Software Engineering

CSC440/640
Prof. Schweitzer
Week 3

Project Management Myths

Myth: *If a task takes 4 hours for one person to complete, than it should take 2 hours for two people to complete.*

Reality: *9 Women Cannot Make a Baby in 1 Month.*

Task Sharing

- If one farm hand can pick a strawberry field in 10 days, ten farm hands can pick the same strawberry field in 1 day
- One elephant can produce a calf in 22 months, but 22 elephants cannot possibly produce that calf in 1 month

- Unlike elephant production, it is possible to share coding tasks between members of a team
- Unlike strawberry picking, team members must interact in a meaningful and effective way

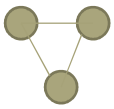
Programming Team Organization

- Example:
  - Sheila and Harry code two modules, m1 and m2, say
- What can go wrong
  - Both Sheila and Harry may code m1, and ignore m2
  - Sheila may code m1, Harry may code m2. When m1 calls m2 it passes 4 parameters; but m2 requires 5 parameters
  - Or, the order of parameters in m1 and m2 may be different
  - Or, the order may be same, but the data types may be slightly different

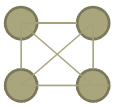
Communication Paths

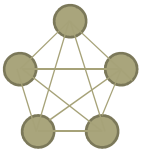
2 People
1 Path

3 People
3 Paths

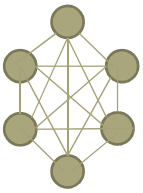


4 People
6 Paths





$$P = \frac{N(N-1)}{2}$$



5 People
10 Paths

6 People
15 Paths

Democratic Team Approach

- Basic underlying concept — egoless programming
- Programmers can be highly attached to their code
  - They even name their modules after themselves
  - They see their modules as extension of themselves
- If a programmer sees a module as an extension of his/her ego, he/she is not going to try to find all the errors in "his"/"her" code
  - If there is an error, it is termed a *bug*
  - The fault could have been prevented if the code had been better guarded against the "bug"
  - "Shoo-Bug" aerosol spray

Democratic Team Approach

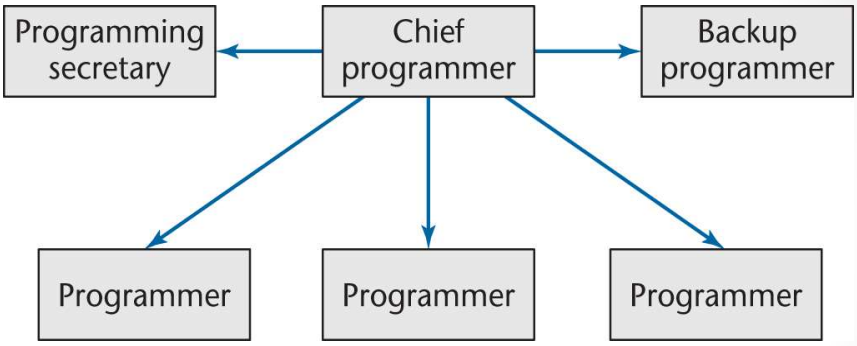
- Egoless programming
  - Restructure the social environment
  - Restructure programmers' values
  - Encourage team members to find faults in code
  - A fault must be considered a normal and accepted event
  - The team as whole will develop an ethos, a group identity
  - Modules will "belong" to the team as whole
  - A group of up to 10 egoless programmers constitutes a democratic team

Strengths & Difficulties with Democratic Team Approach

- Management may have difficulties
  - Democratic teams are hard to introduce into an undemocratic environment
- Democratic teams are enormously productive
- They work best when the problem is difficult
- They function well in a research environment

- Problem:
  - Democratic teams have to spring up spontaneously

Classical Chief Programmer Team



- Six programmers, but now only 5 lines of communication

Classical Chief Programmer Team

- The basic idea behind the concept
  - Analogy: chief surgeon directing an operation, assisted by
    - Other surgeons
    - Anesthesiologists
    - Nurses
    - Other experts, such as cardiologists, nephrologists
- Two key aspects
  - Specialization
  - Hierarchy

Classical Chief Programmer Team

- Chief programmer
  - Successful manager and highly skilled programmer
  - Does the architectural design
  - Allocates coding among the team members
  - Writes the critical (or complex) sections of the code
  - Handles all the interfacing issues
  - Reviews the work of the other team members
  - Is personally responsible for every line of code

Classical Chief Programmer Team

- Back-up programmer
  - Necessary only because the chief programmer is human
  - The back-up programmer must be in every way as competent as the chief programmer, and
  - Must know as much about the project as the chief programmer
  - The back-up programmer does black-box test case planning and other tasks that are independent of the design process

Classical Chief Programmer Team

- Programming secretary
  - A highly skilled, well paid, central member of the chief programmer team
  - Responsible for maintaining the program production library (documentation of the project), including:
    - Source code listings
    - JCL
    - Test data
  - Programmers hand their source code to the secretary who is responsible for
    - Conversion to machine-readable form
    - Compilation, linking, loading, execution, and running test cases (this was 1971, remember!)
    - Much of this has been replaced with automation now

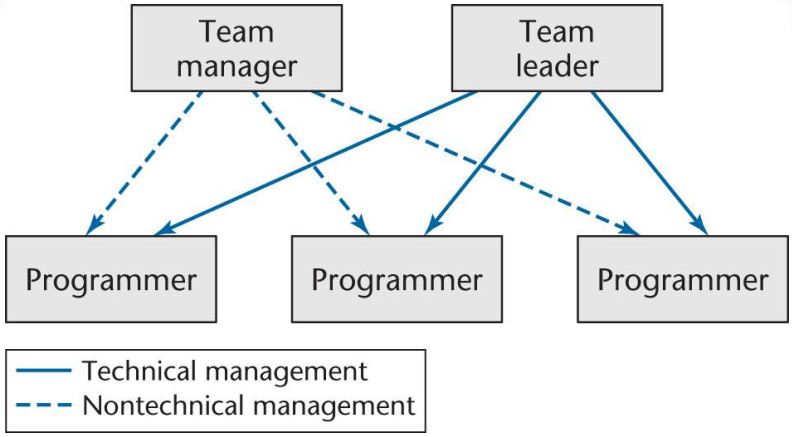
Classical Chief Programmer Team

- Programmers
  - Do nothing but program
  - All other aspects are handled by the programming secretary

Beyond CP and Democratic Teams

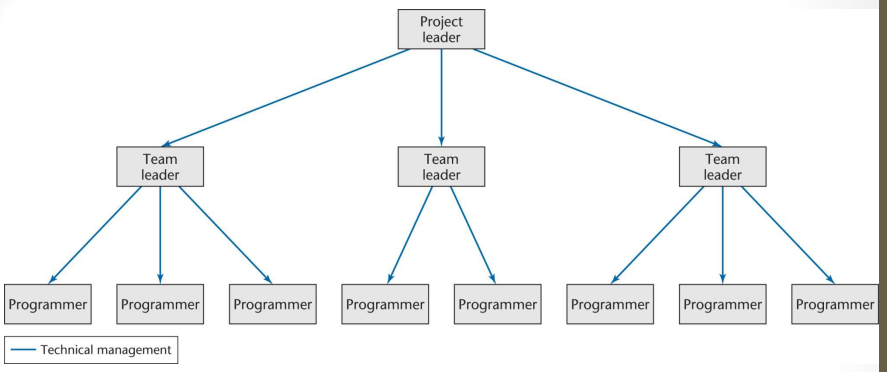
- We need ways to organize teams that
  - Make use of the strengths of democratic teams and chief programmer teams, and
  - Can handle teams of 20 (or 120) programmers
- A strength of democratic teams
  - A positive attitude to finding faults
- Use CPT in conjunction with code walkthroughs or inspections
- The chief programmer is personally responsible for every line of code
  - He/she must therefore be present at reviews
- The chief programmer is also the team manager
  - He/she must therefore *not* be present at reviews!

Beyond CP and Democratic Teams



- Solution
  - Reduce the managerial role of the chief programmer

Larger Projects



- The nontechnical side is similar
  - For even larger products, add additional layers

Strengths of Pair Programming

- Programmers should not test their own code
  - One programmer draws up the test cases, the other tests the code
- If one programmer leaves, the other is sufficiently knowledgeable to continue working with another pair programmer
- An inexperienced programmer can learn from his or her more experienced team member
- Centralized computers promote egoless programming

Choosing an Appropriate Team Organization

- Exceedingly little research has been done on software team organization
  - Instead, team organization has been based on research on group dynamics in general
- Without relevant experimental results, it is hard to determine optimal team organization for a specific product

What Is a Module?

- A lexically contiguous sequence of program statements, bounded by boundary elements, with an aggregate identifier
  - "Lexically contiguous"
    - Adjoining in the code
  - "Boundary elements"
    - `{ ... }`
    - `begin ... end`
  - "Aggregate identifier"
    - A name for the entire module

Composite/Structured Design

- A method for breaking up a product into modules to achieve
  - Maximal interaction within a module, and
  - Minimal interaction between modules
- Module cohesion
  - Degree of interaction within a module
  - *High Cohesion* is Desired
- Module coupling
  - Degree of interaction between modules
  - *Low Coupling* is Desired

Cohesion

- The degree of interaction within a module
- Seven categories or levels of cohesion (non-linear scale)

7.	Informational cohesion	(Good)
6.	Functional cohesion	
5.	Communicational cohesion	
4.	Procedural cohesion	
3.	Temporal cohesion	
2.	Logical cohesion	
1.	Coincidental cohesion	(Bad)

Coincidental Cohesion

- A module has coincidental cohesion if it performs multiple, completely unrelated actions
- Example:
  - print_next_line,
    reverse_string_of_characters_comprising_second_ parameter,
    add_7_to_fifth_parameter,
    convert_fourth_parameter_to_floating_point
- Such modules arise from rules like
  - "Every module will consist of between 35 and 50 statements"

Why Is Coincidental Cohesion So Bad?

- It degrades maintainability
- A module with coincidental cohesion is not reusable
- The problem is easy to fix
  - Break the module into separate modules, each performing one task

Logical Cohesion

- A module has logical cohesion when it performs a series of related actions, one of which is selected by the calling module
- Example 1:

```
function_code = 7;
new_operation (op code, dummy_1, dummy_2, dummy_3);
// dummy_1, dummy_2, and dummy_3 are dummy variables,
// not used if function code is equal to 7
```

- Example 2:
  - An object performing all input and output
- Example 3:
  - One version of OS/VS2 contained a module with logical cohesion performing 13 different actions. The interface contains 21 pieces of data

Why Is Logical Cohesion So Bad?

- The interface is difficult to understand
- Code for more than one action may be intertwined
- Difficult to reuse

Temporal Cohesion

- A module has temporal cohesion when it performs a series of actions related in time
- Example:
  - open_old_master_file, new_master_file, transaction_file, and print_file; initialize_sales_district_table, read_first_transaction_record, read_first_old_master_record (a.k.a. perform_initialization)
- The actions of this module are weakly related to one another, but strongly related to actions in other modules
  - Consider `sales_district_table`
- Not reusable

Procedural Cohesion

- A module has procedural cohesion if it performs a series of actions related by the procedure to be followed by the product
- Example:
  - read_part_number_and_update_repair_record_on_master_file
- The actions are still weakly connected, so the module is not reusable

Communicational Cohesion

- A module has communicational cohesion if it performs a series of actions related by the procedure to be followed by the product, but in addition all the actions operate on the same data
- Example 1:
  - update_record_in_database_and_write_it_to_audit_trail
- Example 2:
  - calculate_new_coordinates_and_send_them_to_terminal
- Still lack of reusability

Functional Cohesion

- A module with functional cohesion performs exactly one action
- Example 1:
  - `get_temperature_of_furnace`
- Example 2:
  - `compute_orbital_of_electron`
- Example 3:
  - `write_to_diskette`
- Example 4:
  - `calculate_sales_commission`

Why Is Functional Cohesion So Good?

- More reusable
- Corrective maintenance is easier
  - Fault isolation
  - Fewer regression faults
- Easier to extend a product

Informational Cohesion

- A module has informational cohesion if it performs a number of actions, each with its own entry point, with independent code for each action, all performed on the same data structure

Coupling

- The degree of interaction between two modules
  - Five categories or levels of coupling (non-linear scale)

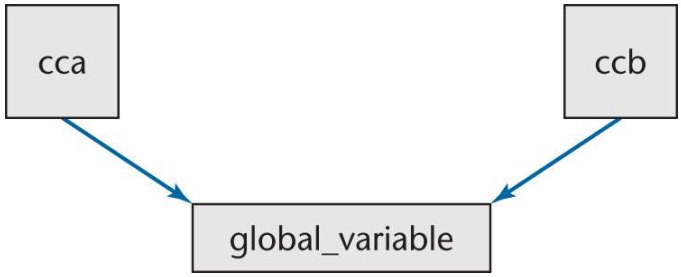
5.	Data coupling	(Good)
4.	Stamp coupling	
3.	Control coupling	
2.	Common coupling	
1.	Content coupling	(Bad)

Content Coupling

- Two modules are content coupled if one directly references contents of the other
- Example 1:
  - Module p modifies a statement of module q
- Example 2:
  - Module p refers to local data of module q in terms of some numerical displacement within q
- Example 3:
  - Module p branches into a local label of module q
- Almost any change to module `q`, even recompiling `q` with a new compiler or assembler, requires a change to module `p`

Common Coupling

- Two modules are common coupled if they have write access to global data



- Example 1
  - Modules cca and ccb can access and change the value of global_variable

Why Is Common Coupling So Bad?

- It contradicts the spirit of structured programming
  - The resulting code is virtually unreadable

```
while (global_variable == 0)
{
    if (argument_xyz > 25)
          modeule_3();
    else
          module_4();
}
```

  - What causes this loop to terminate?

Why Is Common Coupling So Bad?

- Modules can have side-effects
  - This affects their readability
  - Example: edit_this_transaction (record_7)
  - The entire module must be read to find out what it does
- A change during maintenance to the declaration of a global variable in one module necessitates corresponding changes in other modules
- Common-coupled modules are difficult to reuse
- Common coupling between a module p and the rest of the product can change without changing p in any way
  - Clandestine common coupling
  - Example: The Linux kernel
- A module is exposed to more data than necessary
  - This can lead to computer crime

Why Is Common Coupling So Bad?

- Common coupling between a module p and the rest of the product can change without changing p in any way
  - Clandestine common coupling
  - Example: The Linux kernel
- A module is exposed to more data than necessary
  - This can lead to computer crime

Control Coupling

- Two modules are control coupled if one passes an element of control to the other
- Example 1:
  - An operation code is passed to a module with logical cohesion
- Example 2:
  - A control switch passed as an argument

Control Coupling

- Module p calls module q
- Message:
  - I have failed — data
- Message:
  - I have failed, so write error message ABC123 — control
- The modules are not independent
  - Module q (the called module) must know the internal structure and logic of module p
  - This affects reusability
- Associated with modules of logical cohesion

Why Is Control Coupling So Bad?

- The modules are not independent
  - Module q (the called module) must know the internal structure and logic of module p
  - This affects reusability
- Associated with modules of logical cohesion

Stamp Coupling

- Some languages allow only simple variables as parameters
  - `part_number`
  - `satellite_altitude`
  - `degree_of_multiprogramming`
- Many languages also support the passing of data structures
  - `part_record`
  - `satellite_coordinates`
  - `segment_table`
- Two modules are stamp coupled if a data structure is passed as a parameter, but the called module operates on some but not all of the individual components of the data structure

Data Coupling

- Two modules are data coupled if all parameters are homogeneous data items (simple parameters, or data structures all of whose elements are used by called module)
- Examples:
  - `display_time_of_arrival (flight_number);`
  - `compute_product (first_number, second_number);`
  - `get_job_with_highest_priority (job_queue);`
- The difficulties of content, common, control, and stamp coupling are not present
- Maintenance is easier

Key Definitions

Abstract data type: a data type together with the operations performed on instantiations of that data type (Section 7.5)

Abstraction: a means of achieving stepwise refinement by suppressing unnecessary details and accentuating relevant details (Section 7.4.1)

Class: an abstract data type that supports inheritance (Section 7.7)

Cohesion: the degree of interaction within a module (Section 7.1)

Coupling: the degree of interaction between two modules (Section 7.1)

Data encapsulation: a data structure together with the operations performed on that data structure (Section 7.4)

Encapsulation: the gathering together into one unit of all aspects of the real-world entity modeled by that unit (Section 7.4.1)

Information hiding: structuring the design so that the resulting implementation details are hidden from other modules (Section 7.6)

Object: an instantiation of a class (Section 7.7)

Data Encapsulation and Development

- Abstraction
  - Conceptualize problem at a higher level
    - Job queues and operations on job queues
  - Not a lower level
    - Records or arrays

Stepwise Refinement

- 1. Design the product in terms of higher level concepts
  - It is irrelevant how job queues are implemented
- 2. Then design the lower level components
  - Totally ignore what use will be made of them

Stepwise Refinement

- In the 1st step, assume the existence of the lower level
  - Our concern is the behavior of the data structure
    - job_queue
- In the 2nd step, ignore the existence of the higher level
  - Our concern is the implementation of that behavior
- In a larger product, there will be many levels of abstraction

Data Encapsulation and Maintenance

- Identify the aspects of the product that are likely to change
- Design the product so as to minimize the effects of change
  - Data structures are unlikely to change
  - Implementation details may change
- Data encapsulation provides a way to cope with change

Information Hiding

- Data abstraction
  - The designer thinks at the level of an ADT
- Procedural abstraction
  - Define a procedure — extend the language
- Both are instances of a more general design concept, information hiding
  - Design the modules in a way that items likely to change are hidden
  - Future change is localized
  - Changes cannot affect other modules

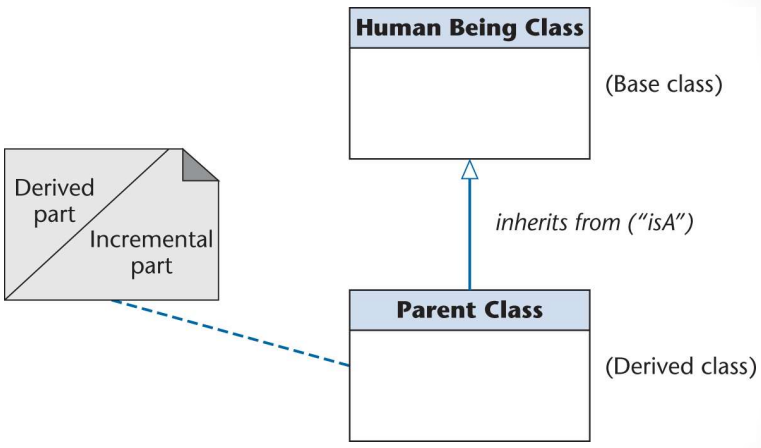
Objects

- First refinement
  - The product is designed in terms of abstract data types
  - Variables (“objects”) are instantiations of abstract data types
- Second refinement
  - Class: an abstract data type that supports inheritance
  - Objects are instantiations of classes

Inheritance

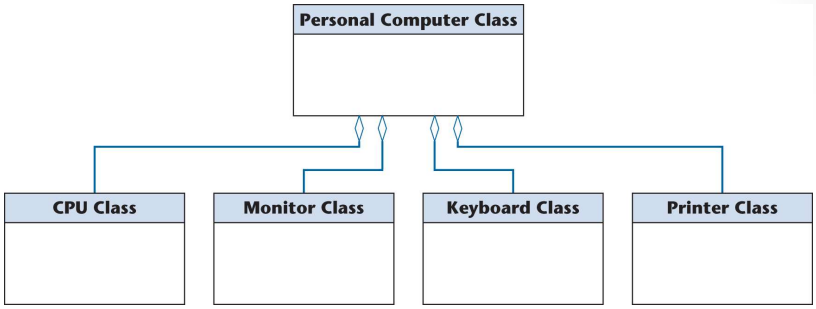
- Define **`HumanBeingClass`** to be a class
  - An instance of **`HumanBeingClass`** has attributes, such as
    - **`age, height, gender`**
  - Assign values to the attributes when describing an object
- Define **`ParentClass`** to be a *subclass* of **`HumanBeingClass`**
  - An instance of **`ParentClass`** has all the attributes of an instance of **`HumanBeingClass`**, plus attributes of his/her own
    - `nameOfOldestChild, numberOfChildren`
  - An instance of **`ParentClass`** inherits all attributes of **`HumanBeingClass`**

Inheritance



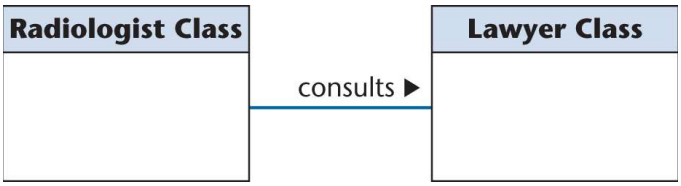
- UML notation
  - Inheritance is represented by a large open triangle

Aggregation



- UML notation for aggregation — open diamond

Association



- UML notation for association — line
  - Optional navigation triangle

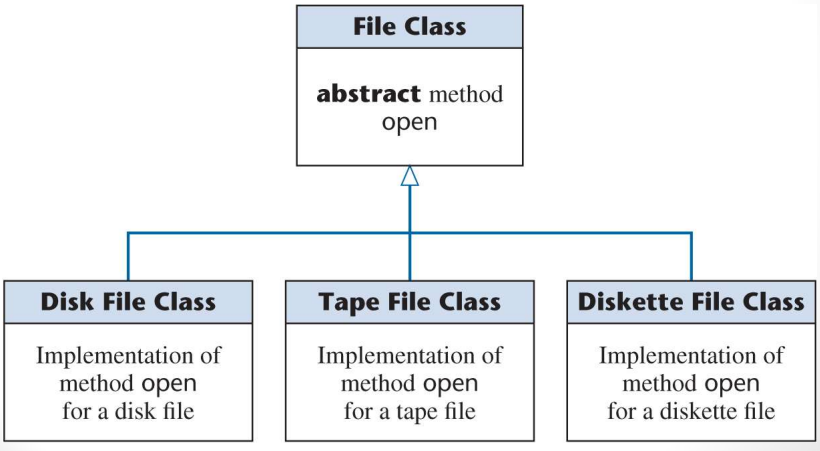
Inheritance, Polymorphism and Dynamic Binding

- Classical code to open a file
  - The correct method is explicitly selected

```
switch (file_type)
{
    case 1:
      open_disk_file ( );          // file_type 1 corresponds to a disk file
      break;
    case 2:
      open_tape_file ( );          // file_type 2 corresponds to a tape file
      break;
    case 3:
      open_diskette_file ( );      // file_type 3 corresponds to a diskette file
       break;
}
```

Inheritance, Polymorphism and Dynamic Binding

- Object-oriented paradigm



Inheritance, Polymorphism and Dynamic Binding

- Polymorphism and dynamic binding
  - Can have a negative impact on maintenance
    - The code is hard to understand if there are multiple possibilities for a specific method
- Polymorphism and dynamic binding
  - A strength and a weakness of the object-oriented paradigm

The Object-Oriented Paradigm

- Reasons for the success of the object-oriented paradigm
  - The object-oriented paradigm gives overall equal attention to data and operations
    - At any one time, data or operations may be favored
  - A well-designed object (high cohesion, low coupling) models all the aspects of one physical entity
  - Implementation details are hidden
- The reason why the structured paradigm worked well at first
  - The alternative was no paradigm at all

The Object-Oriented Paradigm

- How do we know that the object-oriented paradigm is the best current alternative?
  - We don't
  - However, most reports are favorable
    - Experimental data (e.g., IBM [1994])
    - Survey of programmers (e.g., Johnson [2000])
  - Other paradigms are becoming popular for certain problem domains, such as Functional Programming.

Weaknesses of the Object-Oriented Paradigm

- Development effort and size can be large
- One's first object-oriented project can be larger than expected
  - Even taking the learning curve into account
  - Especially if there is a GUI
- However, some classes can frequently be reused in the next project
  - Especially if there is a GUI

Weaknesses of the Object-Oriented Paradigm

- Inheritance can cause problems
  - The fragile base class problem
  - To reduce the ripple effect, all classes need to be carefully designed up front
- Unless explicitly prevented, a subclass inherits all its parent's attributes
  - Objects lower in the tree can become large
  - "Use inheritance where appropriate"
  - Exclude unneeded inherited attributes
  - Different Languages Make Different Choices in This Area… Are Methods overridable by default?

Next Week

- Chapters 8 & 11
  - Weekly Quiz Will Come From This Material
  - We'll Backtrack to Chapter 9 Later

Software Project – Requirements Definition

- Create a Requirements Specification Document Out of Appendix A
- Recommended Approach: Use Cases
- For Each Use Case:
  - Define Actors
  - Title – As a _____ I want to be able to _____
  - Story
    - Define the primary workflow
    - Define alternate paths, such as error conditions
  - Acceptance Criteria
    - What can you test at the end to determine if the story was completely successfully?
- **Due: 10/12/2016 at 6 PM**