04-630
Data Structures and Algorithms for Engineers

David Vernon
Carnegie Mellon University Africa

vernon@cmu.edu
www.vernon.eu

Lecture 28

- Correctness
  - Types of software defects
    - Syntactic, semantic, logical defects
  - Formal verification
  - Static tests
    - Reviews, walkthroughs, inspections
    - Reviewing algorithms and software
  - Dynamic testing
    - Unit tests
    - Test harness, stubs, drivers,
    - Integration testing
    - Regression testing
  - Verification and validation strategies

(The material on correctness was adapted from M. Rosso-Llopart's notes for Computer Science for Practicing Engineers)

Lecture 28

- Abstract Data Types and Object-oriented Programming
  - OOA
  - OOD
  - OOP
  - OOT

- Standard Template Library

Types of Software Defects

Specification

- defective requirements

System Design

- defects introduced during design of the system

Detailed Design

- defects introduced during code module design

Syntactic

- using "," instead of ";" or forgetting to match {}

Semantic

- applying arithmetic operations to non- arithmetic values, order of arithmetic evaluation,... e.g. 5+6 * 2 when you mean (5+6)*2

Logical

- this is when the solution to the algorithmic problem is incorrect, usually for just some of the inputs

Types of Software Defects

- Syntactic defects are relatively easy to find and fix

- Semantic & logical defects are seen at “run time” in three general ways:

  1. In execution that terminates normally but with incorrect outputs

  2. An aborted execution

  3. An execution that does not terminate

- Quality attribute failure (security, performance, availability, maintainability, etc.) – often the most difficult to repair

Detecting Defects

- Formal Verification
  - Rigorously showing that an algorithm is correct
  - Generally referred to as “formal methods” in computer science
- Static Testing
  - Reviews, walkthroughs or inspections of code
- Dynamic Testing
  - Executing programmed code with a given set of test cases and expected results

Static Testing
Reviews, Walkthroughs, Inspections

Structured reviews are a kind of static test that can be used to review designs, code, or algorithms

- algorithm reviews – this is a review where the structure and flow of an algorithm is reviewed by a group of engineers
- code reviews – this is a review where actual code is reviewed by a group of engineers for semantic correctness.

Static Testing
Reviews, Walkthroughs, Inspections

- Structured reviews or inspections
  - are used to check the correctness of algorithmic designs and implementations of a software product
  - aim to find software defects early in the development process to reduce the costs of finding and removing these defects
- The cost of finding and removing defects increases the longer they go undetected

Static Testing
Reviews, Walkthroughs, Inspections

- A review team is selected – typically 3 to 5 reviewers (may or may not include the producer)
- The team receives the algorithm or source code and are given time to privately review the artifact
- A review meeting is scheduled and the review team convenes and roles are assigned:
  - moderator
  - time keeper
  - issue recorder (usually the producer)
- The moderator will lead the review of the code or algorithm a bit at a time
- The members of the review team (including the producer) may raise issues during the review
- The recorder documents the issues – they are addressed later by the producer

Static Testing
Reviews, Walkthroughs, Inspections

- Algorithms should be in a form such as:
  - pseudo-code
  - flow charts
  - formal mathematics,… or some combination…

- When preparing review handouts, include
  - a general description of the algorithm (or algorithms)
  - purpose of the algorithm and its role in the system
  - preconditions and post conditions

Static Testing
Reviews, Walkthroughs, Inspections

- When reviewing the algorithm, each step should be read aloud by the moderator, then:
  - the reviewers should be given an opportunity to ask clarifying questions or raise issues
  - the producer will answer any questions and record any issues that arise during the review
- The way that the algorithm is traced through by the reviewers depends upon how the algorithm is documented.
- It is important that issues are captured and NOT SOLVED during the review

Dynamic Testing

Software testing is an empirical method for finding defects in software systems

- It is clearly the most widely used technique for detecting defects

- Usually involves running the program on several typical and atypical inputs, called test sets

- Certain kinds of dynamic test, under certain operational conditions can be automated

Dynamic Testing

There are many strategies for dynamic software test...

- Black Box testing
  treats the software as a "black box" without any knowledge of internal implementation; focus on specification as test driver

- White Box testing
  when the tester has access to the internal data structures and algorithms and focuses on critical code sections to design tests

- Grey Box testing
  testers have some insight into internal data structures and algorithms and may influence the design of tests

Dynamic Testing

Testing occurs at many levels:

- Unit test – this kind of testing involves testing small code modules
- Integration test – this test involves checking interfaces
- System test – this is test of the entire system
- Acceptance testing – customer tests where acceptance of the product is contingent upon successfully completing agreed to tests
- Regression testing – any type of software testing that seeks to uncover newly introduced defects in software (usually due to maintenance, upgrades, etc.) that was working properly

Unit Test

- In unit testing we isolate the testable software "chunks" from remainder of the code, and determine whether it behaves as expected

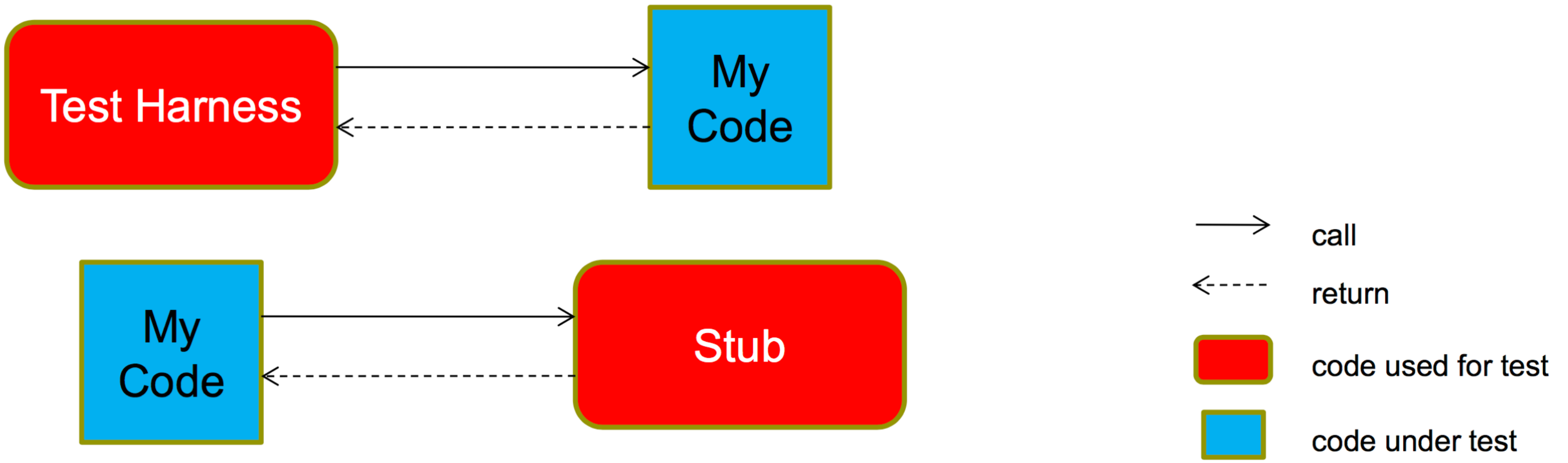
- Units are tested separately before integrating them into larger "chunks" and finally into a complete system

- The most common approach to unit testing requires test harnesses and stubs to be written

Test Harness and Stubs

- Test harnesses simulate the calling unit in order to test methods, functions, procedures

- Stubs simulate a called unit by returning dummy and/or “hardwired” data until the real methods, procedures, functions can be delivered



Test Harness and Stubs

- Test harnesses and stubs play a role in product quality

- Test harnesses and stubs may require significant attention and when there are stringent quality demands:

  – May require high level of design attention

  – Might need to be reviewed/inspected

  – Often require a lot of effort and time to develop

"To achieve the level of quality we need, we write as much test code [harnesses and stubs] as functional, production code – and we review it [the test code]!"

Andy Park G3 Technologies

Test Harness and Stubs

- Advantages
  - A large percentage of operational defects can be identified prior to system integration
  - Unit tests reduce difficulties of discovering errors contained in larger, more complex chunks of the system or application
- Disadvantages
  - The development of test harnesses and stubs can represent a significant investment
    - because of this, unit testing is minimized or skipped because of schedule
    - not a good idea
  - Often leads to code, test, fix cycles rather than thoughtful design and analysis

Integration Test

- As we aggregate “units,” we test the behaviour of the sub-system or the entire system
- Integration testing usually begins in a lab setting where we test the system (in whole or in part) under simulated and ideal conditions
- Integration testing will eventually include a deployment test, testing the system under real environmental conditions
- Integration testing identifies problems that occur when units are combined

Integration Test

- The advantages speak for themselves:

  We must show that the system works!

- Poor practices include not …

  - deliberately planning integration tests
  - testing the system under realistic conditions
  - stress testing the system
  - verifying that the system possesses the required systemic properties
  - budgeting time and schedule for integration tests

Regression Test

- Whenever system software is modified we conduct regression tests to verify we did not introduce defects

- The goal is to provide “sufficient” coverage without wasting time – the real trick is determining what is “sufficient”

  – Spend as little resources as possible in regression test, without reducing the probability that we will find defects

- The regression test strategy we use will often be dictated by the quality needs of our project and product

- Issue is path coverage, how much is reasonable

Regression Test

Factors to consider:

- Design separate regression tests for each defect fixed or enhancement to the system
  - TDD – Design the test first?
- Watch out for side effects of fixes and enhancements
- If two or more tests are similar, determine which is less effective and get rid of it

Regression Test

Factors to consider:

- Develop and maintain tests suites
  - Archive and reuse them
- Test critical systemic properties (performance, security, availability, …)

General Issues with Dynamic Testing

- It's often impossible to test a program on all possible inputs
  - the input sets might be very large, or even infinite
- It can be impossible to test a system without placing life, limb, and material at significant risk
  - You can only really test the software when you fly it, drive it, … That is a terrible time to find defects!
- Dynamic testing focuses on testing functionality, not systemic properties such as modifiability, maintainability, scalability, …

General Issues with Dynamic Testing

- "Testing can only be used to demonstrate the presence of errors in software, not their absence." Dijkstra

- Too often testing is conducted in an ad hoc way and is not planned. It can be difficult to
  - determine the level of coverage
  - know if the important things have been tested

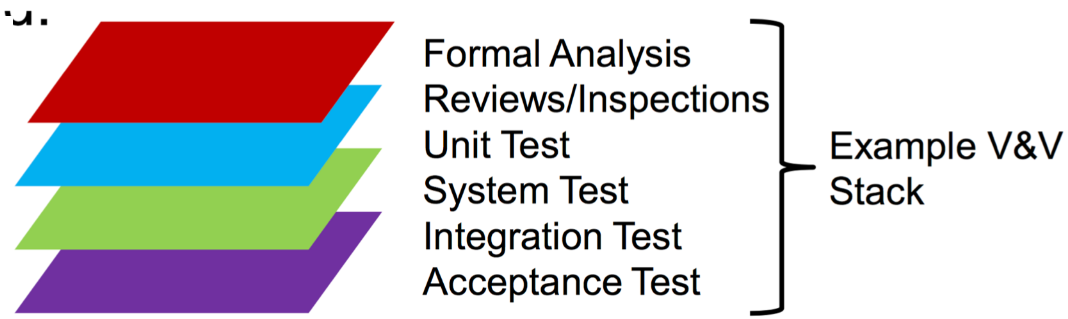
- Testing $\neq$ Quality – you can't "test-in" quality
  - testing is not cheap
  - testing is the last resort
  - achieving quality software requires a quality strategy base on quality goals

Verification and Validation Strategies

- How much quality do you need and when do you need it?
  - Does it have to be perfect when it is delivered?
  - Can we deliver with “good-enough” quality and improve over time?
- Adopting an explicit V&V strategy and creating a plan to achieve quality goals is an essential part of project planning

Verification and Validation Strategies

- What we try to do when we devise a V&V strategy is select multiple layers of techniques to prevent and detect defects before they are deployed



- The combination of techniques you select will vary based on your quality goals

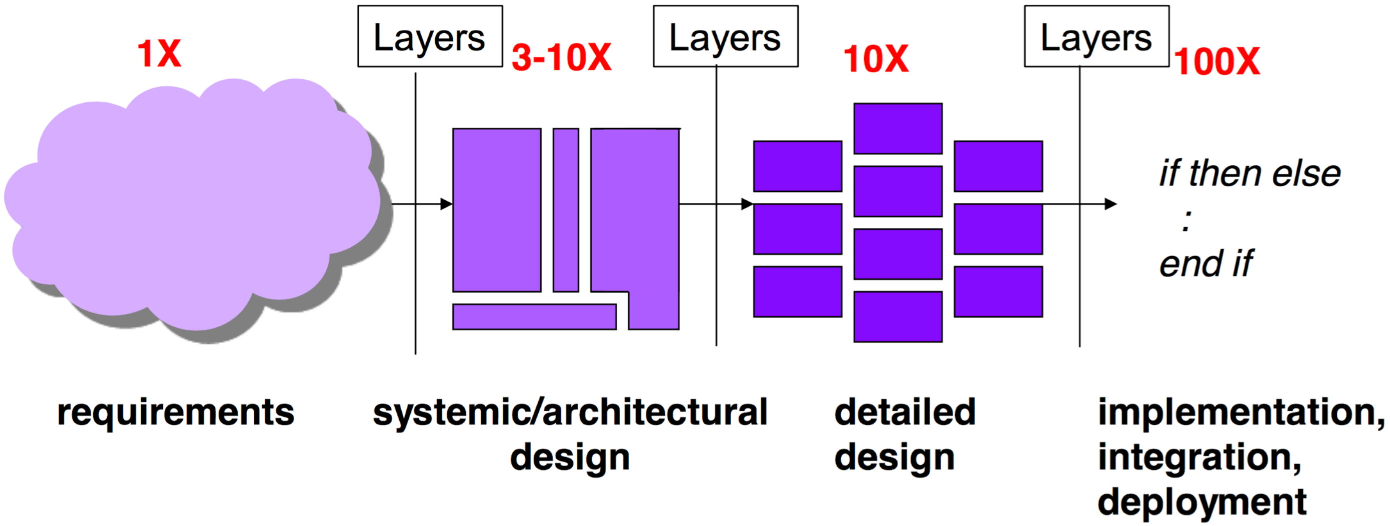
The Cost of Finding and Fixing Defects

- Data shows that the earlier a defect is found the cheaper it is to fix it
- The following table shows the average cost of fixing defect depending on when it was introduced and when it was found
- For example, a defect introduced in design, costs on average 25–100 times more to fix it once deployed

		Time Detected				
		Requirements	**Design**	**Coding**	**Test**	**Deployment**
Time Introduced	**Requirements**	1x	3x	5-10x	10x	10-100x
	Design	-	1x	10x	15x	25-100x
	Coding	-	-	1x	10x	10-25x

The Cost of Finding and Fixing Defects

Validation and verification cuts across the lifecycle …



Verification Considerations

Explore the input domain

- Inputs that force all the errors
- Input messages
- Inputs that force default values
- Explore allowable inputs
- Overflow input buffers
- Test inputs that may interact, and test combinations of their values
- Repeat the same input numerous times

Verification Considerations

Explore the outputs

- try to force different outputs to be generated for each input
- try to force invalid outputs to be generated
- force properties of an output to change
- force the screen to refresh

Verification Considerations

Explore data constraints

- force a data structure to store too many or too few values
- find ways to violate internal data constraints

Verification Considerations

Explore feature interactions

- force invalid operator/operand combinations
- make a function call itself recursively
- force computation results to be too big or too small
- test features that share data

Verification Considerations

Explore the file system conditions

- file system full to capacity
- disk is busy or unavailable
- invalid file name
- invalid disk
- vary file permissions
- vary or corrupt file context

Conclusion

Simple Formal Verification Techniques

Dynamic Testing Techniques

Static Testing Techniques

These methods must be coupled with
disciplined software practices and
a broader verification and validation strategy
to provide practical, cost effective, benefit

“you can’t test quality into software”

References

- Stephen H. Kan, Metrics and Models in Software Quality Engineering (2nd edition) Addison-Wesley, September 2002
- Capers Jones, Measuring Defect Potentials and Defect Removal Efficiency, Cross Talk – The Journal of Defense Software Engineering, Jun 2008
- Algorithmics: The Spirit of Computing (3rd edition), David Harel, Yishai Feldman
- Data Structures and Algorithms, Alfred V. Aho, Jeffrey D. Ullman, John E. Hopcroft
- Introduction to Algorithms (2nd edition), Thomas H. Cormen et al.
- M. E. Fagan, Design and Code Inspections to Reduce Errors in Program Development, IBM Systems Journal, vol. 15, no. 3, pp. 182, 211, 1976.
- W. E. Stephenson, An Analysis of Resources Used on the SAFEGUARD System Software Development, Technical Report, Bell Labs, August 1976.
- E. B. Daly, Management of Software Engineering, IEEE Transactions on Software Engineering, vol. 3, pp. 229, 242, May 1977.
- B.W.Boehm,SoftwareEngineeringEconomics.PrenticeHall,1981.
- J. Whittaker, How to Break Software: A Practical Guide to Testing, 2003

Supplementary Reading

- "Integrating Software Testing and Run-Time Checking in an Assertion Verification Framework," Lopez-Garcıa/Hermenegildo

- All I Really Need to Know about Pair Programming I Learned In Kindergarten," Williams, Kessler

- "Unit Testing– a Very Short Parable," Sanford M. Sorkin http://ww2.cis.temple.edu/sorkin/CIS338UnitTesting.htm

- “Designing Unit Tests,” IPL Information Processing Ltd.
  (With permission from IPL Information Processing Ltd., Eveleigh House, Grove Street Bath, BA1 5LR, United Kingdom)

- Invariant Assertion Method, http://www.rose- hulman.edu/Users/faculty/young/CS-Classes/csse373/current/Resources/Ardis1.pdf

Abstract Data Types and Object-Oriented Programming

- OOA: Object-oriented Analysis (Booch method; Coad and Yourdon method)
- OOD: Object-oriented Design
- OOP: Object-oriented Programming
- OOT: Object-oriented Testing
- What is an object-oriented approach?

  One definition:

  It is the exploitation of class objects, with private data members and associated access functions

Abstract Data Types and Object-Oriented Programming

- Class
  - A class is a 'template' for the specification of a particular collection of entities (e.g. a widget in a Graphic User Interface)
  - More formally, 'a class is an OO concept that encapsulates the data and procedural abstractions that are required to describe the content and behaviour of some real-world entity'
- Attributes
  - Each class will have specific attributes associated with it (e.g. the position and size of the widget)
  - These attributes are queried using associated access functions (e.g. set_position)

Abstract Data Types and Object-Oriented Programming

- Object

  - An object is a specific instance (or instantiation) of a class (e.g. a button or an input dialogue box)

- Data Members
  - The object will have data members representing the class attributes (e.g. int x, y;)

Abstract Data Types and Object-Oriented Programming

- Access functions
  - The values of these data members are accessed using the access functions (e.g. set_position(x, y);)
  - These access functions are called methods (or services)
  - Since the methods tend to manipulate a limited number of attributes (i.e. data members) a given class tends to be cohesive.
  - Since communication occurs only through methods, a given class tends to be decoupled from other objects.

Abstract Data Types and Object-Oriented Programming

- Encapsulation
  - The object (and class) encapsulates the data members (attributes), methods (access functions) in one logical entity
- Data Hiding
  - It allows the implementation of the data members to be hidden
  - Why? Because the only way of getting access to them – of seeing them – is through the methods
  - This is called data hiding

Abstract Data Types and Object-Oriented Programming

- Abstraction
  - This separation, though data hiding, of physical implementation from logical access is called abstraction
- Messages
  - Objects communicate with each other by sending messages
  - This just means that a method from one class calls a method from another method and information is passed as arguments

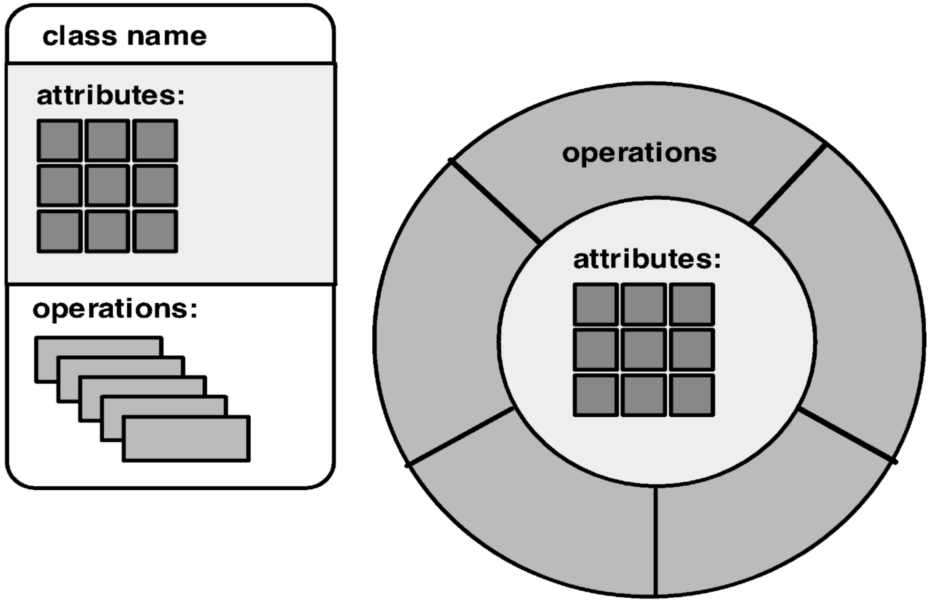
Abstract Data Types and Object-Oriented Programming

Aside: an Alternative definition of object-orientation (Ellis and Stroustrup)

> 'The use of derived classes and virtual functions is often called object-oriented programming'

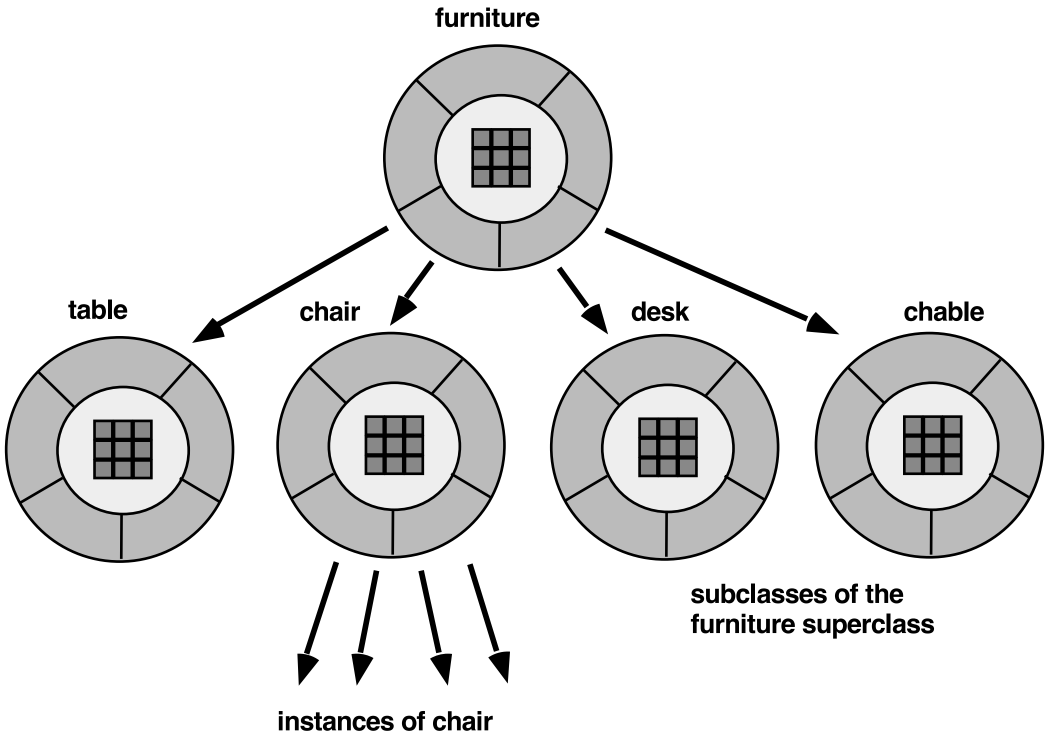
Abstract Data Types and Object-Oriented Programming

Two views of a class:



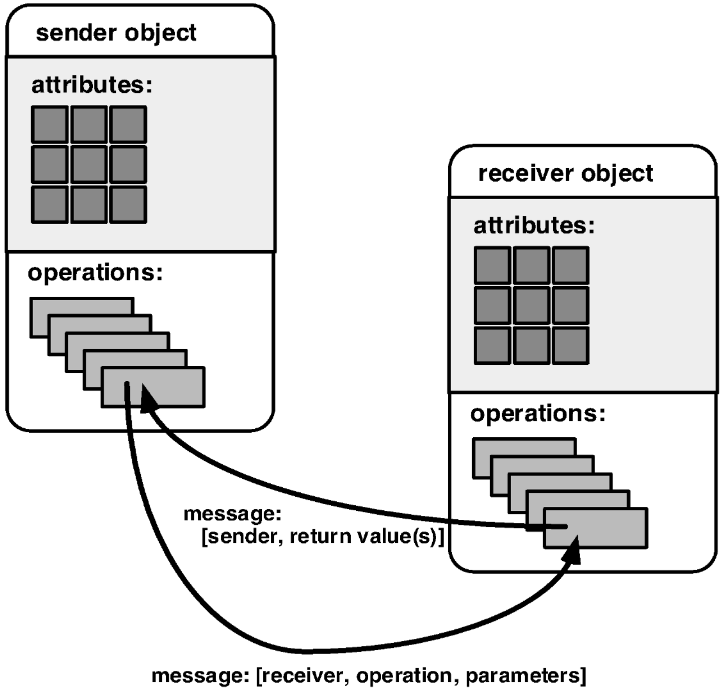
Abstract Data Types and Object-Oriented Programming

Class hierarchy:



Abstract Data Types and Object-Oriented Programming

Message passing between objects



Abstract Data Types and Object-Oriented Programming

OOA: Object-Oriented Analysis

- Booch method
- Coad and Yourdon method
- Jacobson method
- Rambaugh method

Abstract Data Types and Object-Oriented Programming

There are seven generic steps in OOA:

1. Obtain customer requirements

    - identify scenarios or use cases
    - build a requirements model

2. Select classes and objects using basic requirements

3. Identify attributes and operations for each object:

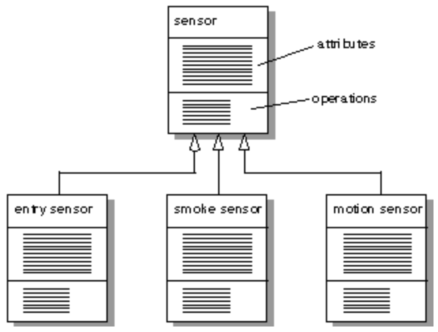
    - Class-Responsibility-Collaborator (CRC) Modelling

Abstract Data Types and Object-Oriented Programming

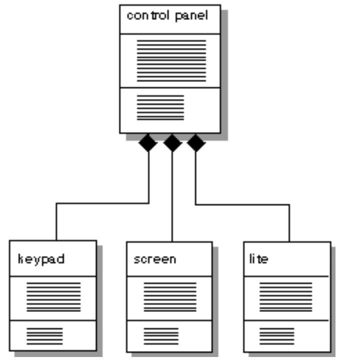
There are seven generic steps in OOA:

4. Define structures and hierarchies that organize classes

   - Generalization-Specialization (Gen-Spec) structure ("is a")



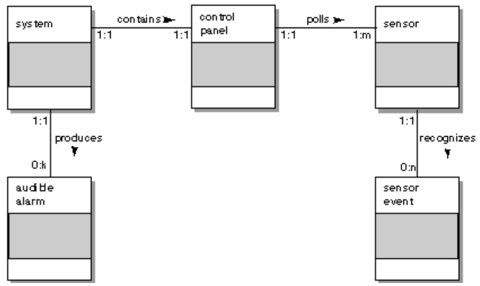
   - Composite-Aggregate (Whole-Part) structure ("has a")



Abstract Data Types and Object-Oriented Programming

There are seven generic steps in OOA:

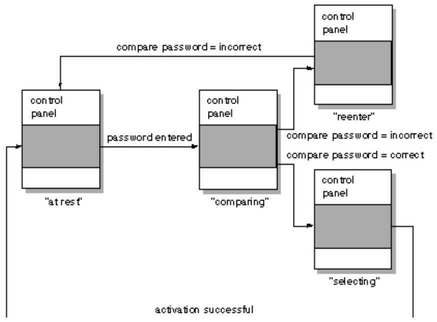
5. Build an object-relationship model

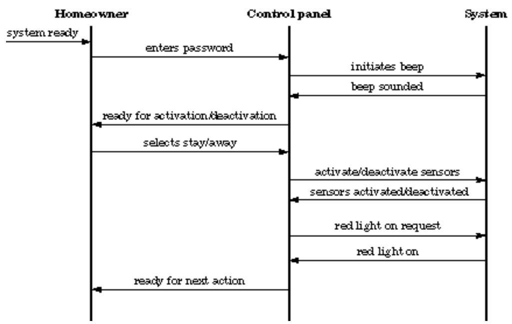


6. Build an object-behaviour model

   State transition diagram

   Event trace diagram





7. Review the OO analysis model against use cases / scenarios

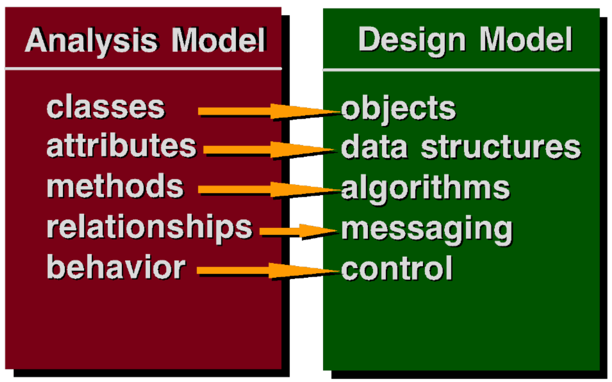
Abstract Data Types and Object-Oriented Programming

OOD: Object-Oriented Design

- ‘Designing object-oriented software is hard, and designing reusable object-oriented software is even harder … a reusable and flexible design is difficult if not impossible to get “right” the first time’

- OOD is a part of an iterative cycle of analysis and design

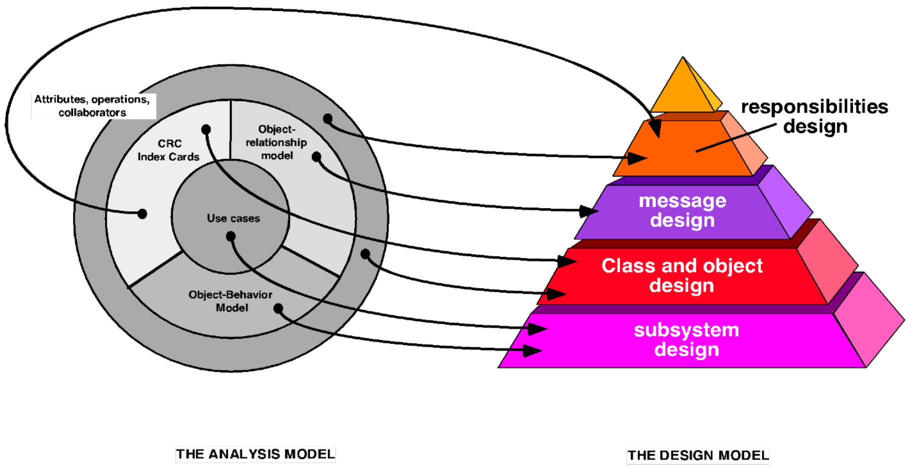
- Several iterations of which may be required before one proceeds to the OOP stage



Abstract Data Types and Object-Oriented Programming

OOD: Object-Oriented Design

- ‘Designing object-oriented software is hard, and designing reusable object-oriented software is even harder … a reusable and flexible design is difficult if not impossible to get “right” the first time’
- OOD is a part of an iterative cycle of analysis and design
- Several iterations of which may be required before one proceeds to the OOP stage



Abstract Data Types and Object-Oriented Programming

```
/*
   Interface file

   Construction of an optimal prefix code using the Huffman binary code tree algorithm

   Course 04-630 Data Structures and Algorithms for Engineers, Assignment 5

   Based on code by David Vernon written originally on 13/03/1997, revised 15/10/2014.
   This code has been rewritten to streamline the object-oriented design.

   In this design, there are four classes:

   Path    a path from the root to a source alphabet symbol
   Map     a code map comprising pairs of source alphabet symbol & prefix code
   Tree    a code tree
   Forest  a forest of code trees used to construct the optimal code tree

   As a general principle, the public access methods do not expose the underlying hidden data structures
   i.e. they present an abstract interface to the data / object

   David Vernon
   19 March 2019
*/
```

Abstract Data Types and Object-Oriented Programming

```
/***********************/
/*                     */
/*     Class Path      */
/*                     */
/***********************/

/* a class to represent the path to a leaf node in a binary tree */
/* a path comprises a sequence of elements of 0s and 1s ...      */
/* 0 means take a left link; 1 means take a right link           */
/* functions are provided to add an element, remove and element, */
/* and print the path to the screen.                             */
/* the maximum number of elements is defined by MAX_PATH_LENGTH  */

class Path {
public:
   Path();                              // constructor: create an empty path
   ~Path();                             // destructor
   void add_to_path(int direction);     // add a direction to the path
   void remove_from_path();             // remove the last direction added to the path
   void print_path(FILE *fp_out);       // print a path to a file
   void to_string(char code[]);         // translate a path to a character string comprising 0s and 1s
private:
   int path_components[MAX_PATH_LENGTH];
   int path_length;
};
```

Abstract Data Types and Object-Oriented Programming

```
/***********************/
/*                     */
/*     Class Map       */
/*                     */
/***********************/

/* a class to represent the code map                                        */
/* This is a dictionary of symbols (the key) and the corresponding paths   */

class Map {
public:
   Map();
   ~Map();
   void add(char symbol, Path path);                        // add a symbol-code pair to the map
   void retrieve(char symbol, char code[]);                 // retrieve the code  corresponding to a given symbol
   void print(FILE *fp_out);                                // print the map to a file
   void encode(char source[], char encoded_source[]);       // encode a source alphabet string as a code alphabet string
private:
   Path path[MAXIMUM_NUMBER_OF_SYMBOLS];
   char symbol[MAXIMUM_NUMBER_OF_SYMBOLS];
   int  size;
};
```

Abstract Data Types and Object-Oriented Programming

```
/************************/
/*                      */
/*     Class Tree       */
/*                      */
/************************/

/* a class to represent the binary code tree where leaf nodes represent the source alphabet symbols */

struct node
{
   char symbol;                 // source alphabet symbol
   float probability;           // source alphabet probability
   node *pleft, *pright;        // links to left and right children nodes
};

class Tree {
public:
   Tree();
   ~Tree();
   void add(char symbol, float probability);          // add a symbol and its probability to the code tree
   void print(FILE *fp_out) const;                    // print a tree to a file
   void delete_tree();                                // delete a tree
   void join_to_tree(Tree &t);                        // join two trees: one is an argument, the other is tree for which the method is called
   float root_probability();                          // return the probability of the symbol at the root
   bool empty_tree() const;                           // test for empty tree
   void compute_map(Map &code_map);                   // store the leaf nodes & the path to the leaf nodes in a map
   void decode(char source[], char decoded_source[]); // decode an encoded message

private:
   node *root;
   void delete_tree(node* &p);
   void add(char symbol, float probability, node* &p);
   void pr(const node *p, int nspace, FILE *fp_out) const;
   node* get_root();
   void cut_off_tree();                                             // break the link to the root without deleting the tree
   void traverse_leaf_nodes(const node *p, Path &path, Map &code_map);  // add to a map the leaf nodes and the path to leaf nodes
   void decode(node *p, char source[], int &i, char decoded_source[], int &j);
};
```

Abstract Data Types and Object-Oriented Programming

```
/***********************/
/*                     */
/*     Class Forest    */
/*                     */
/***********************/

class Forest {
public:
   Forest(int size);
   ~Forest();
   void initialize_forest();
   void add_to_tree(int tree_number, char symbol, float probability);
   void print_forest(FILE *fp_out) const;
   void print_tree(int tree_number, FILE *fp_out);
   void join_trees(int tree_1, int tree_2);
   bool empty_tree(int tree_number);
   float root_probability(int tree_number);
   Map  build_map();
   void build_code_tree(int number_of_symbols, char symbols[], float probabilities[]);
   void decode(char source[], char decoded_source[]);
private:
   Tree tree_array[MAXIMUM_NUMBER_OF_SYMBOLS];
   int forest_size;
};
```

Standard Template Library
STL

```
/* Example of use of STL for

   stack
   queue
   priority queue
   map (the underlying STL implementation is a red-black tree)
   unordered_map (the underlying STL implementation is a hash table)

   */

#include "stdio.h"
#include "string.h"
#include <string.h>
#include <iostream>

#include <iterator>
#include <stack>
#include <queue>
#include <unordered_map>
#include <map>

using namespace std;
```

```
stack<int> s;

printf("stack\n");
printf("-----\n");

s.push(1);
s.push(2);
s.push(3);

printf("%d \n",s.top()); // Note: top() accesses the element
s.pop();                 // but you need pop() to remove it

printf("%d \n",s.top());
s.pop();

printf("%d \n",s.top());
s.pop();

if (s.empty())
  printf("stack is empty\n\n");
```

```
queue<int> q;

printf("queue\n");
printf("-----\n");

q.push(3);
q.push(7);
q.push(1);
q.push(2);

printf("queue size: %d \n",q.size());

printf("%d \n",q.front());
q.pop();

printf("%d \n",q.front());
q.pop();

printf("%d \n",q.front());
q.pop();

printf("%d \n",q.front());
q.pop();

if (q.empty())
   printf("queue is empty\n\n");
```

```
priority_queue<int> pq;

printf("priority queue\n");
printf("--------------\n");

pq.push(1);
pq.push(4);
pq.push(2);
pq.push(8);
pq.push(5);
pq.push(7);

printf("queue size: %d \n",pq.size());

printf("%d \n",pq.top());
pq.pop();

printf("%d \n",pq.top());
pq.pop();

printf("%d \n",pq.top());
pq.pop();

printf("%d \n",pq.top());
pq.pop();

printf("%d \n",pq.top());
pq.pop();

printf("%d \n",pq.top());
pq.pop();

if (pq.empty())
   printf("priority queue is empty\n\n");
```

```
// unordered_map: the underlying STL implementation is a hash table

unordered_map<string, double> umap; // Declaring umap to be of <string, double> type
                                    // key will be of string type and mapped value will be of double type

unordered_map<string, double>:: iterator itr2;

printf("unorded_map\n");
printf("-----------\n");
// inserting values by using [] operator
umap["PI"]    = 3.14;
umap["root2"] = 1.414;
umap["root3"] = 1.732;
umap["log10"] = 2.302;
umap["loge"]  = 1.0;

// inserting value by insert function

umap.insert(make_pair("e", 2.718));

key = "PI";

if (umap.find(key) == umap.end())       // If key not found in map iterator to end is returned
   cout << key.c_str() << " not found\n";
else                                    // If key found then iterator to that key is returned
   cout << "Found " << key.c_str() << "\n";

key = "lambda";
if (umap.find(key) == umap.end())
   cout << key.c_str() << " not found\n";
else
   cout << "Found " << key.c_str() << endl;

 // iterating over all value of umap
 cout << "All elements : \n";
 for (itr2 = umap.begin(); itr2 != umap.end(); itr2++)   {
   // itr works as a pointer to pair<string, double>
   // type itr->first stores the key part and itr->second stroes the value part
   cout << itr2->first.c_str() << " " << itr2->second << endl;
}
cout << endl;
```

Adapted from https://www.geeksforgeeks.org/unordered_map-in-cpp-stl/

```
// map: the underlying STL implementation is a red-black tree

map<string, double> myMap; // Declaring map to be of <string, double> type
                           // key will be of string type and mapped value will be of double type

map<string, double>:: iterator itr;
string key;

printf("map\n");
printf("---\n");

// inserting values by using [] operator
myMap["PI"]    = 3.14;
myMap["root2"] = 1.414;
myMap["root3"] = 1.732;
myMap["log10"] = 2.302;
myMap["loge"]  = 1.0;

// inserting value by insert function

myMap.insert(make_pair("e", 2.718));

key = "PI";

if (myMap.find(key) == myMap.end())       // If key not found in map iterator to end is returned
   cout << key.c_str() << "not found\n\n";
else                                     // If key found then iterator to that key is returned
   cout << "Found " << key.c_str() << "\n";

key = "lambda";
if (myMap.find(key) == myMap.end())
   cout << key.c_str() << " not found\n";
else
   cout << "Found " << key.c_str() << endl;

 // iterating over all value of myMap
 cout << "All elements: \n";
 for (itr = myMap.begin(); itr != myMap.end(); itr++)    {
   // itr works as a pointer to pair<string, double>
   // type itr->first stores the key part and itr->second stroes the value part
   cout << itr->first.c_str() << " " << itr->second << endl;
}
```

Adapted from https://www.geeksforgeeks.org/unordered_map-in-cpp-stl/

Standard Template Library
STL

```
/* this is required for the unordered_map and the map classes */
/* so that they know how to compare the string keys           */

namespace std {
    template<>
    struct equal_to<string> {
        bool operator()(const string& a, const string& b) const {
           if (strcmp(a.c_str(), b.c_str()) == 0)
              return true;
           else
              return false;
        }
    };
}

namespace std {
    template<>
    struct less<string> {
        bool operator()(const string& a, const string& b) const {
           if (strcmp(a.c_str(), b.c_str()) < 0)
              return true;
           else
              return false;
        }
    };
}
```

```
stack
-----
3
2
1
stack is empty

queue
-----
queue size: 4
3
7
1
2
queue is empty

priority queue
--------------
queue size: 6
8
7
5
4
2
1
priority queue is empty

map
---
Found PI
lambda not found
All elements:
PI 3.14
e 2.718
log10 2.302
loge 1
root2 1.414
root3 1.732

unorded_map
-----------
Found PI
lambda not found
All elements :
PI 3.14
root2 1.414
root3 1.732
e 2.718
loge 1
log10 2.302
```

Standard Template Library
STL

STL Containers

1. Introduction to STL
2. Overview of Containers in STL
3. Pair Template
4. Tuple Template
5. Array
6. Vector
7. List
8. Map
9. Stack
10. Queue
11. Priority Queue
12. Deque

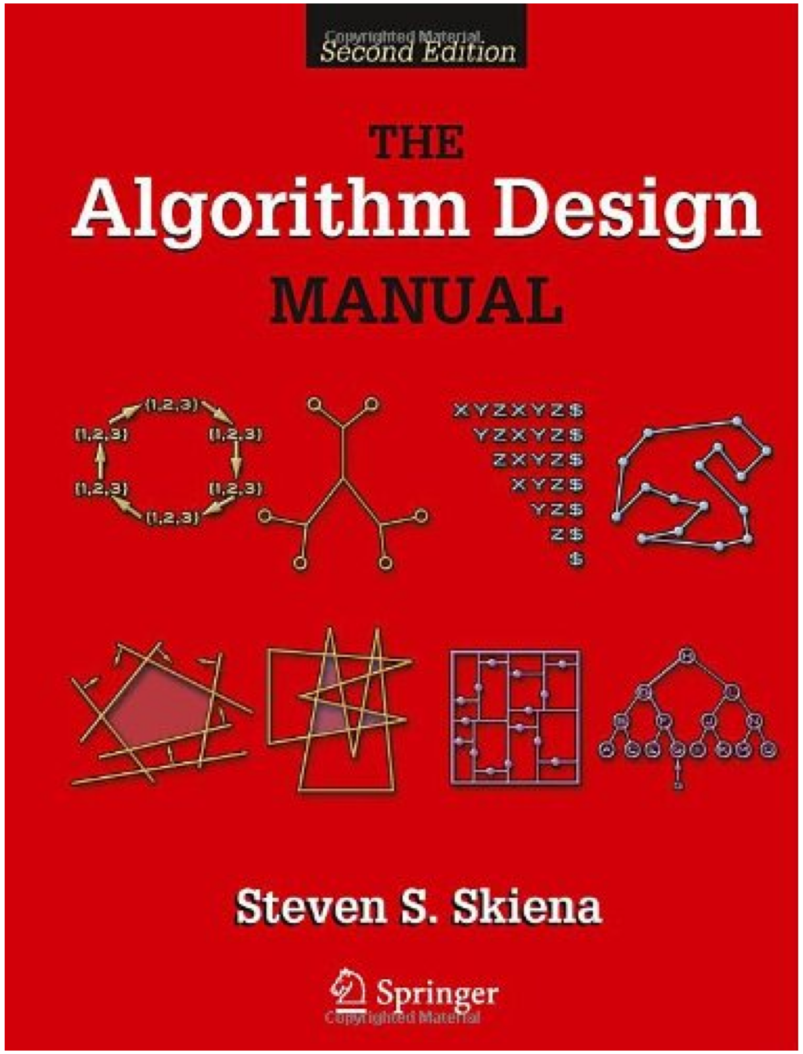
Iterators

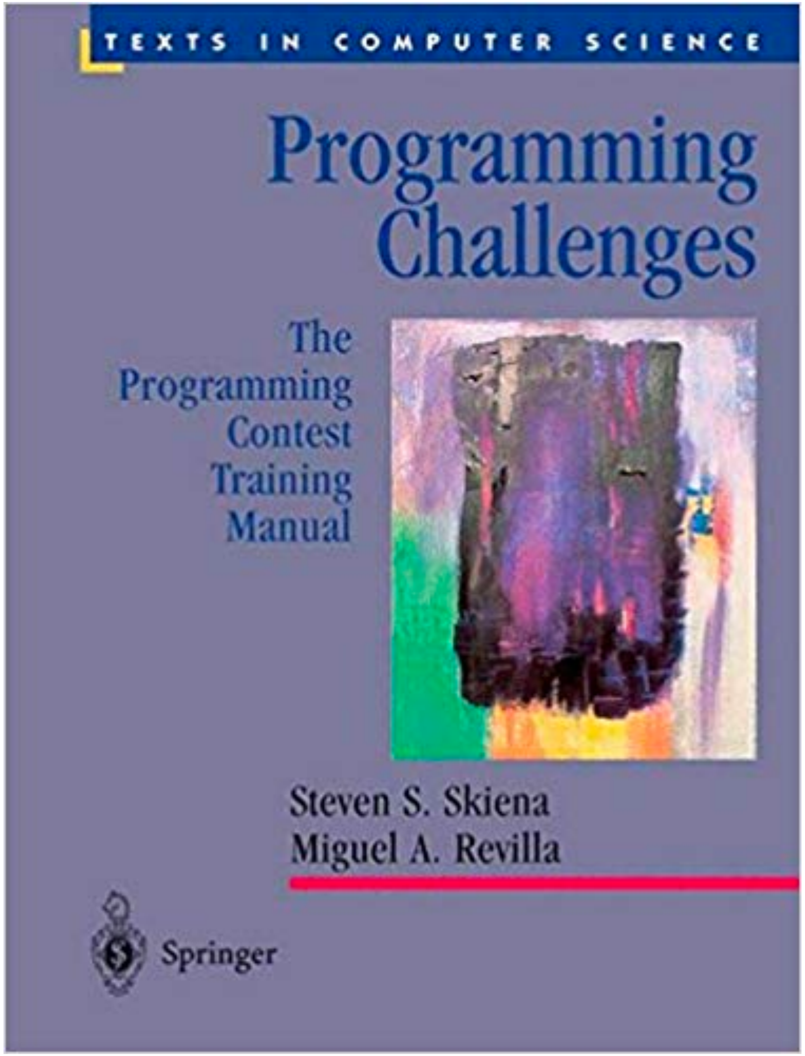
1. Introduction to Iterators
2. Operations on Iterators

Algorithms in STL

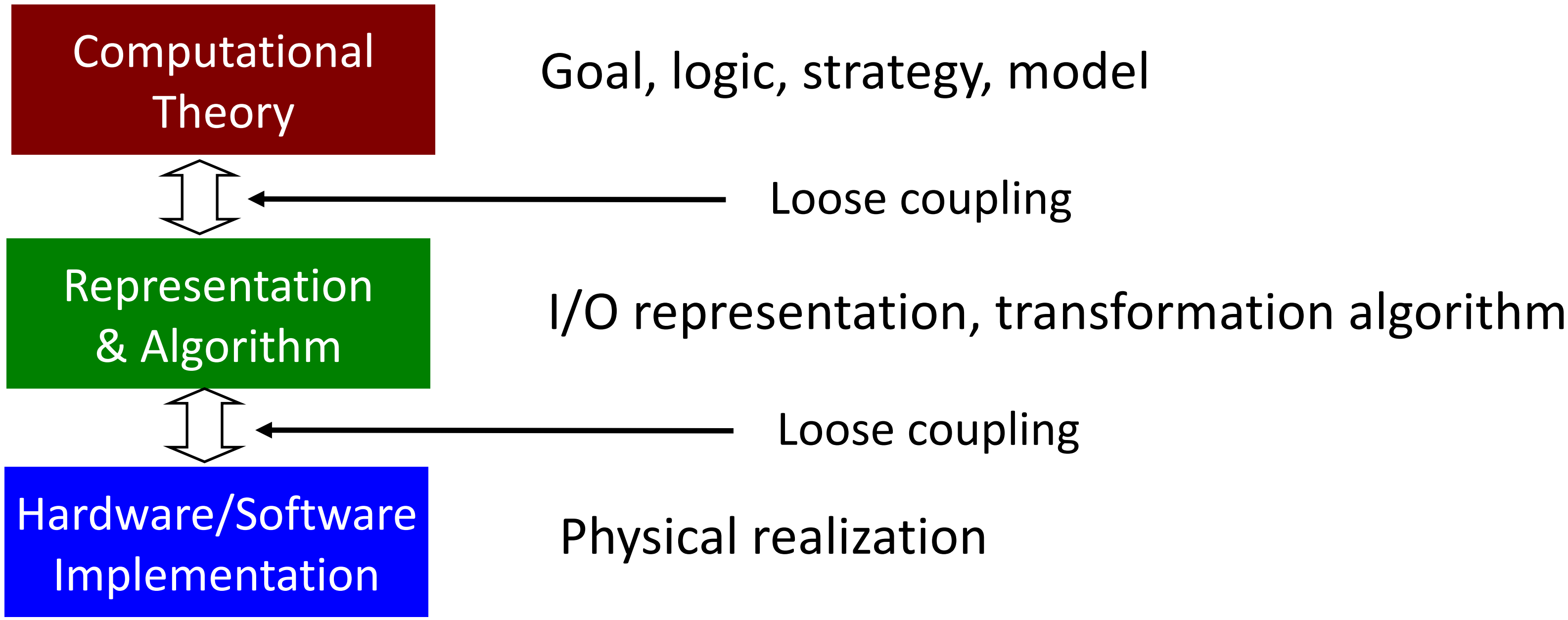
1. Overview of Algorithms
2. Sorting Algorithms
3. Binary Search and Equal Range
4. Upper Bound and Lower Bound
5. Non Modifying Algorithms
6. Modifying Algorithms
7. Numeric Algorithms
8. Minimum and Maximum operations
9. MinMax and Permutation operations

https://www.studytonight.com/cpp/stl/


Copyrighted Material
Second Edition
THE
Algorithm Design
MANUAL
XYZXYZ$
YZXYZ$
ZXYZ$
XYZ$
YZ$
Z$
$
Steven S. Skiena
Springer
Copyrighted Material


TEXTS IN COMPUTER SCIENCE
Programming
Challenges
The
Programming
Contest
Training
Manual
Steven S. Skiena
Miguel A. Revilla
Springer

Marr’s Hierarchy of Abstraction / Levels of Understanding Framework



04-630
Data Structures and Algorithms for Engineers

David Vernon
Carnegie Mellon University Africa

vernon@cmu.edu
www.vernon.eu