California State University, East Bay
Department of Mathematics and Computer Science

CS 4310 SOFTWARE ENGINEERING I

Introduction to the Software Life Cycle

Copyright 2006
Edward A. Billard
Contents

I Requirements ................................................................. 6
  1 System Narrative .......................................................... 8
  2 Interface Inputs ............................................................ 15
  3 Output Reports ............................................................. 17

II Analysis ................................................................. 19
  4 Function-Oriented Analysis ............................................... 24
  5 Object-Oriented Analysis ............................................... 28
  6 User Documentation ...................................................... 35

III Design ................................................................. 37
  7 Design Concepts: Cohesion vs. Coupling .............................. 39
  8 Interface Design .......................................................... 58
  9 Database Design ......................................................... 60
 10 Function-Oriented Design .............................................. 64
 11 Object-Oriented Design ............................................... 70
 12 Data Structure Design .................................................. 82
 13 Architectural Design .................................................... 85
 14 Procedural Design ....................................................... 87

IV Implementation ....................................................... 97
 15 Domain Objects .......................................................... 98
 16 Boot Sequence .......................................................... 102
 17 Presentation Layer ...................................................... 106
 18 Domain Logic ............................................................ 109
 19 Data Structures ........................................................ 118
 20 Database ............................................................... 121
V  Test ................................................................. 122
21 Black Box Test ......................................................... 125
22 White Box Test ......................................................... 128
23 Unit Test ............................................................... 130
24 Integration Test ......................................................... 132
25 Validation Test ......................................................... 135
26 Automated Test ......................................................... 137

VI Project Management ............................................... 141
27 Deliverables Schedule ............................................... 142
28 PERT Tasks ............................................................ 144
29 Project Assessment ................................................... 145
30 Team Assessment ..................................................... 148
The Software Engineering Life Cycle

- Software Engineering is a process of activities
- process must be systematic or repeatable in order to qualify as “engineering”
- goal is to produce software which satisfies the user requirements and is error-free
- “error-free” is a major attribute of quality
- the life cycle is the primary model for this process
- another model is the Spiral Model, which continually iterates on successive prototypes
- simplified version is Build-and-Fix
- life cycle model is viewed as a “waterfall”
- stages of the waterfall: requirements, analysis, design, implementation, test
- difficult to go back up the waterfall: if early stages are wrong, implementation will be wrong
- must start over and redo early stages which is very expensive in time, money, people
- early stages are abstract using general terms, models, diagrams
many standard definitions exist

using the keywords on the previous page, complete this definition:

Software Engineering is
Part I

Requirements

• start with customer-defined goals/constraints

• sometimes “fuzzy” objectives and ideas

• listen/elicit/clarify

• resolve differences of view

• advise on technical issues

• produce a requirements document with the problem definition

• system overview of capabilities

• constraints:
  • schedule, data volume, performance
  • reliability, security, cost

• economic feasibility: development cost vs. income/benefit

• technical feasibility

• development/operating/maintenance environments

• requirements are at the highest level of abstraction

• use more general terms, which would yield various implementations
• system is a black box with **Interface Inputs** and **Output Reports**: 

![Diagram of a black box with interface inputs and output reports]

• black box described by a **System Narrative**
1 System Narrative

- system overview of capabilities
- from the point of view of the user
- why the user will benefit from this product
- marketing description of the desirable features
- may be a “day in the life” presentation
- breakdown of the functional features
- include hints of the different entities and their relationships
Example: System Narrative

- Given this simple description of an employee system:

A company employee system has a set of departments (DEPT# unique), each with a budget (DBUDGET) and manager (MGR#).

Each department has a set of employees (EMP# unique), a set of projects (PROJ# unique), and a set of offices (OFF# unique).

Each employee has a PHONE# (unique but shared by other employees) and is assigned to a project.

Each employee has a job history, each with a DATE, JOBTITLE and SALARY.

Each project has a budget (PBUDGET).

Each office has a set of phones.

- See later for Object-Oriented Analysis, Object-Oriented Design, Database Design

- Note that this is a simplistic narrative, and usually would not contain the explicit FIELDS

- However, the narrative hints at the entities and the relationships between entities
• Given this simple description of a shipping system:

An order-entry system contains information about customers, items, and orders.

For each customer (CUST# unique), there is ship-to address (ADDRESS, several per customer), balance (BAL), credit limit (CREDLIM), and discount (DISCOUNT).

For each order (ORD# unique), there is a CUST#, ADDRESS and DATE.

For each line of the order (LINE#, several per order), there is an ITEM# (unique), quantity ordered (QTYORD) and quantity outstanding (QTYOUT).

For each item, there is a description (DESCN).

For each plant (PLANT# unique) that manufactures the item, there is a quantity on hand (QTYOH) and a stock danger level (DANGER).

For each plant, there is a manager (MGR).

• Perform later: Object-Oriented Analysis, Object-Oriented Design, Database Design
Stock Trader System

• a CASE STUDY will be examined at all phases of the life cycle

The Bestway Stock Trader provides a stock trading environment where investors have accounts with cash and stocks. Offers are made to buy and sell stocks at specific values and the system determines compatible matches. The system includes audit information of completed transactions. The application provides the necessary functionality to the investor to support investment activities.

• note the high level of abstraction of these introductory remarks

• following is the narrative for the Bestway Stock Trader system
BESTWAY STOCK TRADER

Always *The Best Way to Buy and Sell Stocks*

CS 4310 Software Engineering I: Requirements

Team Members:

Instructor: Prof. Ted Billard
Why Bestway Stock Trader

For intelligent investors who need to quickly take advantage of market conditions with proven trading strategies and capable tools at their fingertips, the Bestway Stock Trader is a web-based securities management tool that enables investors to maximize their assets and minimize their risk. Unlike other online trading tools, our product appeals to new and seasoned investors alike by providing an intuitive but powerful interface to a suite of market analytics combined with advanced trading logic for smarter and safer trades.

Create New Investor

New investors are welcomed into the system with an easy interface to provide investor identification, password, name, phone, and other personal data. A first account is then opened for the investor.

Create New Account

Besides the original account, say for growth, the investor may open additional accounts for IRA, college funds, etc. Investors can move between accounts and keep investment activities associated with specific accounts.

Login

Registered investors log into their accounts with an investor ID and password.

Display Account Information

Investors have access to various displays, including pending orders, a history of completed transactions, and a portfolio of stocks and cash. The portfolio computes the market value of all stocks based upon current market conditions.

Trade Stock

Investors can buy and sell stocks in a real-time environment. Before trading, the investor probably wants to examine current market conditions. Quotes are available for individual stocks, including recent trade prices. An investor interested in buying a stock can see the minimum sell price offered. Likewise, an investor interested in selling a stock can see the maximum buy price offered.

After making decisions, an investor can place buy or sell orders for specific stocks, including the number of shares and the desired price. The Bestway Stock Trader will execute these trades within one second and provide confirmation. The investor's portfolio is updated and a transaction is generated. Sometimes the price does not match current market conditions with the result that a pending order is generated for future markets. If the price is good but the number of shares is high, then the system transacts with as many other offers as possible, with the remaining quantity posted as a pending order.
Stock Trader: System Narrative

Investors can view their portfolio, pending orders, and transactions to monitor their trading activities. Investors can change pending orders for price and share, or delete the order.

Investors also have access to real-time graphs which trend the market conditions. Other advanced features include watchlists, stock filters, trading wizards, alert notification, and stock price streaming.

Cash Management

A sale of a stock results in increased cash value within the investor’s account. Of course, stock purchases require cash but investors can trade on margin, using their stocks as security.

Investors can deposit and withdraw cash into a specific account at any time.

Logout

Investors can exit the system at any time.

Conclusions

Our system provides all the functionality, with an easy-to-use interface, that the investor needs for successful stock management.

Open an account today and start trading!

The Bestway Stock Trader is always the best way to buy and sell stocks.
2 Interface Inputs

- system is a black box where inputs enter and outputs exit
- usually a Graphical User Interface (GUI) collects user inputs
- user enters inputs at text fields of GUI
- requirements specify the inputs, often as a data dictionary
<table>
<thead>
<tr>
<th>FIELD</th>
<th>VALUE</th>
<th>COMMENT</th>
</tr>
</thead>
<tbody>
<tr>
<td>Investor ID</td>
<td>jones</td>
<td>Investor’s unique ID</td>
</tr>
<tr>
<td>Password</td>
<td>jones</td>
<td>Secret password to login</td>
</tr>
<tr>
<td>First</td>
<td>John</td>
<td>Investor’s first name</td>
</tr>
<tr>
<td>Last</td>
<td>Jones</td>
<td>Investor’s last name</td>
</tr>
<tr>
<td>Phone</td>
<td>510-111-1111</td>
<td>Investor’s home phone number</td>
</tr>
<tr>
<td>Account#</td>
<td>1000</td>
<td>Investor currently logged into this account</td>
</tr>
<tr>
<td>Cash</td>
<td>1000.00</td>
<td>used to deposit/withdraw money from account</td>
</tr>
<tr>
<td>Stock</td>
<td>IBM</td>
<td>used for stock quote or to buy/sell</td>
</tr>
<tr>
<td>Shares</td>
<td>10</td>
<td>how many shares to buy/sell</td>
</tr>
<tr>
<td>Price</td>
<td>$82.50</td>
<td>how much is investor willing to pay, or must receive</td>
</tr>
</tbody>
</table>
3 Output Reports

- System is a black box where inputs enter and outputs exit
- Outputs are usually reports, especially in table format
- Outputs represent the point, or significance, of the system
- Reports give the user a feel for expected behavior
- Reports use example data so user can understand
Stock Trader: Output Reports

Stock Quote:

Quote: IBM Minimum SELL: $86.00 Maximum BUY: $82.50 Last Sale: $84.00

Transactions:

<table>
<thead>
<tr>
<th>Trans#</th>
<th>Date</th>
<th>Account</th>
<th>Stock</th>
<th>Type</th>
<th>Price</th>
<th>Shares</th>
</tr>
</thead>
<tbody>
<tr>
<td>1001</td>
<td>Fri May 05 15:49:33 PDT 2006</td>
<td>1001</td>
<td>CASH DEPOSIT</td>
<td>BUY</td>
<td>$1000.00</td>
<td>1</td>
</tr>
<tr>
<td>1003</td>
<td>Fri May 05 15:49:33 PDT 2006</td>
<td>1001</td>
<td>IBM</td>
<td>BUY</td>
<td>$80.00</td>
<td>5</td>
</tr>
<tr>
<td>1004</td>
<td>Fri May 05 15:49:33 PDT 2006</td>
<td>1001</td>
<td>IBM</td>
<td>SOLD</td>
<td>$82.50</td>
<td>5</td>
</tr>
<tr>
<td>1005</td>
<td>Fri May 05 15:49:33 PDT 2006</td>
<td>1001</td>
<td>CSCO</td>
<td>BUY</td>
<td>$20.00</td>
<td>100</td>
</tr>
</tbody>
</table>

Pending Orders:

<table>
<thead>
<tr>
<th>Order#</th>
<th>Date</th>
<th>Account</th>
<th>Stock</th>
<th>Type</th>
<th>Price</th>
<th>Shares</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000</td>
<td>Fri May 05 15:49:33 PDT 2006</td>
<td>1000</td>
<td>INTC</td>
<td>BUY</td>
<td>$19.0</td>
<td>50</td>
</tr>
<tr>
<td>1005</td>
<td>Fri May 05 15:49:33 PDT 2006</td>
<td>1000</td>
<td>IBM</td>
<td>SELL</td>
<td>$86.0</td>
<td>5</td>
</tr>
</tbody>
</table>

Portfolio:

<table>
<thead>
<tr>
<th>Stock</th>
<th>Shares</th>
<th>Market</th>
</tr>
</thead>
<tbody>
<tr>
<td>IBM</td>
<td>10</td>
<td>840.00</td>
</tr>
<tr>
<td>CSCO</td>
<td>100</td>
<td>2200.00</td>
</tr>
</tbody>
</table>

$3040.00

Cash

$1000.00

Net Worth

$4040.00
Part II
Analysis

- start with requirements document
  - problem recognition
  - natural language has ambiguities
- analysis steps:
  - evaluation and synthesis
  - modeling
- produce an analysis document for review
- precise, expose ambiguities/inconsistencies
- abstract essential features into a model
- basis for agreement with customer
- what must be done, not how
- design will use analysis model, not requirements document
- bad analysis yields high costs and poor product
• see the forest, not only the trees

• separate functionality (what) from the design (how)

• provide the basis for the creation of the design

• encompass system not just software

• encompass environment of the system

• cognitive model as perceived by user

• tolerant of incompleteness (it’s a model)

• localized and loosely coupled (it’s dynamic)

• if you’re modeling, you’re doing analysis

• if you’re doing analysis, you’re modeling
Analysis: Modeling

- test a physical entity before building it (cheaper)
- communicate with customers
- reduce complexity
- high level of abstraction
  - allows us to deal with complexity
  - selective examination of certain aspects
  - limit the universe
- good model captures crucial aspects of problem
- precise, concise, understandable, correct model of the real world
• function-oriented analysis: Data Flow Diagrams are the model
  • also called classical analysis, or structured analysis
• object-oriented analysis: Class Diagrams are the model
• database analysis: Entity-Relationship Diagrams are the model
• state analysis: State Transition Diagrams are the model
• “a picture is worth 10,000 words”
• overall description

• information description

• functional description
  
  • partitioning
  
  • narrative
  
  • restrictions/limitations
  
  • constraints
  
  • supporting diagrams

• behavioral description

• validation criteria: performance, classes of tests

• preliminary user’s manual

• prototype: extension/alternative to analysis

• allows user to “test-drive” the application

• users are better at “using” than “specifying”
4 Function-Oriented Analysis

• original form of analysis

• appropriate for implementation in FORTRAN, C, Pascal

• software designers think in terms of data flowing between processing sites

• processing sites become functions

• data becomes input/output parameters to functions

• create a Data Flow Diagram (DFD)

• read requirements document for:
  
  • verbs: actions become processing sites

  • nouns: things become external entities, internal data, data storage

• refine the DFD to expose more details of internal processing

• refinement is an essential characteristic of the software engineering process
• highest level is just the outside world and the system

• Investor is an external entity

• Stock Trading System is a processing site

• data flows from the Investor to the Stock Trading System

• confirmation flows from the Stock Trading System back to the Investor
- refine the original DFD to expose more processing detail
- but there still must be a “conservation” of inputs and outputs (as with energy)
- same data flows to/from Investor
- Process Buy is a processing site

- retrieve the minimum selling price data for the specified stock from the pending orders data storage
- adjust the Investor’s holdings of that stock in the holdings data storage
- generate a transaction into the transactions data storage
- update the market value of the stock in the stock data storage
- note the data is “flowing”
- refine DFD for even more processing detail

- make processing sites based on verbs from previous page: retrieve, adjust, generate, update

- in design, these will become functions:
  - ReceiveBuyOrder()
  - RetrieveBestSell()
  - AdjustHoldings()
  - GenerateTransaction()
  - UpdateMarketValue()

- data will become parameters
more modern approach

appropriate for implementation in C++, Java

software designers think in terms of data encapsulated in objects, methods operating on objects

create Class Diagrams which show relationships between classes

use Unified Modeling Language (UML) to draw Class Diagrams

UML is recognized industry standard for Object-Oriented Analysis and Design

the following is a UML tutorial, followed by OOA of employee example and Stock Trader application
Inheritance performs generalization/specialization ("is a"):

```
SuperClass
   ↓
SubClassX    SubClassY
```

- SubClassX inherits from SuperClass but still considered "is a" SuperClass, it "qualifies"
- SubclassX inherits all of the generalized data and methods of SuperClass
- SubClassX can add more specialized data and methods
- SubClassX can even "overload" a method of SuperClass, and reprogram it
- likewise for SubClassY

Relationships demonstrate multiplicity ("has a"):

1. first example: A is related to one B, and B is related to one A
2. second example: A is related to many B’s, but a B is related to only one A
• consider the requirements for the employee example

• the entities described become classes (later, objects will become instances of the classes)

• a Department has many Projects, which each have many Employees

• a Department also provides many Offices, which each have many Phones

• a Phone is shared by many Employees, which each have a resume of many previous Jobs

• very similar to entity-relationship diagrams in database design

• UML allows the naming of these relationships (e.g. “provides”) but “has” usually works

• note the two “many” asterisks towards Employee

• database design calls this an “intersection”

• this means that there is a many-to-many relationship between Projects and Phones
• consider the requirements document for the Stock Trader application

• examine the document for entities (classes) and their relationships

• an Account owns many Stocks (INTC, IBM, ...)

• and a Stock (e.g. IBM) is owned by many different Accounts

• many-to-many relationship between Accounts and Stocks

• in database relational design, an intersection table is inserted between the two

• in fact, there is a very natural intersection: Holding represents the act of ownership

• an Account has many Holdings, a Stock appears in many different Holdings

• one particular Holding belongs to just one Account, and is for just one Stock

• later, this class will track the number of shares this particular Account has of this particular Stock

• all many-to-many relationships should have this standard intersection insertion
- an Investor manages many different types of Accounts
- an Account has three different intersections with Stock
  - an Account
    - has many holdings of Stocks
    - has many pending orders for Stocks
    - has many past transactions for Stocks
- likewise, a Stock appears in many Holdings, Orders, Transactions
- a Holding is for one Account, one Stock
- an Order is for one Account, one Stock
- a Transaction, say BUY, is one Account, one Stock
- (a different Transaction, say SELL, is generated for the seller of the Stock)
- this model captures the high-level abstractions of the Stock Trader application
• the previous showed relationships and multiplicities between classes

• now consider any inheritance relationships

• Account is a super class

• IRA, College, Growth are each subclasses of Account

• IRA “is an” Account

• IRA shares general characteristics with College and Growth, say stock Holdings

• but IRA may have some specialized characteristics, say how taxes are paid, or not paid

• OOA models relationships and inheritance between classes

• OOD extends the model with attributes (variables) and methods

• this concludes OOA, but the following gives a preview of Database Design, OOD, and Java
• abbreviated preview of the following stages: OOA to Database to OOD to Java

• database schema adds fields, identifies primary and foreign keys

• OOD adds fields (attributes, variables) to UML

• Java provides the implementation

• consistency between stages: Investor has many Accounts, an Account belongs to one Investor

• database foreign key indicates that Account belongs to one Investor

• OOD Investor has a Vector of the many Accounts that it owns

• OOD Account has a reference to a single Investor

• Java has the same variables
• analysis describes \textbf{what} the system is supposed to do, not \textbf{how}

• analysis still has the aspect of a black box from the user’s point of view

• this means that user documentation could be specified as to \textbf{what} the black box will due

• usually, the document cannot be produced in its entirety (Interface Design has not occurred yet)

• the User’s Manual is produced later in the life cycle, but it should still be considered analysis
BESTWAY STOCK TRADER
User’s Manual

Function: Buy Stock

Synopsis: Use cash in investor’s account to purchase new shares of stock.

| Inputs: | 
| --- | --- |
| Stock | symbol of stock to purchase |
| Shares | number of shares to purchase |
| Price | highest price to pay for purchase |

Actions: Move cursor to Buy button, click on button.

Outputs: Confirmation: symbol, shares, price, remaining pending order.

Example:

- Stock: INTC
- Shares: 100
- Price: 22.0

Sale completed: 75 shares of INTC at $22.0
Current outstanding order: BUY 25 shares of INTC at $22.0

Function: Sell Stock

- note the emphasis on functionality, actions, inputs, outputs
• start with the analysis document

• goal is to produce a design document for review

• translate customer’s requirements into finished software

• design is a process of sufficient detail to permit realization

• how the software will do the what of analysis

• interface design: Graphical User Interface for inputs/outputs

• database design: relational schema of tables with fields and keys

• function-oriented design: transform DFDs into a control hierarchy of function calling function

• object-oriented design: add details of attributes and methods to class diagrams

• data structure design: select appropriate data structures for capabilities and performance

• architectural design: 3 tier architecture of presentation layer, business logic, storage

• procedural design: one step away from programming with pseudocode, flow charts

• object-oriented procedural design: UML collaboration/sequence diagrams
Design Concepts: Cohesion vs. Coupling

- outgrowth of modularity/abstraction/information hiding
- “single-minded” function
- aversion to excessive interaction
- address specific subfunction of requirements
- simple interface
- easier to develop/code/test/maintain
- functional independence $\Rightarrow$ good design $\Rightarrow$ quality
- qualitative measurements:
  - **cohesion:**
    - extent to which a module is single-minded
    - ranges from low to **high** (single task)
  - **coupling:**
    - extent of complexity of interconnections with other modules
    - ranges from **low** to high
- goal: **high** cohesion and **low** coupling
Design Concept: Cohesion

- extent to which a module is single-minded
Design Concept: Coincidental Cohesion

- multiple, unrelated tasks ("scatter-brained")
- action cannot be defined
- large program “modularized/segmented” into arbitrary partitions
- modularize unrelated, but repeated, code in other modules
- standards which require minimum LOC (Lines Of Code)
  - implode $n$ modules to form 1 module with unrelated tasks
- standards which require maximum LOC
  - explode 1 well-designed module to form $n$ modules
  - increases interaction (coupling)
  - small remainders of explosions get lumped together
- difficult to maintain
- not reusable
- example: $A=0$; write $B$
Design Concept: Logical Cohesion

- multiple, somewhat related tasks
- perform all input and output
- related tasks but one is selected by calling module
- implies passing control parameters
- selective use of data parameters based on control
- complex and interrelated
- difficult to understand and modify
- math library routines
- decompose (explode) into n routines
- example: read A or write B
- more details:

```c
void myFunction(int flag) {
    switch (flag) {
        case 1 : // perform task1
        case 2 : // perform task2
        case 3 : // perform task3
        ...
    }
}
```

- flag is a control parameter
- “calling” function determines (provides flag) which part of the myFunction code to exercise
- explode into individual functions for task1, task2, task3, ...  
- calling function should then call these directly
Design Concept: Temporal Cohesion

- multiple tasks which are executed at the same time
- does not have the selective control of logical cohesion
- program initialization/termination
- weakly related actions
- but strongly related to actions in other modules
- can cause a regression fault
- unlikely to be reused
- example: \( A = 0; B = 0; \)

Design Concept: Procedural Cohesion

- actions related to steps taken by product
- actions must be executed in a specific order
- read part from database and update maintenance file
- example: \texttt{read A; write B}
Design Concept: Communicational Cohesion

- actions related to steps taken by product
- AND performed on SAME data
- refer to the same set of input and/or output data
- example: read A; write A

Design Concept: Sequential Cohesion

- output of one element is input for the next element
- read transaction and update master file
- structure bears close resemblance to problem structure
- example: read A; B = A + 1; write B

Design Concept: Functional Cohesion

- exactly one action, task, goal ("single-minded")
- generally can be reused
- fault isolation
- easier maintenance
- example: compute square root
- example: read A
Design Concept: Informational Cohesion

- "object" cohesion
- contains complex data structure
- AND all operations on object (modify, inspect, etc.)
- EACH operation functionally cohesive
- similar to communication cohesion (same data)
- but communication module executes all code
- ex: public func init A; func read A; func write A; private A
- C++ encapsulation:

class Aclass {
public:
    Aclass(int i) {A = i;} // Constructor to init
    int read() {return A;}
    write(int a) {A = a;}
private:
    int A;
};
main() { Aclass A(7); A.write(5); A.read(); }
Design Concept: Coupling

- extent of complexity of interconnections with other modules
- goal: high cohesion and low coupling
Design Concept: Content Coupling

• one module directly references contents of another
• module \( x \) modifies a statement in module \( y \)
• assembly language and COBOL (\texttt{alter})
• module \( x \) refers to a local variable of \( y \) (by displacement)
• module \( x \) branches to a label within \( y \)

Design Concept: Common Coupling

• two modules both have access to the same global data
• two modules can read/write the same record in a database
• \texttt{common} statement in FORTRAN, COBOL
• global variables in C, Pascal
• contradicts structured programming
• modules can have side effects
• many modules may need to be modified
• difficult to reuse
• difficult and time-consuming to analyze
• not necessarily bad but beware of dangers
Design Concept: Control Coupling

- one module passes an element of control to another
- one module explicitly controls the logic of another
- flags passed as parameters or globals
- generally associated with modules with logical cohesion

Design Concept: Stamp Coupling

- data structure passed as parameter
- but called module operates on only some of the components
- difficult to know which subset it operates on
- pointer to a record

Design Concept: Data Coupling

- all parameters are homogeneous data items
- either simple or data structure (and all elements used)
- easier maintenance
- less likely to get a regression fault
Coupling and Cohesion Example

/* Main "first" reads the data,
   "then" sorts the data, and
   "after" writes the data => SEQUENTIAL COHESION
But main "does everything" => LOW COHESION
In general, the body could get long => make procedures
Array is STATIC and always INTEGER.
No other procedures => no coupling or interaction */

main () {
    int array[100]; int size; int i, j, tmp;
    scanf("%d", &size);
    for (i=0; i<size; i++)
        scanf("%d", &array[i]);
    for (i=0; i<size-1; i++)
        for (j=size-1; j>i; j--){
            if (array[j] < array[j-1]) {
                tmp = array[j];
                array[j] = array[j-1];
                array[j-1] = tmp;
            }
        }
    for (i=0; i<size; i++)
        printf("%d ", array[i]);
    printf("\n");
}
/* Main is now a "controller" from TRANSFORM ANALYSIS.
   Each function does one thing => FUNCTION COHESION
   BUT the functions share GLOBAL data => COMMON COUPLING
   Array is STATIC and always INTEGER
*/
int array[100]; int size; int i,j,tmp;
void read_array() {
    scanf("%d",&size);
    for (i=0; i<size; i++)
        scanf("%d",&array[i]);
}
void sort_array() {
    for (i=0; i<size-1; i++)
        for (j=size-1; j>i; j--){
            if (array[j] < array[j-1]) {
                tmp = array[j];
                array[j] = array[j-1];
                array[j-1] = tmp;
            }
        }
}
void write_array() {
    for (i=0; i<size; i++)
        printf("%d ",array[i]);
    printf("\n");
}
void main () {
    read_array();
    sort_array();
    write_array();
}
Coupling and Cohesion Example

/* Main gives a "code" to processor to EITHER read, sort, or write
   => LOGICAL COHESION for processor and CONTROL COUPLING between functions
*/
int array[100]; int size; int i,j,tmp;
#define READ_ARRAY 0 #define SORT_ARRAY 1 #define WRITE_ARRAY 2
void processor(int code) {
    switch (code) {
    case READ_ARRAY :
        scanf("%d",&size);
        for (i=0; i<size; i++)
            scanf("%d",&array[i]);
        break;
    case SORT_ARRAY :
        for (i=0; i<size-1; i++)
            for (j=size-1; j>i; j--){
                if (array[j] < array[j-1]) {
                    tmp = array[j];
                    array[j] = array[j-1];
                    array[j-1] = tmp;
                }
            }
        break;
    case WRITE_ARRAY: for (i=0; i<size; i++)
        printf("%d ",array[i]);
        printf("\n");
        break;
    }
}
main()
    processor(READ_ARRAY);
    processor(SORT_ARRAY);
    processor(WRITE_ARRAY);
/ * Global data is removed and passed as parameter
   and all data is used => DATA COUPLING
   Each function does one thing => FUNCTIONAL COHESION
   Main is a controller.
   Note that the data is "flowing" and any function,
      including main, can manipulate the structure.
   Array is DYNAMIC at "read" time but always INTEGER
   Code is INSTRUMENTED for debug purposes.
   read_array is a DRIVER of init_array.
   read_array is also a STUB.
   Main is a DRIVER.
   Code is DOCUMENTED for INPUT/OUTPUT/PURPOSE.
/*
#define NODBUG 0
#define PROCENTRY 1
#define IFENTRY 2
int dbug=NODBUG;

typedef int element_type;
struct array_record {
   unsigned int size;
   element_type *array;
};
typedef struct array_record *ARRAY;

/* INPUT : size of a dynamic array
   OUTPUT : pointer to the array structure
   PURPOSE: to create and initialize array
*/
ARRAY init_array(int size) {
   ARRAY A;
   A = (ARRAY) malloc(sizeof(struct array_record));
   A->array = (element_type *) malloc(sizeof(element_type)*(size));
   A->size = size;
   return(A);
}

/* INPUT : none
   OUTPUT : pointer to an array structure filled-in with input values
   PURPOSE: to init and read in the array values
*/
ARRAY read_array() {
   ARRAY A;
   int i;
   if (dbuf >= PROCENTRY) printf("Enter Read\n");
   A = init_array(4);
   /* Note that the function is a STUB and does not really read
      the data but acts as a driver of init_array */
   return A;
}
Coupling and Cohesion Example

/* INPUT : pointer to the array structure
   OUTPUT : pointer to the same structure (but could resize in other problems)
   PURPOSE: to sort values in the array using a bubble sort
*/
ARRAY sort_array(ARRAY A) {
    int i, j;
element_type tmp;
    if (debug >= PROCENTRY) printf("Enter Sort, size=%d\n", A->size);

    /* all values 0..i are sorted */
    for (i=0; i<A->size-1; i++)
        for (j=A->size-1; j>i; j--){
            if (A->array[j] < A->array[j-1]) {
                if (debug >= PROCENTRY) printf("Swap at positions %d and %d\n", i, j);
                tmp = A->array[j];
                A->array[j] = A->array[j-1];
                A->array[j-1] = tmp;
            }
        }
    /* all values 0..size-1 are sorted */
    return A;
}

/* INPUT : pointer to the array structure
   OUTPUT : none
   PURPOSE: write the sorted values to the console
*/
void write_array(ARRAY A) {
    int i;
    if (debug >= PROCENTRY) printf("Enter Write\n");
    for (i=0; i<A->size; i++)
        printf("%d ", A->array[i]);
    printf("\n");
}

/* INPUT : optional level of debug
   OUTPUT : none
   PURPOSE: to act as the main controller or driver
*/
main (int argc, char *argv[]) {
    ARRAY A;
    if (argc > 1) debug = atoi(argv[1]);
    if (debug >= PROCENTRY) printf("Enter Main\n");
    A = read_array();
    A = sort_array(A);
    write_array(A);
}
/* C++: Data is encapsulated (private) and each function does one thing
   => INFORMATIONAL COHESION
   Array can hold different data types => POLYMORPHIC
   Array is DYNAMIC at "read" time
   Initialization occurs for each new object.
   Main is a controller but does not see the structure
   Data is not "flowing"
*/
#include <iostream.h>
template <class Element_Type>
class Array {
   private:
      unsigned int size;
      Element_Type *array;
   public:
      Array(); // constructor
      void ReadArray();
      void SortArray();
      void WriteArray();
};
template <class Element_Type>
Array<Element_Type>::Array() {
   size = 0;
}
template <class Element_Type>
void Array<Element_Type>::ReadArray() {
    int i;
    cin >> size;
    array = new Element_Type[size];
    for (i=0; i<size; i++) cin >> array[i];
}

template <class Element_Type>
void Array<Element_Type>::SortArray() {
    int i, j; Element_Type tmp;
    for (i=0; i<size-1; i++)
        for (j=size-1; j>i; j--){
            if (array[j] < array[j-1]) {
                tmp = array[j];
                array[j] = array[j-1];
                array[j-1] = tmp;
            }
        }
}

template <class Element_Type>
void Array<Element_Type>::WriteArray() {
    int i;
    for (i=0; i<size; i++) cout << array[i] << " ";
    cout << endl;
}

main () {
    Array<int> A;
    A.ReadArray();
    A.SortArray();
    A.WriteArray();
}
• reduce coupling and increase cohesion

• *explode* a module that does more than one task

• if common component in multiple modules, explode the code

• *implode* two or more modules to lower coupling
  
  • reduce passage of control

  • reduce references to global data

  • reduce complexity of interface
• minimize high fan-out
  • can imply complex control code
  • helps to create more layers
  • top-level fan-out OK
    • helps to quickly partition software
    • uncomplicated menu options
• maximize high fan-in at low levels
  • implies good reuse of utility code
  • but at higher levels suggests high coupling

**Design Heuristics: Effect and Control**

• *scope of effect:* module $y$ affected by decision of module $x$
  • setting a global variable
• *scope of control:* module $y$ is subordinate to $x$ in control hierarchy
• keep effect within control (if necessary, move $y$ underneath $x$)
design an interface that allows for convenient inputs/outputs

- Graphical User Interfaces are usually event-driven: GUI responds to user actions
- usually multiple screens to help user through process
- following is a simple one screen GUI
- screen consists of labels, fields, buttons, and a text area for printing
- will be implemented using Java Abstract Windowing Toolkit (AWT)
- resultant StockApplet will be in tier one (presentation layer) of the eventual three-tier architecture
- use the requirements document to make an interface design:
  - add a label/field for each data element of Interface Inputs:
    - InvestorID, Password, Account, First, Last, Phone, Cash, Stock, Shares, Price
  - add a button for each report in Output Reports:
    - Pending Orders, Transactions, Quote, Portfolio
  - add a button for each action (verb) in your narrative:
    - Login, [make] New Investor, [make] New Account,
      Change Account, Deposit, Withdraw, Buy, Sell
  - add a TextArea for reports, confirmations, error messages, etc.
<table>
<thead>
<tr>
<th>Stock</th>
<th>Shares</th>
<th>Market</th>
</tr>
</thead>
<tbody>
<tr>
<td>IBM</td>
<td>10</td>
<td>640.00</td>
</tr>
<tr>
<td>CSCO</td>
<td>100</td>
<td>2200.00</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$3040.00</td>
</tr>
<tr>
<td>Cash</td>
<td></td>
<td>$1000.00</td>
</tr>
<tr>
<td>Net Worth</td>
<td></td>
<td>$4040.00</td>
</tr>
</tbody>
</table>
• database design consists of taking an Entity-Relationship (E-R) diagram and converting to tables

• UML class relationships show the same multiplicity as an E-R diagram

• each class (entity) becomes a table (class names are singular, table names are plural)

• the requirements narrative helps with the fields of the tables

• must establish a primary key that uniquely identifies each record in the table

• sometimes fields must be concatenated together to form a primary key
Example: Database Design

Database Schema of Tables with Fields:

Departments(Department.NO, DBUDGET, MGR_NO)
Projects(PROJ_NO, DEPT_NO, PBUDGET)
Offices(OFF_NO, DEPT.NO, AREA)
Phones(PHONE_NO, OFF_NO)
Employees(EMP_NO, PROJ_NO, PHONE_NO)
Jobs(EMP_NO, DATE, JOB_TITLE, SALARY)

Example Departments data:

<table>
<thead>
<tr>
<th>DEPT_NO</th>
<th>DBUDGET</th>
<th>MGR_NO</th>
</tr>
</thead>
<tbody>
<tr>
<td>D545</td>
<td>$1,000,000</td>
<td>M213</td>
</tr>
<tr>
<td>D246</td>
<td>$2,000,000</td>
<td>M456</td>
</tr>
</tbody>
</table>

Example Projects data:

<table>
<thead>
<tr>
<th>PROJ_NO</th>
<th>DEPT_NO</th>
<th>PBUDGET</th>
</tr>
</thead>
<tbody>
<tr>
<td>P103</td>
<td>D545</td>
<td>$100,000</td>
</tr>
<tr>
<td>P107</td>
<td>D246</td>
<td>$100,000</td>
</tr>
<tr>
<td>P115</td>
<td>D545</td>
<td>$200,000</td>
</tr>
</tbody>
</table>

- primary key of Departments is DEPT_NO (underline all primary keys)
- primary key of Jobs is EMP_NO, DATE (unless Employee started multiple jobs on same DATE)
- one Department has many Projects; a Project belongs to just one Department
- *the primary key of the "one" will appear as a field on the "many" side*
- therefore, DEPT_NO will be a field in the Projects table
- PROJ_NO P103 and P115 both belong to DEPT_NO D545
- this is called a foreign key, it identifies the one Department that the Project belongs to
- overline all foreign keys
- sometimes foreign key fields participate as part of the primary key
- PROJ_NO cannot be a field in the Departments table because Departments have many Projects
Example: Database Design

- Employees is an intersection table between Projects and Phones, both have “many” Employees
- Employees work on one Project, and have one Phone, so these are foreign keys in Employees table
- intersections will have always have the primary keys of both the “one” sides as foreign keys
- Employees has foreign keys: PROJ_NO, PHONE_NO:

  Projects(PROJ_NO)
  Employees(EMP_NO, PROJ_NO, PHONE_NO)
  Phones(PHONE_NO)

- tables should be “normalized”: no redundancy
- do not put the DEPT_NO in the Employees table
- Employees table already has PROJ_NO which determines the DEPT_NO via the Projects table
- every record where the Employee works on PROJ_NO P103, will have exactly the same DEPT_NO
- main points:
  - make class diagrams of entities with multiplicity to other classes
  - many-to-many relationships require an intersection
  - classes becomes tables
  - identify primary and foreign keys
  - the primary key of the “one” will appear as a field on the “many” side
  - fill-out tables with other fields
  - eliminate redundancies, but this is usually common sense
  - identify primary and foreign keys
  - good idea to work on class diagrams and schema at the same time
Database Schema:

Investors([INVESTOR_ID, PASSWORD, FIRST, LAST, PHONE])

Accounts([ACCOUNT_NO, INVESTOR_ID, CASH])

Holdings([ACCOUNT_NO, SYM, QTY])

Orders([ORDER_NO, DATE, ACCOUNT, SYM, TYPE, PRICE, QTY])

Transactions([TRANS_NO, DATE, ACCOUNT, SYM, TYPE, PRICE, QTY])

Stocks([SYM, PRICE])

- Investors have many Accounts, INVESTOR_ID is a foreign key in Accounts
- Accounts and Stocks have three intersections: Holdings, Orders, Transactions
- ACCOUNT_NO, SYM appear as foreign keys in Holdings, Orders, Transactions
- in the case of Holdings, ACCOUNT_NO, SYM also form the primary key
- TYPE is BUY or SELL
- pending Orders and Transactions have the same fields
- but pending Orders will disappear from the table once they are filled (i.e. QTY=0)
- Transactions are a permanent history
- database resides in tier three (storage) of the eventual three-tier architecture
- transition of flow to structure
- function-oriented analysis produced DFDs to model flow
- structure is a control hierarchy of function calling function, calling function
- main point of function-oriented design is to produce the control hierarchy
- control hierarchy can also be considered part of architectural design
- (architectural design and data structure design are often called preliminary design)
- function-oriented analysis performs “transform analysis” on the DFDs to yield a control hierarchy
- DFDs were output of analysis, now a different design methodology (transform analysis) is performed
- OOA and OOD will transition almost transparently
- above is high-level view of a DFD, with processing site 1, 2, ...
- data flows into the system with an external (user) representation
- data moves to the core with a more internal (say binary, or some data structure) representation
- data flows out of the core, and out of the system, back to a more external representation
- software engineer acts as a surgeon, and cuts around the core (may be off some)
- control hierarchy begins with main controller at root, with core (transform flow) controller just below
- incoming flow controller is added, especially if there are multiple streams as in the example
- outgoing flow controller is added, especially for multiple streams
- incoming calls func2, which calls func1, with data returned upwards
- transform calls all core functions (5, 6, 7, 8)
- outgoing calls func9 (which calls 11) and func10, with data flowing downwards as parameters
struct data_record {
    int data2;
    int data4;
};
typedef struct data_record *Data;

int main() {
    // main calls three controllers
    Data data = incomingFlow(); // data flows from incoming
    data = transformFlow(data);  // data flows to/from core
    outgoingFlow(data);          // data flows to outgoing
}

Data incomingFlow() {
    // data is returned upwards
    Data data;
    data->data2 = func2();
    data->data4 = func4();
    return data;
}

int func2() { return 2 + func1(); } // data is returned upwards
int func1() { return 1; }
int func4() { return 4 + func3(); }
int func3() { return 3; }

Data transformFlow(Data data) {
    // data flows to core subfunctions
    func5(data);
    func6(data);
    func7(data);
    func8(data);
    return data;
}
void func5(Data data) {}
void func6(Data data) {}
void func7(Data data) {}
void func8(Data data) {}

void outgoingFlow(Data data) {
    // data flows downwards via param
    func9(data);
    func10(data);
}
void func9(Data data) { func11(data); }
void func10(Data data) {}
void func11(Data data) {}
- starting from design DFD, make a cut of the core

- data moves inwards to ReceiveBuyOrder, and then moves outwards

- convert to control hierarchy
Stock Trader: Control Hierarchy

- no need for controllers because of simple flow

C code:

```c
int ReceiveBuyOrder(account, stock, price, qty) {
    order_ptr minsell = RetrieveBestSell(stock);    // data flows from
    if (minsell == NULL)
        return NO_STOCK_FOR_SALE;
    if (minsell->price > price)
        return NO_STOCK_AT_PRICE;
    AdjustHoldings(account, stock, qty, buy);        // data flows to
    AdjustHoldings(minsell->account, stock, qty, sell);
    GenerateTransaction(account, stock, buy, price, qty);
    GenerateTransaction(minsell->account, stock, sell, price, qty);
    UpdateMarketValue(stock, price);
    return STOCK_PURCHASED;                        // data returns to user
}
```
11 Object-Oriented Design

- OOA and OOD will transition almost transparently
- UML is the industry standard for OOA and OOD
- OOA class diagrams with relationships will transition to OOD class diagrams with attributes and methods
  - it is just a matter of applying more detail (less abstraction)
- although OOD uses class diagrams, the emphasis is on objects, rather than classes
- goal is to produce the Domain Objects, representing the business/application record structures
  - first, a UML tutorial on attributes and methods
  - second, a UML tutorial on the Vector class, which implements one-to-many relationships
  - finally, Domain Objects for the employer example, and for the Stock Trader application
Class Definition:

<table>
<thead>
<tr>
<th>ClassName</th>
</tr>
</thead>
<tbody>
<tr>
<td>attribute</td>
</tr>
<tr>
<td>attribute2:type</td>
</tr>
<tr>
<td>attribute3:type=initial value</td>
</tr>
<tr>
<td>-privateAttribute</td>
</tr>
<tr>
<td>#protectedAttribute</td>
</tr>
<tr>
<td>+publicAttribute</td>
</tr>
<tr>
<td>classAttribute</td>
</tr>
</tbody>
</table>

method1()
method2():type
method3(parameter:type)
-privateMethod()
#protectedMethod()
+publicMethod()
classMethod()

class ClassName {
    int attribute2;
    int attribute3=10;
    private int privateAttribute;
    protected int protectedAttribute;
    public int publicAttribute;
    static int classAttribute;

    void method1() { }
    int method2() {return 0;}
    void method3(int parameter) {}
    private void privateMethod() {}
    protected void protectedMethod() {}
    public void publicMethod() {}
    static void classMethod() {}
}

- UML is “attribute: int” rather than “int attribute”, likewise for parameters
- underline indicates static attributes or methods
- +, #, - indicate visibility
Navigability ("has a") where the Association is "uses":

```
class Client {
    Server s;
    void clientMethod() {
        s.serverOperation();
    }
}
```

- Client "uses" the Server, Client "has a" reference (handle) to Server
- "has a" and "is a" (inheritance) are the two ways of building structure

Multiplicity:

```
class Parent {
    java.util.Vector v1;
    java.util.Vector v2;
    Object o1;
    Object o2;
    Object o3;
}
```

- Java Vector is a container that stores references to "many" objects
**UML: Tutorial on Vector Container Class**

<table>
<thead>
<tr>
<th>Vector</th>
</tr>
</thead>
<tbody>
<tr>
<td>+Vector()</td>
</tr>
<tr>
<td>+addElement(o:Object): void</td>
</tr>
<tr>
<td>+elementAt(i:int): Object</td>
</tr>
<tr>
<td>+insertElementAt(o:Object,i:int): void</td>
</tr>
<tr>
<td>+removeElementAt(i:int): void</td>
</tr>
<tr>
<td>+removeElement(o:Object): void</td>
</tr>
<tr>
<td>+size(): int</td>
</tr>
</tbody>
</table>

- Java’s Vector contains a list of objects
- Vector can implement the “one-to-many relationships” in an application
- Vector is both a list with links and an array with indices
- addElement() to the end of the list
- get the elementAt() an index position (numbered from 0)
- insertElementAt() an index position (increase indices of all later nodes)
- removeElementAt() an index position (decrease indices of all later nodes)
- removeElement() based on the object reference, locate which node contains reference
- size() returns the number of elements in the list
- see the following Java example
import java.util.Vector;

class Person {
    private String name;
    public Person(String name) {
        this.name = name;
    }
    public String getName() {
        return name;
    }
}

public class VectorTest {

    public static void main(String args[]) {
        Vector v = new Vector(); // create a container
        Person p1 = new Person("Jones"); // create new objects
        Person p2 = new Person("Smith");
        Person p3 = new Person("Adams");
        v.addElement(p1); // add objects to the end of the list
        v.addElement(p2);
        v.addElement(p3);
        for (int i=0; i<v.size(); i++) { // iterate through the list 0..size()-1
            Person p = (Person)v.elementAt(i); // cast as Person, retrieve object at i
            System.out.println(p.getName()); // Jones, Smith, Adams
        }
        v.removeElementAt(0); // remove Jones based on index
        v.insertElementAt(p1,1); // insert Jones after Smith
        v.removeElement(p3); // remove Adams based on object

        for (int i=0; i<v.size(); i++) { // print again
            Person p = (Person)v.elementAt(i);
            System.out.println(p.getName()); // Smith, Jones
        }
    }
}
Vector Example

v.addElement(p1); v.addElement(p2); v.addElement(p3)

v.removeElement(0)

v.insertElementAt(p1,1)

v.removeElement(p3)

- Vector is both a list with links and an array with indices
- note the automatic re-numbering with insertions and deletions
Example: Domain Objects

• consider the employee example

• start with the OOA of the classes and relationships

• database schema adds fields, identifies primary and foreign keys

• add the fields from the schema to the class attributes, including types

• make the fields “public” for easy record access (see Java later)

• however, do not add any foreign keys

• remember that a foreign key points to “one” parent

• instead, make an object reference to this “one” parent

• Project belongs to a Department: Project gets a “dept: Department” reference

• literally captures the “one” relationship

• Departments have “many” Projects: Department gets a Vector called projects

• again, a literal translations of “many”

• OOA abstractly indicates one-to-many relationships between classes, OOD details with object references

• following are the Domain Objects for the employee example

• and then a Java example showing the object references
Example: Domain Objects

Diagram of domain objects:

- Department
  - DEPT_NO: String
  - DBUDGET: double
  - MGR_NO: int
  - projects: Vector
  - offices: Vector
  - Department()

- Office
  - OFF_NO: int
  - dept: Department
  - AREA: int
  - phones: Vector
  - Office()

- Project
  - PROJ_NO: int
  - dept: Department
  - PBUDGET: double
  - employees: Vector
  - Project()

- Employee
  - EMP_NO: int
  - project: Project
  - phone: Phone
  - jobs: Vector
  - Employee()

- Phone
  - PHONE.NO: String
  - office: Office
  - employees: Vector
  - Phone()

- Job
  - employee: Employee
  - DATE: String
  - JOBTITLE: String
  - SALARY: double
  - Job()
Example: Java References

Job job;  // assume these variables have been constructed
Employee emp;
Department dept;

// Job does not have EMP_NO, because it is a foreign key, instead use employee reference
System.out.println(job.employee.EMP_NO);

// this is why employee, EMP_NO are both ‘public’ (ease-of-use)
// detail an employee by visiting (with references) all the higher classes
System.out.println(emp.EMP_NO + emp.project.PROJ_NO + emp.phonePHONE_NO +
                   emp.phone.office.OFF_NO + emp.project.dept.DEPT_NO);

// display all the projects for a particular department
for (int i=0; i<dept.projects.size(); i++) {  // iterate through vector
    Project proj = (Project)dept.projects.elementAt(i);  // get reference to ith element
    System.out.println(proj.PROJ_NO);  // use reference
    System.out.println(dept.DEPT_NO + " equals " +
                       proj.dept.DEPT_NO);  // going down to project, back up to dept

    for (int i=0; i<dept.projects.employees.size(); i++) {  // nested iteration
        emp = (Employee)proj.employees.elementAt(i);  // get reference to ith element
        System.out.println(emp.EMP_NO);  // display all employees on project
    }
}

• public attributes make for easy record use (and should only be done for “structs” like these)
• easy to chase up a chain, or to iterate through all children
Stock Trader: Domain Objects

- same methodology used on the employee example is now applied to Stock Trader

- fields from database design are added, but not foreign keys (will be references)

- Investor has many (Vector) Accounts, Account has one reference (account: Account)

- Account has three intersections with Stock: Holding, Order, Transaction

- each is collected/contained in Account with a Vector

- each point back upwards to one parent Account

- remove() a Holding when all QTY is sold

- remove() an Order when it is no longer pending (all QTY transacted)

- Holding, Order, Transaction each has a reference to their parent Stock

- in this design, Stock just needs to know all of the pending Orders, but not Holdings, Transactions

- in data structure design, this Vector will become a Minheap of SELL, Maxheap of BUY Orders

- this will improve performance from linear to logarithmic
- usually inheritance is considered in OOA, but OOD can define/refine inheritance
- Domain Objects (e.g. Investor) are repositories for business data
- Domain Logic is the actual code that operates on Domain Objects
- usually simple operations on Domain Objects will be in the object itself (e.g. remove)
- global operations which transcend more parts of the system are extracted into Domain Logic
- global operation: access Orders and Holdings, adjust cash, generate transactions
- Stock Trader has three subclasses (subcomponents) for business code
- note that DomainLogic superclass is abstract, and only subclasses may be instantiated
- both Domain Objects and Domain Logic with be at tier two of the eventual three-tier architecture
• database design establishes the external memory storage

• data structure design does the same for internal memory storage

• data structure design affects the operations that may be performed

• and the execution speed of those operations

• these are crucial design decisions

• (control hierarchies, as architectural design, and data structure design are often called preliminary design)

• Stock Trader application: Hashtables, Vectors, Minheaps, Maxheaps

• use a Hashtable to quickly lookup an Investor at login

• use another Hashtable to quickly lookup Stock at BUY/SELL

• use a Vector to list all of the Accounts for a particular Investor

• for a particular Account, use 3 Vectors to list all Holdings, Orders, Transactions
- for each Stock, record the pending Orders in a Minheap and Maxheap

- Minheap: smallest SELL price is at the root

- Maxheap: largest BUY price is at the root

- new BUY Order comes in: check the Minheap for that Stock

- new SELL Order comes in: check the Maxheap for that Stock

- easy to check for a match, remove the Order from the Heap, and reheap
### Stock Trader: Performance Analysis

<table>
<thead>
<tr>
<th>DATA STRUCTURE</th>
<th>OPERATION</th>
<th>AVERAGE-CASE</th>
<th>WORST-CASE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stock Hashtable</td>
<td>Find</td>
<td>O(1)</td>
<td>O(x)</td>
</tr>
<tr>
<td></td>
<td>Insert</td>
<td>O(1)</td>
<td>O(x)</td>
</tr>
<tr>
<td></td>
<td>Delete</td>
<td>O(1)</td>
<td>O(x)</td>
</tr>
<tr>
<td>Min/Max Heap</td>
<td>Findmin/max</td>
<td>O(1)</td>
<td>O(1)</td>
</tr>
<tr>
<td></td>
<td>Insert</td>
<td>O(log m)</td>
<td>O(log n)</td>
</tr>
<tr>
<td></td>
<td>Delete</td>
<td>O(log m)</td>
<td>O(log n)</td>
</tr>
<tr>
<td>Match Overall</td>
<td></td>
<td>O(log m)</td>
<td>O(x + log n)</td>
</tr>
</tbody>
</table>

n = total # of pending orders  
x = # of stock symbols  
m = average # of pending/orders per stock = n/x  
n = worst-case # of pending orders per stock (all orders are for just one stock)

- very good average case performance: log m (where m is average number of Orders per Stock)
- worst-case is dominated by linear performance of Hashtable (very unlikely)
- overall performance based on matching algorithm:
  - lookup Stock in Hasetable, look at root of appropriate Heap  
  - if match, delete (get root and reheap)  
  - continue with new root until no more matches  
  - any remaining pending Order quantity goes into the other Heap  
  - see procedural design for more details on the algorithm
control hierarchies are often called architectural design, because they provide structure

however, architectural design also needs to consider overall structure of combining major components

standard way of putting together the components: three-tier architecture

Stock Trader three-tier architecture appears on next page, with the following comments

tier one: thin presentation layer, usually a Graphical User Interface

interface just collects inputs and displays output, no real logic

tier two: domain logic, where the business or application code operates on inputs, and produces outputs

domain logic uses domain objects to store business data in memory

tier three: storage, either permanent in a external SQL database or internal data structures

SQL and data structures can be considered utility code, not business dependent

boot sequence: Stockmain creates StockApplet, creates DomainLogic

StockInit pre-populates domain logic with initial in-memory database

StockTest is a button that triggers Validation Test

consider these actions:

- StockTest triggers a BUY of IBM in the StockApplet
- StockApplet responds with a doOrder to DomainLogic (TradeExecution)
  - TradeExecution does SQL on the external database and/or examines the root of the SELL Minheap
  - (in this project, the external database is not used, just internal data structures)
• procedural (detailed) design is one step away from programming
• algorithm specified at just above a programming language
• low level of abstraction
• designer should be able to give the procedural design to a programmer for coding
• any or all of the following techniques can be used
• pseudocode mainly used for function-oriented design, but can be used for OOD
• pseudocode can be English-like, with some structure for branching/iteration
• there are examples of formal pseudocode syntax such as Program Design Language (PDL)
• flow chart mainly used for function-oriented design
• prototype is a function specification of the inputs and outputs
• UML collaboration diagram shows methods triggering other methods
• UML sequence diagram is similar but along a timeline
scanOrders(account, stock, TYPE, QTY, PRICE) {
    while not done do {
        if this order’s QTY = 0 then done is TRUE  // QTY consumed
        else {
            if TYPE is BUY then {  // and similar for SELL
                if SELL minheap is empty then done is TRUE
                access order1 in the root of the SELL minheap
                if new order’s PRICE < order1’s PRICE then done is TRUE // no more possible matches
                else {
                    q = min of new order’s QTY and order1’s QTY1
                    this order’s QTY -= q
                    order1’s QTY1 -= q
                    update cash and generate transactions for both accounts
                    update stock market value
                    adjust holdings for both accounts
                    - if new stock, insert
                    - if old stock, update
                    - if selling all quantity, delete
                    if order1’s QTY1 = 0 then delete from minheap  // QTY1 consumed
                }
            }
            // end if statement
        }
        // end if statement
    } // end while loop
}

if this order’s QTY > 0 then {  // no more matches, QTY not consumed
    generate order number and insert this order into
    - BUY maxheap
    - account’s vector of orders
}
// end if statement
} //end constructor
- flow chart corresponds to the pseudocode for Stock Trader buy/sell matching
- conditional diamonds used to control branching and iteration
- may result in non-structured code, that is, spaghetti programming
Stock Trader: Prototypes

/* FUNCTION: ReceiveBuyOrder
   PURPOSE : To process a new buy order
   INPUTS :
       account purchaser
       stock to buy
       price offered
       qty of shares to buy
   RETURN : confirmation or error
   ALGORITHM: Given a purchase request, locate the best sell, adjust holdings,
              generate transactions, update the market value
   */
int ReceiveBuyOrder(int account, char *stock, double price, int qty);

Alternatively:

int ReceiveBuyOrder(                     // process a purchase order, return confirmation
    int account,                      // purchaser
    char *stock,                      // to buy
    double price,                     // offered
    int qty                           // shares to buy
);

order_ptr RetrieveBestSell(char *stock);

void AdjustHoldings(int account, char *stock, int qty, int type);

void GenerateTransaction(int account, char *stock, int type, price, qty);

void UpdateMarketValue(char *stock, price);

● prototypes specify the interface to a function in terms of inputs/outputs

● correct interfaces are important so that the control hierarchy works
UML: Tutorial

- UML is the industry standard for OOA and OOD
- OOA uses class diagrams
- OOD details attributes and methods at preliminary design
- OOD uses collaboration and sequence diagrams at procedural design
- following is a tutorial on UML, with examples, and with Stock Trader procedural design
Collaboration Diagram:

- `firstMessage()` in Class1, triggers 7 method calls
- `classMethod()` is a static call to Class2: `Class2.classMethod()`
- `create()` in UML is construction: “new”
- apply a method to an instance: `id2.message1(99)`
- 4 is an “if” statement: `[some boolean condition]`
- 5b is the “else” to the 5a “if”
- “for” loop: 6*: `[i:=1..10]`
- “while” loop (not shown): #*: `[some boolean condition]`
- 7: `size()` applied to collection of objects, not an individual object
class Class1 {
    private Class2 id2;
    private Class2 id3 = new Class2();
    private Class2 id4 = new Class2();
    private java.util.Vector v = new java.util.Vector();
    public Class1() {
        v.addElement(new Class2());
        v.addElement(new Class2());
        v.addElement(new Class2());
    }
    public void firstMessage() {
        boolean test = true;
        boolean test2 = true;
        Class2.classMethod(); // 1:
        id2 = new Class2(); // 2:
        id2.message1(99); // 3:
        if (test) id2.message2(); // 4:
        if (test2)
            id2.message3(); // 5a:
        else
            id3.message4(); // 5b:
        for (int i=1; i<=10; i++)
            id4.message5(); // 6:
        int size = v.size(); // 7:
    }
}
class Class2 {
    public static void classMethod() {}
    public void message1(int parameter){}
    public void message2(){}
    public void message3(){}
    public void message4(){}
    public void message5(){}
}
class Class3 {
    private Class1 id1 = new Class1();
    public Class3() {
        id1.firstMessage();
    }
}
Stock Trader: Collaboration Diagram

- adjust cash for both seller and buyer accounts
- remove seller's holding, create new holding for buyer
- generate transactions for both accounts, and adjust current value of stock

Java:

```java
public class TradeExecution {
    public void doSale(Account a1, Account a2, Stock s, double p, int q) {
        a1.CASH += q*p;                        // 1:
        a2.CASH -= q*p;                       // 2:
        Holding h1 = a1.holdings.elementAt(0); // for example
        h1.remove();                         // 3:
        Holding h2 = new Holding(a2, s, q);  // 4:
        Transaction t1 = new Transaction(s1, s, "SELL", p, q); // 5:
        Transaction t2 = new Transaction(a2, s, "BUY", p, q);   // 6:
        s.PRICE = p;                           // 7:
    }
}
```
Sequence Diagram:

class Sequence1 {
    private Sequence2 id2 = new Sequence2();
    public void message1() { id2.message2(); }
}
class Sequence2 {
    public void message2() {}
}
class Sequence3 {
    private Sequence1 id1 = new Sequence1();
    public Sequence3() {
        id1.message1();
    }
}

- collaboration diagram uses numbers to illustrate triggering events
- sequence diagram uses vertical timeline to illustrate triggering events
- sequence diagram corresponds to the Stock Trader collaboration diagram (and Java)
- implementation results in some form of binary (or interpretive) code

- by definition, this is the lowest level of abstraction

- Stock Trader system implemented in the Java language
15 Domain Objects

- Domain Objects developed in OOA and OOD
- represent the business or application record structures
- at the second tier in the three-tier architecture
- hold the application data
- if purely record structure, may have little actual code
- Stock Trader: Investor, Account, Holding, Order, Stock, Transaction
import java.util.Vector; // Java container class
import java.util.Hashtable; // Java lookup mechanism based on key

public class Investor {
    public String INVESTOR_ID; // primary key
    public String PASSWORD; // public visibility: same as a record struct
    public String FIRST;
    public String LAST;
    public String PHONE;
    public Vector accounts = new Vector(); // list of all account children: 1 to many
    // children add themselves to this container
    public static Hashtable investors = new Hashtable(); // find investor keyed on INVESTOR_ID
    // static: just one hashtable

    public Investor(String INVESTOR_ID, String PASSWORD, String FIRST, String LAST, String PHONE){
        this.INVESTOR_ID = INVESTOR_ID; this.PASSWORD = PASSWORD; this.FIRST = FIRST;
        this.LAST = LAST; this.PHONE = PHONE;

        investors.put(INVESTOR_ID, this); // put itself into the hashtable using primary key
    }
    public String toString() {
        return (INVESTOR_ID + " " + LAST + " " + FIRST + " " + PHONE);
    }
}

public class Account {
    public int ACCOUNT_NO; // primary key
    public Investor investor; // reference to Investor parent: many to 1
    public double CASH = 0; // public visibility: same as record struct
    public Vector holdings = new Vector(); // list of all holding children : 1 to many
    public Vector orders = new Vector(); // list of all order children : 1 to many
    public Vector transactions = new Vector(); // list of all transaction children: 1 to many
    private static int base_acct = 1000; // static var to auto generate account numbers

    public Account(Investor investor, double CASH) {
        this.investor = investor;
        ACCOUNT_NO = base_acct++;
        // auto generate
        this.CASH = CASH;
        investor.accounts.addElement(this); // add itself to the parent's list of children
    }
    public String toString() {
        return (ACCOUNT_NO + " " + investor.FIRST + " " + investor.LAST + " " + CASH);
    }
}
public class Holding {
    public Account account; // part of primary key, but not a string
    public Stock stock; // part of primary key, reference to parent
    public int QTY;

    public Holding(Account account, Stock stock, int QTY) {
        this.account = account; this.stock = stock; this.QTY = QTY;
        account.holdings.addElement(this); // add itself to the parent’s list of children
    }
    public String toString() {
        return (account.ACCOUNT_NO + " " + stock.SYM + " " + QTY); // access primary key string
    }
    public void remove() {
        account.holdings.removeElement(this); // remove itself when QTY is sold off
    }
}

import java.util.Vector; // Java container class
import java.util.Date; // auto generate a date, convert to String

public class Order {
    public int ORDER_NO; // primary key
    public String DATE; // date as a String
    public Account account; // reference to parent
    public Stock stock; // reference to parent
    public String TYPE; // public visibility: same as a record struct
    public double PRICE;
    public int QTY;
    public static int base_order = 1000; // static var to auto generate order numbers

    public Order(Account account, Stock stock, String TYPE, double PRICE, int QTY) {
        this.account = account; this.stock = stock; this.TYPE = TYPE; this.PRICE = PRICE;
        this.QTY = QTY;
        ORDER_NO = base_order++; // auto generate
        Date date = new Date(); // get date/time right now
        DATE = date.toString(); // store as a String
        account.orders.addElement(this); // add itself to the parent’s list of children
        if (TYPE.equals("SELL"))
            stock.sellheap.put(this); // add itself to it’s parent’s heap of orders
        else
            stock.buyheap.put(this);
    }
    public String toString() {
        return (ORDER_NO + " " + DATE + " " + account.ACCOUNT_NO + " " + stock.SYM + " " + TYPE + " " + PRICE + " " + QTY);
    }
    public void remove() {
        account.orders.removeElement(this); // remove itself when the QTY is consumed
    }
}
import java.util.Hashtable; // Java lookup mechanism based on key

public class Stock {
    public String SYM; // primary key
    public double PRICE; // public visibility: same as record struct
    public Heap buyheap = new Heap(true);
    public Heap sellheap = new Heap(false);
    public static Hashtable stocks = new Hashtable(); // find stock keyed on SYM
        // static: just one hashtable

    public Stock(String SYM, double PRICE) {
        this.SYM = SYM; this.PRICE = PRICE;
        stocks.put(SYM, this); // put itself into the hashtable using primary key
    }

    public String toString() {
        return (SYM + " " + PRICE);
    }
}

import java.util.Vector; // Java container class
import java.util.Date; // auto generate a date, convert to String

public class Transaction {
    public int TRANS_NO; // primary key
    public String DATE; // date as a String
    public Account account; // reference to parent
    public Stock stock; // reference to parent
    public String TYPE; // public visibility: same as record struct
    public double PRICE;
    public int QTY;
    public static int base_trans = 1000; // static var to auto generate tranaction numbers

    public Transaction(Account account, Stock stock, String TYPE, double PRICE, int QTY) {
        this.account = account; this.stock = stock; this.TYPE = TYPE; this.PRICE = PRICE;
        this.QTY = QTY;
        TRANS_NO = base_trans++;
            // auto generate
        Date date = new Date();
            // get date/time right now
        DATE = date.toString();
            // store as a String
        account.transactions.addElement(this); // add itself to it's parent's list of children
    }

    public String toString() {
        return (TRANS_NO + " " + DATE + " " + account.ACCOUNT_NO + " " + stock.SYM + " " + TYPE + " " + PRICE + " " + QTY);
    }
}
• to start system: java Stockmain

• Stockmain creates and runs StockApplet

• StockApplet creates StockInit, which pre-populates Domain Objects

• StockApplet creates Domain Logic: AccountManagement, ReportGenerator, TradeExecution

• StockApplet calls Domain Logic in response to the user pushing a button
<html>
<head>
<title>Bestway Stock Trader</title>
</head>
<body>
<applet
   code=StockApplet.class
   id=supp
   width=620
   height=840 />
</applet>
</html>

• alternative start: netscape stock.html

• load an html file, which loads StockApplet
import java.awt.*; // Java GUI classes
import java.applet.*;
import java.awt.event.*;

public class Stockmain{ // to run: java Stockmain
    public static void main(String args[]) {
        Applet applet = new StockApplet(); // create the applet and
        Frame frame = new StockFrame(applet); // put it into a frame
    }
}
class StockFrame extends Frame implements ActionListener {
    public StockFrame(Applet applet) {
        super("Bestway Stock Trader");
        MenuBar menubar = new MenuBar(); // make a menu with Quit
        Menu file = new Menu("File",true);
        menubar.add(file);
        file.add("Quit");
        setMenuBar(menubar);
        file.addActionListener(this);
        add("Center",applet);
        setSize(620,680);
        applet.init(); // start applet
        this.show();
    }
    public void actionPerformed (ActionEvent evt) {
        String arg = evt.getActionCommand();
        if (arg.equals("Quit")) { // if Quit, exit
            System.exit(0);
        }
    }
}
pre-populate the domain objects before GUI comes up

public class StockInit {
    public StockInit() { // called by StockApplet
        // create investors

        // create accounts for the investors above
        Account a1 = new Account(i1, 1000.0);
        Account a2 = new Account(i2, 1000.0);
        Account a3 = new Account(i3, 1000.0);

        // cash is used as the "stock" transaction for DEPOSIT/WITHDRAW
        Stock cash = new Stock("CASH", 0.00);

        // create stocks with market price
        Stock s1 = new Stock("IBM", 84.00);
        Stock s2 = new Stock("CSCO", 22.00);
        Stock s3 = new Stock("INTC", 20.00);

        // give some holdings to the accounts
        // concatenated primary key is really references to parents
        new Holding(a1, s1, 10);
        new Holding(a1, s2, 100);
        new Holding(a2, s3, 100);
        new Holding(a3, s2, 100);

        // generate some existing transactions
        new Transaction(a1, cash, "DEPOSIT", 1000.00, 1);
        new Transaction(a2, cash, "DEPOSIT", 1000.00, 1);
        new Transaction(a3, cash, "DEPOSIT", 1000.00, 1);
        new Transaction(a1, s1, "BUY", 80.00, 5);
        new Transaction(a1, s1, "SOLD", 82.00, 5);
        new Transaction(a1, s2, "BUY", 20.00, 100);
        new Transaction(a2, s3, "BUY", 18.00, 100);
        new Transaction(a3, s2, "BUY", 19.00, 100);

        // generate some outstanding pending orders
        new Order(a1, s3, "BUY", 19.00, 50);
        new Order(a2, s3, "SELL", 23.00, 50);
        new Order(a3, s3, "SELL", 21.00, 75);
        new Order(a3, s1, "BUY", 82.00, 5);
        new Order(a3, s1, "BUY", 82.50, 5);
        new Order(a1, s1, "SELL", 86.00, 5);
    }
}
17 Presentation Layer

- Presentation Layer implements the Interface Design
- at the first tier in the three-tier architecture
- Java provides Applets for Labels, TextFields, Buttons, TextAreas, etc.
- Applets are event-driven: respond to Buttons pushes
- StockApplet presents interface and calls DomainLogic to provide functionality
- StockApplet is “thin”, delegating real functionality to DomainLogic
import java.awt.*;                     // Java GUI classes
import java.applet.*;
import java.awt.event.*;

// Presentation Layer
public class StockApplet extends Applet implements ActionListener {
    
    // buttons
    private Button newinvestor, newacct, login, deposit, withdraw,
        portfolio, orders, trans, quote, buy, sell, test, chgacct;
    
    // labels for fields
    private Label acctLabel, firstLabel, lastLabel, idLabel, phoneLabel,
        passLabel, cashLabel, stockLabel, sharesLabel, priceLabel;
    
    private TextField acctField, firstField, lastField, idField, phoneField,
        passField, cashField, stockField, sharesField, priceField;
    
    // print to textArea
    private TextArea textArea;
    
    // references to Domain Logic
    private AccountManagement management;
    private ReportGenerator report;
    private TradeExecution trade;
    
    public void init() {
        super.init();
        setLayout(null);
        
        // create a label and field, place on the screen
        idLabel = new Label("Investor ID:");
        idLabel.setBounds(20,10,70,30);
        add(idLabel);
        idField = new TextField(130);
        idField.setBounds(100,10,80,30);
        add(idField);
        
        // ETC. for other labels and fields
        
        // create a TextArea for printing
        textArea = new TextArea();
        textArea.setBounds(20,270,580,350);
        Font font = new Font("Courier",Font.PLAIN,10);
        textArea.setFont(font);
        add(textArea);
        
        // create a button, place on the screen, applet "listens" for button push
        login = new Button("Login");
        login.setBounds(20,210,60,25);
        add(login);
        login.addActionListener(this);
        
        // ETC. for other buttons
        
        // pre-populate some Domain Objects
        new StockInit();
        
        // create the Domain Logic classes
        
        // hand over textArea1 so class can print to it (in this case, same TextArea)
        management = new AccountManagement(textArea);
        report    = new ReportGenerator(textArea);
        trade     = new TradeExecution(textArea);
    }
}
// called by actionPerformed whenever a button is pushed
// also called by StockTest to do Automated Test
public void doAction(String arg) {

    // depending on the button, call the appropriate method in Domain Logic
    if (arg.equals("Login"))
        management.doLogin(idField.getText(), passField.getText(), acctField, firstField, lastField, phoneField);
    if (arg.equals("Chg Acct"))
        management.doChgAcct(acctField.getText());
    if (arg.equals("New Investor"))
        management.doNewInvestor(idField.getText(), passField.getText(), firstField.getText(), lastField.getText(), phoneField.getText(), acctField);
    if (arg.equals("New Account"))
        management.doNewAccount(acctField);
    if (arg.equals("Deposit"))
        management.doCash("DEPOSIT", cashField.getText());
    if (arg.equals("Withdraw"))
        management.doCash("WITHDRAW", cashField.getText());
    if (arg.equals("Portfolio"))
        report.doPortfolio();
    if (arg.equals("Pending Orders"))
        report.showOrders();
    if (arg.equals("Transactions"))
        report.showTrans();
    if (arg.equals("Quote"))
        report.showQuote(stockField.getText());
    if (arg.equals("Buy"))
        trade.doOrder("BUY", stockField.getText(), sharesField.getText(), priceField.getText());
    if (arg.equals("Sell"))
        trade.doOrder("SELL", stockField.getText(), sharesField.getText(), priceField.getText());
    if (arg.equals("Test"))
        new StockTest(this, textArea1, acctField, firstField, lastField, idField, phoneField, passField, cashField, stockField, sharesField, priceField);
}

// called by the window manager whenever a button is pushed
public void actionPerformed(ActionEvent event) {
    String arg = event.getActionCommand();
    doAction(arg);
}
} // class StockApplet
• DomainLogic provides the functionality in response to user inputs

• at the second tier in the three-tier architecture

• super class DomainLogic is abstract and must be subclassed

• DomainLogic provides current investor/account for subclasses to share statically

• also provides some helper methods to share

• subclass AccountManagement provides new investor/account and login

• subclass ReportGenerator provides printouts of orders, transactions, quotes, portfolio

• subclass TradeExecution responds to new orders by scanning existing pending orders for matches
import java.awt.*; // Java GUI classes
import java.applet.*;

public abstract class DomainLogic { // super class, abstract => must be subclassed

    // protected variables so subclasses can access
    protected TextArea textarea1; // print to textarea1
    protected static Account account = null; // current account, static to share among subclasses
    protected static Investor investor = null; // current investor, static also

    public DomainLogic(TextArea textarea1) {
        this.textarea1 = textarea1;
    }

    // protected methods so subclasses can access
    protected boolean chkLogin() { // reports and trading can only occur after a login
        boolean ok = false;
        if (investor==null)
            textarea1.append("Must login first.
"");
        else
            if (account==null)
                textarea1.append("Must have an account.
"");
            else
                ok = true;
        return ok;
    }

    // helper methods for text processing
    protected double toDouble(String s) {
        Double d = new Double(0.0);
        try {
            d = Double.valueOf(s);
        } catch(Exception e) {return 0.0;}
        return d.doubleValue();
    }

    protected int toInt(String s) {
        Integer i = new Integer(0);
        try {
            i = Integer.valueOf(s);
        } catch(Exception e) {return 0;}
        return i.intValue();
    }

    protected String pad(String s, int width, String padChar, boolean padLeft) {
        for (int i=s.length()+1; i<=width; i++)
            if (padLeft)
                s = padChar + s;
            else
                s = s + padChar;
        return s;
    }
}
import java.awt.*;
import java.applet.*;

public class AccountManagement extends DomainLogic { // inherits from DomainLogic
    public AccountManagement(TextArea textArea1) {
        super(textArea1);
    }
    public void doChgAcct(String acct) { // change to a different account
        if (investor==null)
            textArea1.append("Must login first.\n");
        else {
            int acctno = toInt(acct);
            account = null;
            // iterate through the list of this investor's accounts
            for (int i=0; i<investor.accounts.size(); i++) {
                Account acct1 = (Account)(investor.accounts.elementAt(i));
                if (acct1.ACCOUNT_NO == acctno) {
                    account = acct1;
                    textArea1.append("Change Account: " + acctno +"\n");
                    return;
                }
            }
            textArea1.append("No Account #: " + acctno +"\n");
        }
    }
    public void doNewAccount(TextField acctField) { // generate a new account
        if (investor==null)
            textArea1.append("Must login first.\n");
        else {
            account = new Account(investor, 0.0);
            acctField.setText(account.ACCOUNT_NO+"");
            textArea1.append("New Account #: " + account.ACCOUNT_NO+"\n");
        }
    }
    public void doNewInvestor(String id, String password, String first, // generate new investor
        String last, String phone, TextField acctField) {
        investor = null;
        if (id.equals("") || password.equals("") || first.equals("") ||
            last.equals("") || phone.equals(""))
            textArea1.append("Must provide Investor ID, Password, First, Last, Phone.\n");
        else
            if (((Investor)Investor.investors.get(id) != null)
                textArea1.append("Investor ID already exists: "+id+"\n");
            else {
                investor = new Investor(id, password, first, last, phone);
                textArea1.append("New Investor Complete.\n");
                account = new Account(investor, 0.0);
                textArea1.append("New account number: "+account.ACCOUNT_NO+"\n");
                acctField.setText(account.ACCOUNT_NO+" ");
            }
    }
}
public void doLogin(String id, String passGiven, TextField acctField,  // validate login
TTextField firstField, TextField lastField, TextField phoneField) {
    investor = null;
    account = null;
    if (!id.equals("")) || (passGiven.equals(""))
            textArea1.append("Must enter Investor ID and Password.\n");
        else {
            investor = (Investor) Investor.investors.get(id);
            if (investor == null)
                textArea1.append("No such Investor ID: "+id+"\n");
            else
                if (!investor.PASSWORD.equals(passGiven))
                    textArea1.append("Incorrect password.\n");
                else {
                    if (investor.accounts.size() > 0) {
                        account = (Account)investor.accounts.elementAt(0);
                        acctField.setText(account.ACCOUNT_NO+"\n");
                    }
                    else
                        textArea1.append("No open accounts for: "+id+"\n");
                firstField.setText(investor.FIRST);
            lastField.setText(investor.LAST);
            phoneField.setText(investor.PHONE);
            textArea1.append("Investor Login: "+id+"\n");
        }
}

public void doCash(String type, String amount) {  // DEPOSIT or WITHDRAW cash
    if (chkLogin())
        if (amount.equals(""))
            textArea1.append("Empty Cash field.\n");
        else {
            double dollars = toDouble(amount);
            if (type.equals("DEPOSIT"))
                account.CASH = account.CASH + dollars;
            else
                account.CASH = account.CASH - dollars;
                Stock cash = (Stock) Stock.stocks.get("CASH");
                new Transaction(account, cash, type, dollars, 1);
                textArea1.append(type+ " completed.\n");
            }
    }
} // AccountManagement class
import java.awt.*;
import java.applet.*;

// Domain Logic
public class ReportGenerator extends DomainLogic { // inherits from DomainLogic
    public ReportGenerator(TextArea textarea) {
        super(textarea);
    }
    public void doPortfolio() { // display portfolio
        if (chkLogin()) {
            textarea.append("PORTFOLIO:\n");
            double sum = 0.0;
            textarea.append("Stock Shares Market:\n");
            textarea.append("---- -----
");
            for (int i=0; i<account.holdings.size(); i++) {
                Holding holding = (Holding) account.holdings.elementAt(i);
                String stockStr = pad(holding.stock.SYM, 7, " ", false);
                String qty = pad(String.valueOf(holding.QTY), 7, " ", true);
                double market = holding.QTY * holding.stock.PRICE;
                String marketStr = String.valueOf(market);
                textarea.append(stockStr + qty + marketStr + "\n");
            }
            String sumStr = Double.toString(sum);
            textarea.append("
"");
            textarea.append("\n");
            String cashStr = String.valueOf(account.CASH);
            textarea.append("Cash \n");
            textarea.append("\n");
            sum = sum * account.CASH;
            String sumStr = Double.toString(sum);
            textarea.append("Net Worth \n");
        }
    }
    public void showOrders() { // display orders
        if (chkLogin()) {
            textarea.append("ORDERS:\n");
            for (int i=0; i<account.orders.size(); i++) {
                Order order = (Order)account.orders.elementAt(i);
                textarea.append(order.toString() + "\n");
            }
            textarea.append("\n");
        }
    }
}
public void showTrans() { // display transactions
    if (chkLogin()) {
        textView.append("TRANSACTIONS:\n");
        for (int i=0; i<account.transactions.size(); i++) {
            Transaction trans = (Transaction)account.transactions.elementAt(i);
            textView.append(trans.toString() + "\n");
        }
        textView.append("\n");
    }
}
private String lastPrice(Stock stock) { // last traded price is stored in stock class
    return "$" + stock.PRICE;
}
private String maxPrice(Stock stock) { // maximum offered price is root of buyheap
    String price = "NONE";
    if (stock.buyheap.size() > 0) {
        Order order = stock.buyheap.testRoot();
        price = "$" + order.PRICE;
    }
    return price;
}
private String minPrice(Stock stock) { // minimum offered price is root of sellheap
    String price = "NONE";
    if (stock.sellheap.size() > 0) {
        Order order = stock.sellheap.testRoot();
        price = "$" + order.PRICE;
    }
    return price;
}
public void showQuote(String sym) { // lookup the stock in the hashtable, and display prices
    sym = sym.toUpperCase();
    if (sym.equals(""))
        textView.append("Empty stock field.\n");
    else {
        Stock stock = (Stock) Stock.stocks.get(sym);
        if (stock == null)
            textView.append("Invalid stock symbol: " + sym + "\n");
    else
        textView.append("Quote: " + sym + " Minimum SELL: " + minPrice(stock) +
        " Maximum BUY: " + maxPrice(stock) +
        " Last Sale: " + lastPrice(stock) + "\n\n");
    }
} // class ReportGenerator
public class TradeExecution extends DomainLogic {  // inherits from DomainLogic
    public TradeExecution(TextArea textArea) { super(textArea); }
    public void chgCash(Account account, double amount) {  // adjust cash after BUY/SELL
        account.CASH = account.CASH + amount;
    }
    private Holding getHolding(Account account, Stock stock) {  // find the holding record
        Holding holding = null;
        for (int i=0; i<account.holdings.size(); i++) {
            Holding holding1 = (Holding)account.holdings.elementAt(i);
            if (holding1.stock == stock) {
                holding = holding1;
                break;
            }
        }
        return holding;
    }
    // adjust QTY of holding, or remove
    private void chgHolding(Account account, Stock stock, int qty) {
        Holding holding = getHolding(account, stock);
        if (holding == null)
            new Holding(account, stock, qty);
        else {
            holding.QTY += qty;
            if (holding.QTY <= 0)
                holding.remove();
        }
    }
    private void chkOrder(Order order, int qty) {  // adjust QTY of order, or remove
        if (qty <= 0)
            order.remove();
        else
            order.QTY = qty;
    }
    // make sure a seller actually has the stock in the correct amount
    private boolean validOrder(Account account, Stock stock, String type, String price, String shares) {
        boolean OK = true;
        if (type.equals("SELL")) {
            OK = false;
            Holding holding = getHolding(account, stock);
            if (holding == null)
                textArea1.append("Account does not have any shares of " + stock.SYM + 
            else
                if (toInt(shares) > holding.QTY)
                    textArea1.append("Account only has " + holding.QTY + " shares of " + stock.SYM + 
                else
                    OK = true;
        }
        return OK;  
}
private boolean chkPrices(String type, double p1, double p2) {
    boolean OK = false;
    if (type.equals("BUY")) {
        if (p1 >= p2)
            OK = true;
    } else // SELL
        if (p1 <= p2)
            OK = true;
    return OK;
}

// process a sale for buyer/seller updating cash, holdings, transaction.
// Also, this sale price represents the stocks new "value", used in all portfolios
private void doSale(Account sellAccount, Account buyAccount, Stock stock, int qty, double price)
    {  
        chgCash(sellAccount, qty * price);  
        chgCash(buyAccount, -qty * price);  
        chgHolding(sellAccount, stock, -qty);  
        chgHolding(buyAccount, stock, +qty);  
        new Transaction(sellAccount, stock, "SELL", price, qty);  
        new Transaction(buyAccount, stock, "BUY", price, qty);  
        stock.PRICE = price;
    }

// StockApplet calls doOrder when a BUY/SELL button is pushed
// Validate the input fields and lookup the stock in the hashtable
public void doOrder(String type, String sym, String shares, String price) {
    sym = sym.toUpperCase();
    if (!chkLogin())
        if ((sym.equals("")) || (shares.equals("")) || (price.equals("")))
            textArea1.append("Must enter Stock, Shares, and Price.\n");
    else {
        Stock stock = (Stock)stock.stocks.get(sym);
        if (stock != null)
            textArea1.append("Stock does not exist: " + sym);
        else
            if (validOrder(account, stock, type, price, shares))
                scanOrders(account, stock, type,toInt(shares), toDouble(price));
    }
}
// main algorithm (see pseudocode) for matching buys with sells
private void scanOrders(Account account, Stock stock, String type, int qty1, double price) {
  int qty2, qty;
  Heap heap = stock.buyheap;
  if (type.equals("BUY"))
    heap = stock.sellheap;
  while (true) {
    if (qty1 <= 0) break;
    Order order = (Order) heap.testRoot();
    if (order == null) break;
    if (!chkPrices(type, price, order.PRICE)) break;

    order = heap.getRoot();
    qty2 = order.QTY;
    Account acct2 = order.account;
    if (qty1 <= qty2)
      qty = qty1;
    else
      qty = qty2;
    Account buyer;
    Account seller;
    if (type.equals("BUY")) {
      buyer = account;
      seller = acct2;
    }
    else {
      seller = account;
      buyer = acct2;
    }
    qty1 -= qty;
    qty2 -= qty;
    doSale(seller, buyer, stock, qty, price);
    chkOrder(order, qty2);
    textArea1.append("Sale completed: " + qty + " shares of " + stock.SYM +
                    " at $" + price + " with account: " + acct2.ACCOUNT_NO + 
                    "\n");
  } // while
  if (qty1 <= 0)
    textArea1.append(type + " completed.\n");
  else {
    new Order(account, stock, type, price, qty1);
    textArea1.append("Current outstanding order: " + type + " " + qty1 +
                    " shares of " + stock.SYM + " at $" + price + "\n");
  }
} // class TradeExecution
Data Structures

- data structures are the heart of any program
- determine the capabilities and performance of the system
- at the third tier in the three-tier architecture
- provide storage in internal memory
- can be re-used for other systems
- Java’s Vector is a container class, keeping lists of objects (e.g. Investor’s Accounts)
- Java’s Hashtable allows keyed lookup (e.g. Investor, Stock)
- Heap provides for Minheap (sell Orders) and Maxheap (buy Orders)
- hard-coded for Orders so cannot be re-used
- could be rewritten to use general objects, with comparable values
public class Heap {
    private boolean maxheap; // true => maxheap, false => minheap
    private static final int MAXSIZE = 100;
    private Order[] array = new Order[MAXSIZE];
    private int size = 0;
    private int maxsize = MAXSIZE - 1;

    public Heap(boolean maxheap) {
        this.maxheap = maxheap;
    }

    public int size() {
        return size;
    }

    public void put(Order order) { // add new element at end, restore heap
        int i;
        if (size < maxsize) {
            i = ++size;
            if (maxheap)
                while ((i > 1) && (array[i/2].PRICE < order.PRICE)) {
                    array[i] = array[i/2];
                    i /= 2;
                    // i becomes the parent
                }
            else
                while ((i > 1) && (array[i/2].PRICE > order.PRICE)) {
                    array[i] = array[i/2];
                    i /= 2;
                }
            array[i] = order;
        }
    }

    public Order testRoot() { // retrieve root
        if (size == 0)
            return null;
        else
            return (array[1]);
    }
}
public Order getRoot() { // retrieve root, remove root, restore heap
    if (size == 0)
        return null;
    else {
        Order root = array[1];
        Order last_element = array[size--];
        int i = 1;
        while (2*i <= size) { // 2*i is left child
            int child = 2*i;
            if (child != size) // find smaller child
                if (maxheap) {
                    if (array[child+1].PRICE > array[child].PRICE) // child+1 is right child
                        child++;
                }
                else
                    if (array[child+1].PRICE < array[child].PRICE)
                        child++;
                if (last_element.PRICE < array[child].PRICE) // percolate one level
                    array[i] = array[child];
                else
                    break;
            else
                if (last_element.PRICE > array[child].PRICE)
                    array[i] = array[child];
                else
                    break;
            i = child;
        }
        array[i] = last_element;
        return(root);
    }
}
• at the third tier in the three-tier architecture

• provide storage in external (permanent) memory

• not implemented in the Stock Trader prototype

• Stock Trader starts fresh each time from StockInit's pre-populated Domain Objects

• could be implemented using Java Database Connectivity (JDBC)

• supports SQL queries

• example 7 lines of JDBC code:

```java
import java.sql.*;
public class jdbc {
    public static void main(String args[]) {
        try {
            // 1: load specific code to talk to mysql
            Class.forName("org.gjt.mm.mysql.Driver");

            // 2: establish network connection to server
            Connection con = DriverManager.getConnection("jdbc:mysql://gold.mcs.csueastbay.edu:3306/SuppDB", "4311", "4311");

            // 3: environment for queries
            Statement stmt = con.createStatement();

            // 4: SQL query of a supplier table
            stmt.execute("select * from S where STATUS > 10");

            // 5: rs encapsulates rows which match the query
            ResultSet rs = stmt.executeQuery();

            // 6: iterate through the rows with a cursor
            while (rs.next()) {
                // 7: extract columns from the row pointed to by the cursor
                System.out.println(rs.getString("S_NO") + " " + rs.getInt("STATUS");
            }
        } catch (Exception e) {System.out.println(e.getMessage());}
    }
}
```
Part V
Test

Test
- Black Box Test
  - Equivalence Partitioning
    - Boundary Value Analysis
  - White Box Test
    - Path Coverage
  - Unit Test
    - JUnit
  - Integration Test
    - Top-Down
      - Bottom-Up
        - Sandwich
  - Validation Test
    - Validation Test Plan
  - Automated Test
    - Stock Test
• intention is to find errors/defects (NOT to NOT find errors)

• cannot prove the absence of defects

• good test case ⇒ high probability of finding undiscovered error

• successful test case ⇒ found undiscovered error

• minimum time and effort

• test to the specs

• main phases: UNIT, INTEGRATION, VALIDATION TEST

• other phases follow: SYSTEM, ALPHA, BETA, STRESS, PERFORMANCE, REGRESSION TEST

• applicable techniques: WHITE and BLACK BOX TEST

• if WHITE BOX TEST is done, usually at UNIT TEST

• BLACK BOX TEST can usually be applied at any phase

• BLACK BOX TEST is always applied at VALIDATION TEST

• tests may be AUTOMATED
Test Strategies

• verification: “Are we building the product right?”

• validation: “Are we building the right product?”

• when are we done testing?
  • never (burden shifts to customer)
  • out of time/money
  • number of new errors is statistically “small”

• software development: large to small

• software testing: small to large

<table>
<thead>
<tr>
<th>1. requirements</th>
<th>↔</th>
<th>4. system test</th>
</tr>
</thead>
<tbody>
<tr>
<td>2. analysis</td>
<td>↔</td>
<td>3. validation test</td>
</tr>
<tr>
<td>3. design</td>
<td>↔</td>
<td>2. integration test</td>
</tr>
<tr>
<td>4. code</td>
<td>↔</td>
<td>1. unit test</td>
</tr>
</tbody>
</table>
• focuses on functional requirements
  • incorrect or missing functions
  • interface errors
  • errors in data structures or external database access
  • performance errors
  • initialization/termination errors
• black box is usually applied later than white box
• what *classes* of input make good test cases?
• how are boundaries of a data class isolated?
• what data rates and volume can be tolerated?
• what effects of specific combinations of data?
Black Box Test: Equivalence Partitioning

- divide input domain into classes of data

- generate test cases (ideal: one case discovers class of errors)

- If the input specifies a RANGE of valid values, define one valid EC (within range) and two invalid ECs (one below the range and one above the range). Example: If the input requires a month in the range 1..12,

  Valid EC: 1..12   Test Case: 4
  Invalid EC: <1   Test Case: -1
  Invalid EC: >12   Test Case: 20

- If the input specifies an exact NUMBER (N) of inputs, define one valid EC (with N inputs) and two invalid ECs (<N and >N). Example: If the input requires 3 book titles,

  Valid EC: 3   Test Case: 3
  Invalid EC: <3   Test Case: 1
  Invalid EC: >3   Test Case: 5

- If the input specifies a SET of valid values, define one valid EC (within set) and one invalid EC (outside set). Example: If the input requires one of the names TOM, DICK, or HARRY,

  Valid EC: in set   Test Case: TOM
  Invalid EC: not in set   Test Case: JOE

- If the input specifies a "must be" situation, define one valid EC and one invalid EC. Example: If the first character of the input must be a digit,

  Valid EC: 0..9   Test Case: 4
  Invalid EC: not 0..9   Test Case: x
• errors tend to be at boundaries, not in “center”

• use equivalence but use values at “edges” of class

• If the input specifies a RANGE of valid values, select around the edges. Example: If the input requires a month in the range 1..12,

  Valid EC: 1..12  Test Cases: 1,12
  Invalid EC: <1  Test Case: 0
  Invalid EC: >12  Test Case: 13

• If the input specifies an exact NUMBER (N) of inputs, select around the edges. Example: If the input requires 3 book titles,

  Valid EC: 3  Test Case: 3
  Invalid EC: <3  Test Case: 2
  Invalid EC: >3  Test Case: 4
• instead of using a function specification, examine the sources to develop test cases

• loop testing
  • simple loop iterations: 0, 1, 2, \( m, n \), \( n - 1 \), \( n \), \( n + 1 \)
  • nested loops
    • start at innermost, set others to minimum
    • do simple loop test
    • work outward, set inner loops to typical

• code coverage
  • try to execute all code by following every path in the flow chart
  • on next page, the flow chart has 5 predicates, 6 regions
  • cyclometric complexity = # predicates + 1 = # regions = 6
  • graph theory says that there are exactly 6 independent paths
  • not all of these paths are feasible, as shown in the table
  • note that the 6 paths listed “touch” a new edge each time
  • this guarantees that the paths are independent
Assuming TYPE= “BUY” stock:

<table>
<thead>
<tr>
<th>#</th>
<th>PATH</th>
<th>PRICE</th>
<th>PRICE1</th>
<th>QTY</th>
<th>QTY1</th>
<th>COMMENT</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0-1-10-11-12</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>infeasible, done=T initially</td>
</tr>
<tr>
<td>2</td>
<td>0-1-10-12</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>infeasible, QTY &gt;0 initially</td>
</tr>
<tr>
<td>3</td>
<td>0-1-2-3-1-10-12</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>infeasible, QTY &gt;0 initially</td>
</tr>
<tr>
<td>4</td>
<td>0-1-2-4-5-6-1-10-11-12</td>
<td>$48</td>
<td>$50</td>
<td>*</td>
<td>*</td>
<td>no match at all</td>
</tr>
<tr>
<td>5</td>
<td>0-1-2-4-5-7-8-1-2-3-1-10-12</td>
<td>$50</td>
<td>$48</td>
<td>100sh</td>
<td>110sh</td>
<td>only QTY consumed</td>
</tr>
<tr>
<td>6</td>
<td>0-1-2-4-5-7-8-9-1-2-4-5-6-1-10-11-12</td>
<td>$50</td>
<td>$48</td>
<td>100sh</td>
<td>90sh</td>
<td>only QTY1 consumed</td>
</tr>
</tbody>
</table>
• test of smallest unit of software: module, function, object

• verify interface flow-in/flow-out

• do local data structures work?

• are boundary conditions handled?

• appropriate error handling?

• white box: use independent paths

• black box: use input/output parameters and return value

• need a driver:
  • calls module with input
  • receives output

• stubs: replace subordinate modules

• may have to wait until integration
JUnit is a package that supports the BLACK BOX test of Java code UNITS (Objects)

- acts as a “test harness”
- create an object, call methods of the object
- check the returned values using assertEquals()
- throw an Exception if the assertion fails
- run a test suite which contains all of the tests to perform
- TestResult captures the thrown Exceptions and stores in a log

```java
import junit.framework.*;
class MyCalendar {
    public static String getName(int month) { // int becomes corresponding String month
        switch (month) {
            case 1: return "January";
            case 2: return "February";
            case 3: return "March";
            case 4: return "April";
            case 5: return "May";
            case 6: return "June";
            case 7: return "July";
            case 8: return "August";
            case 9: return "September"; // THIS IS NOT SPelled CORRECTLY
            case 10: return "October";
            case 11: return "November";
            case 12: return "December";
            default: return "UNKNOWN";
        }
    }
}

class MyCalendarTest extends TestCase { // MyCalendarTest is a TestCase
    public MyCalendarTest(String s) {
        super(s);
    }

    public void testGetName() {
        String name = MyCalendar.getName(9); // call a method with parameter 9
        assertEquals(name,"September"); // examine the return value
    }
}

class AllTests {
    public static void main(String args[]) {
        TestSuite suite = new TestSuite(); // create a new test suite
        suite.addTest(new MyCalendarTest("testGetName")); // add just one test to the suite
        // indicate method to call
        TestResult result = new TestResult(); // create a log
        suite.run(result); // run all test cases
        result.print(); // print the log
    }
}
```

OUTPUT:
Errors:
    MyCalendarTest.testGetName(): MyCalendarTest // discovered the problem
• program structure dictated by design

• errors associated with interfacing

• errors associated with indirect effects

• incremental integration

**Top-Down:**

• BFS, DFS from the main program

• replace subordinate stubs with real modules one at a time

• good: verifies major control points early

• bad: stubs may delay real testing

```c
void A() {
    B();
    C();
    D();
}
void B() {
    stub1();
}
void C() {
    stub2();
}
void D() {
    stub3();
}
```
**Integration Test**

**Bottom-Up:**

- low-level modules are combined into clusters (builds)
- good: do not need stubs, just driver
- bad: program does not exist until last module

```c
void driver() {
    D();
    E();
    F();
}
void E() {
}
void F() {
}
void G() {
}
```
Sandwich:

- bad: need both drivers and stubs, but maybe middle layers were developed first

```c
void driver() {
    G();
    H();
    I();
}
void G() {
    stub1();
}
void H() {
    stub2();
}
void I() {
    stub3();
}
```
• “reasonable expectations” of user-visible attributes by customer

• specified early in Validation Criteria (Analysis)

• always black box

• test outcome of function or performance:
  • accepted
  • not accepted: deficiency list (negotiate)

• create a Validation Test Plan that details all interactions and expected results
<table>
<thead>
<tr>
<th>TEST</th>
<th>INPUTS</th>
<th>EXPECTED OUTPUTS</th>
</tr>
</thead>
</table>
| 1.   | Investor ID: jones  
      | Password: jones   
      | push Login        | First: John       
      | Last: Jones       
      | Account: 1000    
      | Phone: 510-111-1111 | |
| 2.   | push Pending Orders  
      | push Transactions  
      | push Portfolio     | Pending Orders: 2  
      | Transactions: 4  
      | Cash: 1000.00    
      | Net Worth: 4040.00 | |
| 3.   | Cash: 1000  
      | push Deposit     | Cash: 1500.00     
      | Cash: 500         | push Withdraw     
      | push Portfolio    | |
| 4.   | Stock: INTC  
      | push Quote       | Min SELL: 21.00   
      | Cash: 1000        | Max BUY: 19.00    
      | Last Sale: 20.00  | |
| 5.   | Stock: INTC  
      | Shares: 100      | Outstanding Order: 100 sh | |
      | Price: 19.50    | push Buy         | |
| 6.   | Price: 22       | push Buy         | Sale completed: 75 sh  
      | push Buy         | Outstanding Order: 25 sh | |
| 7.   | Shares: 25      | push Buy         | Sale completed: 25 sh | |
| 8.   | Stock: IBM      | push Buy         | Sale completed: 5 sh   
      | Shares: 10       | push Buy         | Sale completed: 5 sh | |
| 9.   | push Pending Orders  
      | push Transactions  
      | push Portfolio     | Pending Orders: 4  
      | Transactions: 10  
      | Cash: 95.00       
      | Net Worth: 4595.00 | |
| 10.  | push New Account  
      | Account: 1003    | Account: 1003      
      | push Chg Acct     | Cash: 0.00        
      | push Portfolio    | Net Worth: 0.00    | |
| 11.  | Investor ID: james  
      | First: Henry     | Account: 1004      
      | Last: James       | Cash: 0.00        
      | Phone: 510-444-4444 | Net Worth: 0.00    | push New Investor  
      | push Portfolio    | push Portfolio    |
• instead of manually running test cases, automate the test cases

• JUnit is a harness that supports automated test

• scripting often used to execute programs and verify results

• following page shows StockTest Java source to automatically run the Validation Test Plan

• StockTest called by GUI when “Test” button is pushed

• StockTest sets fields of GUI just like a user

• StockTest calls doAction() in GUI just like a button push

• system behaves as if the user was manually running test cases

• StockTest is not fully automated: tester must examine the printed results
// automatically run the same test cases in the Validation Test Plan

import java.awt.*;
import java.applet.*;

public class StockTest {

    public StockTest( // called by StockApplet when Test button is pushed

        // must have access to applet to set fields and textarea to print
        StockApplet applet, TextArea ta, TextField acctField, TextField firstField,
        TextField lastField, TextField idField, TextField phoneField, TextField passField,
        TextField cashField, TextField stockField, TextField sharesField, TextField priceField) {

        ta.append("---------------------------------------------------------------------\n");  
        ta.append("BEGIN: AUTOMATED TEST \n");  
        ta.append("---------------------------------------------------------------------\n");  

        ta.append("---------------------------------------------------------------------\n");  
        ta.append("TEST: 1 | Investor ID: jones | First: John \n");  
        ta.append("Password: jones | Last: Jones \n");  
        ta.append("push Login | Account: 1000 \n");  
        ta.append(" | Phone: 510-111-1111 \n");  
        ta.append("---------------------------------------------------------------------\n");  

        // set the fields in the applet, call doAction just like a button push
        // let applet do the usual calls to StockReport
        idField.setText("jones");  
        passField.setText("jones");  
        applet.doAction("Login");  

        ta.append("---------------------------------------------------------------------\n");  
        ta.append("TEST: 2 | push Pending Orders | Pending Orders: 2 \n");  
        ta.append(" | push Transactions | Transactions: 4 \n");  
        ta.append(" | push Portfolio | Cash: 1000.00 \n");  
        ta.append(" | pull Portfolio | Net Worth: 4040.00 \n");  
        ta.append("---------------------------------------------------------------------\n");  

        applet.doAction("Pending Orders");  
        applet.doAction("Transactions");  
        applet.doAction("Portfolio");  

        // ETC.

        ta.append("---------------------------------------------------------------------\n");  
        ta.append("END: AUTOMATED TEST \n");  
        ta.append("---------------------------------------------------------------------\n");  
    }
}
BEGIN: AUTOMATED TEST

TEST: 1
<table>
<thead>
<tr>
<th>Investor ID: jones</th>
<th>First: John</th>
</tr>
</thead>
<tbody>
<tr>
<td>Password: jones</td>
<td>Last: Jones</td>
</tr>
<tr>
<td>push Login</td>
<td>Account: 1000</td>
</tr>
<tr>
<td></td>
<td>Phone: 510-111-1111</td>
</tr>
</tbody>
</table>

Investor Login: jones

TEST: 2
<table>
<thead>
<tr>
<th>push Pending Orders</th>
<th>Pending Orders: 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>push Transactions</td>
<td>Transactions: 4</td>
</tr>
<tr>
<td>push Portfolio</td>
<td>Cash: 1000.00</td>
</tr>
<tr>
<td></td>
<td>Net Worth: 4040.00</td>
</tr>
</tbody>
</table>

ORDERS:
1000 Fri May 12 08:30:08 PDT 2006 1000 INTC BUY 19.0 50
1005 Fri May 12 08:30:08 PDT 2006 1000 IBM SELL 86.0 5

TRANSACTIONS:
1000 Fri May 12 08:30:08 PDT 2006 1000 CASH DEPOSIT 1000.0 1
1003 Fri May 12 08:30:08 PDT 2006 1000 IBM BUY 80.0 5
1004 Fri May 12 08:30:08 PDT 2006 1000 IBM SOLD 82.0 5
1005 Fri May 12 08:30:08 PDT 2006 1000 CSC0 BUY 20.0 100

PORTFOLIO:
<table>
<thead>
<tr>
<th>Stock</th>
<th>Shares</th>
<th>Market</th>
</tr>
</thead>
<tbody>
<tr>
<td>IBM</td>
<td>10</td>
<td>840.0</td>
</tr>
<tr>
<td>CSC0</td>
<td>100</td>
<td>2200.0</td>
</tr>
</tbody>
</table>

\[\begin{array}{c}
\text{Cash} \\
\$ 1000.0 \\
Net Worth \\
\$ 4040.0 \\
\end{array}\]

ETC.

END: AUTOMATED TEST
Other Tests

- SYSTEM: performed on all hardware/software components
- STRESS: check demands by quantity, frequency, volume
- PERFORMANCE: instrument the code to measure speed, etc.
- SECURITY: attempt to prevent hackers
- RECOVERY: resumption of processing after crash
- ALPHA: performed in-house by developer with simple developer data
- BETA: performed at site by trusted client with live data
- REGRESSION: performed before new releases to insure “old” code still works
Part VI
Project Management

- large-scale software engineering projects require extensive and detailed management
- resources (money, time, people) are precious and scarce
- the following pages answer these important management questions:
  - what are the deliverables, and when are they due?
  - what are the effects of one task on another task?
  - how will the project be assessed?
  - how will the team members be accessed?
- managers in industry need to do the same activities
Deliverables Schedule

- the following page shows the deliverables schedule
- due date is established for each of the 15 tasks
- allocate a team member as lead on each task
  - 10 non-Java deliverables
  - 3 novice Java deliverables using example templates
  - 1 optional Java programming of data structures including hashing
  - 1 extensive Java implementation
- many ways to contribute, each member leads about 5 deliverables
- allocate other members (or none) as contributors on each task
- the lead member needs to:
  - provide the key input into achieving the goal of the deliverable
  - coordinate the activities of the contributors
  - pay attention to timelines
  - keep track of the STATUS
    - TO DO → EXPECT date → DONE date
  - pay attention to the project assessment (delivery, complete, correct, systematic)
    (see Project Assessment)
  - pay attention to the team member assessment (leadership, support, work habits)
    (see Team Assessment)
  - make sure the goals are met
- deliverables schedule will be turned in at the end of the project
<table>
<thead>
<tr>
<th>#</th>
<th>DELIVERABLE</th>
<th>DUE</th>
<th>STATUS</th>
<th>LEAD</th>
<th>CONTRIBUTORS</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>Requirements</td>
<td>Week 2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>[Skill: Writing]</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2.</td>
<td>Object-Oriented Analysis</td>
<td>Week 3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>[Class Relationships]</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3.</td>
<td>Interface Design</td>
<td>Week 4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>[Screen Layout]</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4.</td>
<td>Database Design</td>
<td>Week 4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>[Tables/Fields]</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5.</td>
<td>Object-Oriented Design</td>
<td>Week 4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>[Extend OOA]</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6.</td>
<td>Domain Objects</td>
<td>Week 6</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>[Novice Java/Code OOD]</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7.</td>
<td>Data Structure Design</td>
<td>Week 6</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>[Hashing]</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8.</td>
<td>Presentation Layer</td>
<td>Week 6</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>[Novice Java/Code IF]</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9.</td>
<td>Procedural Design</td>
<td>Week 7</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>[Pseudocode]</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10.</td>
<td>Boot Sequence</td>
<td>Week 7</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>[Novice Java/Code new()]</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11.</td>
<td>Three-Tier Architecture</td>
<td>Week 7</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>[Copy Template]</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>12.</td>
<td>Data Structures</td>
<td>Week 9</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>[Optional Java]</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>13.</td>
<td>Domain Logic</td>
<td>Week 10</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>[Extensive Java]</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>14.</td>
<td>Validation Test</td>
<td>Week 10</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>[Detailed Black Box]</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>15.</td>
<td>User Documentation</td>
<td>Week 11</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>[Writing]</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
• Program Evaluation and Review Technique (PERT)

• shows the timeline and task dependencies

• if a task is late, that affects other tasks downstream

• how does a project get one year behind? one day at a time
following page shows the project assessment table for the 15 deliverables

- each deliverable is assessed by criteria: delivery, complete, correct, systematic

- each criteria is evaluated by the Project Manager (Instructor) as EXCELLENT, GOOD, FAIR, POOR

- the page following shows the definitions of each criteria

- this assessment, along with team/team member assessment, will be used to determine grades
<table>
<thead>
<tr>
<th>#</th>
<th>TASK</th>
<th>DELIVERY</th>
<th>COMPLETE</th>
<th>CORRECT</th>
<th>SYSTEMATIC</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>Req</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2.</td>
<td>OOA</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3.</td>
<td>IF</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4.</td>
<td>DB</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5.</td>
<td>OOD</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6.</td>
<td>Obj</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7.</td>
<td>DSD</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8.</td>
<td>GUI</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9.</td>
<td>Proc</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10.</td>
<td>Boot</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11.</td>
<td>Arch</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>12.</td>
<td>DS</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>13.</td>
<td>Logic</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>14.</td>
<td>Valid</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>15.</td>
<td>User</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Overall</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Project Assessment Criteria

<table>
<thead>
<tr>
<th>Project Criteria</th>
<th>EXCELLENT</th>
<th>GOOD</th>
<th>FAIR</th>
<th>POOR</th>
</tr>
</thead>
<tbody>
<tr>
<td>Delivery*</td>
<td>at due date</td>
<td>up to 1 week late</td>
<td>1-2 weeks late</td>
<td>over 2 weeks late</td>
</tr>
<tr>
<td>Complete</td>
<td>all areas finished</td>
<td>simple omissions</td>
<td>some areas missing</td>
<td>most areas missing</td>
</tr>
<tr>
<td>Correct</td>
<td>Code: outputs match inputs only minor faults</td>
<td>a few noticeable faults reasonable behavior</td>
<td>faulty but runs some areas deficient</td>
<td>unpredictable crash</td>
</tr>
<tr>
<td></td>
<td>Documents: interfaces to prev doc: premises match conclusions of previous document new conclusions follow premises good language syntax good diagram syntax</td>
<td>ditto</td>
<td>some interface faults faulty interface</td>
<td>faulty interface</td>
</tr>
<tr>
<td>Systematic</td>
<td>process-oriented highly organized exhibits clarity consistent intelligent approach</td>
<td>process-oriented organized some lack of clarity inconsistent sometimes approach slightly off</td>
<td>process breakdowns some disorganization unclear at times inconsistent sometimes some wrong directions</td>
<td>no process disorganized unclear inconsistent wrong direction</td>
</tr>
</tbody>
</table>

*No materials will be accepted after the official end of the course.*

- each deliverable is assessed by criteria: delivery, complete, correct, systematic
- each criteria is defined in terms of EXCELLENT, GOOD, FAIR, POOR
- code vs. document deliverables are evaluated differently for correctness
### TEAM MEMBER:

<table>
<thead>
<tr>
<th>Team Member Criteria</th>
<th>EXCELLENT</th>
<th>GOOD</th>
<th>FAIR</th>
<th>POOR</th>
</tr>
</thead>
<tbody>
<tr>
<td>Leadership</td>
<td>many new ideas</td>
<td>new ideas</td>
<td>few ideas</td>
<td>no ideas</td>
</tr>
<tr>
<td></td>
<td>project manager</td>
<td>good team lead</td>
<td>passive</td>
<td>disruptive</td>
</tr>
<tr>
<td>Support</td>
<td>actively helped lead</td>
<td>helped lead</td>
<td>helped when asked</td>
<td>no help</td>
</tr>
<tr>
<td></td>
<td>sought out other opinions</td>
<td>listened to others</td>
<td>sometimes listened</td>
<td>never listened</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>dominated others</td>
</tr>
<tr>
<td>Work Habits</td>
<td>worked hard</td>
<td>good work effort</td>
<td>worked</td>
<td>did not work</td>
</tr>
<tr>
<td></td>
<td>delivered on-time</td>
<td>close to on-time</td>
<td>trouble with on-time</td>
<td>could not deliver</td>
</tr>
</tbody>
</table>

### TEAM #:

<table>
<thead>
<tr>
<th>Overall Team Criteria</th>
<th>EXCELLENT</th>
<th>GOOD</th>
<th>FAIR</th>
<th>POOR</th>
</tr>
</thead>
<tbody>
<tr>
<td>Synergy</td>
<td>more than sum of parts</td>
<td>sum of parts</td>
<td>spinning gears</td>
<td>non-existent</td>
</tr>
<tr>
<td>Operation</td>
<td>very smoothly</td>
<td>good coordination</td>
<td>managed to get by</td>
<td>crisis mode</td>
</tr>
<tr>
<td>Workload</td>
<td>equally balanced</td>
<td>mostly balanced</td>
<td>one did not help</td>
<td>one did all</td>
</tr>
<tr>
<td>Goals</td>
<td>achieved</td>
<td>mostly achieved</td>
<td>partial success</td>
<td>not achieved</td>
</tr>
</tbody>
</table>

- each team member is assessed by criteria: leadership, support, work habits
- each criteria is defined in terms of EXCELLENT, GOOD, FAIR, POOR
- each team is assessed by criteria: synergy, operation, workload, goals
- each criteria is defined in terms of EXCELLENT, GOOD, FAIR, POOR