding Standards (15.4)
- Implementation and Integration (15.6)
  - Top-down, bottom-up, and sandwich implementation and integration (15.6.1, 15.6.2)
- Testing during implementation and Integration (15.9, 15.10, 15.11, 15.3)
- Integration testing (15.20)
- Product testing (15.21)
- Acceptance Testing (15.22)

Choice of Programming Language

- Two times this issue comes up
  - Beginning of a brand new project
  - Rewriting 'ancient' works
    - So essentially a new project
- You would not normally switch in the middle of development
  - Though you might pick a new toolset for the same language

Language 'Taxonomy'

- First generation languages
  - Machine languages
- Second generation languages
  - Assemblers
- Third generation languages
  - High-level languages (COBOL, FORTRAN, C++, Java)
- Forth generation
  - Goal: Anybody can program!

Choice of Programming Language

- Do you really get to pick?
- Normally few choices
  - How is the target programmed?
    - Example: Android uses java
- What development tools are presently used in your organization?

Choice of Programming Language

- Does it ever make sense to move from something the teams are used to?
  - Times change
  - Technologies change
  - Standards change
- Sometime you have to “bight the bullet”

Choice of Programming Language

- Suppose you were moving your application to the 'cloud'
- You could possibly be starting from scratch
- Google App Engine (Platform as a service)
  - Python
  - Java
  - Go

Choice of Programming Language

- If you are switching environments
  - New environment means there **must be some benefit**
  - Does is solve a problem?
  - Does it really make life better?
  - Can it expand your product features or reach?
- There has to be a reason to switch

Choice of Programming Language

- Bottom Line
  - Tools already in house may be adequate, so no choice
  - Target has specific tool requirements, so no choice
  - Long standing problem exists, new language solves
    - Example, designing UI's with Cobol really hard
    - Designing UI's with C# really easy

Next slides form

- http://blog.teamtreehouse.com/choose-programming-language
- Discussion of languages

HTML/CSS

- People often begin by learning HTML and CSS. Why? These two languages are essential for creating static web pages. HTML (Hypertext Markup Language) structures all the text, links, and other content you see on a website.
- CSS is the language that makes a web page look the way it does—color, layout, and other the visuals we call style.
- Really no “logic artifacts”

PHP

- PHP is one of the most popular web languages.
- It runs massive sites such as Facebook and Etsy. WordPress and Drupal are both written in PHP, and those two platforms power a huge number of the sites online today.
- Because of its popularity, learning PHP will serve you well if you intend to code for the Web.

Java Script

- JavaScript is the first full programming language for many people.
- Why? It is the logical next step after learning HTML and CSS. JavaScript provides the behavior portion of a website. For example, when you see a form field indicate an error, that's probably JavaScript at work.
- JavaScript has become increasingly popular, and it now lives outside web browsers as well.
- Learning JavaScript will put you in a good place as it becomes a more general-purpose language.
- JavaScript seems to be everywhere lately.

Java Script

- An important component of cloud computing
- HTML5
- Etc.

Python

- Python is a general-purpose language
  - used for everything from server automation to data science.
- Python is a great language for beginners
  - it is easy to read and understand.
- You can also do so many things with Python that it's easy to stick with the language for quite a while before needing something else.
- Python finds itself at home both creating Web apps like Instagram and helping researchers make sense of their data.

Python

- Google App Engine likes Python!

Ruby

- Ruby is often associated with the Rails framework that helped popularize it.
- Used widely among web startups and big companies alike, Ruby and Rails jobs are pretty easy to come by.
- Ruby and Rails make it easy to transform an idea into a working application, and they have been used to bring us Twitter, GitHub, and Treehouse.

Swift

- Apple released Swift in June, 2014 as a modern language for developing Mac, iPad, iPhone, Apple Watch, and Apple TV applications.
- If you want to enter the world of iOS, Swift is the language with which Apple intends to move forward.
- Yes, many apps are already written in Objective-C, but Swift is here to stay.
- If the Apple ecosystem lures you in, you'll need some understanding of both Objective-C and Swift.

Objective-C

- Like Java, Objective-C can be used to write desktop software and mobile apps. However, Objective-C is essentially Apple territory.
- Until the recent release of the Swift programming language, Objective-C was the language for developing native iPhone and iPad apps.
- Many major apps are still written in Objective-C, and programmers for these apps are in high-demand.
- If you want to work on iPhone and iPad apps, it's a good idea to learn Objective-C.

Java

- Despite its name, Java is not related to JavaScript in any meaningful way.
- Java can be used for anything from web applications to desktop and mobile apps.
- Java has a strong presence among large enterprise applications—think bank, hospital, and university software.
- It *still* powers Android apps, so it's a good choice for those inclined toward mobile development.

Kotlin

- Addresses several Java issues
- Does not mimic java / c syntax
- Secondary (as of now) language for Android

```
fun eatACake() = println("Eat a Cake")
fun bakeACake() = println("Bake a Cake")

fun main(args: Array<String>) {
    var cakesEaten = 0
    var cakesBaked = 0

    while (cakesEaten < 5) {                    // 1
        eatACake()
        cakesEaten ++
    }

    do {                                        // 2
        bakeACake()
        cakesBaked++
    } while (cakesBaked < cakesEaten)

}
```

https://play.kotlinlang.org/byExample/02_control_flow/02_Loops

Good Programming Practice (15.3)

- Use of consistent and meaningful variable names
- “Self documenting code”
  - We always thought this a joke.
- Avoid “magic numbers” (15.3.3)
  - “Use of parameters”
- Code layout (15.3.4)
- Nested if (15.3.5)

Good Programming Practice (15.3)

- Use of *consistent* and *meaningful* variable names
  - "Meaningful" to future maintenance programmers
  - "Consistent" to aid future maintenance programmers

Good Programming Practice (15.3)

- A code artifact includes the variable names `freqAverage`, `frequencyMaximum`, `minFr`, `frqncyTotl`
- A maintenance programmer has to know if `freq`, `frequency`, `fr`, `frqncy` all refer to the same thing
  - If so, use the identical word, preferably `frequency`, perhaps `freq` or `frqncy`, but *not* `fr`
  - If not, use a different word (e.g., `rate`) for a different quantity

Good Programming Practice (15.3)

- We can use `frequencyAverage, frequencyMaximum, frequencyMinimum, frequencyTotal`
- We can also use `averageFrequency, maximumFrequency, minimumFrequency, totalFrequency`
- But all four names must come from the same set
- So be consistent!

Good Programming Practice (15.3)

- **"Self documenting code" --- We always thought this a joke.**
- **You should never need to write a comment to explain what you're doing.**
- **If you feel you have to, rewrite the code until it doesn't need explaining in a comment.**

```
# this method manipulates dimensions
def method_x(a, b, c, d)
  # a is the width
  # b is the height
  # c is the depth
  # d is a list of options
  ...
end
```

That could obviously be re-written as:

```
def manipulate_dimensions(width, height, depth, options)
  ...
end
```

https://news.ycombinator.com/item?id=4381371

Good Programming Practice (15.3)

- Comments are essential whenever the code is written in a non-obvious way, or makes use of some subtle aspect of the language
- Nonsense!
  - Recode in a clearer way
  - We must never promote/excuse poor programming
  - However, comments can assist future maintenance programmers

Good Programming Practice (15.3)

- Avoid “magic numbers” (15.3.3) -“Use of parameters”
- A 'magic number' is a literal value that appears in a program. For example:

```
total = 1.08 * price;
```

- 1.08 is a magic number because it appears out of the blue, and it's unclear from this line of code what it means. It's generally better to replace magic numbers with NamedConstants, e.g.

```
const double TAX_RATE_IN_TEXAS = 1.08;
total = TAX_RATE_IN_TEXAS * price;
```

- Even better, this should be stored in a configuration file
- http://c2.com/cgi/wiki?MagicNumber

Good Programming Practice (15.3)

- Code layout (15.3.4)
- Each language has an established layout
- Easy to find guidelines
- http://tech.dolhub.com/article/computer/code-Layout
- http://geosoft.no/development/javastyle.html
- http://www.infoq.com/news/2013/08/objective-c-coding-style

Code Layout – nice to read??

```
        try (BufferedReader inputReader = Files.newBufferedReader(
                        Paths.get(new URI
                          ("file:///C:/home/docs/users.txt")),
                        Charset.defaultCharset());
                    BufferedWriter outputWriter = Files.
newBufferedWriter(
                        Paths.get(new URI("file:///C:/home/docs/
users.bak")),
                        Charset.defaultCharset())) {
             String inputLine;
             while ((inputLine = inputReader.readLine()) != null) {
                 outputWriter.write(inputLine);
                 outputWriter.newLine();
}
             System.out.println("Copy complete!");
}
catch (URISyntaxException | IOException ex) {
             ex.printStackTrace();
}
```

From google images

Nested if

```
if (latitude > 30 && longitude > 120) {if (latitude <= 60 && longitude <= 150)
mapSquareNo = 1; else if (latitude <= 90 && longitude <= 150) mapSquareNo = 2
else print "Not on the map";} else print "Not on the map";
```

- So how do you test something like this?
- How do you visually verify it is correct?

```
if (latitude > 30 && longitude > 120)
{
   if (latitude <= 60 && longitude <= 150)
      mapSquareNo = 1;
   else
      if (latitude <= 90 && longitude <= 150)
         mapSquareNo = 2;
      else
         print "Not on the map";
}
else
   print "Not on the map";
```

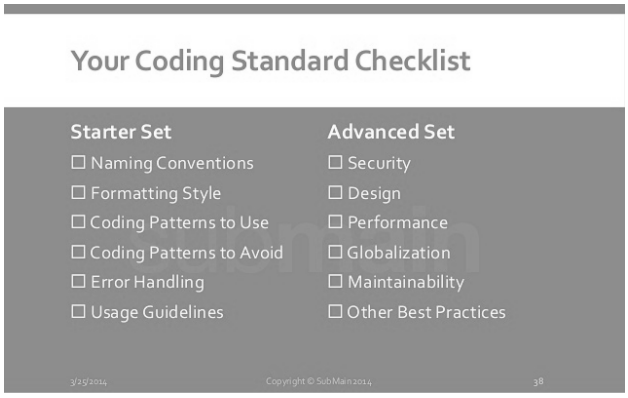
Good Programming Practice (15.3)

- Nested if (15.3.5)

```
public Boolean userHasAccessToFeature( User u, Feature f) {

        Boolean result = false;
        if ( u.isAdminUser() || u.isSystemAccount()) {
                result = true;
        }
        else {
                Boolean isSecured = f.isSecured();
                if ( !isSecured) {
                        result = true;
                }
                else {
                        List<Integer> acceptableProfiles = f.getAssignedProfiles();
                        If ( acceptableProfiles != null && acceptableProfiles.size() > 0) {
                                List<Integer> assignedProfiles = u.getAssignedProfiles();
                                If ( assignedProfiles != null && assignedProfiles.size() > 0) {
                                        for (int idx = 0; idx < assignedProfiles.size(); idx++) {
                                                Integer profile = assignedProfiles.get(idx);
                                                if ( acceptableProfiles.contains( profile) {
                                                        result = true;
                                                        break;
                                                }
                                        }
                                }
                        }
                }
        }
}
```

Coding Standards (15.4)

- Bundles all the above into a nice neat package
- Your organization might have one
- Ask for it.



Remarks on Programming Standards

- The aim of standards is to make maintenance easier
  - If they make development difficult, then they must be modified
  - Overly restrictive standards are counterproductive
  - The quality of software suffers

15.6.1 Top-down Integration

- If code artifact `mAbove` sends a message to artifact `mBelow`, then `mAbove` is implemented and integrated before `mBelow`
- One possible top-down ordering is
  - `a, b, c, d, e, f, g, h, i, j, k, l ,m`

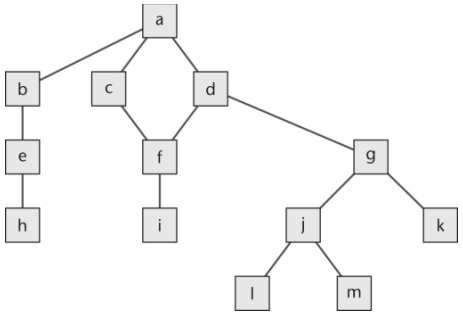


Figure 15.6 (again)

Top-down Integration (contd)

- Advantage 1: Fault isolation
  - A previously successful test case fails when `mNew` is added to what has been tested so far
    - The fault must lie in `mNew` or the interface(s) between `mNew` and the rest of the product
- Advantage 2: Stubs are not wasted
  - Each stub is expanded into the corresponding complete artifact at the appropriate step

Top-down Integration

- Advantage 3: Major design flaws show up early
- Logic artifacts include the decision-making flow of control
  - In the example, artifacts `a, b, c, d, g, j`
- Operational artifacts perform the actual operations of the product
  - In the example, artifacts `e, f, h, i, k, l, m`
- The logic artifacts are developed before the operational artifacts

Top-down Integration

- Problem 1
  - Reusable artifacts are not (necessarily) properly tested
  - Lower level (operational) artifacts are not tested frequently
  - The situation is aggravated if the product is well designed
- Defensive programming (fault shielding)
  - Example:

```
if (x >= 0)
   y = computeSquareRoot (x, errorFlag);
```

  - IF `computeSquareRoot` is never tested with `x < 0`
  - This has implications for reuse

15.6.2 Bottom-up Integration

- If code artifact `mAbove` calls code artifact `mBelow`, then `mBelow` is implemented and integrated before `mAbove`
- One possible bottom-up ordering is

```
l, m, h, i, j, k, e,
f, g, b, c, d, a
```

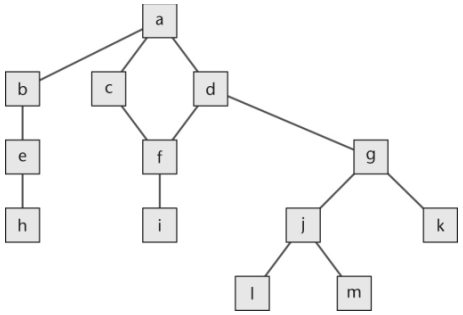


Figure 15.6 (again)

15.6.2 Bottom-up Integration

- Another possible bottom-up ordering is

```
h, e, b
i, f, c, d
l, m, j, k, g      [d]
a     [b, c, d]
```

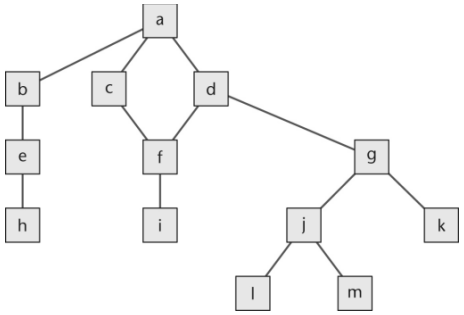


Figure 15.6 (again)

Bottom-up Integration (contd)

- Advantage 1
  - Operational artifacts are thoroughly tested
- Advantage 2
  - Operational artifacts are tested with drivers, not by fault shielding, defensively programmed artifacts
- Advantage 3
  - Fault isolation

Bottom-up Integration

- Difficulty 1
  - Major design faults are detected late
- Solution
  - Combine top-down and bottom-up strategies making use of their strengths and minimizing their weaknesses

15.9 The Test Workflow: Implementation

- Unit testing
  - Minimally: Informal unit testing by the programmer
  - Methodical unit testing by the SQA group
- There are two types of methodical unit testing
  - Non-execution-based testing
  - Execution-based testing

15.10 Test Case Selection

- Worst way — random testing
  - "Haphazard test data"
  - There is likely no time to test all paths
  - **BUT random data testing not completely bad**
- We need a systematic way to construct test cases

15.10.1 Testing to Specifications versus Testing to Code

- There are two extremes to testing
- *Test to specifications* (also called black-box, data-driven, functional, or input/output driven testing)
  - Ignore the code — use the specifications to define test cases
- *Test to code* (also called glass-box, logic-driven, structured, or path-oriented testing)
  - Ignore the specifications — use the code to define test cases

15.10.2 Feasibility of Testing to Specifications

- Example:
  - The specifications for a data processing product include 5 types of commission and 7 types of discount
  - 35 test cases
- We cannot say that commission and discount are computed in two entirely separate artifacts
- Remember: blackbox testing… the structure is irrelevant

Feasibility of Testing to Specifications

- Suppose the specifications include 20 factors, each taking on 4 values
  - There are 4^{20} or 1.1 *10^{12} possible test cases
  - If each takes 30 seconds to run, running all test cases takes more than 1 million years
- The "combinatorial explosion" makes testing to specifications impractical

15.10.3 Feasibility of Testing to Code

- Assume each path through a artifact must be executed at least once
  - Combinatorial explosion

Feasibility of Testing to Code

- Code example:

```
read (kmax)                          // kmax is an integer between 1 and 18
for (k = 0; k < kmax; k++) do
{
   read (myChar)                     // myChar is the character A, B, or C
   switch (myChar)
   {
     case 'A':
       blockA;
       if (cond1) blockC;
       break;
     case 'B':
       blockB;
       if (cond2) blockC;
       break;
     case 'C':
       blockC;
       break;
   }
   blockD;
}
```

Figure 15.9

Feasibility of Testing to Code

- The flowchart has over 10^{12} different paths

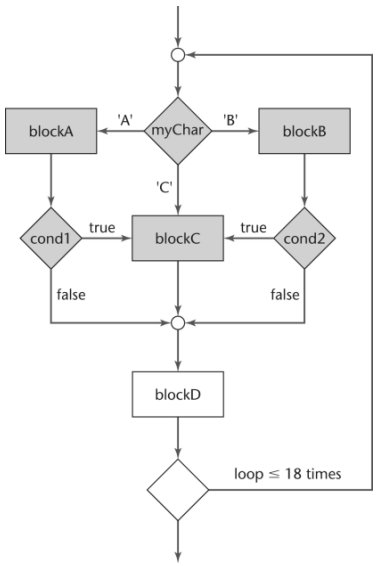


Figure 15.10

Feasibility of Testing to Code

- Testing to code may not be reliable

```
if ((x + y + z)/3 == x)
   print "x, y, z are equal in value";
else
   print "x, y, z are unequal";
```

Test case 1: x = 1, y = 2, z = 3
Test case 2: x = y = z = 2

- We can exercise every path without detecting every fault

Feasibility of Testing to Code

- A path can be tested only if it is present
- A programmer who omits the test for d = 0 in the code probably is unaware of the possible danger

```
if (d == 0)
   zeroDivisionRoutine ();
else
   x = n/d;
```

(a)

```
x = n/d;
```

(b)

Figure 15.12

Feasibility of Testing to Code

- Criterion "exercise all paths" is not *reliable*
  - Products exist for which some data exercising a given path detect a fault, and other data exercising the same path do not

15.11 **Black-Box** and Unit-testing Techniques

- Neither exhaustive testing to specifications nor exhaustive testing to code is feasible
- The art of testing:
  - Select a small, manageable set of test cases to
  - Maximize the chances of detecting a fault, while
  - Minimizing the chances of wasting a test case
- Every test case must detect a previously undetected fault

Black-Box and Unit-testing Techniques

- We need a method that will highlight as many faults as possible
  - First black-box test cases (testing to specifications)
  - Then glass-box methods (testing to code)

Equivalence Testing

- Any one member of an equivalence class is as good a test case as any other member of the equivalence class
- Range (1..16,383) defines three different equivalence classes:
  - Equivalence Class 1: Fewer than 1 record
  - Equivalence Class 2: Between 1 and 16,383 records
  - Equivalence Class 3: More than 16,383 records

Boundary Value Analysis

- Select test cases on or just to one side of the boundary of equivalence classes
  - This greatly increases the probability of detecting a fault

Equivalence Testing of Output Specifications

- We also need to perform equivalence testing of the output specifications
- Example:

  In 2008, the minimum Social Security (OASDI) deduction from any one paycheck was $0, and the maximum was $6,324
  - Test cases must include input data that should result in deductions of exactly $0 and exactly $6,324
  - Also, test data that might result in deductions of less than $0 or more than $6,324
  - **Test cases for something in between!**

Overall Strategy

- Equivalence classes together with boundary value analysis to test both input specifications and output specifications
  - This approach generates a small set of test data with the potential of uncovering a large number of faults

15.11.2 Functional Testing

- Functions are tested by feeding them input and examining the output, and internal program structure is rarely considered.

15.11.2 Functional Testing

- An alternative form of black-box testing for classical software
  - We base the test data on the functionality of the code artifacts
- Each item of functionality or function is identified
- Test data are devised to test each (lower-level) function separately
- Then, higher-level functions composed of these lower-level functions are tested

Functional Testing

- In practice, however
  - Higher-level functions are not always neatly constructed out of lower-level functions using the constructs of structured programming
  - Instead, the lower-level functions are often intertwined
- Also, functionality boundaries do not always coincide with code artifact boundaries
  - The distinction between unit testing and integration testing becomes blurred
  - This problem also can arise in the object-oriented paradigm when messages are passed between objects

15.13 Glass-Box *Unit-Testing* Techniques

- We will examine
  - Statement coverage
  - Branch coverage
  - Path coverage
  - Linear code sequences
  - All-definition-use path coverage

15.13.1 Structural Testing: Statement, Branch, and Path Coverage

- *Statement coverage*:
  - Running a set of test cases in which every statement is executed at least once
  - Tools needed to keep track
- Weakness
  - Branch statements
- Both statements can be executed without the fault showing up

```
if (s > 1 && t == 0)
   x = 9;
```

Test case: s = 2, t = 0.

Figure 15.15

Structural Testing: Branch Coverage

- Running a set of test cases in which every branch is executed at least once (as well as all statements)
  - This solves the problem on the previous slide
  - Again, tools are needed to keep track

Structural Testing: Path Coverage

- Running a set of test cases in which every path is executed at least once (as well as all statements)
- Problem:
  - The number of paths may be very large
- We want a weaker condition than all paths but that shows up more faults than branch coverage

Linear Code Sequences

- Identify the set of points L from which control flow may jump, plus entry and exit points
- Restrict test cases to paths that begin and end with elements of L
- This uncovers many faults without testing every path

All-Definition-Use-Path Coverage

- Each occurrence of variable, `zz` say, is labeled either as
  - The *definition* of a variable

    `zz = 1` or `read (zz)`
  - or the *use* of variable

    `y = zz + 3` or `if (zz < 9) errorB ()`
- Identify all paths from the definition of a variable to the use of that definition
  - This can be done by an automatic tool
- A test case is set up for each such path

All-Definition-Use-Path Coverage

- Disadvantage:
  - Upper bound on number of paths is 2^d, where d is the number of branches
- In practice:
  - The actual number of paths is proportional to d
- This is therefore a practical test case selection technique

Infeasible Code

- It may not be possible to test a specific statement
  - We may have an infeasible path ("dead code") in the artifact
- Frequently this is evidence of a fault

```
if (k < 2)
{
  if (k > 3)              [should be k > –3]
       ↑
    x = x * k;
}
```

(a)

```
for (j = 0; j < 0; j++)   [should be j < 10]
          ↑
  total = total + value[j];
```

(b)

15.13.2 Complexity Metrics

- A quality assurance approach to glass-box testing
- Artifact `m1` is more "complex" than artifact `m2`
  - Intuitively, `m1` is more likely to have faults than artifact `m2`
- If the complexity is unreasonably high, redesign and then reimplement that code artifact
  - This is cheaper and faster than trying to debug a fault-prone code artifact

Lines of Code

- The simplest measure of complexity
  - Underlying assumption: There is a constant probability p that a line of code contains a fault
- Example
  - The tester believes each line of code has a 2% chance of containing a fault.
  - If the artifact under test is 100 lines long, then it is expected to contain 2 faults
- The number of faults is indeed related to the size of the product as a whole

Other Measures of Complexity

- Cyclomatic complexity *M* (McCabe)
  - Essentially the number of decisions (branches) in the artifact
  - Easy to compute
  - A surprisingly good measure of faults (but see next slide)
- In one experiment, artifacts with *M* > 10 were shown to have statistically more errors

Problem with Complexity Metrics

- Complexity metrics, as especially cyclomatic complexity, have been strongly challenged
- SO WHAT
- It is a tool and can be a useful tool.
- It should not make final decisions for you

Code Walkthroughs and Inspections

- Code reviews lead to rapid and thorough fault detection
  - Up to 95% reduction in maintenance costs

15.20 Integration Testing

- The testing of each new code artifact when it is added.
  - to what has already been tested
- Special considerations testing graphical user interfaces

Integration Testing of Graphical User Interfaces

- GUI test cases include
  - Mouse clicks, and
  - Key presses
- These types of test cases cannot be stored in the usual way
  - We need special tools
- Examples:
  - Selenium
  - http://www.seleniumhq.org/

15.21 Product Testing

- Product testing for COTS software
  - Alpha, beta testing
- Product testing for Custom Software
  - The SQA group must ensure that the product passes the acceptance test
  - Failing an acceptance test has bad consequences for the development organization

Product Testing for Custom Software

- The SQA team must try to approximate the acceptance test
  - Black box test cases for the product as a whole
  - Robustness of product as a whole
    - *Stress testing* (under peak load)
    - *Volume testing* (e.g., can it handle large input files?)
  - All constraints must be checked
  - All documentation must be
    - Checked for correctness
    - Checked for conformity with standards
    - Verified against the current version of the product

Product Testing for Custom Software

- The product (code plus documentation) is now handed over to the client organization for acceptance testing

15. 22 Acceptance Testing

- The client determines whether the product satisfies its specifications
- Not that just if the product is broken, but does it do what the client needs

- Acceptance testing is performed by
  - The client organization, or
  - The SQA team in the presence of client representatives, or
  - An independent SQA team hired by the client

Acceptance Testing (contd)

- The four major components of acceptance testing are
  - **Correctness**
  - Robustness
  - Performance
  - Documentation

- These are precisely what was tested by the developer during product testing

Acceptance Testing (contd)

- The key difference between product testing and acceptance testing is
  - Acceptance testing is performed on actual data (maybe)
  - Product testing is preformed on test data, which can never be real, by definition
  - Acceptance testing is performed to see that the software does what it is supposed to, not just that it works.

The End

CHAPTER 16

POSTDELIVERY MAINTENANCE

Agenda

- Development and Maintenance (16.1)
- Why Postdelivery Maintenance is Necessary (16.2)
- What Is Required of Postdelivery Maintenance Programmers? (16.3)
- Management of Postdeliver Maintenance (16.5)
- Postdelivery Maintenance Skills vs. Devleopment Skills (16.7)
- Testing During Post Delivery Maintenance (16.9)
- Metrics for Postdelivery Maintenance (16.11)

16.1 Postdelivery Maintenance

- Postdelivery maintenance
  - *Any* change to *any* component of the product (including documentation) after it has passed the acceptance test
- This is a short chapter
  - But the whole book is essentially on postdelivery maintenance
- In this chapter we explain how to ensure that maintainability is not compromised during postdelivery maintenance

16.2 Why Postdelivery Maintenance Is Necessary

- **Corrective maintenance**
  - To correct residual faults
    - Analysis, design, implementation, documentation, or any other type of faults

Why Postdelivery Maintenance Is Necessary

- Perfective maintenance
  - Client requests changes to improve product effectiveness
    - Add additional functionality
    - Make product run faster
    - Improve maintainability

Why Postdelivery Maintenance Is Necessary.

- Adaptive maintenance
  - Responses to changes in the environment in which the product operates
    - The product is ported to a new compiler, operating system, and/or hardware
    - A change to the tax code
    - 9-digit ZIP codes

16.3 What Is Required of Postdelivery Maintenance Programmers?

- At least 67% of the total cost of a product accrues during postdelivery maintenance
- Maintenance can be a major income source
- Nevertheless, even today many organizations assign maintenance to
  - Unsupervised beginners, and
  - Less competent programmers

16.3 What is Required of Postdelivery Maintenance Programmer ?

- Postdelivery maintenance is one of the most difficult aspects of software production because
  - Postdelivery maintenance incorporates aspects of all other workflows

16.3 What is Required of Postdelivery Maintenance Programmer ?

- Corrective Maintenance
- What tools does the maintenance programmer have to find the fault?
  - The defect report filed by user
  - The source code
  - And often nothing else

16.3 What is Required of Postdelivery Maintenance Programmer ?

- A maintenance programmer must therefore have superb debugging skills
  - The fault could lie anywhere within the product
  - The original cause of the fault might lie in the by now non-existent specifications or design documents

16.3 What is Required of Postdelivery Maintenance Programmer ?

- Suppose that the maintenance programmer has located the fault
- Problem:
  - How to fix it without introducing a regression fault

16.3 What is Required of Postdelivery Maintenance Programmer ?

- How to minimize regression faults
  - Consult the detailed documentation for the product as a whole
  - Consult the detailed documentation for each individual module
- What usually happens
  - There is no documentation at all, or
  - The documentation is incomplete, or
  - The documentation is faulty

16.3 What is Required of Postdelivery Maintenance Programmer ?

- The programmer must deduce from the source code itself all the information needed to avoid introducing a regression fault
- The programmer now changes the source code

16.3 What is Required of Postdelivery Maintenance Programmer ?

- The Programmer Now Must
- Test that the modification works correctly
  - Using specially constructed test cases
- Check for regression faults
  - Using stored test data
- Add the specially constructed test cases to the stored test data for future regression testing
- Document all changes

16.3 What is Required of Postdelivery Maintenance Programmer ?

- Major skills are required for corrective maintenance
  - Superb diagnostic skills
  - Superb testing skills
  - Superb documentation skills

16.3 What is Required of Postdelivery Maintenance Programmer ?

- Adaptive and Perfective Maintenance
- The maintenance programmer must go through the
  - Requirements
  - Specifications
  - Design
  - Implementation and integration

  workflows, using the existing product as a starting point

16.3 What is Required of Postdelivery Maintenance Programmer ?

- Adaptive and Perfective Maintenance
- When programs are developed
  - Specifications are produced by analysis experts
  - Designs are produced by design experts
  - Code is produced by programming experts
- But a maintenance programmer must be expert in all three areas, and also in
  - Testing, and
  - Documentation

The Rewards of Maintenance

- Maintenance is a thankless task in every way
  - Maintainers deal with dissatisfied users
  - If the user were happy, the product would not need maintenance
  - The user's problems are often caused by the individuals who developed the product, not the maintainer
  - The code itself may be badly written
  - Postdelivery maintenance is despised by many software developers
  - Unless good maintenance service is provided, the client will take future development business elsewhere
  - Postdelivery maintenance is the most challenging aspect of software production — and the most thankless

16.3 What is Required of Postdelivery Maintenance Programmer ?

- No form of maintenance
  - Is a task for an unsupervised beginner, or
  - Should be done by a less skilled computer professional

16.5 Management of Postdelivery Maintenance

- Various issues regarding management of postdelivery maintenance are now considered

16.5.1 Management of Postdelivery Maintenance

- Defect Reports
- We need a mechanism for changing a product
- If the product appears to function incorrectly, the user files a defect report
  - It must include enough information to enable the maintenance programmer to recreate the problem
- Ideally, every defect should be fixed immediately
  - In practice, an immediate preliminary investigation is the best we can do

If the Defect Has Been Previously Reported

- How was it handled?
- Is there a work-around?
- Is there a patch?

If it Is a New Defect

- The maintenance programmer should try to find
  - The cause,
  - A way to fix it, and
  - A way to work around the problem
- The new defect is now filed in the defect report file, together with supporting documentation
  - Listings
  - Designs
  - Manuals

Management of Postdelivery Maintenance

- In an ideal world
  - We fix every defect immediately
  - Then we distribute the new version of the product to all the sites
- In the real world
  - We distribute defect reports to all sites
  - We do not have the staff for instant maintenance
  - It is more practical/cheaper to make a number of changes at the same time, particularly if there are multiple sites

16.5.2 Authorizing Changes to the Product

- Corrective maintenance
  - Assign a maintenance programmer to determine the fault and its cause, then repair it
  - Test the fix, test the product as a whole (regression testing)
  - Update the documentation to reflect the changes made
  - Update the prologue comments to reflect
    - What was changed,
    - Why it was changed,
    - By whom, and
    - When

16.5.2 Authorizing Changes to the Product

- Adaptive and perfective maintenance
  - Same as with corrective maintenance, except there is no defect report
  - There is a change in requirements instead

Authorizing Changes to the Product

- What if the programmer has not tested the fix adequately?
  - Before the product is distributed, it must be tested by the SQA group
- Postdelivery maintenance is extremely hard
- Testing is difficult and time consuming
  - Performed by the SQA group

Authorizing Changes to the Product

- The technique of baselines and private copies must be followed
- The programmer makes changes to private copies of code artifacts, tests them
- The programmer freezes the previous version, and gives the modified version to SQA to test
- SQA performs tests on the current baseline version of all code artifacts

16.5.3 Ensuring Maintainability

- Maintenance is not a one-time effort
- We must plan for maintenance over the entire life cycle
  - Design workflow — use information-hiding techniques
  - Implementation workflow — select variable names meaningful to future maintenance programmers
  - Documentation must be complete and correct, and reflect the current version of every artifact

16.5.3 Ensuring Maintainability

- During postdelivery maintenance, maintainability must not be compromised
  - Always be conscious of the inevitable further maintenance
- Principles leading to maintainability are equally applicable to postdelivery maintenance itself

16.5.4 Problem of Repeated Maintenance

- The moving target problem
- The problem is exacerbated during postdelivery maintenance
- The more changes there are
  - The more the product deviates from its original design
  - The more difficult further changes become
  - Documentation becomes even less reliable than usual
  - Regression testing files are not up to date
  - A total rewrite may be needed for further maintenance

16.5.4 Problem of Repeated Maintenance

- The moving target problem
- Apparent solution
  - Freeze the specifications once they have been signed off until delivery of the product
  - After each request for perfective maintenance, freeze the specifications for (say) 3 months or 1 year
- In practice
  - The client can order changes the next day
  - If willing to pay the price, the client can order changes on a daily basis
- "He who pays the piper calls the tune"

16.6 Maintenance of Object-Oriented Software

- In general
  - Can be better partitioned
  - Can have better cohesion
  - Can be easier to understand
- It is easier to maintain, in general
- However, even object-oriented can be done badly.
- Page 560

16.7 Postdelivery Maintenance versus Development Skills

- The skills needed for maintenance include
  - The ability to determine the cause of failure of a large product
    - Also needed during integration and product testing
  - The ability to function effectively without adequate documentation
    - Documentation is rarely complete until delivery
  - Skills in analysis, design, implementation, and testing
    - All four activities are carried out during development

Postdelivery Maintenance vs. Development Skills

- The skills needed for postdelivery maintenance are the same as those for the other workflows
- Key Point
  - Maintenance programmers must not merely be skilled in a broad variety of areas, they must be *highly* skilled in *all* those areas
  - Specialization is impossible for the maintenance programmer
- Postdelivery maintenance is the same as development, only more so

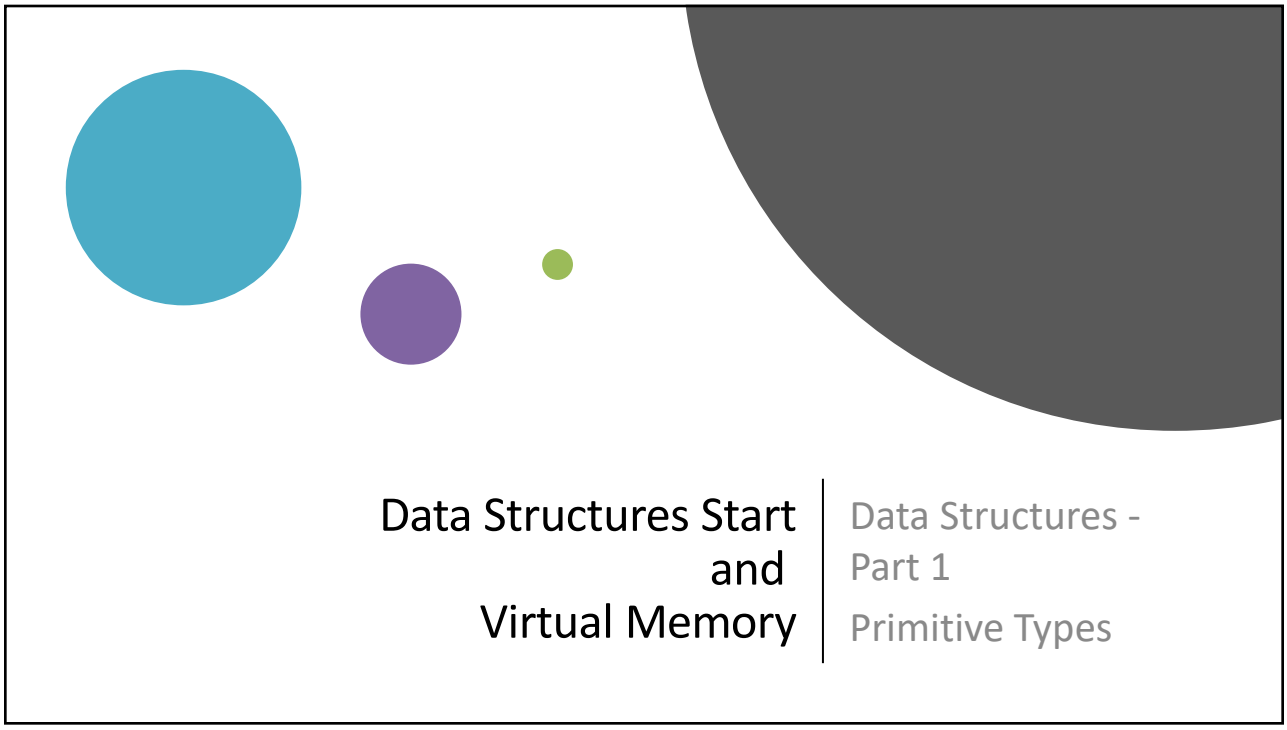
16.9 Testing during Postdelivery Maintenance

- Maintainers tend to view a product as a set of loosely related components
  - They were not involved in the development of the product
- Regression testing is essential
  - Store test cases and their outcomes, modify as needed

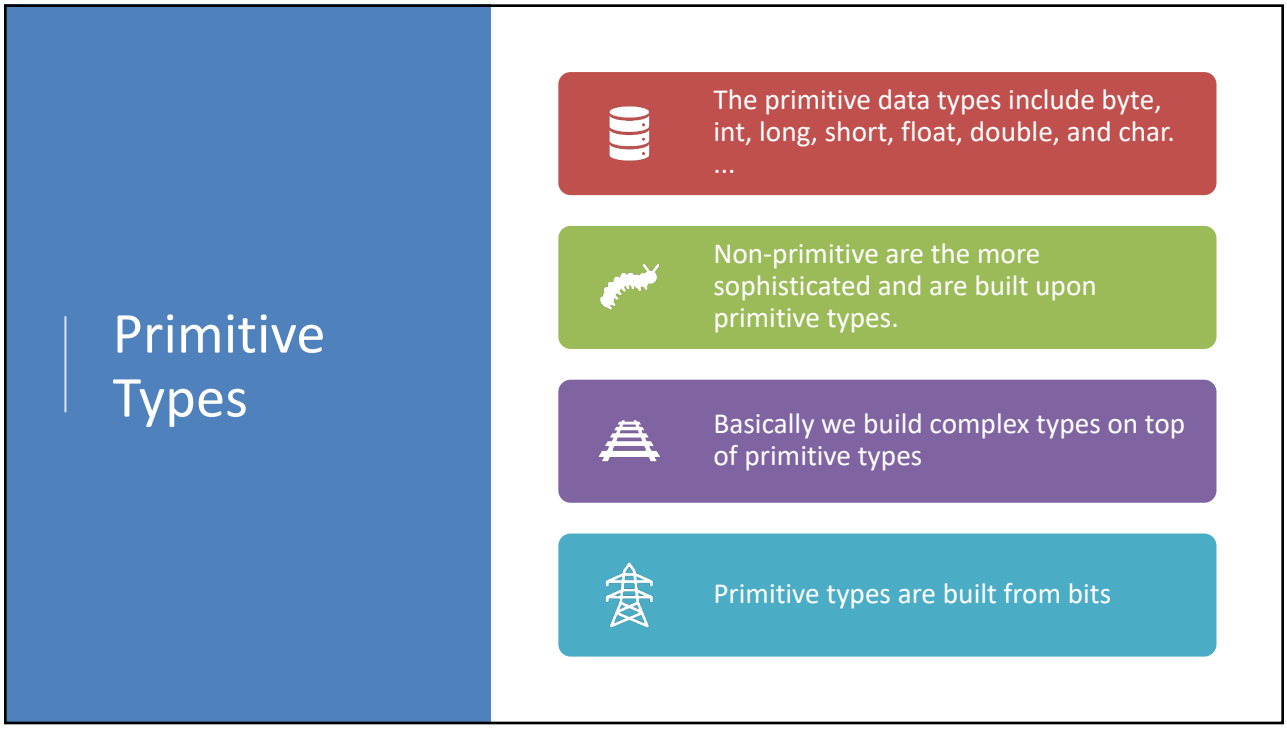
16.11 Metrics

- Page 566
- Code artifacts with high complexity is a likely candidate for inducing a regression fault.
- Measures relating to software defect reports,
- Number of defects reported and classification of those defects by severity and type.

The End


Data Structures Start and Virtual Memory
Data Structures - Part 1
Primitive Types

1


Primitive Types
The primitive data types include byte, int, long, short, float, double, and char. ...
Non-primitive are the more sophisticated and are built upon primitive types.
Basically we build complex types on top of primitive types
Primitive types are built from bits

2

Bit

- Tiniest data structure
- Can be on or off
  - 1 or 0
- Fundamental building block of all data types
- Bits are used to build binary numbers

3

Binary Numbers

- Build from a collection of bits
- Count the places from right to left
- We start counting from place zero
- Multiply each digit by 2 to the power of its place **number**
- For example:
  - If place #1 from the right is a 1, you would multiple 1 by 2 to the power of 1 and get 2
  - if place #2 from the right is a 1, you would multiply 1 by 2 to the power of 2 to get 4
- Example: 0b110
  - Place #0, is 0,
  - Place #1 is 1, so we multiple 1 by 2 to the power of 1 and get 2
  - Place #2 is 1, so we multiple 1 by 2 to the power of 2 and get 4
  - 4 + 2 = 6
- Example: b1000
  - Place # 3 is 1, so we multiple 1 by 2 to the power of 3 and get 8
- https://www.wikihow.com/Read-Binary

4

"Byte"

- 8 bits
- Bytes store numbers
- We use numbers to represent everything else
- Numbers are stored, ultimately in binary
- Example:
  00000001 is 1
  00000100 is 4
  00001100 is 12

5

Nybble / nyble/nibble

- Half a byte, 4 bits
- Image by sipa from Pixabay

6

Byte

- A byte can contain a number
- A byte can contain a character
- Byte = 25;
- Byte = ‘A’
- What if we want a word?

7

Array of bytes

char bytes[5] ;

bytes[0] = ‘M’

Bytes[1] = ‘I’

Bytes[2] = ‘K’

Bytes[3] = ‘E’

Bytes[4] = 0

(Remember , computers like to start counting from zero)

8

Big Numbers

- Integers
- What if we want a number bigger than the largest value a byte can hold
- Integer types:
  - Short - 2bytes (packaged as 1)
  - Int - (4 bytes packaged as 1)
  - Long (8 bytes packaged as 1)

9

Hex Numbers

Decimal	Binary	Hexadecimal
0	0000	0
1	0001	1
2	0010	2
3	0011	3
4	0100	4
5	0101	5
6	0110	6
7	0111	7
8	1000	8
9	1001	9
10	1010	A
11	1011	B
12	1100	C
13	1101	D
14	1110	E
15	1111	F

10

Ascii

- extended ascii chart - Barta.innovations2019.org

Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char
0	00	Null	32	20	Space	64	40	@	96	60	`
1	01	Start of heading	33	21	!	65	41	A	97	61	a
2	02	Start of text	34	22	"	66	42	B	98	62	b
3	03	End of text	35	23	#	67	43	C	99	63	c
4	04	End of transmit	36	24	$	68	44	D	100	64	d
5	05	Enquiry	37	25	%	69	45	E	101	65	e
6	06	Acknowledge	38	26	&	70	46	F	102	66	f
7	07	Audible bell	39	27	'	71	47	G	103	67	g
8	08	Backspace	40	28	(	72	48	H	104	68	h
9	09	Horizontal tab	41	29	)	73	49	I	105	69	i
10	0A	Line feed	42	2A	*	74	4A	J	106	6A	j
11	0B	Vertical tab	43	2B	+	75	4B	K	107	6B	k
12	0C	Form feed	44	2C	,	76	4C	L	108	6C	l
13	0D	Carriage return	45	2D	-	77	4D	M	109	6D	m
14	0E	Shift out	46	2E	.	78	4E	N	110	6E	n
15	0F	Shift in	47	2F	/	79	4F	O	111	6F	o
16	10	Data link escape	48	30	0	80	50	P	112	70	p
17	11	Device control 1	49	31	1	81	51	Q	113	71	q
18	12	Device control 2	50	32	2	82	52	R	114	72	r
19	13	Device control 3	51	33	3	83	53	S	115	73	s
20	14	Device control 4	52	34	4	84	54	T	116	74	t
21	15	Neg. acknowledge	53	35	5	85	55	U	117	75	u
22	16	Synchronous idle	54	36	6	86	56	V	118	76	v
23	17	End trans. block	55	37	7	87	57	W	119	77	w
24	18	Cancel	56	38	8	88	58	X	120	78	x
25	19	End of medium	57	39	9	89	59	Y	121	79	y
26	1A	Substitution	58	3A	:	90	5A	Z	122	7A	z
27	1B	Escape	59	3B	;	91	5B	[	123	7B	{
28	1C	File separator	60	3C	<	92	5C	\	124	7C	|
29	1D	Group separator	61	3D	=	93	5D	]	125	7D	}
30	1E	Record separator	62	3E	>	94	5E	^	126	7E	~

11

Virtual Memory

SEIS-610

A small taste!

12

- Paging and Page Replacement
- Many of these slides came from
- http://pages.cs.wisc.edu/~mattmcc/cs537/notes/Replacement.ppt
- **Thank you to:**

Matt McCormick
1302 Computer Science Building
University of Wisconsin - Madison
1210 West Dayton Street
Madison, WI 53706

13

Paging

- In general..
  - Computer Memory is broken up into pages
  - A page can be in physical memory (a frame)
  - A page can be in persistent storage (our disk)
- Since somethings can be in physical memory and some things can be on a disk, we have the illusion of unlimited memory.

14

Paging

- If a page is not in physical memory (frame)
  - find the page on disk
  - find a free frame
  - bring the page into memory
- What if there is no free frame in memory?

15

Page Replacement

- Basic idea
  - if there is a free page in memory, use it
  - if not, select a *victim* frame
  - write the victim out to disk
  - read the desired page into the now free frame
  - update page tables
  - restart the process

16

Page Replacement

- Main objective of a good replacement algorithm is to achieve a low *page fault rate*
  - ensure that heavily used pages stay in memory
  - the replaced page should not be needed for some time
- Secondary objective is to reduce latency of a page fault
  - efficient code
  - replace pages that do not need to be written out

17

Virtual Memory -- Paging

- Remember –
  - Virtual Memory can give the illusion of large memory space.
  - Virtual Memory, the entire program does not need to be in volatile memory
- Once again, what if a page is not in physical memory???
  - find the page on disk
  - find a free frame
  - bring the page into memory

18

Page Replacement

- REMEMBER
  - Frames – Physical
  - Pages – Virtual
- Idea
  - if there is an open frame in memory, use it
  - if not, select a *victim* frame
  - write the victim out to disk
  - read the desired page into the now free frame
  - update page tables
  - restart the process

19

Reference String

- Reference string is the sequence of pages being referenced
- If user has the following sequence of addresses
  - 123, 215, 600, 1234, 76, 96
- If the page size is 100, then the reference string is
  - 1, 2, 6, 12, 0, 0
  - Divide by 100
- Pretending the page number is the first digits
- I will just give you reference strings, not addresses

20

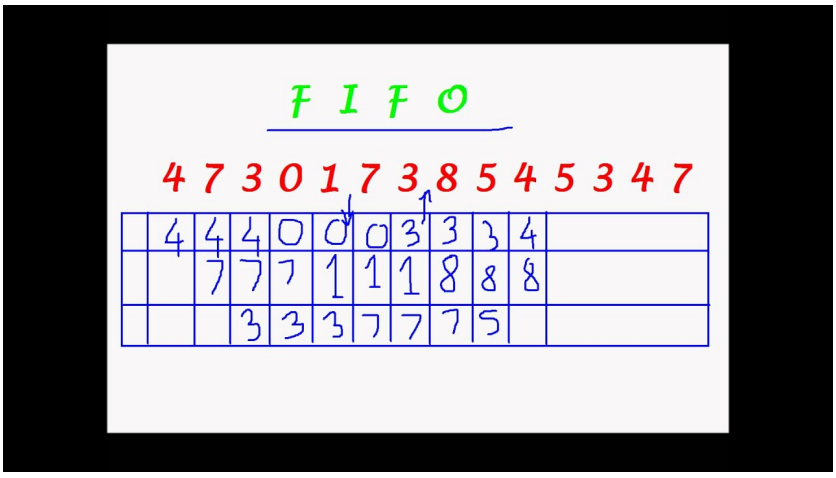
First-In, First-Out (FIFO)

- The oldest page in physical memory is the one selected for replacement
- Very simple to implement
  - keep a list
    - victims are chosen from the tail
    - new pages in are placed at the head

21

First-in, First Out Example

https://www.youtube.com/watch?v=KejyTiATz18



22

FIFO Issues

- Poor replacement policy
- Evicts the oldest page in the system
  - usually a heavily used variable should be around for a long time
  - FIFO replaces the oldest page - perhaps the one with the heavily used variable
- FIFO does not consider page usage

23

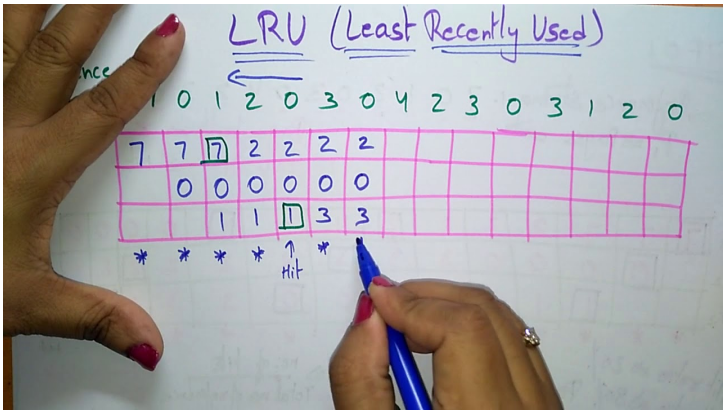
Least Recently Used (LRU)

- Basic idea
  - replace the page in memory that has not been accessed for the longest time
- Optimal policy looking back in time
  - as opposed to forward in time
  - fortunately, programs tend to follow similar behavior

24

Least Recently Used

https://www.youtube.com/watch?v=u23ROrISK_g



25

Dirty Pages

- If a page has been written to, it is *dirty*
- Before a dirty page can be replaced it must be written to disk
- A *clean* page does not need to be written to disk
  - the copy on disk is already up-to-date
- We would rather replace an old, clean page than and old, dirty page

26

- The end

27