Software Engineering

Robert M. Dondero, Ph.D.
Princeton University
Objectives

- You will learn/review these software engineering topics:
  - Requirements analysis
  - Design
  - Implementation
  - Debugging
  - Testing
  - Evaluation
  - Maintenance
  - Methodologies

Stages of SW dev
How to sequence the stages
Objectives

- In other words...
- Describe some software engineering techniques and tools that:
  - You \textit{might have used} in your COS 333 project
  - You \textit{might still use} in your COS 333 project
  - You might use \textit{beyond} COS 333
Writing code is a small part of what a programmer does. The other parts are important, but sometimes overlooked in academia. Let's consider all of the parts...
Requirements Analysis

**Who** are the users?

**What** should the system do to fulfill the users' needs?
Kinds of Requirements

- **Functional requirements**
  - What should the system do?

- **Data requirements**
  - How much data must the system store?
  - How long must the data persist?
  - How accurate must the data be?
  - How volatile will the data be?
Kinds of Requirements

- **Environmental requirements**
  - What will be the system's:
    - Physical env? (lighting, noise, ...)
    - Social env (collaboration, coordination, ...)
    - Organizational env (support, ...)
    - Technical env (power, compatibilities, ...)

Kinds of Requirements

- **User** requirements
  - Will the system's users be:
    - Domain novices/experts?
    - Computer novices/experts?

- **Usability** requirements
  - How efficient/learnable/memorable must the system be?

Academic vs. Broad-World

- In the **academic** world:
  - (Student) programmers often are **given** requirements

- In the **broader** world:
  - (Senior) programmers often must know how to **gather** and **structure** requirements
Q: How to gather requirements?
A: Some techniques...
Requirements Gathering

- Questionnaires
- Interviews
- Focus groups
- Naturalistic observation
- Studying documentation

Users visit the programmers

Pgmmers visit the users

Q: How to **structure** requirements?
A: Some techniques...
Requirements Structuring

- Create **models** of the user's domain
  - A popular set of modeling notations...
  - **Unified Modeling Language (UML)**
Unified Modeling Lang (UML)

- **Who**: Grady Booch, James Rumbaugh, Ivar Jacobson
- **When**: 1980s
- **What**: A set of notations
- **Why**: To model user domains and systems

"The three amigos"
Create **Class Model(s)**

- A UML notation
- Describes classes of objects in the user's domain
- Does not (necessarily) refer to the system to be built
Class model example:

- **Vehicle**
  - **Plane**
  - **Ship**
- **Camera**
- **ReconPlane**
- **Carrier**
Create **Scenario(s)**

- A story describing a user interaction with the system to achieve some goal
  - In one scenario/path all goes well
  - In another scenario/path an error occurs
  - ...
Requirements Structuring

- Create **Prototype(s)**
  - Bridge from requirements analysis to design
  - Low-fidelity
    - Storyboards, index cards, ...
  - High-fidelity
    - Shallow HTML docs, "Wizard of Oz", ...
    - System with only the most common logical paths implemented
Design

How should the system work?
Design Models

- Create *specification* **Class Model(s)**
  - An elaboration of a *conceptual* class model
  - Models concepts in the user's domain
    - Vehicles, Ships, ...
  - ... and in the programmer's domain
    - Fields, methods, ...
    - ArrayList objects, Thread objects, Database objects, ...
Design Heuristic 1

- Level resource management
  - A module should free a resource iff it has allocated that resource
  - Resources: memory, file handles, ...
  - Document all violations clearly

Design Heuristic 2

- **Detect errors low; handle errors high**
- A module should:
  - Detect errors
  - Handle errors if it can; otherwise…
  - Report errors to its clients
- A module often cannot assume what error-handling action its clients prefer

Design Heuristic 3

- Seek **strong coherence** within modules
  - The components (fields, methods) of a module should be related to each other

- Empirically: not significant
Design Heuristic 4

- Seek **weak coupling** among modules
  - Minimize interfaces
  - Encapsulate data

- Empirically: significant
Design Heuristic 4a

- Seek weak design-time coupling

**Strong** design-time coupling (bad!)

Simulator calls many methods in Plane
Design Heuristic 4a

Weak design-time coupling (good!)

Simulator calls few methods in Plane

Recall COS 217 shell program
Design Heuristic 4b

- Seek weak *run-time coupling*

**Strong run-time coupling (bad!)**

Client

- run()
- sort()

Collection

- getItem()
- setItem()

Client makes **many** calls to Collection methods
Weak run-time coupling (good!)

Client makes few calls to Collection methods
Design Heuristic 4b

- Seek weak **maintenance-time coupling**

**Strong** maintenance-time coupling (**bad**!)

Maintenance programmer changes Client and MyModule together **frequently**
Design Heuristic 4b

**Weak** maintenance-time coupling (**good**!)

Maintenance programmer changes Client and MyModule together **infrequently**
Use composition to model "has a"; use inheritance to model "is a"

- Example: Plane and Camera
- Example: Plane and Vehicle
- Example: Stack and Vector
Use the **Liskov substitution principle**

"Let q(x) be a property provable about objects x of type T. Then q(y) should be true for objects y of type S where S is a subtype of T"

Barbara Liskov and Jeannette Wing.  
"A behavioral notion of subtyping,"  
*ACM Transactions on Programming Languages and Systems*, volume 16, issue 6 (November 1994), pp. 1811 - 1841.
Liskov substitution principle (cont)

- Informally: If S inherits from T, then s should be usable in place of t
- Informally: A subclass should be a subtype
- Example: Plane and Vehicle
- Counter example: Stack and Vector

- Favor composition over inheritance

Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley. Reading, MA. 1995.
Inheritance:

- Relationship among classes
- Determined at compile-time
- White box reuse
  - Subclass sees internals of superclass
  - Change superclass => (often) change subclass
  - "Inheritance breaks encapsulation"

Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley. Reading, MA. 1995.
Composition:

- Relationship among **objects**
- Determined at **run-time**
  - More flexible
- **Black box reuse**
  - Neither containing nor contained object sees internals of the other

Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley. Reading, MA. 1995.
Design Patterns

- **Who:** Christopher Alexander
  - Architect, not computer scientist
- **When:** 1970s
- **Why:** Help people create rooms, buildings, towns, ...

![Christopher Alexander](image.png)
Patterns in Architecture

- Example Pattern: Entrance Room

"Arriving in a building, or leaving it, you need a room to pass through, both inside the building and outside it. This is the entrance room."

"At the main entrance to a building, make a light-filled room which marks the entrance and straddles the boundary between indoors and outdoors, covering some space outdoors and some space indoors. The outside part may be like an old-fashioned porch; the inside like a hall or sitting room."

Christopher Alexander et al.  
A Pattern Language.  
Example Pattern: **Private Terrace on the Street**

"The relationship of a house to a street is often confused: either the house opens entirely to the street and there is no privacy; or the house turns its back on the street, and communion with street life is lost."

"Let the common rooms open onto a wide terrace or a porch which looks into the street. Raise the terrace slightly above street level and protect it with a low wall, which you can see over if you sit near it, but which prevents people on the street from looking into the common rooms."

---

Christopher Alexander et al.  
*A Pattern Language.*  
OO Design Patterns

The Gang of Four

Ralph Johnson
Richard Helm
Erich Gamma
John Vlissides
<table>
<thead>
<tr>
<th>Creational Patterns</th>
<th>Behavioral Patterns</th>
</tr>
</thead>
<tbody>
<tr>
<td>Abstract factory</td>
<td>Chain of responsibility</td>
</tr>
<tr>
<td>Builder</td>
<td>Command</td>
</tr>
<tr>
<td>Factory method</td>
<td>Interpreter</td>
</tr>
<tr>
<td>Prototype</td>
<td>Iterator</td>
</tr>
<tr>
<td>Singleton</td>
<td>Mediator</td>
</tr>
<tr>
<td></td>
<td>Memento</td>
</tr>
<tr>
<td>Structural Patterns</td>
<td>Observer</td>
</tr>
<tr>
<td>Adapter</td>
<td>State</td>
</tr>
<tr>
<td>Bridge</td>
<td>Strategy</td>
</tr>
<tr>
<td>Composite</td>
<td>Template method</td>
</tr>
<tr>
<td>Decorator</td>
<td>Visitor</td>
</tr>
<tr>
<td>Facade</td>
<td></td>
</tr>
<tr>
<td>Flyweight</td>
<td></td>
</tr>
<tr>
<td>Proxy</td>
<td></td>
</tr>
</tbody>
</table>

Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley. Reading, MA. 1995.
Example: **Decorator**

"Sometimes we want to add responsibilities to individual objects, not to an entire class. A GUI toolkit, for example, should let you add properties like borders or behaviors like scrolling to any user interface component."

"One way to add responsibilities is with inheritance. Inheriting a border from another class puts a border around every subclass instance. This is inflexible, however, because the choice of border is made statically. A client can't control how and when to decorate the component with a border."

Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley. Reading, MA. 1995.
Example: **Decorator** (cont.)

"A more flexible approach is to enclose the component in another object that adds the border. The enclosing object is called a **decorator**. The decorator conforms to the interface of the component it decorates so that its presence is transparent to the component's clients. The decorator forwards requests to the component and may perform additional actions (such as drawing the border) before or after forwarding. Transparency lets you nest decorators recursively, thereby allowing an unlimited number of additional responsibilities."

Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley. Reading, MA. 1995.
Decorator

Component
operation()

ConcreteComponent
operation()

Decorator
operation()

ConcreteDecoratorA
operation() addedState

ConcreteDecoratorB
operation() addedBehavior()

Decorator::operation()

Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley. Reading, MA. 1995.
Where have we seen Decorator?
Example: **Observer**

"A common side-effect of partitioning a system into a collection of cooperating classes is the need to maintain consistency between related objects. You don't want to achieve consistency by making the classes tightly coupled, because that reduces their reusability."

Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley. Reading, MA. 1995.
"The Observer pattern describes how to establish these relationships. The key objects in this pattern are subject and observer. A subject may have any number of dependent observers. All observers are notified whenever the subject undergoes a change in state. In response, each observer will query the subject to synchronize its state with the subject's state."

Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley. Reading, MA. 1995.
**Subject**
- `attach(observer)`
- `detach(observer)`
- `notify()`

**Observer**
- `update()`

**ConcreteSubject**
- `getState()`
- `subjectState`

**ConcreteObserver**
- `update()`
- `observerState`

_for all o in observers o->update()_ 

`observerState = subject->getState()` 

Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley. Reading, MA. 1995.
Where have we seen Observer?
Observer pattern relates to:

- MVC architecture
  - Model = Subject; View = Observer
- XML

Common idea: separate data (model, subject) from presentation (view, observer)
- Allow same data to be presented in multiple ways
Patterns in Other Fields

- **User Interface Patterns**

- **Pedagogical Patterns!!!**
Implementation

How should I code the system?
Should I code the system, or not?
Reuse

- How should I code the system?
  - What we've been discussing all semester
- But first...
- Should I code the system, or not?!!!
  - Build or buy (or use open-source)?
  - Create new code or reuse exiting code?
Kinds of Reuse

- Copy and paste
  - Beware of code cloning
- Call a library function
- Use a library class
  - Call a static method
  - Instantiate an object of a library class
  - Define a class that inherits from a library class
Reuse Challenges

- Finding the code to be reused!
  - How to **categorize** reusable code?
  - How to **search for** reusable code?

- See **Google Code**
Reuse Challenges

- The Reusability Paradox
  - From the field of "learning objects"
  - **Large** modules are **more** useful, but can be used in **fewer** programs
  - **Small** modules are **less** useful, but can be used in **more** programs
- Designing for reuse inherently involves compromise

David Wiley. "The Reusability Paradox." http://cnx.org/content/m11898/latest/
Debugging

How can I fix the system?
Debugging Techniques

- Recall from COS 217...
- Debugging techniques
  - Divide and conquer
  - Validate function parameters
  - Check invariants
    - At beginning and end of each public method
  - Display output
    - Remember to write to stderr, not stdout
Debugging Techniques

- Debugging techniques (cont.)
  - Use a debugger
    - Recall gdb, jdb, pdb from Appendix of "Languages" lecture
  - Focus on recent changes
    - Use a version control system
Testing

How can I break the system?
Debugging vs. Testing

- **Debugging**: How can I **fix** the system?
- **Testing**: How can I **break** the system?
  - A programmer should write at least as much test code as production code
External Testing Techniques

- Recall from COS 217...

- **External testing**
  - Boundary condition testing
  - Stress testing
  - Path testing
  - Statement testing
Statement Testing Tools

- For C: gcc with "coverage" option
  - See fraction.h, fraction.c from previous lectures, plus Fraction_equals()

```
$ gcc -D TEST_FRACTION --coverage
    intmath.c fraction.c -o fraction
$ fraction
$ gcov fraction.c
$ emacs fraction.c.gcov
```

Data accumulates in fraction.c.gcov file
Statement Testing Tools

- For Java: **Emma**
  - See `Fraction.java` from previous lectures

```
$ javac Fraction.java
$ java emmarun -r html -cp . -sp . Fraction
$ cd coverage
$ firefox index.html
```
For Python: **python-coverage**

- See *frac.py* from previous lectures

```
$ python-coverage -x frac.py
$ python-coverage -rma
$ emacs frac.py,cover
```

- Very important for Python
  - Could detect syntax errors!!!
Recall from COS 217...

**Internal testing**

- Checking invariants
- Checking function return values
- Changing code temporarily
- Leaving testing code intact
General Testing Strategies

- Recall from COS 217...

- **General testing strategies**
  - Testing incrementally
  - Comparing implementations
  - Bug-driven testing
    - Use a bug tracking system (Bugzilla, flat file)
  - Fault injection
  - Automation
For C: **CUnit**

- Recall fraction.h, fraction.c
- See testfraction.c

```
$ gcc testfraction.c fraction.c intmath.c -lcunit -o testfraction
$ testfraction
```
Tools for Test Automation

- For Java: **JUnit**
  - Recall Fraction.java
  - See TestFraction.java

```
$ javac TestFraction.java
$ java org.junit.runner.JUnitCore TestFraction
```

- Note Java annotations
  - New with JDK 1.5
  - See Horstmann & Cornell Chapter 11
Tools for Test Automation

- For Python: **PyUnit** (the `unittest` module)
  - Recall `frac.py`
  - See `testfraction.py`

```
$ testfraction.py
```
Tools for Test Automation

- For GUls:
  - Many logical paths
    - "Microsoft WordPad has 325 possible GUI operations" – Wikipedia
  - Test automation is difficult
  - See http://en.wikipedia.org/wiki/List_of_GUI_testing_tools
Tools for Test Automation

- GUI testing approaches:
  - User action capture & playback
    - Using version X: Capture user actions & resulting screens (bitmaps)
    - Using version X+1: Playback user actions; compare resulting screens to previous
  - User action generate & play
    - Compose scripts to generate user actions
    - Play user actions; analyze screens
  - But same data ("model") can have multiple presentations ("views")
GUI testing approaches:

- **Event capture & playback**
  - Using version X: Capture significant events & resulting model states
  - Using version X+1: Playback events, compare resulting model states to previous

- **Event generation & play**
  - Compose scripts to generate events
  - Play events; analyze model states
Evaluation

How can I tell if the system fulfills the users' needs?
Testing vs. Evaluation

- **Testing**
  - Checks the **implementation**
  - Does the implementation work?

- **Evaluation**
  - Checks the **system**
  - Does the system fulfill the user's needs?
Some research communities:
  - "Three of my colleagues liked it"
Other research communities:
  - Does the system consistently and measurably help its users?
    - Often involves experiments with control group and experimental group
    - Often involves statistical analysis
Real-World Eval by Users

- Questionnaires
- Interviews
- Focus groups
- Naturalistic observation

Recall requirements gathering techniques
Heuristic Evaluation

For evaluating the system as a whole

Visibility of system status
Match between system and the real world
User control and freedom
Consistency and standards
Help users recognize, diagnose, and recover from errors
Error prevention
Recognition rather than recall
Flexibility and efficiency of use
Aesthetic and minimalist design
Help and documentation

Cognitive Walkthrough

For evaluating part of a system in detail

Repeatedly:

1. Will the correct action be sufficiently evident to the user?
2. Will the user notice that the correct action is available?
3. Will the user associate and interpret the response from the action correctly?

Maintenance

How can I ensure that the system continues to fulfill the users' needs through time?

"Continuance" might be a better term
Continuing Tasks

- Fix bugs
- Add new functionality
- Improve performance
  - Using an execution profiling tool
- Refactor

- First **profiling**, then **refactoring**...
For C: **gprof** (as in COS 217)
- Recall `concord.c`

```bash
$ gcc -pg concord.c symtablehash.c -o concord
$ concord Bible.txt
$ gprof concord > reportc.txt
$ emacs reportc.txt
```
Execution Profiling Tools

- For Java: `-Xrunhprof` option
  - Recall `Concord2.java`

```
$ javac Concord2.java
$ java -Xrunhprof:cpu=times Concord2 Bible.txt
$ cp java.hprof.txt reportjava.txt
$ emacs reportjava.txt
```

- Very slow!!!
Execution Profiling Tools

- For Python: **profile** module
- Recall *concord.py*

```
$ python
>>> import profile
>>> from concord import main
>>> argv = ['concord', 'Bible.txt']
>>> profile.run('main(argv)')
>>> quit()
$ # screen scrape to reportpython.txt
$ emacs reportpython.txt
```
### Bad smells in code

- Duplicated code
- Long method
- Long parameter list
- Divergent change
- **Shotgun surgery**
- Feature envy
- Data clumps
- Primitive obsession
- Switch statements
- Parallel inheritance hierarchies
- Lazy class

<table>
<thead>
<tr>
<th>Speculative generality</th>
<th>Temporary field</th>
</tr>
</thead>
<tbody>
<tr>
<td>Message chains</td>
<td>Middle man</td>
</tr>
<tr>
<td><strong>Inappropriate intimacy</strong></td>
<td></td>
</tr>
<tr>
<td>Alternative classes with diff interfaces</td>
<td></td>
</tr>
<tr>
<td>Incomplete library class</td>
<td></td>
</tr>
<tr>
<td><strong>Data class</strong></td>
<td></td>
</tr>
<tr>
<td><strong>Refused bequest</strong></td>
<td></td>
</tr>
<tr>
<td>Comments</td>
<td></td>
</tr>
</tbody>
</table>
Refactoring categories:

- Composing methods (9)
- Moving features between methods (8)
- Organizing data (16)
- Simplifying conditional expressions (8)
- Making method calls simple (15)
- Dealing with generalization (12)
- Big refactorings (4)
- TOTAL (72)
Extract Method

```java
void printOwing(double amount)
{
    printBanner();
    // print details
    System.out.println("name: " + name);
    System.out.println("amount: " + amount);
}

void printOwing(double amount)
{
    printBanner();
    printDetails(amount);
}

void printDetails(double amount)
{
    System.out.println("name: " + name);
    System.out.println("amount: " + amount);
}
```

Martin Fowler.  
Refactoring Example

- Inline Temp

```java
double basePrice = anOrder.basePrice();
return (basePrice > 1000);  
```

```java
return (anOrder.basePrice() > 1000);  
```

Martin Fowler.  
*Refactoring: Improving the Design of Existing Code.*  
Refactoring Example

- Replace Type Code with Subclass

```java
public class Employee {
    public static int ENGINEER = 0;
    public static int SALESPERSON = 1;
    private int type;
    ...
}
```

```java
public class Employee {
    ...
}
public class Engineer extends Employee {
    ...
}
public class SalesPerson extends Employee {
    ...
}
```

Martin Fowler.
*Refactoring: Improving the Design of Existing Code.*
Methodologies

How should I sequence those stages?
(And much more)
Methodologies Spectrum

- Predictive: Waterfall Model
- Rational Unified Process
- Extreme Programming

Example SW dev methodologies
The Waterfall Model

- Requirements
  - Requirements analysis
- Design
  - Design
- Implementation
  - Implementation, Debugging
- Verification
  - Testing, Evaluation
- Maintenance
  - Maintenance
The Waterfall Model

- Completely predictive (non-adaptive)
  - From manufacturing industry
- Used by many early software dev projects
  - No other choices!
- Required by many funding agencies
  - Agency defines requirements
  - SW company does the rest, while agency monitors progress
Waterfall Model Editorial

- Not realistic
  - Programming involves discovery
  - Any project must adapt to unplanned events
Extreme Programming


Planning/Feedback Loops

- Release Plan
- Iteration Plan
- Acceptance Test
- Stand Up Meeting
- Pair Negotiation
- Unit Test
- Pair Programming
- Code

Requirements analysis
Evaluation
Design
Testing
Impl., Debugging

Maintenance
Extreme Programming

- As adaptive (non predictive) as possible
  - "Extremely" adaptive
  - "Embrace change"
- Essentially, code is the only artifact produced
Extreme Programming Practices

- The planning game
- Small releases
- Metaphor
- Simple design
- Testing
- Refactoring
- Pair programming

- Collective ownership
- Continuous integration
- 40-hour work week
- On-site customer
- Coding standards

Appealing!

Too extreme?

- An excuse for programmers not to do some tasks that they find less fun?
Rational Unified Process

Rational Unified Editorial

- Balances prediction and adaptation
- More reasonable?
Methodologies Editorial

Frederick Brooks
"All software involves essential tasks, the fashioning of the complex conceptual structures that compose the abstract software entity, and accidental tasks, the representation of those abstract entities in programming languages and the mapping of these onto machine languages within space and speed constraints. Most of the big gains in software productivity have come from removing artificial barriers that have made the accidental tasks inordinately hard."

Frederick Brooks.  
_The Mythical Man Month: Essays on Software Engineering_  
"How much of what software engineers now do is still devoted to the accidental, as opposed to the essential? Unless it is more than 9/10 of all effort, shrinking all the accidental activities to zero time will not give an order of magnitude improvement."

"There is no single development, in either technology or management technique, which by itself promises even one order of magnitude improvement in productivity, in reliability, in simplicity."

Software Methodology and Snake Oil
– Each methodology has the germ of a useful idea
– Each claims to solve major programming problems
– Some are promoted with religious fervor
– In fact most don't seem to work well
– Or don't seem to apply to all programs
– Or can't be taught to others
– A few are genuinely useful and should be part of everyone's repertoire
In summary...

(Kernighan) Some methodologies offer good ideas, but...

(Brooks) Software development is inherently hard, and...

(Kernighan) Many methodologies are over-hyped, so...

(Kernighan) View methodologies with healthy skepticism
A programmer bridges the gap between computers and people

A good programmer has:

- (1) Knowledge of computers
- (2) Knowledge of people (empathy)

If you have/develop (1), you'll make a living

If you also have/develop (2), you'll make a difference
Summary

- We have covered these software engineering topics:
  - (1) Requirements analysis
  - (2) Design
  - (3) Implementation
  - (4) Debugging
  - (5) Testing
  - (6) Evaluation
  - (7) Maintenance
  - (8) Methodologies
We have covered:

- Programming languages (C, Java, Python)
- Database programming (SQL)
- GUI programming (Swing, Tkinter)
- Network programming

(Continued on next slide)
We have covered (cont.):

- Server-side web programming
  - CGI
  - Programming the web server (PHP, JSP)
  - Web app frameworks (Django)
  - Security issues
- Client-side web programming (Applets, JavaScript)
- (Continued on next slide)
Course Summary

- We have covered (cont.):
  - XML programming (DOM and SAX)
  - Concurrent programming (processes and threads)
  - Programming paradigms (C++)
  - Software engineering
The Rest of the Course

- May 5 or May 6
  - Project presentation and final demo
- May 10 (Dean's Date) at 5PM
  - Final website
  - User's guide document
  - Design document
  - Compressed source code file
  - Final report document
  - Presentation/demo slides
Course Conclusion

- Thank you
  - I learned; I had fun
- I hope you also
  - Learned; had fun
- I hope your project concludes well
  - I hope it transcends COS 333!
- I hope you enjoy the summer
- I wish you all the best