CS3157: Advanced Programming

Lecture #13

Dec 5

Shlomo Hershkop

shlomo@cs.columbia.edu

Overview

- Last lecture
  - Software engineering
    - Will cover most in class, you are responsible for understanding high level overview
  - PHP
    - Will cover in class and next lab.

What is Software Engineering?

- Stephen Schach: “Software engineering is a discipline whose aim is the production of fault-free software, delivered on time and within budget, that satisfies the user's needs.”
- includes:
  - requirements analysis
  - human factors
  - functional specification
  - software architecture
  - design methods
  - programming for reliability
  - programming for maintainability
  - team programming methods
  - testing methods
  - configuration management

Why

- in school, you learn the mechanics of programming
- you are given the specifications
- you know that it is possible to write the specified program in the time allotted
- but not so in the real world...
  - what if the specifications are not possible?
  - what if the time frame is not realistic?
  - what if you had to write a program that would last for 10 years?
- in the real world:
  - software is usually late, over budget and broken
  - software usually lasts longer than employees or hardware
- the real world is cruel and software is fundamentally brittle

Who

- the average manager has no idea how software needs to be implemented
- the average customer says: “build me a system to do X”
- the average layperson thinks software can do anything (or nothing)
- most software ends up being used in very different ways than how it was designed to be used

Time

- you never have enough time
- software is often under budgeted
- the marketing department always wants it tomorrow
- even though they don’t know how long it will take to write it and test it
- “Why can’t you add feature X? It seems so simple...”
- “I thought it would take a week...”
- “We’ve got to get it out next week. Hire 5 more programmers...”

People

- you can’t do everything yourself
- e.g., your assignment: “write an operating system”
- where do you start?
- what do you need to write?
- do you know how to write a device driver?
- do you know what a device driver is?
- should you integrate a browser into your operating system?
- how do you know if it’s working?

Complexity

- software is complex!
- or it becomes that way
  - feature bloat
  - patching
- e.g., the evolution of Windows NT
  - NT 3.1 had 6,000,000 lines of code
  - NT 3.5 had 9,000,000
  - NT 4.0 had 16,000,000
  - Windows 2000 has 30-60 million
  - Windows XP has at least 45 million...

Necessity

- you will need these skills!
- risks of faulty software include
  - loss of money
  - loss of job
  - loss of equipment
  - loss of life

Therac-25

- http://sunnyday.mit.edu/papers/therac.pdf
- therac-25 was a linear accelerator released in 1982 for cancer treatment by releasing limited doses of radiation
- it was software-controlled as opposed to hardware-controlled (previous versions of the equipment were hardward-controlled)
- it was controlled by a PDP-11; software controlled safety
- in case of error, software was designed to prevent harmful effects

---

- BUT
- in case of software error, cryptic codes were displayed to the operator, such as:
- “MALFUNCTION xx”
- Where 1 < xx < 64
- operators became insensitive to these cryptic codes
- they thought it was impossible to overdose a patient
- however, from 1985-1987, six patients received massive overdoses of radiation and several died

---

- main cause:
- a race condition often happened when operators entered data quickly, then hit the up-arrow key to correct the data and the values were not reset properly
- the manufacturing company never tested quick data entry— their testers weren’t that fast since they didn’t do data entry on a daily basis
- apparently the problem had existed on earlier models, but a hardware interlock mechanism prevented the software race condition from occurring
- in this version, they took out the hardware interlock mechanism because they trusted the software

Example2: Ariane 501

- next-generation launch vehicle, after ariane 4
- presigious project for ESA
- maiden flight: june 4, 1996
- inertial reference system (IRS), written in ada
  - computed position, velocity, acceleration
  - dual redundancy
  - calibrated on launch pad
  - relibration routine runs after launch (active but not used)
- one step in recalibration converted floating point value of horizontal velocity to integer
- ada automatically throws out of bounds exception if data conversion is out of bounds
- if exception isn't handled... IRS returns diagnostic data instead of position, velocity, acceleration

---

- perfect launch
- ariane 501 flies much faster than ariane 4
- horizontal velocity component goes out of bounds
- IRS in both main and redundant systems go into diagnostic mode
- control system receives diagnotic data but interprets it as wierd position data
- attempts to correct it...
- ka-boom!
- failure at altitiude of 2.5 miles
- 25 tons of hydrogen, 130 tons of liquid oxygen, 500 tons of solid propellant

---

- expensive failure:
  - ten years
  - $7 billion
- horizontal velocity conversion was deliberately left unchecked
- who is to blame?
- "mistakes were made"
- software had never been tested with actual flight parameters
- problem was easily reproduced in simulation, after the fact

---

Mythical man-month

- Fred Brooks (1975)
- book written after his experiences in the OS/360 design
- major themes:
  - Brooks' Law: "Adding manpower to a late software project makes it later."
  - the "black hole" of large project design: getting stuck and getting out
  - organizing large team projects and communication
  - documentation!!!
  - when to keep code; when to throw code away
  - dealing with limited machine resources
- most are supplemented with practical experience

No silver bullet

- paper written in 1986 (Brooks)
- "There is no single development, in either technology or management technique, which by itself promises even one order-of magnitude improvement within a decade of productivity, in reliability, in simplicity."
- why? software is inherently complex
- lots of people disagreed, but there is no proof of a counter-argument
- Brooks' point: there is no revolution, but there is evolution when it comes to software development

SE Mechanics

- well-established techniques and methodologies:
  - team structures
  - software lifecycle / waterfall model
  - cost and complexity planning / estimation
  - reusability, portability, interoperability, scalability
  - UML, design patterns

Team Structures

- why Brooks' Law?
  - training time
  - increased communications: pairs grow by
- while people/work grows by
  - how to divide software? this is not task sharing
- types of teams
  - democratic
  - "chief programmer"
  - synchronize-and-stabilize teams
  - eXtreme Programming teams

Lifecycles

- software is not a build-one-and-throw-away process
- that's far too expensive
- so software has a lifecycle
- we need to implement a process so that software is maintained correctly
- examples:
  - build-and-fix
  - waterfall

Software lifestyle cycle

- 7 basic phases (Schach):
  - requirements (2%)
  - specification/analysis (5%)
  - design (6%)
  - implementation (module coding and testing) (12%)
  - integration (8%)
  - maintenance (67%)
  - retirement
- percentages in ()'s are average cost of each task during 1976-1981
- testing and documention should occur throughout each phase
- note which is the most expensive!

Requirements

- what are we doing, and why?
- need to determine what the client needs, not what the client wants or thinks they need
- worse— requirements are a moving target!
- common ways of building requirements include:
  - prototyping
  - natural-language requirements document
- use interviews to get information (not easy!)
- example: your online store

Specifications

- the "contract"— frequently a legal document
- what the product will do, not how to do it
- should NOT be:
  - ambiguous, e.g., "optimal"
  - incomplete, e.g., omitting modules
  - contradictory
- detailed, to allow cost and duration estimation
- classical vs object-oriented (OO) specification
  - classical: flow chart, data-flow diagram
  - object-oriented: UML
- example: your online store

Design Phase

- the "how" of the project
- fills in the underlying aspects of the specification
- design decisions last a long time!
- even after the finished product
  - maintenance documentation
  - try to leave it open-ended
- architectural design: decompose project into modules
- detailed design: each module (data structures, algorithms)
- UML can also be useful for design
- example: your online store

Implementation

- implement the design in programming language(s)
- observe standardized programming mechanisms
- testing: code review, unit testing
- documentation: commented code, test cases
- integration considerations
  - combine modules and check the whole product
  - top-down vs bottom-up ?
  - testing: product and acceptance testing; code review
  - documentation: commented code, test cases
  - done continually with implementation (can't wait until the last minute!)
- example: your online store

Maintenance Phase

- defined by Schach as any change
- by far the most expensive phase
- poor (or lost) documentation often makes the situation even worse
- programmers hate it
- several types:
  - corrective (bugs)
  - perfective (additions to improve)
  - adaptive (system or other underlying changes)
- testing maintenance: regression testing (will it still work now that I've fixed it?)
- documentation: record all the changes made and why, as well as new test cases
- example: your on-line store— how might the system change once it's been implemented?

Retirement phase

- the last phase, of course
- why retire?
  - changes too drastic (e.g., redesign)
  - too many dependencies ("house of cards")
  - no documentation
  - hardware obsolete
- true retirement rate: product no longer useful

Planning and Estimation

- we still need to deal with the bottom line
  - how much will it cost?
  - can you stick to your estimate?
  - how long will it take?
  - can you stick to your estimate?
- how do you measure the product (size, complexity)?

Reusability

- impediments:
  - lack of trust
  - logistics of reuse
  - loss of knowledge base
  - mismatch of features
- how to:
  - libraries
  - APIs
  - system calls
  - objects (OOP)
  - frameworks (a generic body into which you add your particular code)

Portability

- Java and C#
- Java: uses a JVM
  - write once, run anywhere (sorta, kinda)
- C#: also uses a JVM
  - emphasizes mobile data rather than code
- winner?
  - betting against Microsoft is historically a losing proposition...

interoperability

- e.g., CORBA
- define abstract services
- allow programs in any language to access services in any language in any location
- object-ish

Scalability

- something to keep in mind
- don’t worry about scaling beyond the abilities of the machine
- avoid unnecessary barriers
- from single connection to forking processes to threads...

PHP

History

- developed in the latter 1990's
- originally created as "Personal Home Page" tools, by Rasmus Lerdorf
- at first, was a quick tool for embedding sql queries in a web page (v1.0)
- then structured code was added (v2.0), but with a buggy language parser
- official release (v3.0) fixed parser bugs - June 1998
- by Jan 1999, 100,000 web pages were using php!!!
- php is better than cgi because:
  - it runs as part of the web server process and doesn't require forking (unlike cgi)
  - it runs faster than cgi
  - it's faster to write...
- php was designed to run with apache web server on unix
  - but also runs on windows and mac
- it's free!

- php is coded in C
  - has a well-defined API
  - extensible
- the way it runs:
  - a php engine is installed as part of a web server
  - the engine runs the php script and produces html, which gets passed back to the browser

---

- hello.php (plain php)
- hello2.php (php embedded in html)
- hello3.php (uses <?php start tag)

Hello.php

```
<?
print "hello world!";
?>
```

Hello2.php

```
<html>
<body bgcolor=#000000 text=#ffffff>
<?
print "hello world!";
?>
</body>
</html>
```

Hello3.php

```
<html>
<body bgcolor=#000000 text=#ffffff>
<?php
print "hello world!";
?>
</body>
</html>
```

basics

- php start and end tags: <? ... ?>
- also: <?php ... ?>
- semi-colon ends a statement (like C)
- string constants surrounded by quotes (") or (')
- you can embed multiple php blocks in a single html file
- variable names are preceded by dollar sign ($)
- user input is through html forms
- the language is case-sensitive, but calls to built-in functions are not (not sure if that's true for all built-in functions)
- identifiers are made of letters, numbers and underscore (_); and cannot begin with a number
- expressions are just like in C

Data types

- integers
- floating-point numbers
- strings
- loosely typed (you don't have to declare a variable before you use it)
- conversion functions: intval, doubleval, strval, settype
- settype(<value>, <newtype>) where newtype="integer", "double" or "string"
- typecasting: (integer), (string), (double), (array), (object)

operators

- mathematical: +, -, *, /, %, ++, --
- relational: <, >, <=, >=, ==, !=
- logical: AND, &&, OR, ||, XOR, !
- bitwise: &, |, ˆ (xor), ˜ (ones complement), >>, <<
- assignment: =, =, -=, *=, /=,
- other:
  - . concatenate
  - -> references a class method or property
  - => initialize array element index

Conditionals

- if/elseif/else:

```
if ( <expression1> ) {
<statement(s)>
}
elseif ( <expression2> ) {
<statement(s)>
}
else {
<statement(s)>
}
```

Conditional II

- tertiary operator:

```
<conditional-expression> ?
<true-expression> : <false-expression>;
```

- switch:

```
switch( <root-expression> ) {
case <case-expression>:
<statement(s)>;
break;
default:
<statement(s)>;
break;
}
```

loops

- while

```
while ( <expression> ) {
<statement(s)>;
}
```

- do-while

```
do {
<statement(s)>;
} while ( <expression> );
```

- for

```
for ( <initialize> ; <continue> ; <increment> ) {
<statement(s)>;
}
```

- break:
  - execution jumps outside innermost loop or switch

other

- exit() function
  - halts execution, meaning that no more code (php or html) is sent to the browser
- built-in constants
  - PHP_VERSION
  - __FILE__, __LINE__
  - TRUE = 1, FALSE = 0
  - M_PI = pi (3.1415927....)

Writing your own functions

- declared just like C:

```
function <name> ( args ) {
<body>
[return <value>]
}
```

- called just like C
- arguments (and local variables) are local, and don't exist when you exit the function; but you can use "static" to declare a variable so that when you call a function again, the value is retained
- use the "global" statement to declare global variables that you want to be able to access from within a function, or the GLOBALS array (which is like a perl hash)
  e.g., GLOBALS['username']
- recursion is okay, but be careful!

code

```
<?
$today = date("l F d, Y");
$yourname = $_POST['yourname'];
$cost     = doubleval( $_POST['cost'] );
$numdays  = intval( $_POST['numdays'] );
?>

<html>
<body>
today is:

<?
PRINT( "$today<br>" );
priNT( "$yourname, you will be out \$" );
print( doubleval( $cost * $numdays ));
print( " for buying lunch this week!" );
?>
</body>
</html>
```

arrays

- indexed using [...]
- indeces can be integers or strings (like a perl hash)
- when strings are indeces, it’s called an “associative array”
- array() function can be used to initialize an array
- e.g., $var = array(value0, value1, value2, ...);
- use the => operator to define the index:

```
$var = array( 1=>value1, value2, ... );
$var = array( "a"=>value1, "b"=>value2, ...
  );
```

- multidimensional arrays are okay (like C)

code

```
<html>
<body bgcolor=#ffffff>
<?
$states = array( "CA","NY" );
print "here are the states:<br>";
for ( $i=0; $i<count( $states ); $i++ ) {
  print "-- $states[$i]<br>";
}
print "<p>";
$cities = array( "CA"=>array( "san francisco","los angeles" ),
                 "NY"=>array( "new york","albany","buffalo" ));
print "here are the CA cities:<br>";
for ( $i=0; $i<count( $cities["CA"] ); $i++ ) {
  print( "-- ".$cities["CA"][$i]."<br>" );
}
print "here are the NY cities:<br>";
for ( $i=0; $i<count( $cities["NY"] ); $i++ ) {
  print( "-- ".$cities["NY"][$i]."<br>" );
}
```

Code II

```
print "<p>";
$states[] = "MA";
print "now here are the states:<br>";
for ( $i=0; $i<count( $states ); $i++ ) {
  print "-- $states[$i]<br>";
}
$cities[] = "MA";
$cities["MA"][] = "boston";
print "here are the MA cities:<br>";
for ( $i=0; $i<count( $cities["MA"] ); $i++ ) {
  print( "-- ".$cities["MA"][$i]."<br>" );
}

?>
</body>
</html>
```

classes

- defining a class:

```
class <class-name> {
// declare properties
// declare methods
}
```

- use just like java and c++
- example: myclass.php and userclass.php
- note use of include statement

myclass.php

```
<html>
<body>
<?

include "userclass.php";

$currentuser = new user;
$currentuser->init( "yaddi","cat" );

print( "name = ".$currentuser->name."<br>" );
print( "last login = ".$currentuser->getLastLogin() );

?>
</body>
</html>
```

userclass.php

```
<?
class user {

  // properties
  var $name;
  var $password;
  var $last_login;

  // methods
  function init( $inputname, $inputpassword ) {
    $this->name = $inputname;
    $this->password = $inputpassword;
    $this->last_login = time();
  }

  function getLastLogin() {
    return( date( "M d Y", $this->last_login ));
  }

}
```

I/O

- get input from html forms using

```
$_POST['<name>']
$_GET['<name>']
$_REQUEST['<name>']
```

- file I/O
  - basically just like C:

```
$fp = fopen( "filename","w" );
fwrite( $fp,"stuff" );
fclose( $fp );
```

  - note that fopen second argument mode is like C)

Closing Remarks

- Will still meet last lab this week
- Hope you enjoyed the whirlwind tour of different types of programming languages and projects
- Hope you had fun
- If you like this…..just the beginning
- If you didn't ….. You now know how complicated it is….never trust a program ☺

Next step

- More Computer science courses
  - theory and practice
- If anyone is interested in doing research over winter break, spring semester, over the summer, please contact me once you are done with finals.

- Thank You!