Lecture 1 - Introduction and Overview

CSCI4448 - Object Oriented Analysis and Design

Dana Hughes

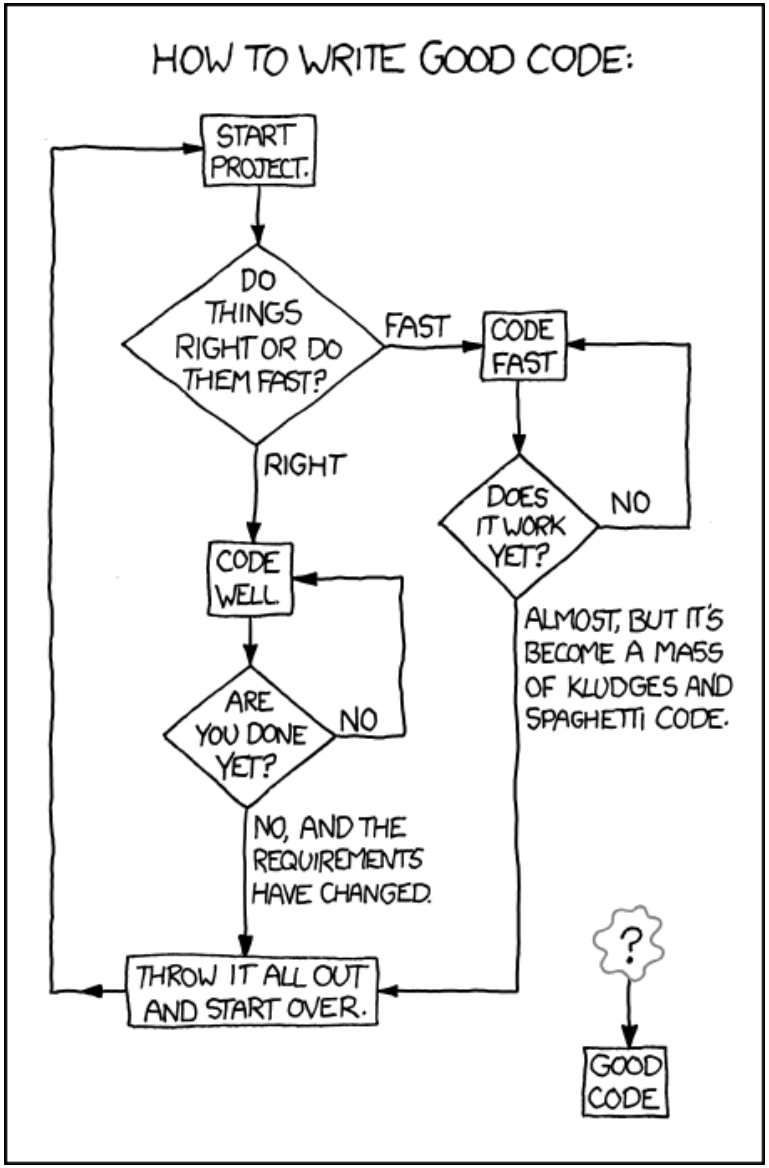
University of Colorado - Department of Computer Science

January 14, 2014

Announcements

Course Overview

- This course is designed to help reach the "Good Code" box.
- Software design isn't a black art.
- There exists design techniques which lead to successful creation of large scale software systems.



About Me

- 2nd Year PhD Student
  - Correll Lab (Robotics)
  - Amorphous Materials / Sensor Networks
  - Swarm Intelligence
- MS in Computer Science College of Charleston
  - Artificial Intelligence
  - Computer Music Lab
  - Music Information Retrieval
  - Computer Assisted Composition and Performance
- MS in Electrical Engineering University of Missouri Rolla
  - RF and Microwave Engineering
  - Embedded Sensors for Structural Health Monitoring
- BS in Electrical Engineering Colorado State University
  - Digital Systems
  - Microprocessor / Microcontroller Design

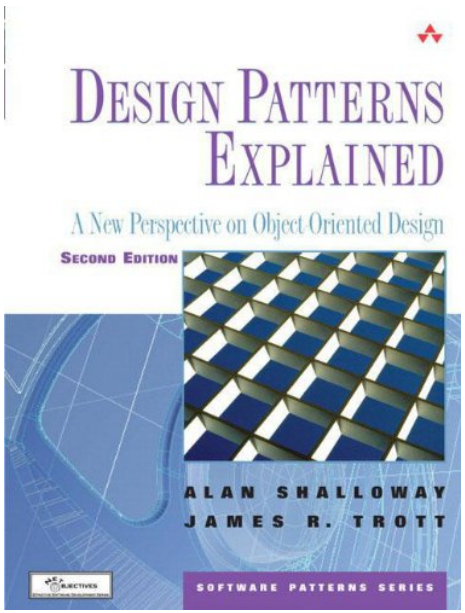
Office Hours and Contact

- Office: ECCS 102 (during official office hours only)
  The robotics lab is **not** my office, please don't show up there for office hours
- Office Hours: Monday 1:30 - 2:45pm, Thursday 2:00 - 3:15pm
  If needed, I am more than willing to make appointments as well. Please send me an email.
- Email: dana.hughes@colorado.edu
  For my convenience, please indicate this is course related in the subject line (e.g., "CSCI4448:" or "OOAD:")

Webpages

- `http://csel.cs.colorado.edu/~dahu6681/courses/csci4448/`
  - Contains syllabus, schedule, presentations, code, etc.
- `http://piazza.com/colorado/spring2014/csci4448/`
  - Mirrors website information
  - Primary means of out-of-class discussion
  - An email to sign up should have been sent to your CU email address, email me if you want to use a different address
- Github (or equivalent)
  - Definitely for class project, possibly for homeworks as well
  - I will either make notes on using git, or dedicate some lecture time

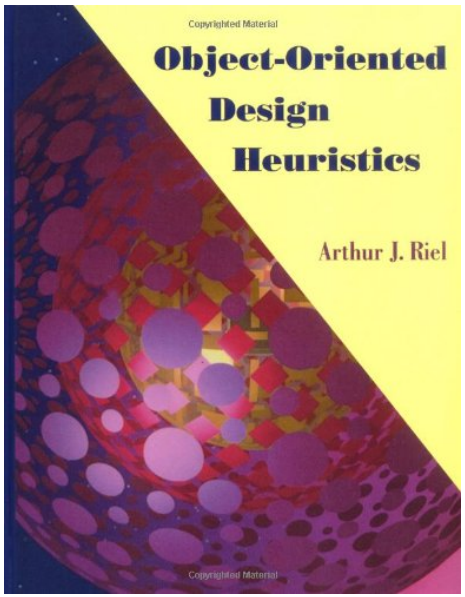
Textbook Resources



Design Patterns Explained
A New Perspective on Object Oriented Design

- Alan Shalloway and James R. Trott (2nd Edition)
- ISBN-10: 0321247140
  ISBN-13: 978-0321247148
  - OO Design
  - Design Patterns
  - UML

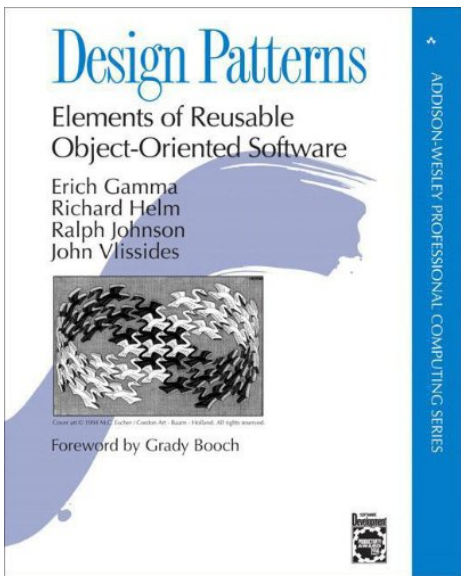
Textbook Resources



Object-Oriented Design Heuristics

- Arthur J. Riel
- ISBN-10: 020163385X
  ISBN-13: 978-0201633856
  - Discusses proper object oriented design through use of heuristics
  - Somewhat older (First printing - 1996)
  - Doesn't cover patterns, UML

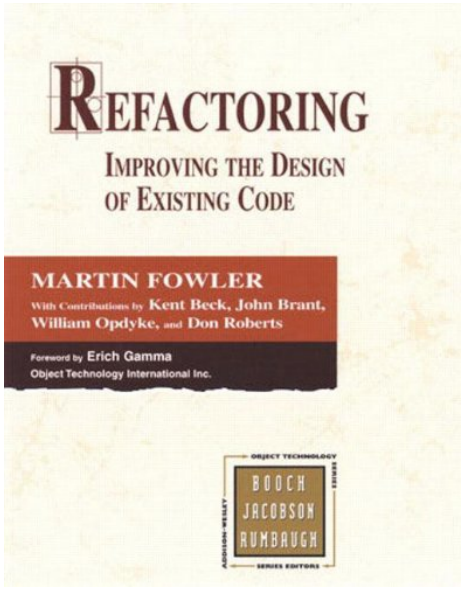
Textbook Resources



Design Patterns
Elements of Reusable Object-Oriented Software

- Erich Gamma, Richard Helm, Ralph Johnson and John Vissides
  (The "Gang of Four" book)
- ISBN-10: 0201633612
  ISBN-13: 978-0201633610
  - Seminal book on design patterns
  - Not the textbook for the course, but still good
  - Lacks OO design and UML
  - Examples in C++ or Smalltalk

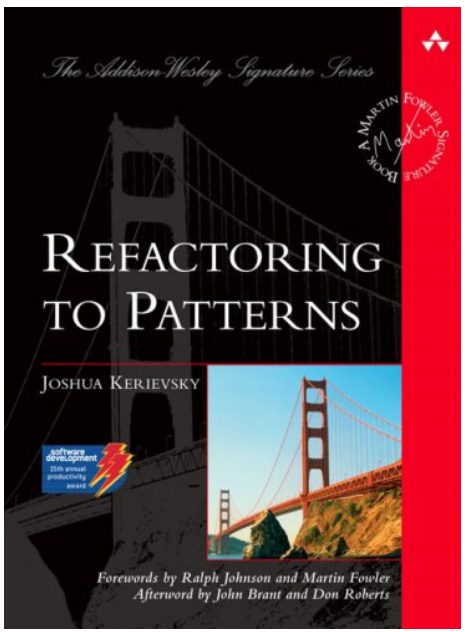
Textbook Resources



Refactoring
Improving the Design of Existing Code

- Martin Fowler, Kent Beck, John Brant, William Opdyke and Don Roberts
- ISBN-10: 0201485672
  ISBN-13: 978-0201485677
  - Canonical book on code refactorization
  - Useful for improving design of existing code
  - Integral part of software development in eXtreme Programming

Textbook Resources



Refactoring to Patterns

- Joshua Kerievsky
- ISBN-10: 0321213351
  ISBN-13: 978-0321213358
  - Refactoring for object-oriented programming
  - Focus is to refactor to established design patterns (see the GoF book)

Prior Courses

- Prior Course Requirements
  I'm assuming the majority of individuals have taken the following
  - Principles of Programming Languages (CSCI3155)
  - Computer Structure (CSCI2400)
  - Data Structures (CSCI2270)
  - Introduction to Programming (CSCI1300)
- Object-Oriented Language
  You should be comfortable with some OO language
  - **Java**, **C++**, C#, Objective-C, **Python**, Ruby
  - Other OO languages okay, but check with me first
  - Examples will typically be in Java (for OO examples), Python (for procedural examples), Scala (for functional examples)

Goals

This course should provide knowledge and skill in

- Object-oriented concepts
- OO analysis, design and implementation techniques
- Design patterns
- Test driven development and refactoring
- Concurrency and distributed issues

Object oriented programming is not simply a means of implementation, but rather a software engineering process with specific tools and techniques.

Tentative Schedule

- **Weeks 1 & 2** - Object oriented fundamentals
- **Week 3** - Unified Modeling Language
- **Week 4** - Object-based analysis and design from use cases
- **Weeks 5 - 9** - Design Patterns, First Midterm Exam
- **Week 10** - Second Midterm Exam
- **Week 11** - Test Driven Development
- **Week 12** - Refactoring
- **Week 13** - Concurrency
- **Week 14** - Object Relational Mapping and Dependency Injection
- **Week 15** - TBD

This schedule is subject to change, but provides an overview of the major topics.

Evaluation

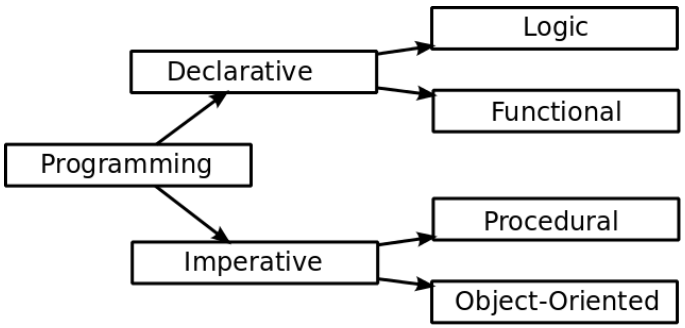
- **Class Participation** - 5%
  Preparation / participation in class, posting on Piazza, surveys
- **Homework** - 25%
  Assigned every 2 - 3 weeks
- **Quizzes** - 10%
  In class, usually after homeworks are due
- **Midterm Exams** - 30%
- **Group Project** - 30%
  Second half of semester, replace final exam

Policies

- Late Policy
  - Maximum score deducted 15% per day after due date, up to 3 days.
  - Homeworks can be resubmitted once for 50% credit on improvement.
- Syllabus
  Various university-level policies are in place to address specific situations, including
  - Disability accomidations
  - Religious Observances
  - Classroom Behavior
  - Discrimination and Harassment
  - Honor Code

  These policies are detailed in the syllabus, please email or see me if these apply to you.

Programming Paradigms



- Declarative Programming
  - Functional - Haskell, Lisp, Erlang
  - Logical - Prolog
- Imperative Programming
  - Procedural - C, Fortran, Pascal
  - Object-oriented - Smalltalk, Objective-C, C++, Java, C#

Programming Paradigms

- Software development is an inherently complex task
  - Some complexity is *essential*, some is *accidental*
- Various paradigms attempt to manage the essential complexity associated with software, for example
  - Imperative languages subdivide a problem into various subproblems
  - Declarative languages ignores implementation details
  - Functional langauges try to minimize or eliminate side effects
  - Object oriented languages try to distribute responsibility among several objects

Motivation

- Software is developed in several phases
  - **Analysis** - Analyze use cases, determine the requirements of the software
  - **Design** - Determine the architecture and design of the software system and subsystems
  - **Implementation** - Build / code the system
  - **Test** - Ensure and demonstrate that the software fulfills the requirements
  - **Maintainence** - Update the software to match changes in requirements
- Different SE models treat these phases differently
- This course focuses primarily on Analysis and Design, Testing and Maintainence will be discussed
- Implementation is *not* the focus

Motivation

- Software development involves fulfilling the requirements of a client
- One constant is that software requirements always **change**
  - Mismatch between client and developer understanding
  - Unforseen use scenarios
  - New features
  - etc.
- Software should be designed for reuse in future projects

Procedural Approach

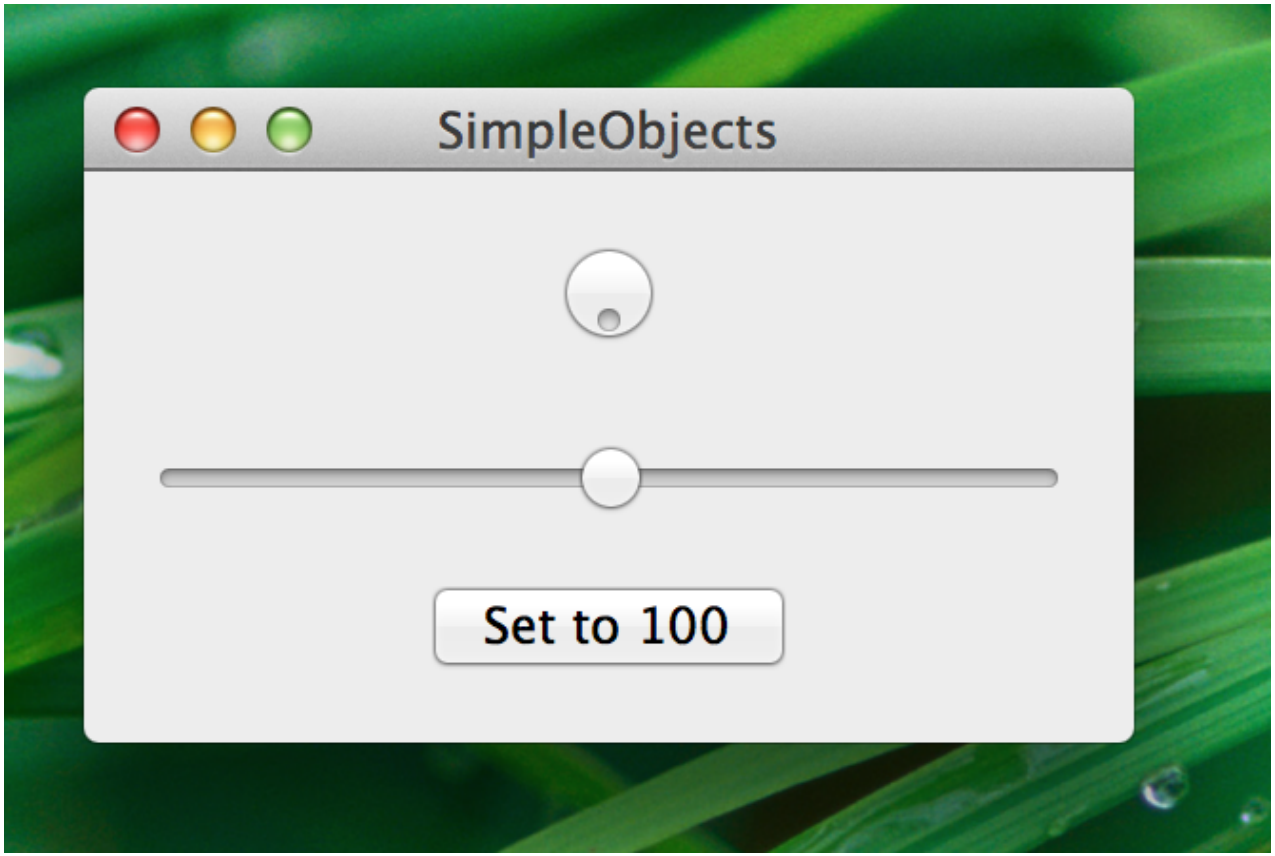
- Procedural languages decompose a problem into
  - Structured data
  - Subprocesses which operate on some part of the structured data
  - Data + Algorithm = Program
- Subprocesses can be reused later in other programs, assuming the same data structure
- How robust is this approach to change?
  - Different algorithm for some subprocess may require change in the data structure
  - Change in the data structure may require altering other subprocesses

Object Oriented Approach

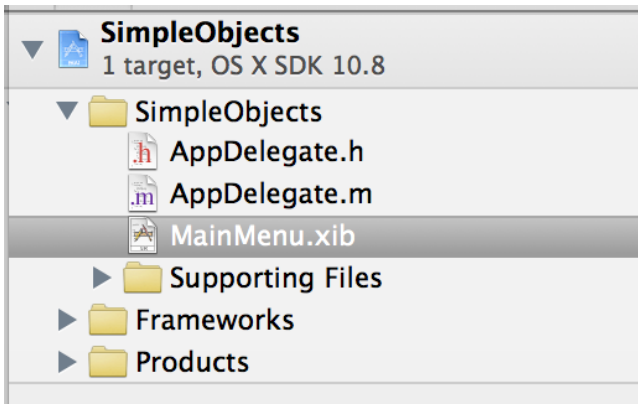
- Object oriented languages decompose a problem into
  - A collection of objects, each with well defined responsibilities
  - Data and behavior associated with individual objects
  - A method of interaction among objects
- Individual objects can be reused later in other programs, without regard to data structure
- Objects can later be specialized for other tasks
- Objects can be composed of other objects dynamically
- Changes in the data structure (should) only require adjusting the associated object

Object Oriented Example - GUI

Consider this small GUI application. How many objects are needed to make this work?

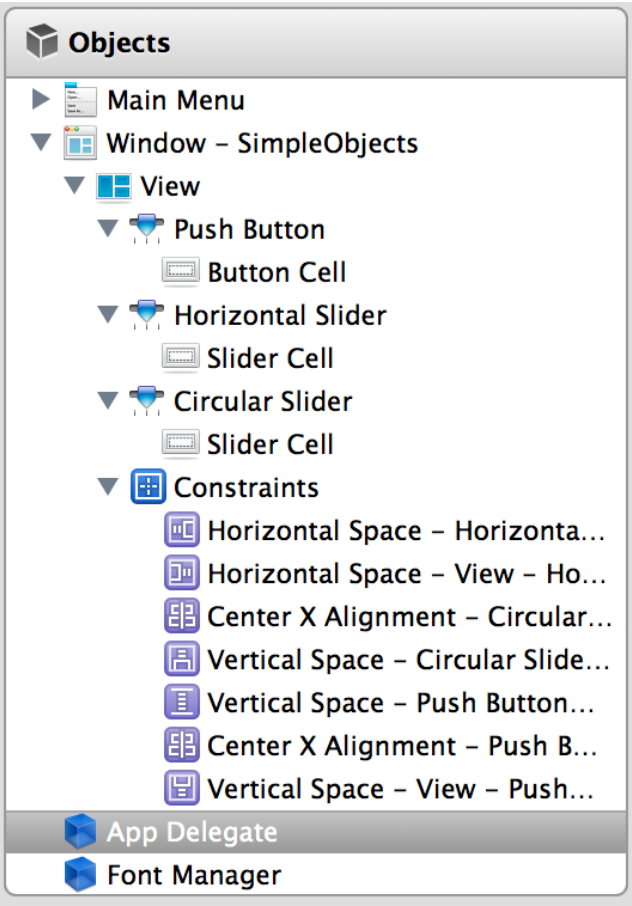


Object Oriented Example - GUI



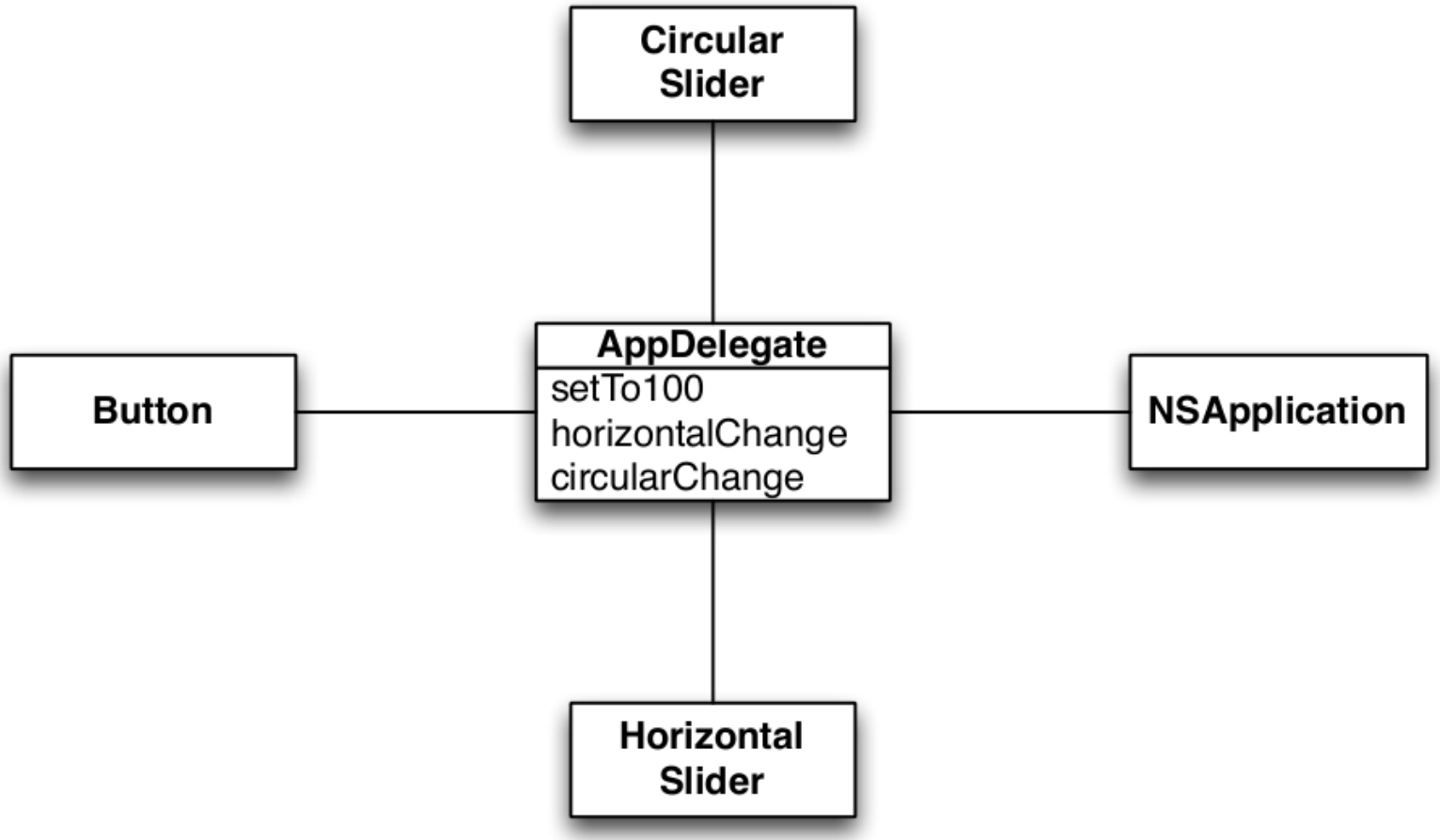
- From the source code, it looks like there's just one object (AppDeligate)
- What is MainMenu.xib?
- .xib is a way Cocoa can store GUIs

Object Oriented Example - GUI



- Looking into the actual GUI yields an additional 19 objects!
- Note that there's still a Main Menu object, which is going to have component objects...

Object Oriented Example - GUI



Object Oriented Example - GUI

- In Cocoa applications, there's a main.m file consisting of a C function

```
int main(int argc, char **argv) {
    return NSApplicationMain(argc, (const char **)argv);
}
```

- This is a bootstrap - procedural code which creates a single object
- When the object is created, it load the .xib file, creates the GUI and starts processing events

Procedural Approach

Functional Decomposition

1. Initialization
   - 1.1 Set Game Board to All Blanks
   - 1.2 Set Current Player to 'X'
2. Repeat Until a Player Wins
   - 2.1 Current Player Selects Valid Move
   - 2.2 If Three in a Row Present
     - 2.2.1 Current Player Wins
   - 2.3 Print Board to Screen
3. Print Winner

Procedural Approach

```
# Initialize the Board and Player
board = [' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ']
currentPlayer = 'X'
playerWon = False
numberOfMoves = 0

printBoard(board)

# Let players move until someone wins, or the board is filled
while not playerWon and numberOfMoves < 9:
    # Current player selects valid move
    index = getValidMove(board, currentPlayer)
    board[index] = currentPlayer

    # Did the current player win?
    playerWon = hasWon(board)

    # Print the board
    printBoard(board)

    # If the player didn't win, swap the players
    if not playerWon:
        currentPlayer = 'O' if currentPlayer == 'X' else 'X'
    numberOfMoves += 1

# Print if there's a winner, otherwise a tie
if playerWon:
    print "%s has won the game" % currentPlayer
else:
    print "It was a tie - neither player won"
```

Tic-Tac-Toe Program

- Good so far, but the requirements have changed so that an AI player can be used...
  - Not too hard to change, just add a few if/else blocks and additional variables

Tic-Tac-Toe Program

- Good so far, but the requirements have changed so that an AI player can be used...
  - Not too hard to change, just add a few if/else blocks and additional variables
  - But, this does increase cyclomatic complexity
  - So, the code becomes just a bit harder to maintain

Tic-Tac-Toe Program

- Good so far, but the requirements have changed so that an AI player can be used...
  - Not too hard to change, just add a few if/else blocks and additional variables
  - But, this does increase cyclomatic complexity
  - So, the code becomes just a bit harder to maintain
- Let's start using objects in our code!
  - After all, all the cool kids are doing it
  - And, OO *is* the silver bullet for software development, right?
  - And, we're in an object oriented design and analysis class, right?

Tic-Tac-Toe Program

- Good so far, but the requirements have changed so that an AI player can be used...
  - Not too hard to change, just add a few if/else blocks and additional variables
  - But, this does increase cyclomatic complexity
  - So, the code becomes just a bit harder to maintain
- Let's start using objects in our code!
  - After all, all the cool kids are doing it
  - And, OO *is* the silver bullet for software development, right?
  - And, we're in an object oriented design and analysis class, right?
- So, what would make a good object / class?

Tic-Tac-Toe Program

- Good so far, but the requirements have changed so that an AI player can be used...
  - Not too hard to change, just add a few if/else blocks and additional variables
  - But, this does increase cyclomatic complexity
  - So, the code becomes just a bit harder to maintain
- Let's start using objects in our code!
  - After all, all the cool kids are doing it
  - And, OO *is* the silver bullet for software development, right?
  - And, we're in an object oriented design and analysis class, right?
- So, what would make a good object / class?
  - There are many choices
  - Hopefully, making a *Player* class was one of them
  - Then, we can use *polymorphism* or *inheritance* or one of those other OO terms, right?

Tic-Tac-Toe Program, version 2

- Great! Not too much change to the code, and the functionality we want!

```
class Player:
    def __init__(self, symbol):
        self.symbol = symbol

    def getValidMove(self, board):
        index = -1
        validMove = False
        while not validMove:
            row = input("%s, select a row: "
                        % self.symbol)
            col = input("%s, select a column: "
                        % self.symbol)
            index = 3*(row-1) + (col-1)
            validMove = board[index] == ' '

        return index

class AIPlayer(Player):
    def getValidMove(self, board):
        index = -1
        validMove = False
        while not validMove:
            index = random.randrange(0,9)
            validMove = board[index] == ' '

        return index
```

Tic-Tac-Toe Program, version 2

- Great! Not too much change to the code, and the functionality we want!
- Good thing too, because there's a lot of ways to design game AI:

```
class Player:
    def __init__(self, symbol):
        self.symbol = symbol

    def getValidMove(self, board):
        index = -1
        validMove = False
        while not validMove:
            row = input("%s, select a row: "
                        % self.symbol)
            col = input("%s, select a column: "
                        % self.symbol)
            index = 3*(row-1) + (col-1)
            validMove = board[index] == ' '

        return index

class AIPlayer(Player):
    def getValidMove(self, board):
        index = -1
        validMove = False
        while not validMove:
            index = random.randrange(0,9)
            validMove = board[index] == ' '

        return index
```

Tic-Tac-Toe Program, version 2

- Great! Not too much change to the code, and the functionality we want!
- Good thing too, because there's a lot of ways to design game AI:

- HumanPlayer
- LookupTablePlayer
- MinimaxPlayer
- AlphaBetaPlayer
- NaiveBayesPlayer
- BayesNetPlayer
- AStarPlayer
- RandomPlayer
- TreeSearchPlayer
- HeuristicPlayer
- QLearningPlayer
- MonteCarloPlayer
- ANNPlayer
- ...

```
class Player:
    def __init__(self, symbol):
        self.symbol = symbol

    def getValidMove(self, board):
        index = -1
        validMove = False
        while not validMove:
            row = input("%s, select a row: "
                        % self.symbol)
            col = input("%s, select a column: "
                        % self.symbol)
            index = 3*(row-1) + (col-1)
            validMove = board[index] == ' '

        return index

class AIPlayer(Player):
    def getValidMove(self, board):
        index = -1
        validMove = False
        while not validMove:
            index = random.randrange(0,9)
            validMove = board[index] == ' '

        return index
```

Tic-Tac-Toe Program, version 3

- The project manager likes the AI players you just spent six months developing
- Now, she would like people to be able play this game in a web browser
- Why not do the same thing we did with players?

Tic-Tac-Toe Program, version 3

- The project manager likes the AI players you just spent six months developing
- Now, she would like people to be able play this game in a web browser
- Why not do the same thing we did with players?
- Let's create an object which handles drawing the board
  - Each kind of Board class handles the specific drawing needs
  - Use different Board objects based on context
- Let's create a Board class!
- and an HTMLBoard class!

Tic-Tac-Toe Program, version 3

- The project manager likes the AI players you just spent six months developing
- Now, she would like people to be able play this game in a web browser
- Why not do the same thing we did with players?
- Let's create an object which handles drawing the board
  - Each kind of Board class handles the specific drawing needs
  - Use different Board objects based on context
- Let's create a Board class!
- and an HTMLBoard class!
- ...and an AndroidPhoneBoard class
- ...and an iOSBoard class
- ...and an XBoxBoard class
- ...and a PS4Board class
- ...

Backgammon Program, version 1

- The Tic-Tac-Toe program is quite a success! Now, how about a Backgammon program?
- Same features are desired, just a different game

Backgammon Program, version 1

- The Tic-Tac-Toe program is quite a success! Now, how about a Backgammon program?
- Same features are desired, just a different game
- No problem, you'll have it done in two months!
  - Make a class for the Dice
  - A little logic to pick valid moves
  - Reuse the Tic-Tac-Toe classes

Backgammon Program, version 1

- The Tic-Tac-Toe program is quite a success! Now, how about a Backgammon program?
- Same features are desired, just a different game
- No problem, you'll have it done in two months!
  - Make a class for the Dice
  - A little logic to pick valid moves
  - Reuse the Tic-Tac-Toe classes
- First, let's reuse the AI classes from Tic-Tac-Toe

Backgammon Program, version 1

- The Tic-Tac-Toe program is quite a success! Now, how about a Backgammon program?
- Same features are desired, just a different game
- No problem, you'll have it done in two months!
  - Make a class for the Dice
  - A little logic to pick valid moves
  - Reuse the Tic-Tac-Toe classes
- First, let's reuse the AI classes from Tic-Tac-Toe
  - Err, those were written *for* Tic-Tac-Toe, *not* Backgammon
  - In fact, *all* those classes need to be rewritten from scratch!
  - Done in eight months...

Backgammon Program, version 1

- The Tic-Tac-Toe program is quite a success! Now, how about a Backgammon program?
- Same features are desired, just a different game
- No problem, you'll have it done in two months!
  - Make a class for the Dice
  - A little logic to pick valid moves
  - Reuse the Tic-Tac-Toe classes
- First, let's reuse the AI classes from Tic-Tac-Toe
  - Err, those were written *for* Tic-Tac-Toe, *not* Backgammon
  - In fact, *all* those classes need to be rewritten from scratch!
  - Done in eight months...
- Well, let's reuse the Board classes

Backgammon Program, version 1

- The Tic-Tac-Toe program is quite a success! Now, how about a Backgammon program?
- Same features are desired, just a different game
- No problem, you'll have it done in two months!
  - Make a class for the Dice
  - A little logic to pick valid moves
  - Reuse the Tic-Tac-Toe classes
- First, let's reuse the AI classes from Tic-Tac-Toe
  - Err, those were written *for* Tic-Tac-Toe, *not* Backgammon
  - In fact, *all* those classes need to be rewritten from scratch!
  - Done in eight months...
- Well, let's reuse the Board classes
  - But, those classes draw Tic-Tac-Toe boards, not Backgammon boards
  - So, gotta redo those as well (done in a year...)

For Next Time

- Sign up for the course Piazza page `http://piazza.com/colorado/spring2014/csci4448/`
- Create a post on Piazza introducing yourself
  - Name, major, current year (Junior, Senior)
  - Specialization / interest
  - Possible project topics
- Fill out the entry survey
  - Link will be provided on course webpage and Piazza
  - Results will be used to determine which language(s) to use for examples, assignments, etc.
- Please try to have these finished by Friday, January 17 at 6:00pm. (Before class on Thursday would be better)